Week 1. Lecture Notes

Topics: Insertion Sort

Analysis of Insertion Sort

Recurrence of Merge sort

Substitution Method

The problem of Sorting

Input: a sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers

Output: a permutation $\langle a_1', a_2', \ldots, a_n' \rangle$

such that

$a_1' \leq a_2' \leq \ldots \leq a_n'$

Example:

Input: 9 3 5 0 4 7

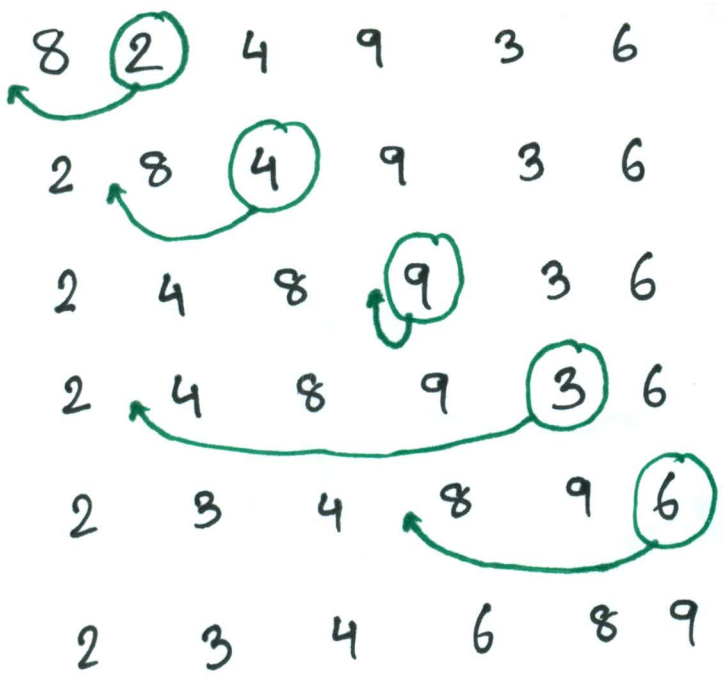
Output: 0 3 4 5 7 9

Pseudo Code: Insertion Sort

pseudo-code

```
INSERTION SORT (A, n)      ▷ A[1,... n]
    for j ← 1 to n
      do  key ← A[i]
          i ← j-1
          while  i>0 and A[i] > key
            do  A[i+1] ← A[i]
                i ← i-1
          A[i+1] = key
```

Example of Insertion Sort



done

Running Time

- The running time depends on the input: an already sorted sequence is easier to sort.

- Parameterize the running time by the size of the input, since short sequences are easier to sort than the long ones.

- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Types of Analysis

Worst-case: (Usually)

$T(n)$ = maximum time of algorithm on any input of size 'n'

Average-case: (Sometimes)

$T(n)$ = expected time of algorithm on any input of size 'n'

Best Case

Cheat with a slow algorithm that works fast on 'some' input

Machine-Independent Time

What is Insertion sort's worst-case time?

- It depends on the speed of our computer:
  - relative speed (on the same machine)
  - absolute speed (on different machines)

Big Idea

Ignore machine-dependent constants

Look at 'growth' of $T(n)$ as $n \to \infty$

"Asymptotic Analysis"

Θ notation

Math:

$\Theta(g(n)) = \{ f(n)$: $\exists$ positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ $\forall\ n \ge n_0 \}$

Engineering

Drop low-order terms; ignore leading constants

Example

$3n^3 - 90n^2 + 5n - 1024 = \Theta(n^3)$

O notation

$O(g(n)) = \{ f(n)$: $\exists$ positive constants c and n_0 such that $f(n) \le cg(n)$ $\forall\ n \ge n_0 \}$

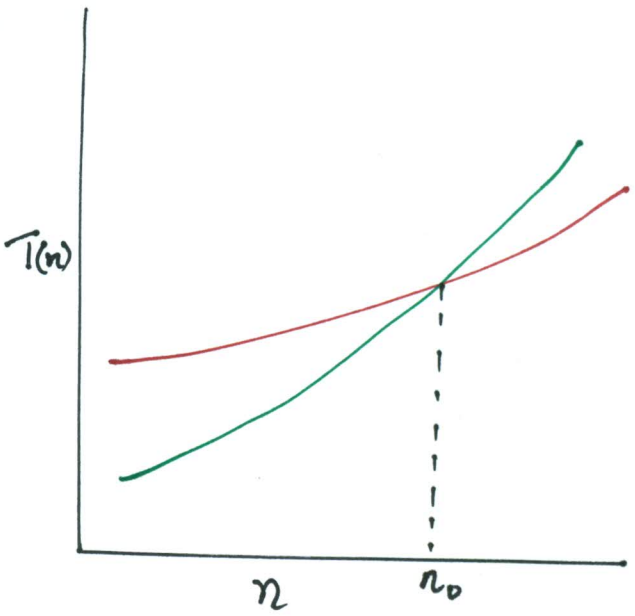
Ω notation

$\Omega(g(n)) = \{ f(n)$: $\exists$ positive constants c and n_0 such that $f(n) \ge cg(n)$ $\forall\ n \ge n_0 \}$

Asymptotic Performance

When n gets large enough a $\Theta(n^2)$ algorithm 'always' beats a $\Theta(n^3)$ algorithm

- We shouldn't ignore asymptotically slower algorithms
- Real world designs situation often calls for a careful balancing of engineering objectives
- Asymptotic analysis is a useful tool to help to structure our thinking



Insertion Sort Analysis

Worst Case: Input inverse sorted

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2) \quad \text{[arithmetic series]}$$

Average Case: All permutations equally likely

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

- Insertion sort is moderately fast for small 'n'
- It is not at all fast for large 'n'.

Merge Sort

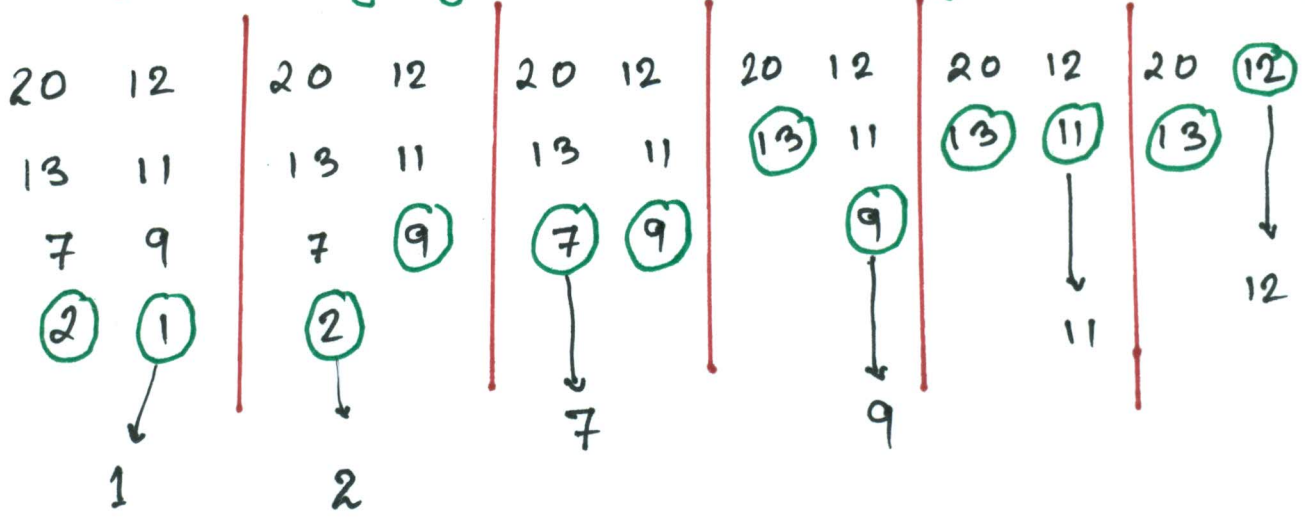
MERGE-SORT $A[1,\ldots,n]$

To sort n numbers

1. If $n=1$, done
2. Recursively sort $A[1,\ldots,\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil+1,\ldots,n]$
3. "Merge" the 2 sorted lists

Key subroutine : MERGE

Example: Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time)

Analyzing Merge Sort

MERGE-SORT (A, n) ▷ $A[1, \ldots, n]$

$T(n)$	To sort n numbers
$\Theta(1)$	1. If $n=1$ done
$2T(n/2)$	2. Recursively sort $A[1, \ldots, \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1, \ldots, n]$
$\Theta(n)$	3. Merge the 2 sorted lists

Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$ but it turns out not to matter asymptotically

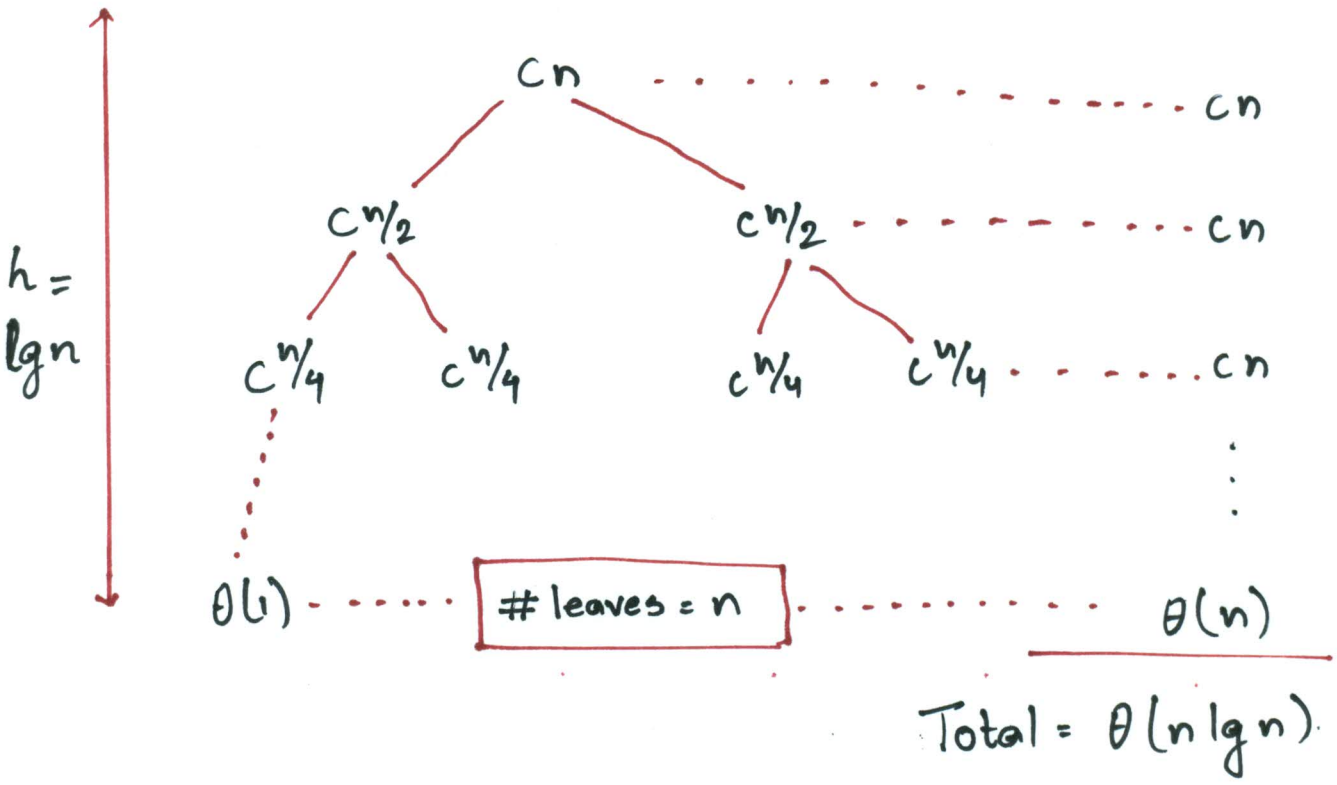
Recurrence for Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- We shall usually omit the base case when $T(n) = \Theta(1)$ for sufficiently small n and when it has no effect on the solutions to the recurrence.

Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



Conclusions

- $\theta(n \lg n)$ grows more slowly than $\theta(n^2)$

  Therefore, merge sort asymptotically beats insertion sort in the worst case

- In practice, merge sort beats insertion sort for $n > 30$ or so

Solving Recurrences : Substitution Method

It is the most general method:

1. Guess the form of solution
2. Verify by induction
3. Solve for constants

Example

$T(n) = 4T(n/2) + n$

- [Assume that $T(1) = \Theta(1)$]
- Guess $O(n^3)$
- Assume that $T(k) \leq ck^3$ for $k < n$
- Prove $T(n) \leq cn^3$ by induction.

Example of Substitution

$$T(n) = 4T(n/2) + n$$

$$\leq 4c(n/2)^3 + n$$

$$= (c/2)n^3 + n$$

$$= cn^3 - ((c/2)n^3 - n) \quad \leftarrow \text{desired - residual}$$

$$\leq cn^3 \quad \leftarrow \text{desired}$$

whenever $((c/2)n^3 - n) \geq 0$, for example if $c \geq 2$, $n \geq 1$

↑ residual

Example (Continued)

- We must also handle the initial conditions that is, ground the induction with base cases.
- **Base**: $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant
- For $1 \le n \le n_0$, we have "$\Theta(1)$" $\le cn^3$, if we pick c big enough

But this bound is not tight

A tighter upper bound

We shall prove that $T(n) = O(n^2)$

Assume that $T(k) \le ck^2$ for $k < n$:

$$T(n) = 4T(n/2) + n$$
$$\le 4cn^2 + n$$
$$= \cancel{O(n)}$$ wrong! We must prove the I.H.
$$= cn^2 - (-n) \quad [\text{desired} - \text{residual}]$$
$$\le cn^2$$

for no choice of $c > 0$

We lose

A tighter upper bound

IDEA: Strengthen the inductive hypothesis

- Subtract a low-order term.

Induction hypothesis:

$$T(k) \le c_1 k^2 - c_2 k \text{ for } k < n$$

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\le 4\left(c_1 (n/2)^2 - c_2 (n/2)\right) + n \\ &= c_1 n^2 - 2c_2 n + n \\ &= c_1 n^2 - c_2 n - (c_2 n - n) \\ &\le c_1 n^2 - c_2 n \quad \text{if } c_2 \ge 1 \end{aligned}$$

We pick c_1 big enough to handle this situation.

Week 2. Lecture Notes

Topics: The Master Method

Divide and Conquer

Strassen's Algorithm

Quick Sort

The Master Method

The Master Method applies to recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ and f is asymptotically positive

Three Common Cases

Compare $f(n)$ with $n^{\log_b a}$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$,

$f(n)$ grows polynomially slower than $n^{\log_b a}$ by a n^{ε} factor

Solution: $T(n) = \Theta(n^{\log_b a})$

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$

$f(n)$ and $n^{\log_b a}$ grows at similar rates

Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$

$f(n)$ grows polynomially faster than $n^{\log_b a}$, by a factor n^{ε} factor)

and $f(n)$ satisfies the 'regularity condition' that of $af(n/b) \leq cf(n)$ for some constant $c < 1$

Solution: $T(n) = \Theta(f(n))$

Examples

1. $T(n) = 4T(n/2) + n$

here $a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n$

Case 1: $f(n) = O(n^{2-\varepsilon})$, for $\varepsilon = 1$

$\therefore T(n) = \Theta(n^2)$

2. $T(n) = 4T(n/2) + n^2$

$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2$

Case 2: $f(n) = \Theta(n^2 \log^0 n)$, i.e. $k = 0$

$\therefore T(n) = \Theta(n^2 \log n)$

3. $T(n) = 4T(n/2) + n^3$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^3$

Case 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(cn/2)^2 \le cn^3$ (reg cond) for $c = 1/2$

$\therefore T(n) = \Theta(n^3)$

4. $T(n) = 4T(n/2) + \frac{n^2}{\lg n}$

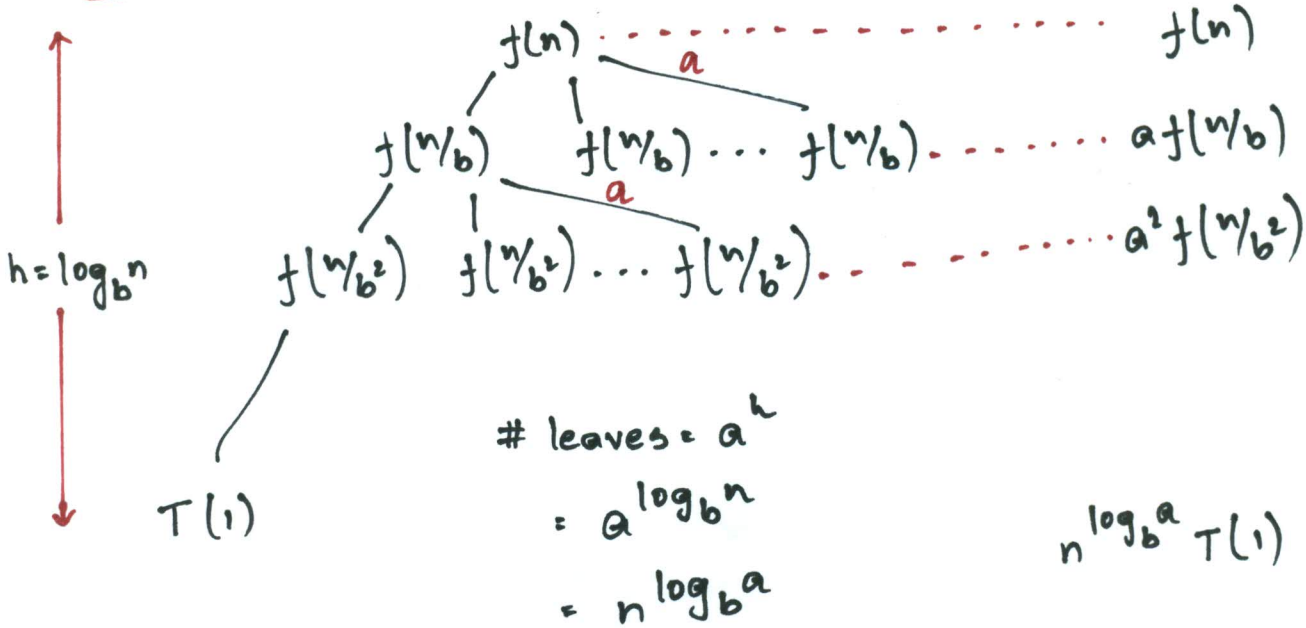
$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = \frac{n^2}{\lg n}$

Master method doesnot apply.

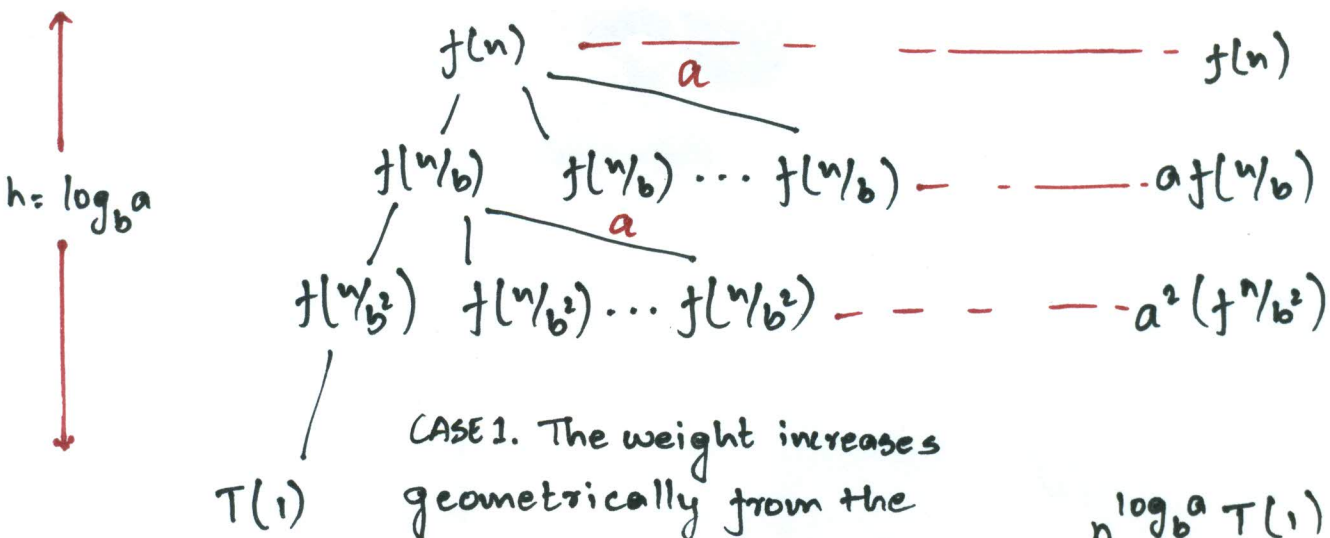
In particular for every constant $\varepsilon > 0$, we have
$n^\varepsilon = \omega(\lg n)$

The Idea of Master Theorem

Recursion Tree:



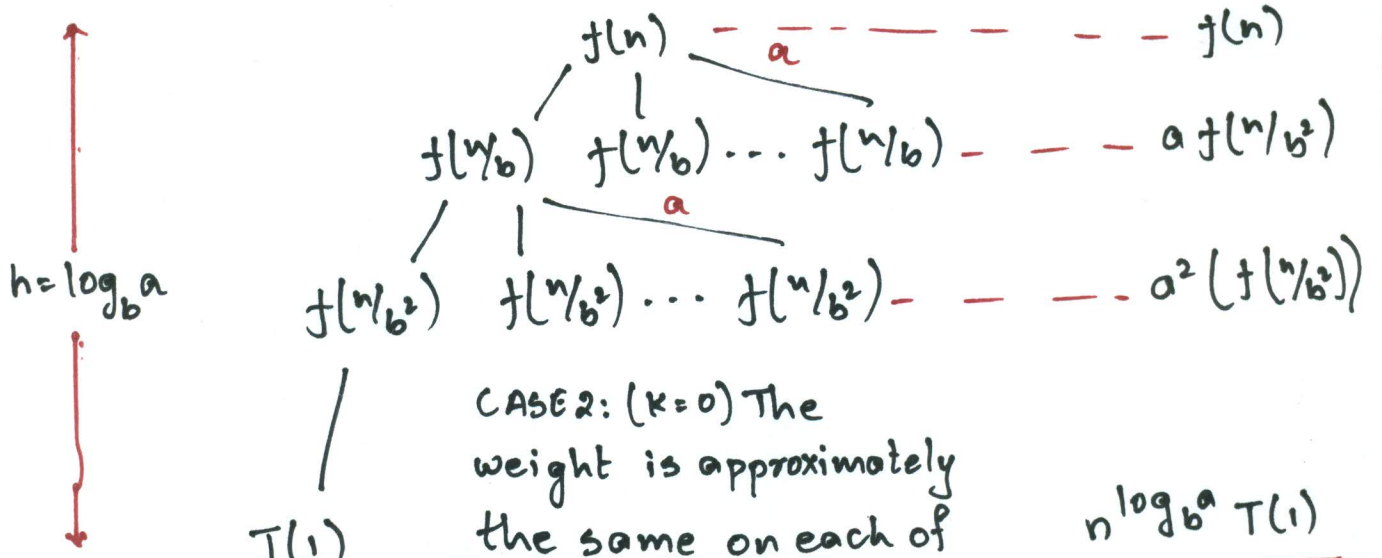
Recursion Tree



CASE 1. The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight

$$\frac{n^{\log_b a} T(1)}{\Theta(n^{\log_b a})}$$

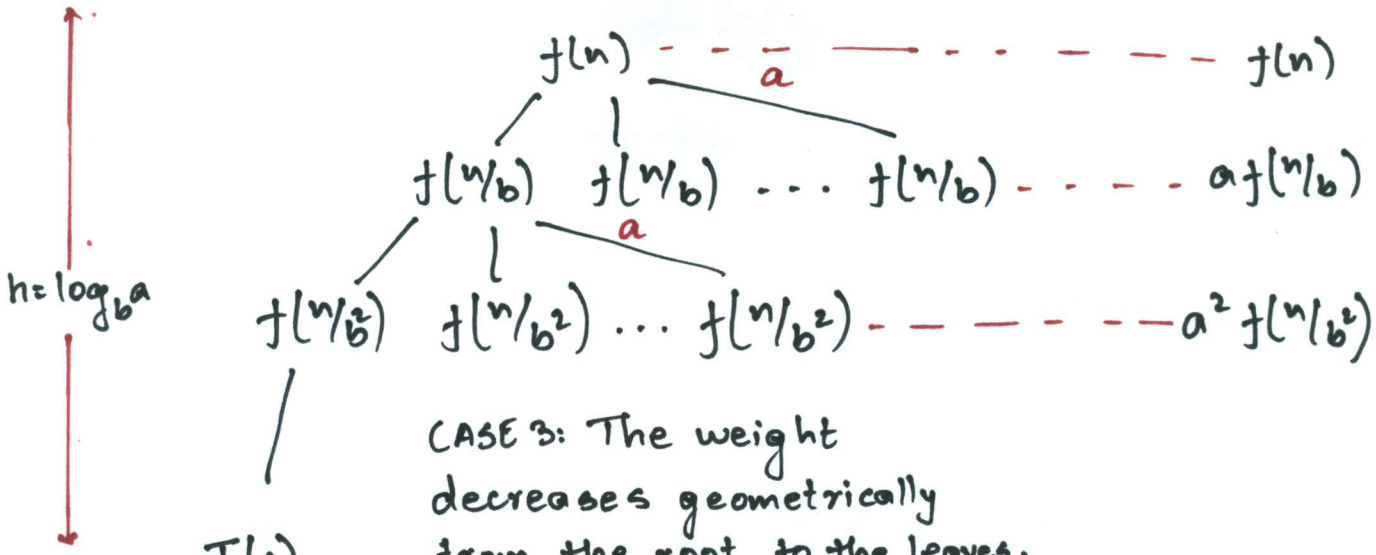
Recursion Tree



CASE 2: (k = 0) The weight is approximately the same on each of the $\log_b n$ levels

$$\frac{n^{\log_b a} T(1)}{\Theta(n^{\log_b a} \lg n)}$$

Recursion Tree



CASE 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

$$\frac{n^{\log_b a} T(1)}{\Theta(f(n))}$$

The divide-and-conquer design paradigm

1. Divide the problem (instance) into subproblems
2. Conquer the subproblems by solving them recursively
3. Combine subproblem solutions

Example: Merge Sort

1. Divide: Trivial
2. Conquer: Recursively sort 2 subarrays
3. Combine: Linear-time merge

$$T(n) = 2T(n/2) + \Theta(n)$$

- 2: # subproblems
- $n/2$: subproblem size
- $\Theta(n)$: work dividing and combining

Binary search

Find an element in a sorted array

1. Divide - Check middle element
2. Conquer - Recursively search 1 subarray
3. Combine - Trivial

Example:

Find 9 in

3 5 7 8 9 12 15

3 5 7 (8) 9 12 15

3 5 7 8 [9 12 15]

3 5 7 8 9 (12) 15

3 5 7 8 (9) 12 15

3 5 7 8 [9] 12 15

Recurrence for binary search

$$T(n) = 1\,T(n/2) + \Theta(1)$$

$$n^{\log_b a} = n^{\log 2.1} = n^{\log 2} = n^0 = 1 \Rightarrow \text{CASE 2 } (k=0)$$

$$\Rightarrow T(n) = \Theta(\log n)$$

Powering a Number

Problem: Compute a^n, $n \in \mathbb{N}$

Naive algorithm: $\Theta(n)$

Divide and conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1)$$

$$\Rightarrow T(n) = \Theta(\lg n)$$

Fibonacci Numbers

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

0 1 1 2 3 5 8 13 21 ...

Naive recursive algorithm: $\Omega(\phi^n)$ (exponential time), where $\phi = (1+\sqrt{5})/2$ is the 'golden ratio.'

Computing Fibonacci Numbers

Naive recursive squaring:

$F_n = \phi^n / \sqrt{5}$, rounded to nearest integer

- Recursive squaring: $\Theta(\lg n)$ time
- This method is unreliable, since floating-point arithmetic is prone to round-off errors.

Bottom-up:

- Compute $F_0, F_1, F_2, \ldots, F_n$ in order, forming each number by summing the two previous.
- Running time: $\Theta(n)$

Recursive Squaring

Theorem:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Algorithm: Recursive squaring

Time = $\Theta(\lg n)$

Proof of theorem: (Induction on n)

For $n = 1$: $\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1$, which is true

Inductive step ($n \geq 2$)

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

■

Matrix Multiplication

Input: $A = [a_{ij}]$, $B = [b_{ij}]$

Output: $C = [c_{ij}] = A \cdot B$, $i, j = 1, 2, \ldots, n$

$$c_{i,j} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \quad , \quad i, j = 1, 2, \ldots, n$$

Standard Algorithm

```
1.  for i ← 1 to n
2.      do for j ← 1 to n
3.          do c_ij ← 0
4.             for k ← 1 to n
5.                 do c_ij ← c_ij + a_ik · b_kj
```

Running time = $\Theta(n^3)$

Divide-and-Conquer Algorithm

IDEA:

$n \times n$ matrix = 2×2 matrices of $(n/2) \times (n/2)$ submatrices

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dh \\ u &= cf + dh \end{aligned}\right\} \begin{array}{l} 8 \text{ mults of } (n/2) \times (n/2) \text{ submatrices} \\ 4 \text{ adds of } (n/2) \times (n/2) \text{ submatrices} \end{array}$$

Analysis of Divide and Conquer algorithm

$$T(n) = 8\,T(n/2) + \Theta(n^2)$$

- 8: # submatrices
- $n/2$: Submatrix size
- $\Theta(n^2)$: work adding submatrices

$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \text{CASE 1: } T(n) = \Theta(n^3)$$

No better than the ordinary algorithm

Strassen's Idea

Multiply 2x2 matrices with only 7 recursive multiplications

$P_1 = a.(f-h)$

$P_2 = (a+b).h$

$P_3 = (c+d).e$

$P_4 = d.(g-e)$

$P_5 = (a+d).(e+h)$

$P_6 = (b-d).(g+h)$

$P_7 = (a-c).(e+f)$

$r = P_5 + P_4 - P_2 + P_6$

$s = P_1 + P_2$

$t = P_3 + P_4$

$u = P_5 + P_1 - P_3 - P_7$

7 mults, 18 adds/subs

Note: No reliance on commutativity of mults.

Here,

$r = P_5 + P_4 - P_2 + P_6$

$= (a+d)(e+h) + d(g-e) - (a+b)h + (b-d)(g+h)$

$= ae + ah + de + dh + dg - de - ah - bh + bg + bh - dg - dh$

$= ae + bg$

Similarly, others can be proved

Strassen's Algorithm

1. Divide: Partition A and B into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using + and -
2. Conquer: Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively
3. Combine: Form C using + and - on $(n/2) \times (n/2)$ submatrices.

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

Analysis of Strassen's Algorithm

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

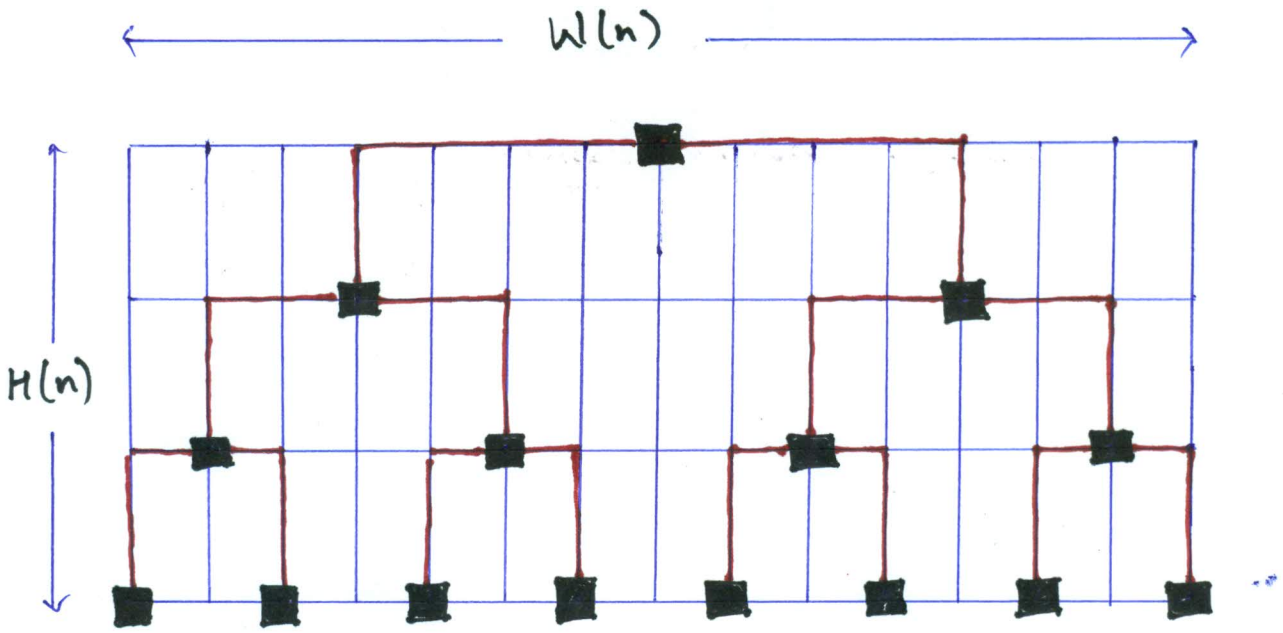
$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow$ CASE 1: $T(n) = \Theta(n^{\log 7})$

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 30$ or so.

Best upto date: $\Theta(n^{2.376\ldots})$

VLSI Layout

Problem: Embed a complete binary tree with 'n' leaves in a grid using minimal area.

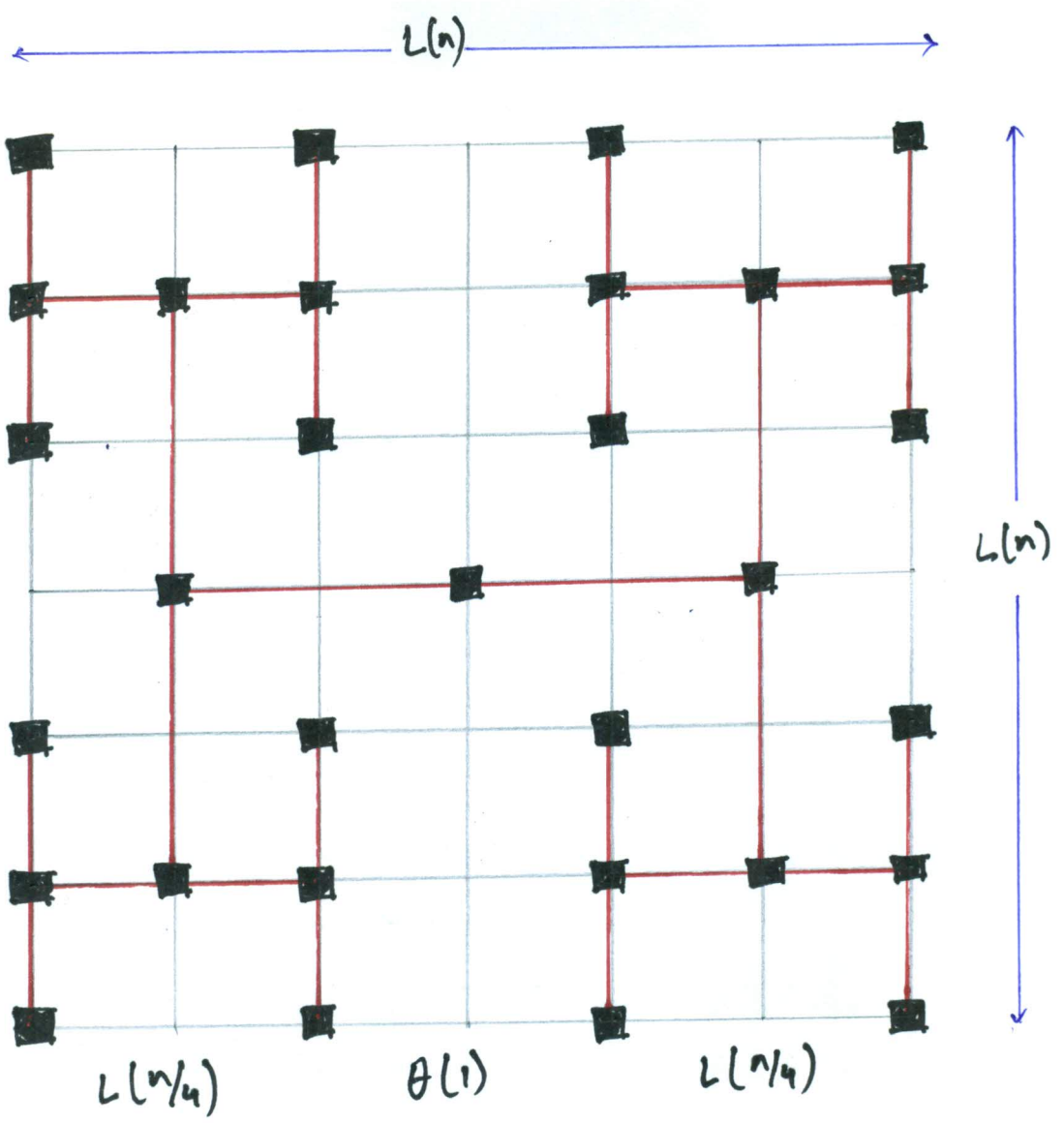


$$H(n) = H(n/2) + \theta(1)$$
$$= \theta(\lg n)$$

$$W(n) = 2W(n/2) + \theta(1)$$
$$= \theta(n)$$

$$\text{Area} = \Theta(n \log n)$$

H-tree embedding



$$L(n) = 2L(n/4) + \Theta(1)$$
$$= \Theta(\sqrt{n})$$

$$\text{Area} = \Theta(n)$$

H-tree embedding

Quick Sort

- Proposed by C.A.R. Hoare in 1962
- Divide and Conquer Algorithm
- Sorts "in place" (like insertion sort)
- Very practical

Divide and Conquer

Quicksort an n-element array:

1. Divide: Partition the array into two subarrays around a pivot x such that elements in lower subarray $\leq x \leq$ elements in upper subarray

2. Conquer: Recursively sort the two subarrays

3. Combine: Trivial

Key: Linear-time partitioning subroutine

Partitioning Subroutine

```
PARTITION (A,p,q) ▷ A[p... q]
1.    x ← A[p] ▷ pivot = A[p]
2.    i ← p
3.    for j ← p+1 to q
4.        do if A[j] ≤ x
5.           then i ← i+1
6.                exchange A[i] ↔ A[j]
7.    exchange A[p] ↔ A[i]
8.    return i
```

Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
2	5	3	6	8	13	10	11
			↑				
			i				

Week 3 - Lecture Notes

Topics: - Analysis of QuickSort
Randomized QuickSort
Heap
Heap Sort
Decision Tree

Pseudo-code for QuickSort

```
QUICKSORT (A, p, r)
1. if p<r
2.     then q ← PARTITION (A, p, r)
3.         QUICKSORT (A, p, q)
4.         QUICKSORT (A, q+1, r)
```

Initial call: QUICKSORT (A, 1, n)

Analysis of Quick sort

- Assume all input elements are distinct
- In practice, there are better partitioning algorithms for when duplicate input may exist.
- Let T(n) = worst case running time on an array of n elements

Worst case of Quick Sort

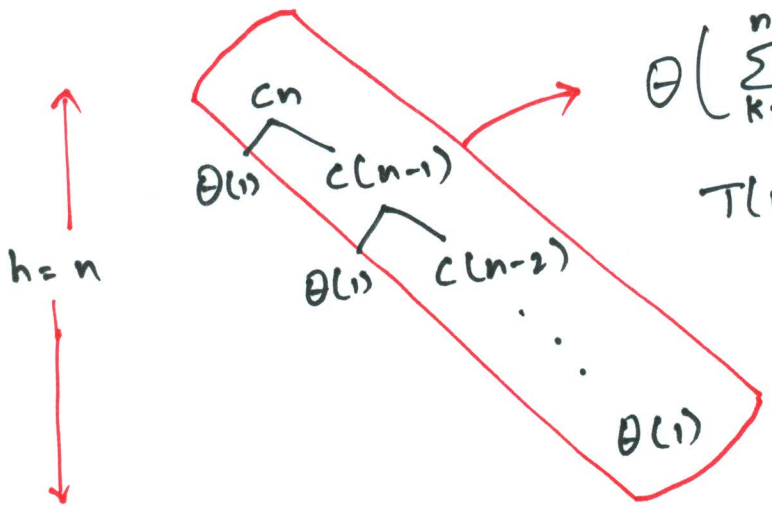
- Input is sorted or reverse sorted
- Partition around minimum or maximum element
- Split : $0 : n-1$,

  one side of the partition always has no element.

$$T(n) = T(0) + T(n-1) + \Theta(n)$$
$$= \Theta(1) + T(n-1) + \Theta(n)$$
$$= \Theta(n) + T(n-1)$$
$$= \Theta(n^2)$$

Worst Case Recursion Tree

$$T(n) = T(0) + T(n-1) + cn$$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

$$T(n) = \Theta(n) + \Theta(n^2)$$
$$= \Theta(n^2)$$

Best-Case Analysis

If we are lucky, PARTITION splits the array evenly $(\frac{1}{2}:\frac{1}{2})$

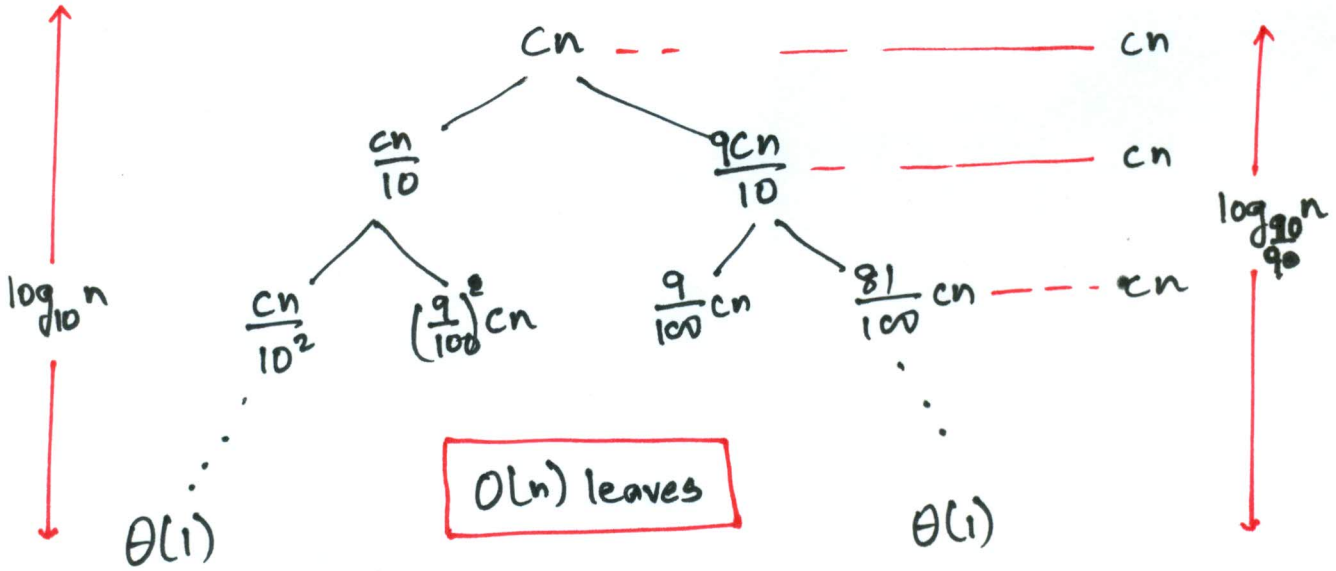
$$T(n) = 2T(n/2) + \theta(n)$$

$$= \theta(n \log n) \quad \text{[same as Merge Sort]}$$

Analysis of "almost-best" case

Consider the split is always $\frac{1}{10}:\frac{9}{10}$.

$$T(n) = T(\tfrac{1}{10}n) + T(\tfrac{9}{10}n) + \theta(n)$$



$$cn\log_{10} n \le T(n) \le cn\log_{\frac{10}{9}} n + O(n)$$

$\theta(n \log n)$ Lucky

More intuition

Let us consider a case in which we are alternate lucky, unlucky, lucky, unlucky, lucky, ...

$$L(n) = 2U(n/2) + \Theta(n) \quad \text{lucky}$$

$$U(n) = L(n-1) + \Theta(n) \quad \text{unlucky}$$

Solving we get.

$$L(n) = 2\left(\left(L(n/2 - 1)\right) + \Theta(n/2)\right) + \Theta(n)$$

$$= 2L(n/2 - 1) + \Theta(n)$$

$$= \Theta(n \log n) \quad \text{Lucky}$$

So, even in this case we are lucky.

How can we make sure we are usually lucky?

Randomized QuickSort

Idea: Partition around a random element

- Running order is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst case behaviour.
- The worst case is determined only by the output of a random-number generator.

Randomized Quick Sort Analysis

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size n, assuming random numbers are independent.

For $K = 0, 1, \ldots, n-1$, define the indicator random variable as:

$$X_K = \begin{cases} 1 & \text{if PARTITION generates } K:n-K-1 \text{ spit} \\ 0 & \text{otherwise} \end{cases}$$

$E[X_K] = Pr\{X_K = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split} \\ \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split.} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n) \right)$$

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n) \right) \right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k \left(T(k) + T(n-k-1) + \Theta(n) \right) \right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \, E\left[T(k) + T(n-k-1) + \Theta(n) \right]$$

$$= \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n)$$

$$= \frac{2}{n} \sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

$$= \frac{2}{n} \sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

The $k=0,1$ terms can be absorbed in the $\Theta(n)$

Prove: $E[T(n)] \le an\lg n$ for constant $a > 0$

Choose 'a' large enough so that $an\lg n$ dominates $E[T(n)]$ for sufficiently small $n \ge 2$.

Use fact: $\sum_{k=2}^{n-1} k\lg k \le \frac{1}{2}n^2\lg n - \frac{1}{8}n^2$

$$E[T(n)] \le \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n)$$

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n)$$

$$= an\lg n - \left(\frac{an}{4} - \Theta(n)\right)$$

$$\le an\lg n$$

if a is chosen large enough so than $\frac{an}{4}$ dominates $\Theta(n)$

Quicksort in Practice

- Quicksort is a great general-purpose sorting algorithm
- Quicksort can benefit substantially from code tuning
- Quicksort is typically over twice as fast as merge sort.

Priority Queue

A data structure implementing a set S of elements, each associated with a key, supporting the following operations.

insert (S, x) : insert element x into set S

max (S) : return element of S with largest key

extract_max (S): return element of S with largest key and remove it from S

increase_key (S, x, K): increase the value of element x's key to new value K.

Heap

- Implementation of a priority queue
- An array, visualized as a nearly complete binary tree
- Max Heap Property: The key ~~node~~ of a node is $\geq$ than the keys of its children

(Min Heap defined analogously)

Heap as a Tree

root of tree : first element in the array, corresponding $i=1$

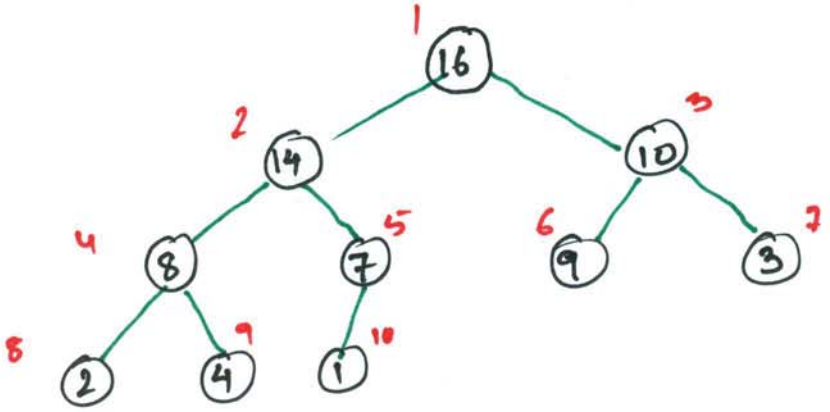
parent $(i) = i/2$: returns index of node's parent

left $(i) = 2i$: returns index of node's left child.

right $(i) = 2i+1$: returns index of node's right child.

Example:

16 14 10 8 7 9 3 2 4 1



No pointers required.

Height of a binary heap is $O(\lg n)$

Heap Operations:

max-heapify : correct a single violation of the heap property in a subtree at its root

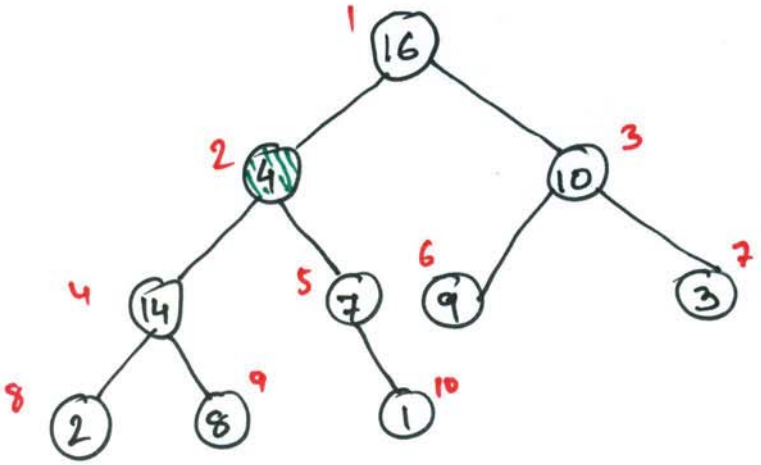
build-max-heap: produce a max-heap from an unordered array.

insert, extract-max, heapsort.

Max-heapify

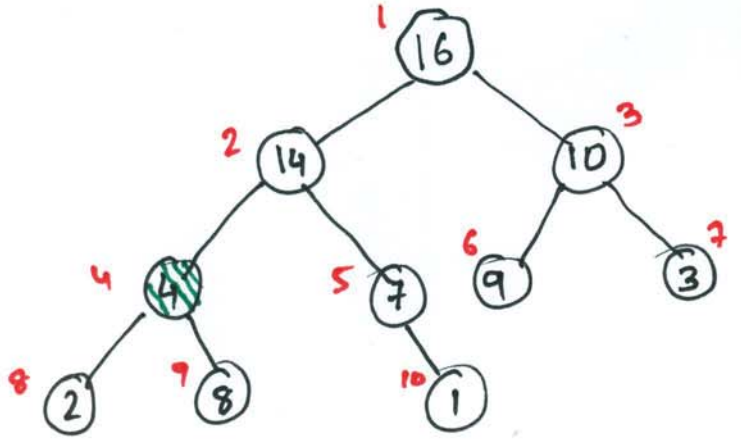
- Assume that the trees rooted at left(i) and right(i) are max-heaps
- If A[i] violates the max-heap property, correct violation by "tricking" element A[i] down the tree, making the subtree rooted at index i a max-heap.

Max_heapify (Example)

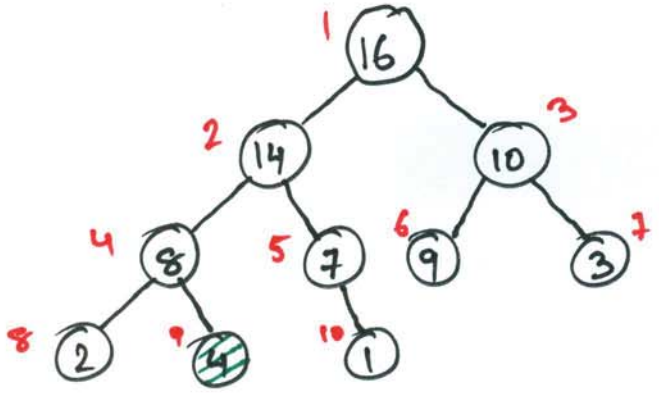


MAX. HEAPIFY (A,2)
heap-size [A] = 10

Node 10 is the left child of node 5 but is drawn to the right for convenience.



Exchange A[2] with A[4]
Call MAX_HEAPIFY (A,4)
because max-heap property is violated



Exchange A[4] with A[9]

No more calls.

Time = $O(\log n)$

Max. Heapify Pseudocode

```
1.  l = left (i)
2.  r = right (i)
3.  if (l ≤ heap-size [A] and A[l] > A[i])
4.      then largest = l
5.      else largest = i
6.  if [r ≤ heap-size (A) and A[r] > A[largest]]
7.      then largest = r
8.  if largest ≠ i
9.      then exchange A[i] and A[largest]
10.     Max-Heapify (A, largest)
```

Build_Max_Heap (A)

Converts $A[1, \ldots, n]$ to a max heap

Build_Max_Heap (A):

for $i = n/2$ down to 1

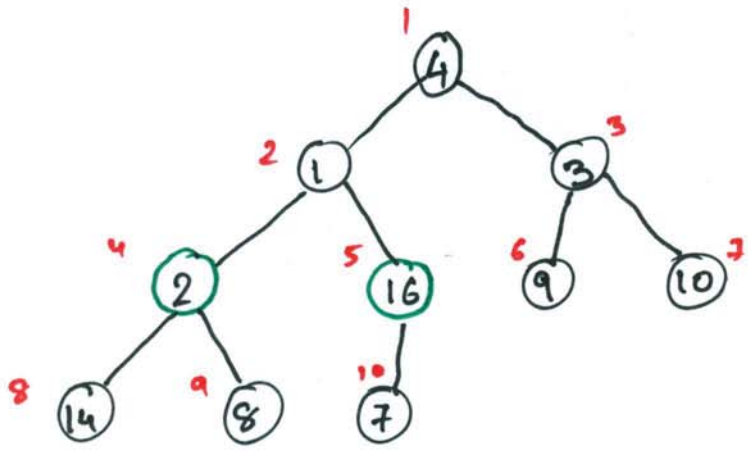
do Max_Heapify (A, i)

- We start at $i = n/2$ because elements $A[n/2+1, \ldots, n]$ are all leaves of the tree

$2i > n$, for $i > n/2 + 1$

Build_Max_Heap Example

4 1 3 2 16 9 10 14 8 7

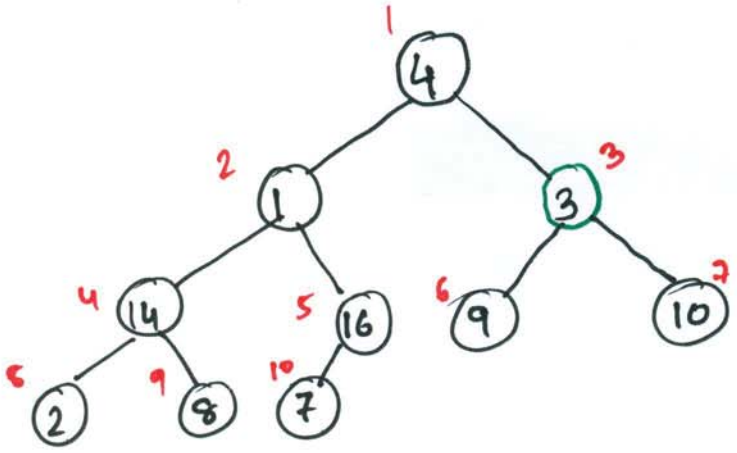


Max_Heapify (A, 5)

no change

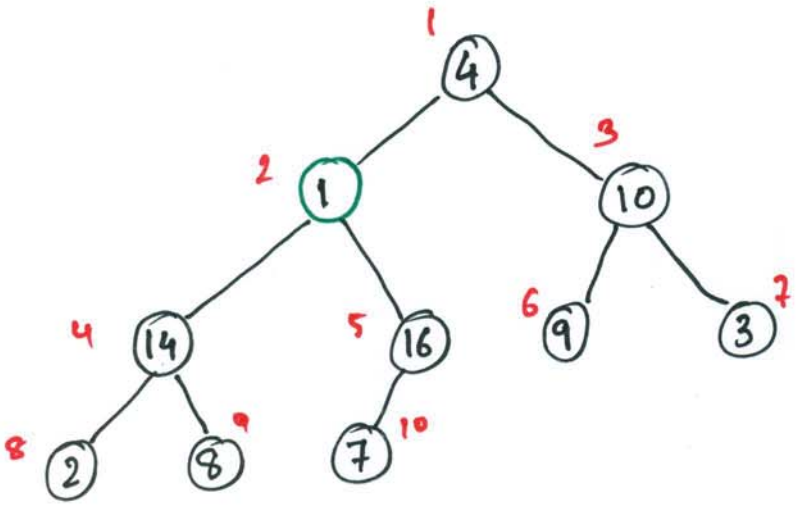
Max_Heapify (A, 4)

Swap A[4] and A[8]



Max-Heapify (A, 3)

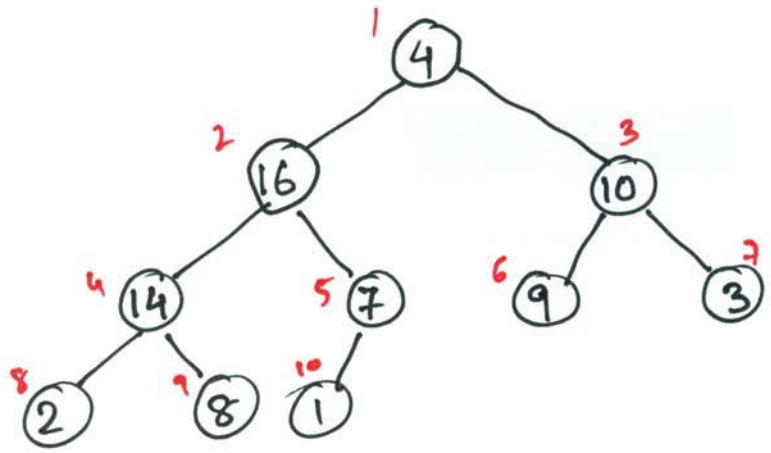
Swap A[3] and A[7]



Max-Heapify (A, 2)

Swap A[2] and A[5]

Swap A[5] and A[10]



Max-Heapify (A,1)

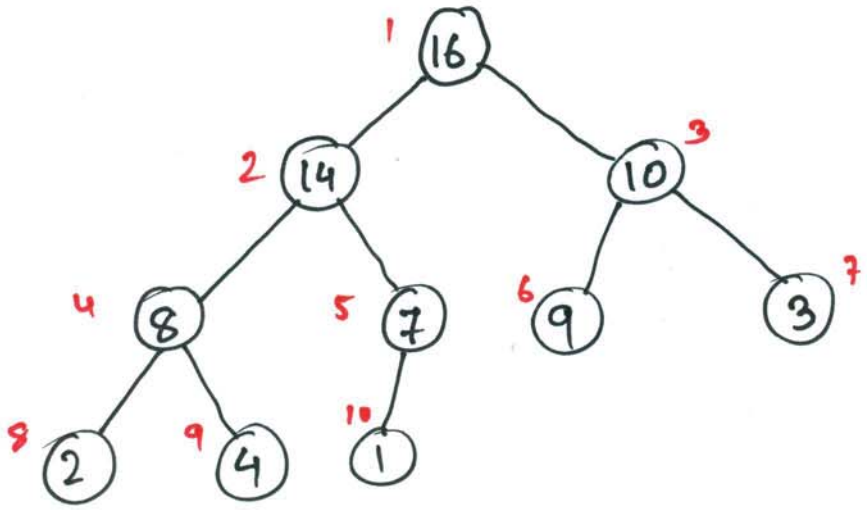
Swap [A,1] with A[2]

Swap [A,2] with A[4]

Swap [A,4] with A[9]

So,

A: 4 1 3 2 16 9 10 14 8 7



Build-Max-Heap (A) Analysis

We can observe that Max-Heapify takes $O(1)$ for nodes that are one level above the leaves, and in general, $O(l)$ for the nodes that are l levels above the leaves.

We have $n/4$ nodes with level 1, $n/8$ with level 2, and so on till we have one root node that is $\lg n$ levels above the leaves.

So, total amount of work in the for loop can be summed as:

$$n/4\,(1c) + n/8\,(2c) + n/16\,(3c) + \cdots + 1(\lg c)$$

setting $n/4 = 2^k$ and simplifying we get

$$c\,2^k\left(1/2^0 + 2/2^1 + 3/2^2 + \cdots + (k+1)/2^k\right)$$

The term in brackets is bounded by a constant.

This means that Build-Max-Heap is $O(n)$

Heap - Sort

Sorting Strategy

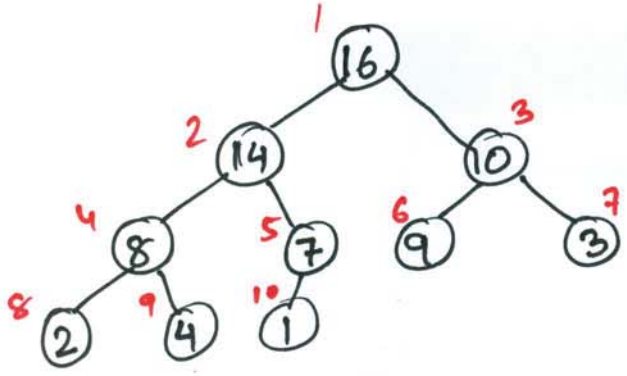
1. Buid Max Heap from unordered array;
2. Find maximum element A[i]
3. Swap elements A[n] and A[i]
   now max element is at the end of array.
4. Discard node n from heap
   (by decrementing heap-size variable)
5. New root may violate max heap property, but it's children are max heaps. Run max-heap to fix this.
6. Go to Step 2 unless heap is empty.

Heap Sort Running Time

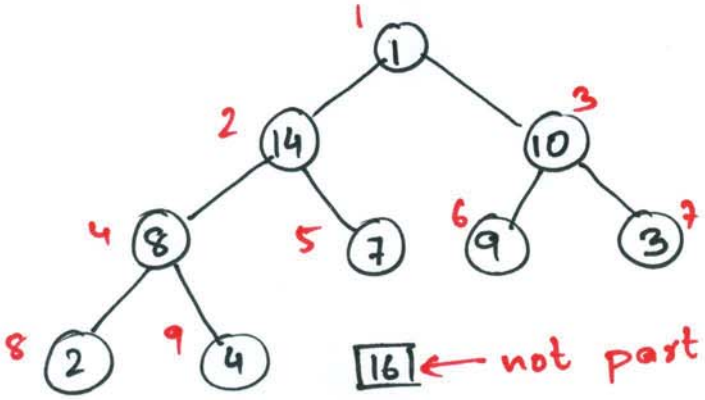
- after n iterations Heap is empty
- every iteration involves a swap and a max. heapify operation; O(logn) time.

Hence, overall : O(nlogn).

Heap-Sort Example



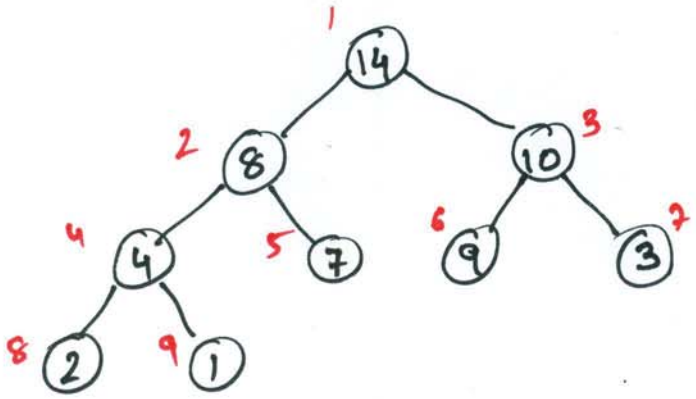
Swap A[10] and A[1]

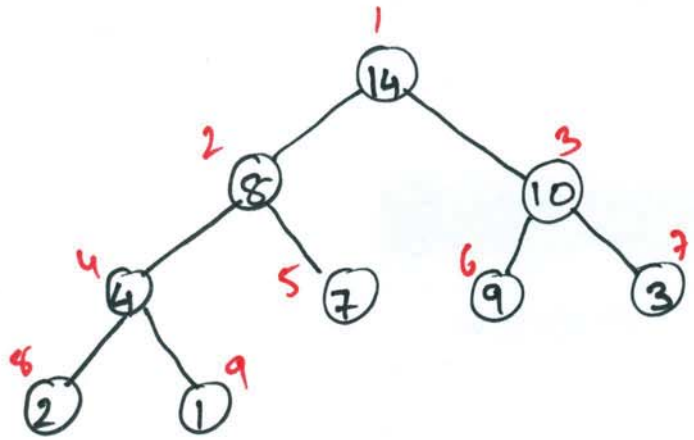


Heap size = 9

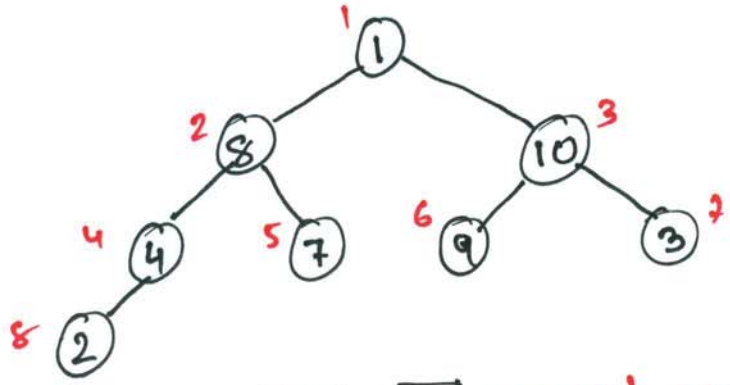
[16] ← not part of heap

Max-heapify (A,1)





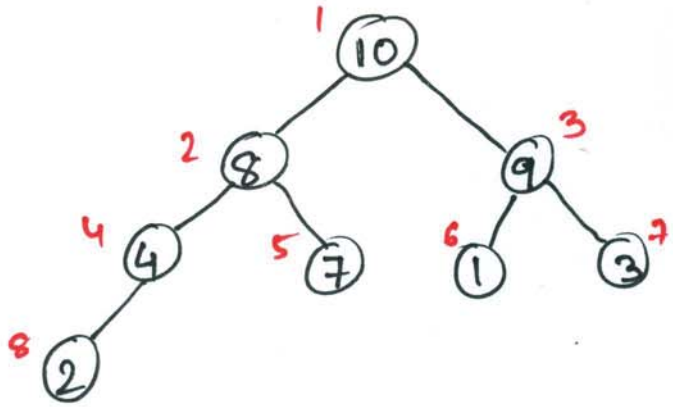
Swap A[9] and A[1]

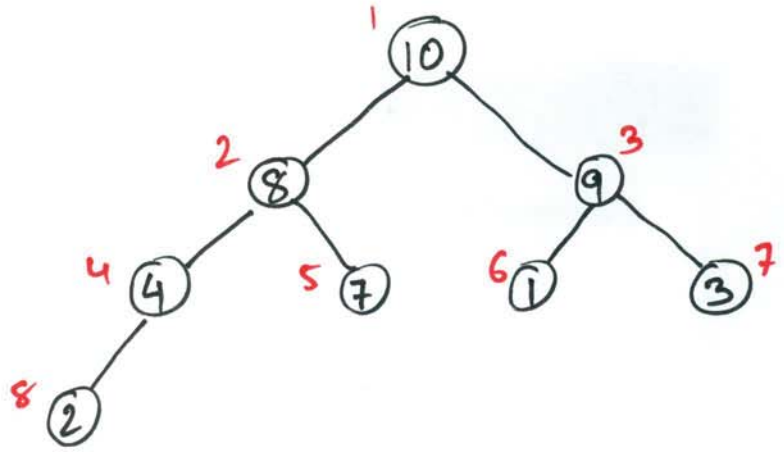


heap size = 8

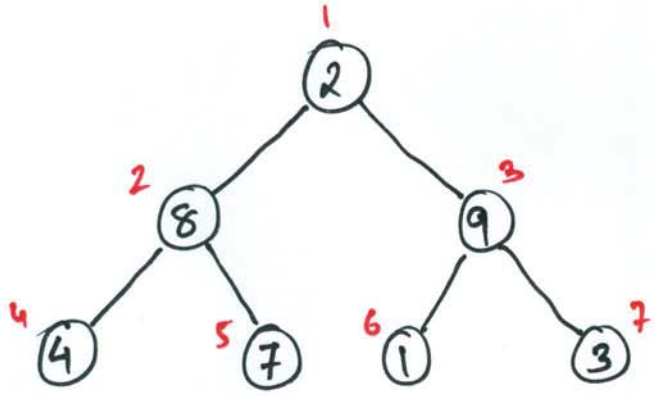
14 16 ← not part of heap

Max_heapify (A,1)





Swap A[8] and A[1]



8 9 10

10 14 16 ← not part of heap.

How fast can we Sort?

All the sorting algorithms we have seen so far are comparison sorts: only use comparisons to determine the relative order of elements.

- E.g.: insertion sort, merge sort, quicksort, heap sort.

The best worst case running time that we have seen for comparison sorting is $O(n \log n)$

Is $O(n \log n)$ the best we can do?

→ Decision Trees can help answer this question

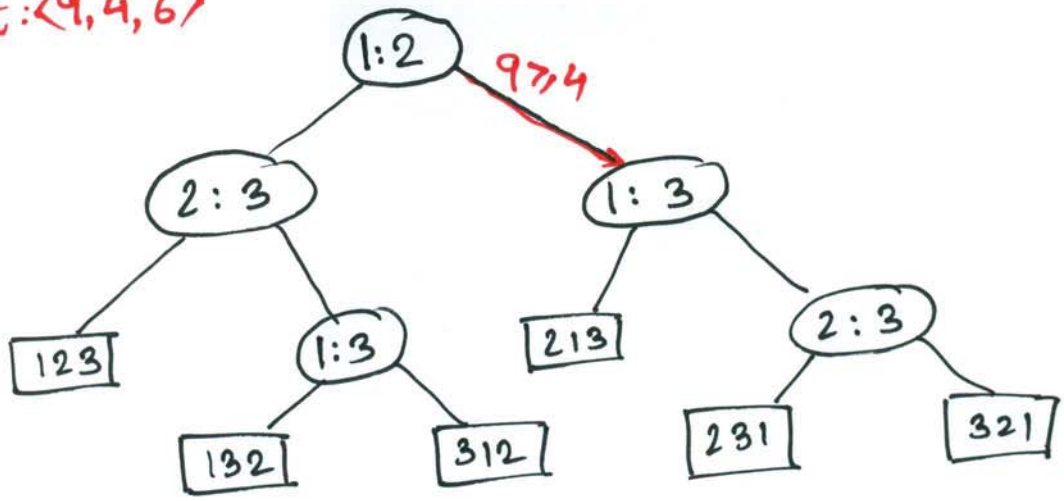
Decision Tree Model

A decision tree can model the execution of any comparison sort:

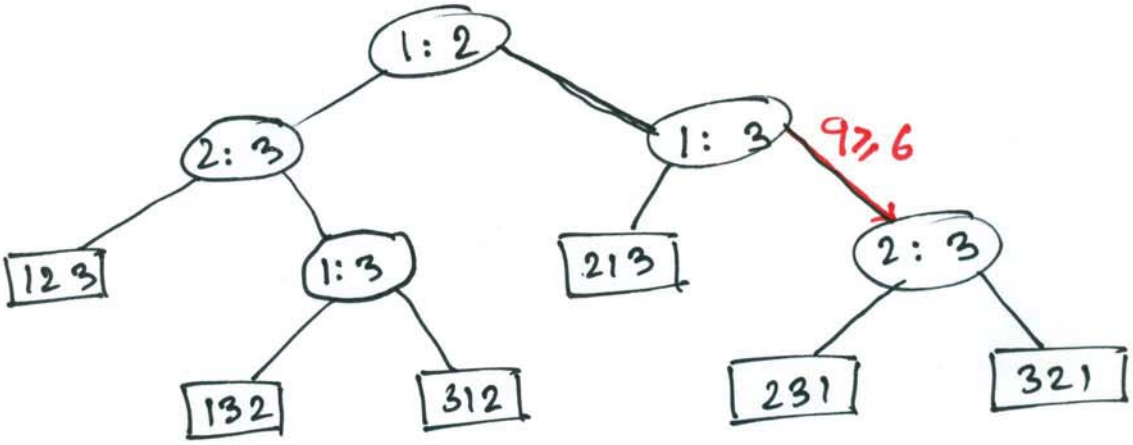
- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of algorithm = the length of the path taken.
- Worst case running time = height of the tree.

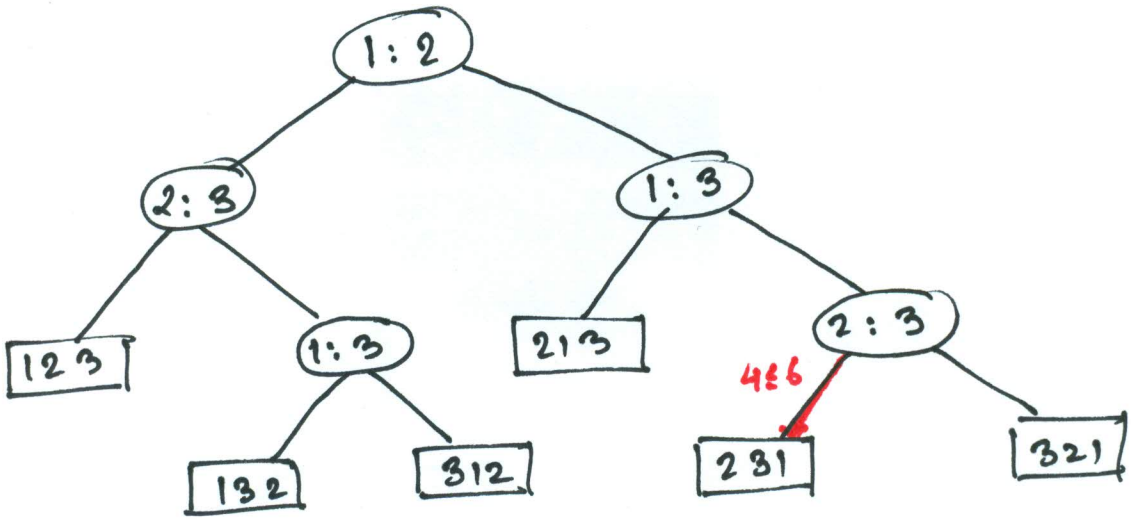
Decision Tree Example

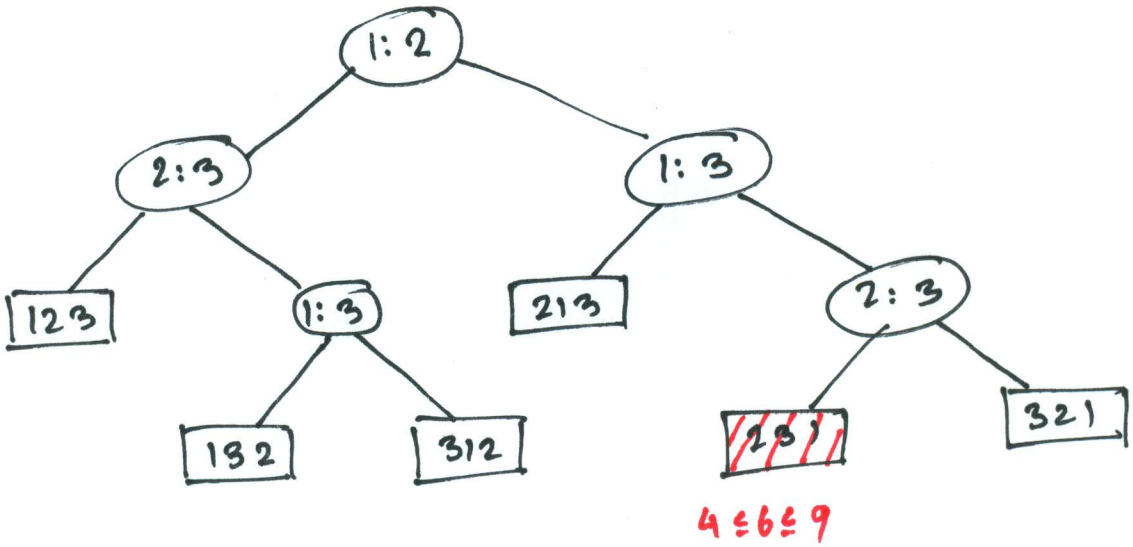
Sort: ⟨9, 4, 6⟩



- Each internal node is labelled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$
- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i \geq a_j$







Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate the ordering $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ has been established.

Lower bound for decision tree Sorting

Theorem:

Any decision tree that can sort n elements must have height $\Omega(n \lg n)$

Proof:

The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations.
A height-h binary tree has $\leq 2^h$ leaves.

Thus $n! \leq 2^h$

$\therefore \quad h \geq \log(n!)$

$\geq \log((n/e)^n)$ [Stirling's formula]

$= n \lg n - n \lg e$

$= \Omega(n \lg n)$

Corollary:

Heapsort and Merge sort are asymptotically optimal comparisons sorting algorithm.

Week 4. Lecture Notes

Topics: Linear Time Sorting
Counting Sort
Radix Sort and Bucket Sort
Order Statistics
Randomized Order Statistics
Worst Case Linear time order Statistics.

Sorting in Linear Time

Counting Sort

No comparison between elements.

Input: $A[1, \ldots, n]$, where $A[j] \in \{1, 2, \ldots, K\}$

Output: $B[1, \ldots, n]$, sorted

Auxiliary Storage

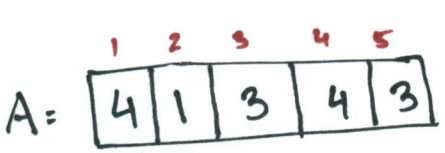
$C[1, \ldots, k]$

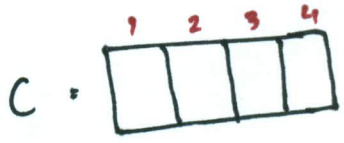
Counting Sort : Pseudocode

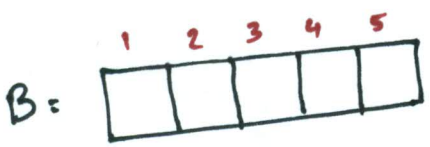
```
1.    for i ← 1 to K
2.        do C[i] ← 0
3.    for j ← 1 to n
4.        do C[A[j]] ← C[A[j]] + 1
5.    for i ← 2 to K
6.        do C[i] ← C[i] + C[i-1]
7.    for j ← n down to 1
8.        do B[C[A[j]]] ← A[j]
9.           C[A[j]] ← C[A[j]] - 1
```

Example:

A = | 4 | 1 | 3 | 4 | 3 |







Loop 1:

```
for i ← 1 to K
    do C[i] ← 0
```

A: | 4 | 1 | 3 | 4 | 3 |

C: | 0 | 0 | 0 | 0 |

B: | | | | | |

Loop 2:

```
for j ← 1 to n
  do C[A[j]] ← C[A[j]] + 1
```

A: | 4 | 1 | 3 | 4 | 3 |

C: | 1 | 0 | 2 | 2 |

B: | | | | | |

Loop 3:

```
for i ← 2 to K
  do C[i] ← C[i] + C[i-1]
```

A: | 4 | 1 | 3 | 4 | 3 |

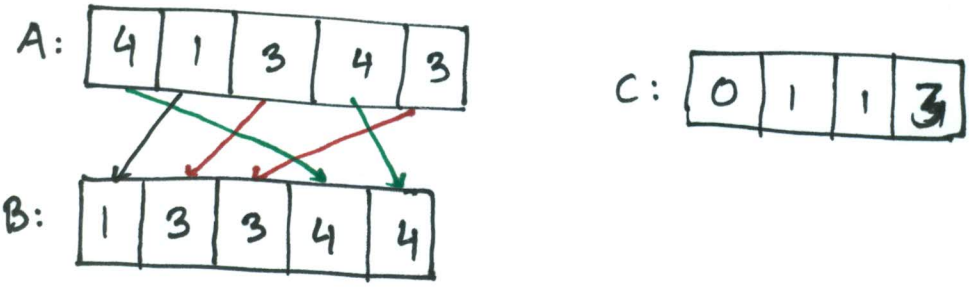
C: | 1 | 0 | 2 | 2 |

B: | | | | | |

C': | 1 | 1 | 3 | 5 |

Loop 4

```
for j ← n down to 1
  do B[C[A[j]]] ← A[j]
     C[A[j]] ← C[A[j]] - 1
```



Analysis of the Pseudocode

$\theta(k)$
```
for i ← 1 to k
  do C[i] ← 0
```

$\theta(n)$
```
for j ← 1 to n
  do C[A[j]] ← C[A[j]] + 1
```

$\theta(k)$
```
for i ← 2 to k
  do C[i] ← C[i] + C[i-1]
```

$\theta(n)$
```
for j ← n down to 1
  do B[C[A[j]]] ← A[j]
     C[A[j]] ← C[A[j]] - 1
```

$\theta(n+k)$

Running Time

If $K = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \log n)$ time.

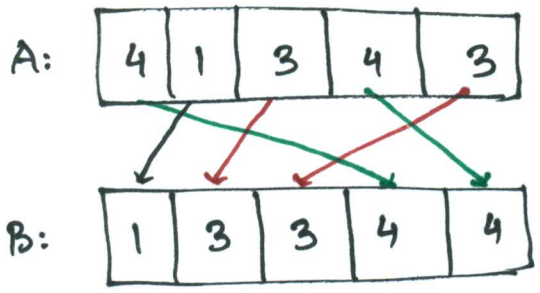
  Where is the fallacy?

Answer:

- Comparison sorting takes $\Omega(n \log n)$ time
- Counting sort is not a comparison sort
- In fact, not a single comparison between element occurs.

Stable Sorting

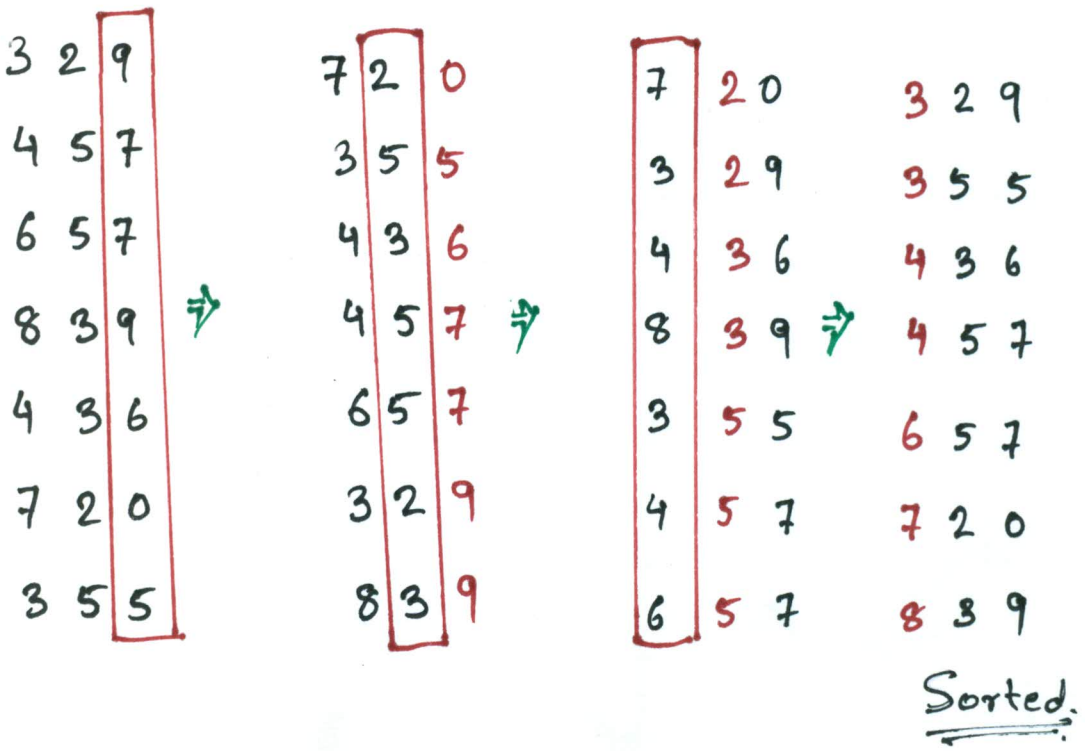
Counting sort is a stable sort, i.e. it preserves the input order among equal elements.



Radix Sort

- Origin: Herman Hollerith's card-sorting machine for the 1890 U.S. Census.
- Digit-by-digit sort
- Hollerith's original (bad) idea: sort on most-significant digit first
- Good idea - Sort on least-significant digit first with auxiliary stable sort.

Operation of Radix Sort



Correctness of Radix Sort

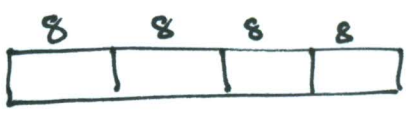
Induction on digit position.

- Assume that the numbers are sorted by their low-order $t-1$ digits
- Sort on digit t.
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input
    $\Rightarrow$ Correct order.

Analysis of Radix Sort

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

Example: 32 bit word



$r = 8 \Rightarrow b/r = 4$ passes of Counting sort on base-2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of Counting sort on base-2^{16} digits.

How many passes should we make?

Recall:

Counting Sort takes $\Theta(n+k)$ time to sort n numbers in the range 0 to $k-1$

If each b-bit word is broken in b/r equal pieces, each pass of counting sort takes $\Theta(n+2^r)$ time. Since there are b/r passes, we have

$$T(n, b) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

Choose r to minimize $T(n,b)$

- Increasing r means fewer passes but as $r \geq \log n$, the time grows exponentially.

Choosing r

$$T(n, b) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

Minimize $T(n,b)$ by differentiating and setting to 0.

Or just observe that we don't want $2^r >> n$, and there's is no harm asymptotically in choosing r as large as possible, subject to this constraint.

Choosing $r = \log n \Rightarrow T(n,b) = \Theta(bn/\log n)$

- For numbers in the range from 0 to $n^d - 1$, we have $b = d\log n \Rightarrow$ radix sort runs in $\Theta(dn)$ time

Conclusions

In practice Radix Sort is fast for large inputs as well as simple to code and maintain.

Example (32-bit numbers)

- At most 3 ~~phases~~ passes when sorting $\geq$ 2000 numbers.
- Merge sort and quicksort do atleast $\lceil \log 2000 \rceil = 11$ passes.

Downside:

Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

Bucket Sort

Idea:

- Divide the interval $[0, n)$ into n equal-sized subinterval, or buckets
- Distribute the n input numbers into the buckets.

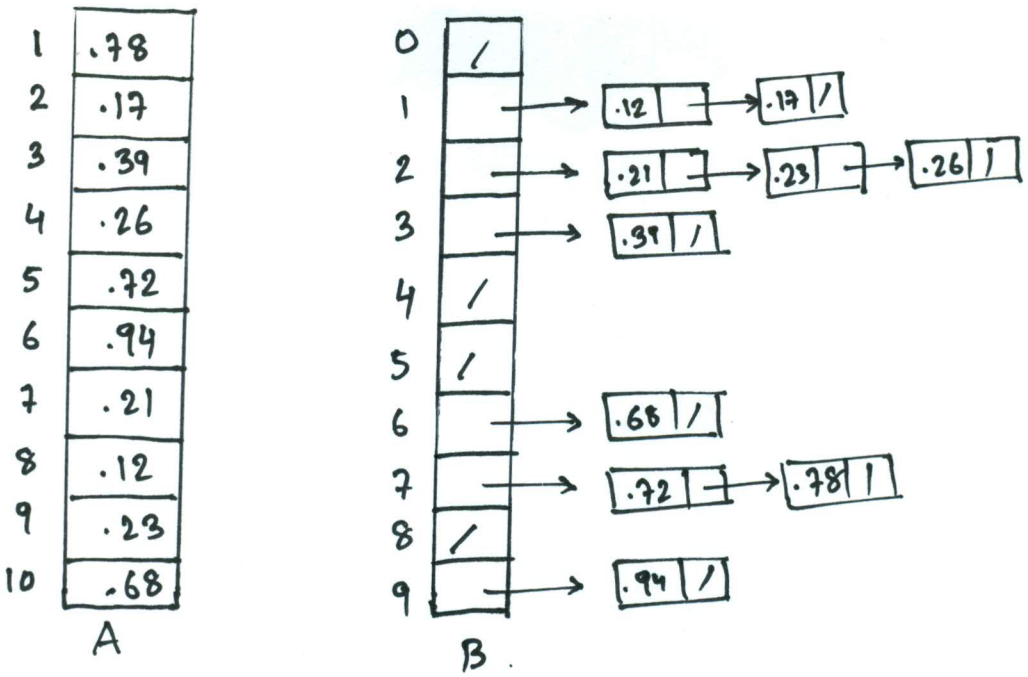
Since the inputs are assumed to be uniformly distributed over $[0,1)$, many numbers do not fall into each bucket.

To produce the output, simply sort the numbers in each bucket and then go through the buckets, in order, listing the elements in each.

Pseudocode of Bucket sort

```
BUCKET-SORT(A)
1.  n ← length[A]
2.  for i ← 1 to n
3.      do insert A[i] into list B[⌊n A[i]⌋]
4.  for i ← 0 to n-1
5.      do sort list B[i] with insertion sort
6.  Concatenate the lists B[0], B[1] ...
        B[n-1] together in order.
```

Example



A: input array

B: array of sorted lists (buckets) after line 5 of the above algorithm.

Correctness of Pseudocode

Consider two elements $A[i] \leq A[j]$

Since $\lfloor n A[i] \rfloor \leq \lfloor n A[j] \rfloor$, element $A[i]$ is placed either into the same bucket as $A[j]$ or a bucket with lower index.

If they are placed in same bucket then the for loop of lines 4-5 puts them in proper order.

If they are placed in different buckets line 6 puts them in proper order.

Therefore, bucket sort works correctly.

Analysis of Running Time

- Observe that all lines except line 5 takes $O(n)$ time in worst case.

- We need to balance the total time taken by the n calls to insertion sort in line 5.

Let n_i be the random variable denoting the number of elements placed in bucket $B[i]$.

So, running of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \quad \left[\because \text{insertion sort runs in } O(n^2)\right]$$

$$\Rightarrow E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \quad \text{——— (1)}$$

We claim that

$$E[n_i^2] = 2 - \frac{1}{n} \quad \text{for } i = 0, 1, \ldots, n-1. \quad \text{——— (2)}$$

Define

$$X_{ij} = I\{A[j] \text{ falls in bucket } i\}$$

for $i = 0, 1, \ldots n-1$, $j = 1, 2, \ldots, n$.

$$\Rightarrow n_i = \sum_{j=1}^{n} X_{ij}$$

$$\Rightarrow \quad E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij} X_{ik}\right]$$

$$= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\neq j}} X_{ij} X_{ik}\right]$$

$$= \sum_{j=1}^{n} E\left[X_{ij}^2\right] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\neq j}} E\left[X_{ij} X_{ik}\right]$$

As, Indicator variable X_{ij} is 1 with probability $1/n$ and 0 other wise, so

$$E\left[X_{ij}^2\right] = 1\cdot 1/n + 0\,(1-1/n) = \frac{1}{n}$$

When $k \neq j$, X_{ij} and X_{ik} are independent, so

$$E[X_{ij} X_{ik}] = E[X_{ij}]\, E[X_{ik}]$$

$$= \frac{1}{n}\cdot\frac{1}{n} = \frac{1}{n^2}$$

So,

$$E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\neq j}} \frac{1}{n^2}$$

$$= n\cdot\frac{1}{n} + n(n-1)\,1/n$$

$$= 1 + \frac{n-1}{n}$$

$$= 2 - \frac{1}{n}$$

which proves (2)

Using this expected value in (1),
We can say that the running time of bucket sort is expected to be

$$T(n) = \theta(n) + n \cdot O(2 - 1/n)$$

$$= \theta(n).$$

- Thus, the entire bucket sort algorithm runs in linear expected time.

Conclusion

- Bucket Sort runs in linear time when the input is drawn from a uniform distribution.
- Like Counting sort, it is fast.
- Even if the input is not drawn from a uniform distribution, bucket sort may run in linear time, as long as the input has the property that the sum of the squares of the bucket ~~sor~~ sizes is linear in the total number of elements.

Order Statistics

Select the i^{th} smallest of n elements (the element with rank i).

- $i=1$, minimum
- $i=n$, maximum
- $i = \lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$, median.

Naive Algorithm:

Sort and index i^{th} element.

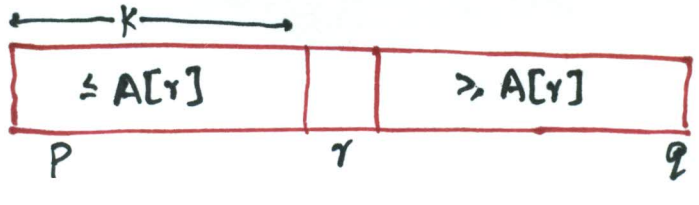
Worst case running time = $\theta(n \log n) + \theta(1)$

$= \theta(n \log n)$

using Merge sort or Heapsort (not Quicksort)

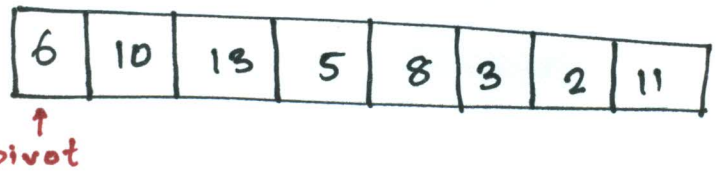
Randomized divide-and-conquer algorithm.

```
RAND-SELECT (A, p, q, i)
if p = q return A[p]
r ← RAND-PARTITION (A, p, q)
k ← r - p + 1
if i = k return A[r]
if i < k
    then return RAND-SELECT (A, p, r-1, i)
else return RAND-SELECT (A, r+1, q, i-k)
```

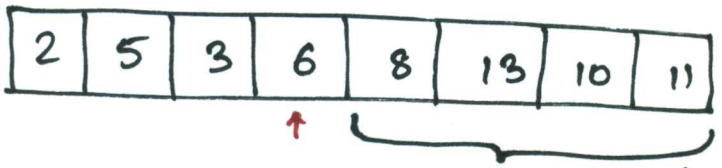


Example

Select the $i = 7^{th}$ smallest



Partition:



Select the $i = 7-4 = 3^{rd}$ element recursively

Intuition for Analysis

All our analysis here assume that all the elements are distinct.

Lucky:

$$T(n) = T(9n/10) + \Theta(n) \quad \left[n^{\log_{10/9} 1} = n^0 = 1, \text{ case 3} \right]$$

$$= \Theta(n)$$

Unlucky:

$$T(n) = T(n-1) + \Theta(n) \quad [\text{arithmetic series}]$$

$$= \Theta(n^2)$$

Worse than sorting

Analysis of Expected Time

The analysis follows that of randomized quicksort, but it's a little different.

Let $T(n)$ = the random variable for the running time of RAND-SELECT on an input of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the indicator random variable X_k as

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split} \\ 0 & \text{otherwise} \end{cases}$$

To obtain an upper bound, assume that the i^{th} element always fall in the larger side of the partition.

$$T(n) = \begin{cases} T(\max\{0, n-1\}) + \Theta(n) & \text{if } 0 : n-1 \text{ split} \\ T(\max\{1, n-2\}) + \Theta(n) & \text{if } 1 : n-2 \text{ split} \\ \vdots \\ T(\max\{n-1, 0\}) + \Theta(n) & \text{if } n-1 : 0 \text{ split} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n) \right).$$

Calculating Expectation

$$E(T(n)) = E\left[\sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k]\, E\left[T(\max\{k, n-k-1\}) + \Theta(n)\right]$$

$$= \frac{1}{n} \sum_{k=0}^{n-1} E\left[T(\max\{k, n-k-1\})\right] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n)$$

$$\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \Theta(n)$$

Prove: $E[T(n)] \leq cn$ for constant $c > 0$

- The constant c can be chosen large enough so that $E[T(n)] \leq cn$ for the base cases.
- Use fact: $\sum_{k=\lfloor n/2 \rfloor}^{n-1} k \leq \frac{3}{8} n^2$

We have,

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n) \quad \left[\begin{array}{l}\text{Substituting} \\ E(T(n)) \leq cn\end{array}\right.$$

So, using the fact stated above

$$E[T(n)] \leq \frac{2c}{n}\left(\frac{3}{8}n^2\right) + \Theta(n)$$

$$= cn - \left(\frac{cn}{4} - \Theta(n)\right)$$

[Expressed as desired - residual]

$$\leq cn$$

if c is chosen large enough so that $cn/4$ dominates the $\Theta(n)$.

Summary of randomized order-statistic selection

- Works fast: linear expected time
- Excellent algorithm in practice
- But the worst case is very bad: $\Theta(n^2)$

Q. Is there an algorithm that runs in linear time in the worst case?

A. Yes, due to Blum, Floyd, Pratt, Rivest and Tarjan [1973]

Idea: Generate a good point recursively.

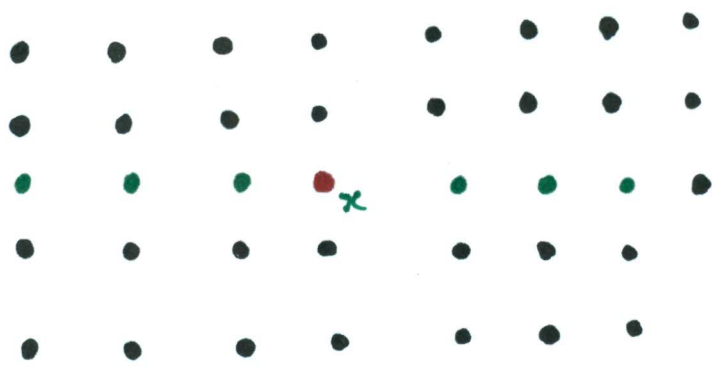
Worst Case linear Time order statistics

SELECT (i, n)

1. Divide the n elements into groups of 5. Find the median of each 5-element group
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
3. Partition around the pivot x. Let $k = \text{rank}(x)$.
4. if $i = k$ then return x
   else if $i < k$
   then recursively SELECT the i^{th} smallest element in lower part.
   else recursively SELECT the $(i-k)^{th}$ smallest element in upper part.

} Same as RAND-SELECT

Choosing the Pivot



- Divide n into groups of 5
- Recursively SELECT the median 'x' of $\lfloor n/5 \rfloor$ group medians as pivot.

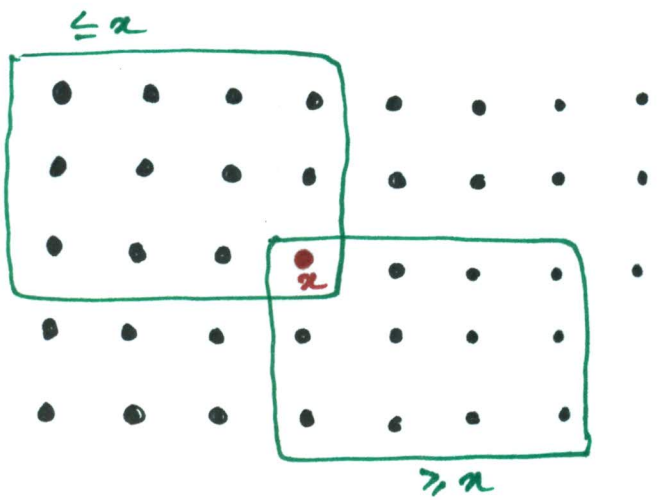
Analysis

Assume all elements are distinct

Atleast half the group medians $\leq x$, which is atleast $\lfloor \lceil n/5 \rceil / 2 \rfloor = \lfloor n/10 \rfloor$ group medians.

Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\leq x$

Similarly, at least $3\lfloor n/10 \rfloor$ elements are $\geq x$



Minor Simplification

- For $n \geq 50$, we have $3\lfloor n/10 \rfloor \geq n/4$
- Therefore, for $n \geq 50$, the recursive call to SELECT in step 4 is executed on $\leq 3n/4$ elements.
- Thus, the recurrence for running time can assume that step 4 takes time $T(3n/4)$ in the worst case.
- for $n < 50$, we know that the wort case time is $T(n) = \theta(1)$

Developing the recurrence

SELECT (i, n)

$\theta(n)$ 1. Divide the n elements into groups of 5. Find the median of each 5-element group.

$T(n/5)$ 2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

$\theta(n)$ 3. ~~Pivot~~ Partition around the pivot x.
Let $k = \text{rank}(x)$.

$T(3n/4)$ 4. if $i = k$ then return x
else if $i < k$
then recursively SELECT the i^{th} smallest element in the lower part.
else recursively SELECT the $(i-k)^{th}$ smallest element in the upper part.

Thus the recurrence is,

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) + \theta(n).$$

Solving the recurrence.

We have

$$T(n) = T(n/5) + T(3n/4) + \theta(n)$$

Substituting $T(n) \leq cn$

$$T(n) \leq \frac{1}{5}cn + \frac{3}{4}cn + \theta(n)$$

$$= \frac{19}{20}cn + \theta(n)$$

$$= cn - \left(\frac{1}{20}cn - \theta(n)\right)$$

$$\leq cn$$

if c is chosen large enough to handle both the $\theta(n)$ and the initial conditions.

Conclusion

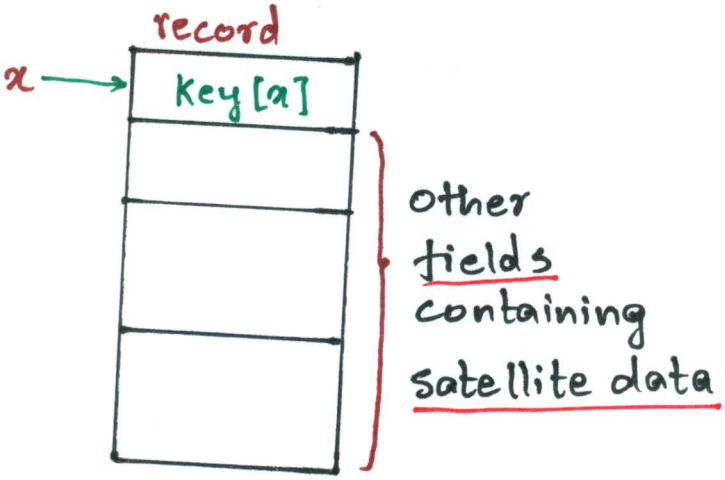
- Since work at each level of recursion is a constant fraction $(19/20)$ smaller, the work per level is a geometric series dominated by the linear work at the root.
- In practice, this algorithm runs slowly, because the constant in front of n is large.
- The randomized algorithm is far more practical.

Week 5. Lecture Notes

Topics: Hash Function
Open Addressing
Universal Hashing
Perfect Hashing
Binary Search Tree (BST) Sort

Symbol Table Problem

Symbol table T holding n records:



How should the data structure T be organized?

Operations on T:

- INSERT (T, x)
- DELETE (T, x)
- SEARCH (T, K)

Direct-access table

IDEA: Suppose that the set of keys is $K \subseteq \{0, 1, \ldots, m-1\}$, and the keys are distinct.

Setup an array $T[0, \ldots, m-1]$:

$$T[k] = \begin{cases} x, \text{ if } x \in K \text{ and } key[x] = k; \\ NIL, \text{ otherwise.} \end{cases}$$

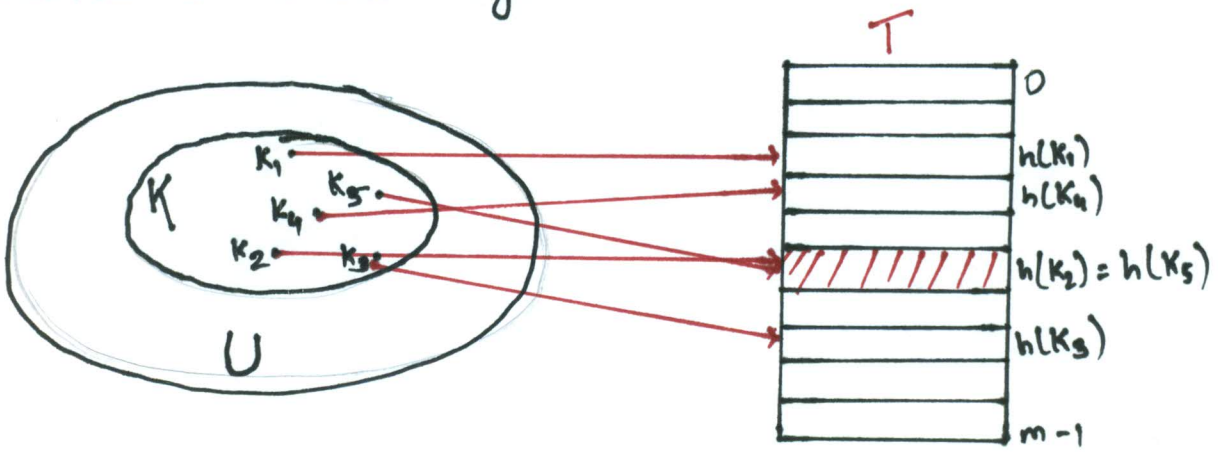
Then the operations take $\Theta(1)$ time.

PROBLEM: The range of keys can be large:

- 64 bit numbers (which represent 64,446,744,073,709,616 different keys),
- character strings (even larger)

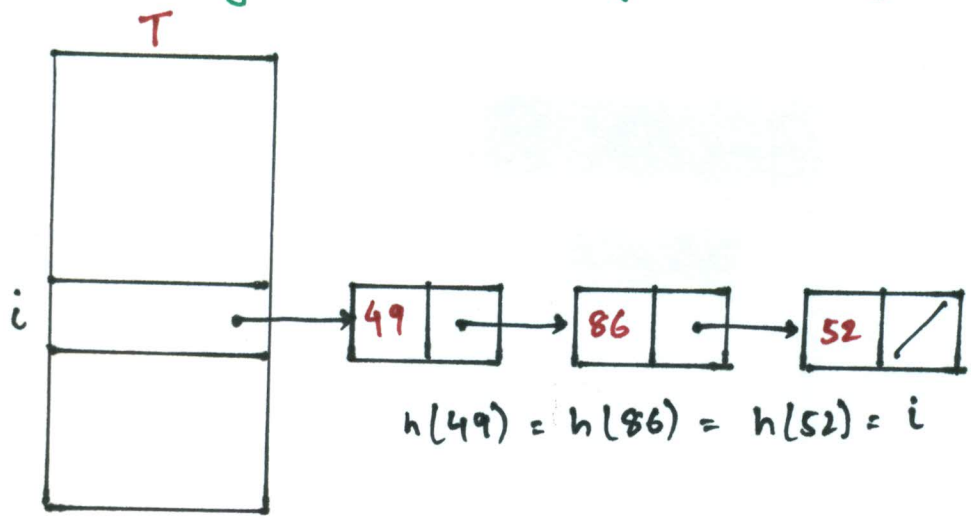
Hash Functions

Solution: Use a hash function h to map the universe U of all keys into $\{0, 1, \ldots, m-1\}$:



When a record to be inserted maps to an already occupied slot in T, a **collision** occurs.

Resolving Collisions by Chaining



Analysis of Chaining

We make the assumption of simple uniform hashing:

- Each key $k \in K$ of keys is equally likely to be hashed to any slot of table T, independent of where other keys are hashed.

Let n be the number of keys in the table, and let m be the number of slots.

Define the load factor of T to be

$$\alpha = n/m$$

= average number of keys per slot.

Search Cost

Expected time to search for a record with a given key = $\Theta(1 + \alpha)$

- 1: apply hash function and access slot
- α: search the list.

- Expected search time = $\Theta(1)$ if $\alpha = O(1)$, or equivalently, if $n = O(m)$

Choosing a hash function

The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.

Desirata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.

Division Method

Assume all keys are integers, and define

$$h(k) = k \bmod m$$

Deficiency: Don't pick an m that has a small divisor d. A preponderance of keys that are congruent modulo d can adversely affect uniformity.

Extreme deficiency: If $m = 2^r$, then the hash doesnot even depend on all bits of k:

- If

$$k = 10111000111\underbrace{011010}_{h(k)}{}_2 \text{ and } r = 6$$

then

$$h(k) = 011010_2$$

Division Method (Continued)

$$h(k) = k \bmod m$$

Pick m to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

Annoyance:

Sometimes, making the table size a prime is inconvenient

But, this method is popular, although the next method we'll see is usually superior

Multiplication Method

Assume that all keys are integers, $m = 2^r$ and our computer has w-bit words. Define

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w-r)$$

where rsh is the "bit-wise right-shift" operator and A is an odd integer in the range $2^{w-1} < A < 2^w$.

- Don't pick A too close to 2^w
- Multiplication modulo 2^w is fast
- The rsh operator is fast.

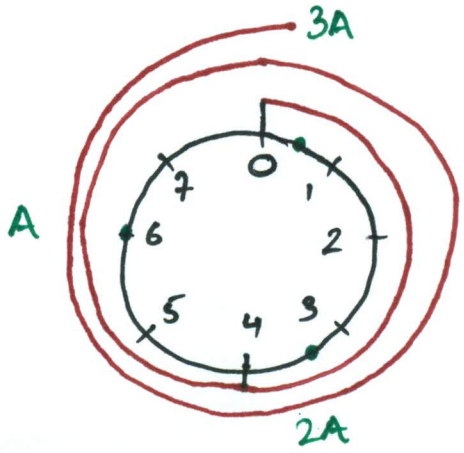
Multiplication method example

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w-r)$$

Suppose that $m = 8 = 2^3$ and that our computer has $w = 7$-bit words.

$$\begin{array}{rl} 1011001 & = A \\ 1101011 & = k \\ \hline 1001010\underbrace{011}_{h(k)}0011 & \end{array}$$

Modular wheel:



Dot product method

Randomized Strategy:

Let m be prime. Decompose key K into $r+1$ digits, each with value in the set $\{0,1,\ldots, m-1\}$.

That is, let $K = \langle k_0, k_1, \ldots, k_{m-1}\rangle$, where $0 \le k_i < m$.

Pick $a = \langle a_0, a_1, \ldots, a_{m-1}\rangle$ where each a_i is chosen randomly from $\{0,1,\ldots, m-1\}$.

Define: $h_a(K) = \sum_{i=0}^{r} a_i k_i \bmod m$

Resolving collisions by open addressing

No storage is used outside the hash table itself.

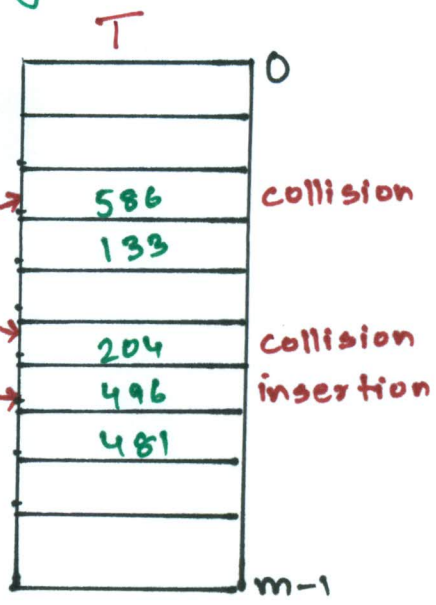
- Insertion systematically probes the table until an empty slot is found
- The hash function depends on both the key and probe number

  $h: U \times \{0,1,\ldots, m-1\} \to \{0,1,\ldots, m-\}$.
- The probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ should be a permutation of $\{0,1,\ldots, m-1\}$.
- The table may fill up, and deletion is difficult (but not impossible)

Example of open addressing

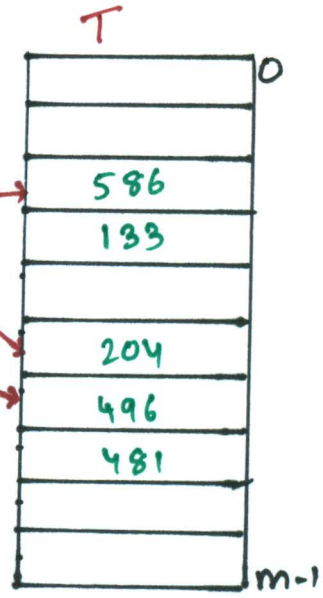
- Insert key K = 496

0. Probe h(496, 0)
1. Probe h(496, 1)
2. Probe h(496, 2)



- Search for key K = 496

0. Probe h(496, 0)
1. Probe h(496, 1)
2. Probe h(496, 2)



Search uses the same probe sequence, terminating successfully if it finds the key an unsuccessfully if it encounters an empty slot.

Probing Strategies

Linear probing:

Given an ordinary hash function $h'(k)$, linear probing uses the hash function

$$h(k,i) = (h'(k) + i) \bmod m$$

This method, though simple, suffers from primary clustering, where long runs of occupied slots build up, clustering the average search time. Moreover, the long runs of occupied slots tends to get longer.

Double hashing

Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m$$

This method generally produces excellent results, but $h_2(k)$ must be relatively prime to m. One way is to make m a power of 2 and design $h_2(k)$ to produce only odd numbers.

Analysis of open addressing

We make the assumption of uniform hashing:

Each key is equally likely to have any one of the $m!$ permutations as its probe-sequence.

Theorem:

Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$

Proof:

- At least one probe is necessary, always.
- With probability n/m, the first probe hits an occupied slot and a second probe is necessary
- With probability $(n-1)/(m-1)$, the second probe hits an occupied slot and a third probe is necessary.
- With probability $(n-2)/(m-2)$, the third probe hits an occupied slot, etc..

Observe that $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$ for $i = 1, 2, \ldots, n$

Therefore, expected number of probes is:

$$1 + \frac{n}{m}\left(1 + \frac{n-1}{m-1}\left(1 + \frac{n-2}{m-2}\left(\cdots\left(1 + \frac{1}{m-n+1}\right)\cdots\right)\right)\right)$$

$$\leq 1 + \alpha\left(1 + \alpha\left(1 + \alpha\left(\cdots\left(1+\alpha\right)\cdots\right)\right)\right)$$

$$\leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$$

$$= \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1-\alpha}$$

$\leq$.

Implications of the theorem

- If α is constant, then accessing an open-addressed hash table takes constant time.

- If the table is half-full, then the expected number of probes is

$$1/(1-0.5) = 2$$

- If the table is 90% full, then the expected number of probes is

$$1/(1-0.9) = 10$$

A weakness of hashing

Problem: For any hash function h, a set of keys exists that can cause the average access time of a hash table to skyrocket.

An adversary can pick all keys from $\{k \in U : h(k) = i\}$ for some slot i.

IDEA: Choose the hash function at random, independently of the keys.

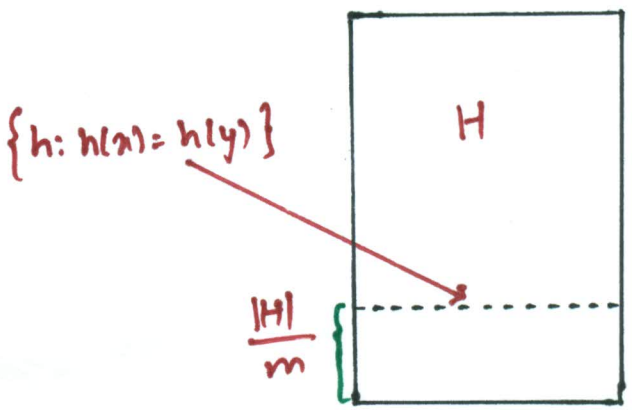
- Even if an adversary can see your code, he or she cannot find a bad set of keys since he or she doesnot know exactly which hash function will be chosen.

Universal Hashing

Definition: Let U be a universe of keys, and let H be a finite collection of hash functions, each mapping U to $\{0,1,\ldots,m-1\}$. We say H is universal if for all $x, y \in U$ where $x \neq y$, we have

$$|\{h \in H : h(x) = h(y)\}| = |H|/m$$

That is, the chance of a collision between x and y is $1/m$ if we choose h randomly from H



Universality is good

Theorem:
Let h be a hash function chosen (uniformly) at random from a universal set H of hash functions. Suppose h is used to hash n arbitrary keys into the m slots of a table T. Then, for a given key x, we have

$$E[\#\text{collisions with } x] < n/m$$

Proof:
Let C_x be the random variable denoting the total number of collisions of keys in T with x and let

$$C_{xy} = \begin{cases} 1, & \text{if } h(x) = h(y); \\ 0, & \text{otherwise.} \end{cases}$$

Note: $E[C_{xy}] = 1/m$ and $C_x = \sum_{y \in T - \{x\}} C_{xy}$

So,

$$E[C_x] = E\left[\sum_{y \in T-\{x\}} C_{xy}\right] \quad \text{. Take expectations of both sides}$$

$$= \sum_{y \in T-\{x\}} E[C_{xy}] \quad \text{. Linearity of expectation}$$

$$= \sum_{y \in T-\{x\}} 1/m \quad \text{. } E[C_{xy}] = 1/m$$

$$= \frac{n-1}{m} \quad \text{. Algebra}$$

Constructing a set of universal hash functions

Let m be prime. Decompose key K into $r+1$ digits, each with value in the set $\{0,1,\ldots,m-1\}$. That is, let $K = \langle k_0, k_1, \ldots, k_r \rangle$, where $0 \le k_i < m$.

Randomized Strategy:

Pick $a = \langle a_0, a_1, \ldots, a_r \rangle$ where each a_i is chosen randomly from $\{0,1,\ldots,m-1\}$

Define $h_a(K) = \sum_{i=0}^{r} a_i k_i \bmod m$ (Dot product modulo m)

How big is $H = \{h_a\}$? $|H| = m^{r+1}$ ← REMEMBER THIS

Universality of dot-product hash functions

Theorem: The set $|H| = \{h_a\}$ is universal.

Proof: Suppose that $x = \langle x_0, x_1, \ldots, x_r \rangle$ and $y = \langle y_0, y_1, \ldots, y_r \rangle$ be distinct keys. Thus, they differ in at least one digit position, say position 0.

For how many $h_a \in H$ do x and y collide?

We must have $h_a(x) = h_a(y)$, which implies that

$$\sum_{i=0}^{r} a_i x_i \equiv \sum_{i=0}^{r} a_i y_i \pmod m$$

Equivalently, we have

$$\sum_{i=0}^{r} a_i(x_i - y_i) \equiv 0 \pmod{m}$$

$$\Rightarrow a_0(x_0 - y_0) + \sum_{i=1}^{r} a_i(x_i - y_i) \equiv 0 \pmod{m}$$

$$\Rightarrow a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod{m}$$

A fact from Number Theory

Theorem: Let m be prime. For any $z \in Z_m$ such that $z \neq 0$, there exists a unique $z' \in Z_m$ such that

$$z \cdot z^{-1} \equiv 1 \pmod{m}$$

Example: $m = 7$

z	1	2	3	4	5	6
z^{-1}	1	4	5	2	3	6

Proof continued:

We have $a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod{m}$

Since $x_0 \neq y_0$, an inverse $(x_0 - y_0)^{-1}$ must exist, so

$$a_0 \equiv \left(-\sum_{i=1}^{r} a_i(x_i - y_i)\right) \cdot (x_0 - y_0)^{-1} \pmod{m}$$

Thus for any choices of $a_1, a_2, \ldots, a_r$, exactly one choice of a_0 causes x and y to collide.

Proof completed:

Q: How many h_a's cause x and y to collide?

A: There are m choices for each $a_1, a_2, \ldots, a_r$, but once these are chosen, exactly one choice for a_0 causes x and y to collide, namely

$$a_0 \equiv \left(\left(-\sum_{i=1}^{r} a_i (x_i - y_i) \right) \cdot (x_0 - y_0)^{-1} \right) \bmod m$$

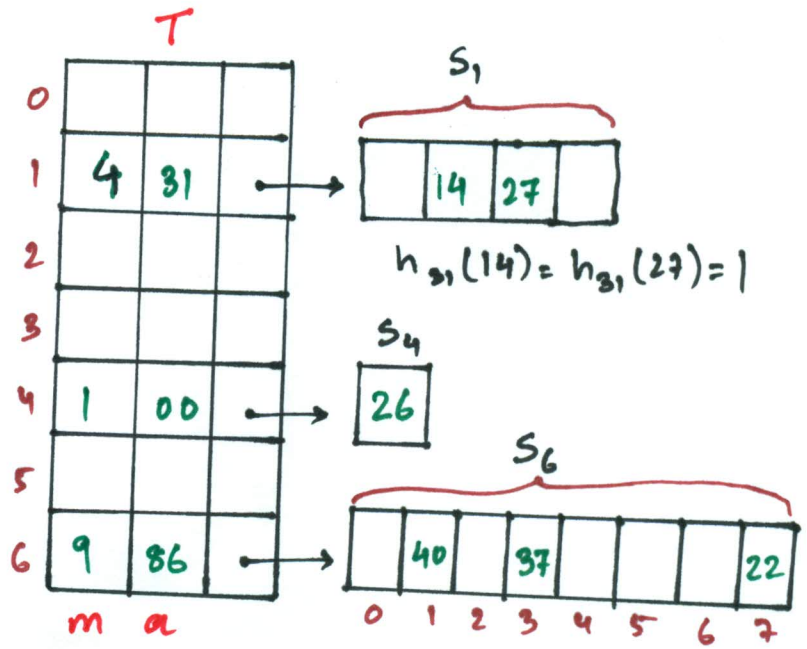
Thus the number of h_a's that cause x and y to collide is $\underline{m^r \cdot 1 = m^r = |H|/m}$.

Perfect Hashing

Given a set of n keys, construct a static hash table of size $m = O(n)$ such that SEARCH takes $\Theta(1)$ time in the worst case.

IDEA:
Two-level scheme with universal hashing at both levels.

No collisions at level 2!



Collisions at level 2

Theorem:

Let H be a class of universal hash functions for a table of size $m = n^2$. Then if we use a random $h \in H$ to hash n keys into the table, the expected number of collisions is at most $1/2$.

Proof: By the definition of universality, the probability that 2 given keys in the table collide under h is $\frac{1}{m} = \frac{1}{n^2}$. Since there are $\binom{n}{2}$ pairs of keys that can possibly collide, the expected number of collisions is

$$\binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2}.$$

No collisions at level 2

Corollary: The probability of no collisions is at least $1/2$.

Proof: Markov's inequality says that for any non-negative random variable X, we have

$$Pr\{X \geq t\} \leq E[X]/t$$

Applying this inequality with $t = 1$, we find that the probability of 1 or more collisions is at most $1/2$

Thus, just by testing random hash functions in H, we will quickly find one that works.

Analysis of Storage

For the level-1 hash table T, choose $m = n$, and let n_i be random variable for number of keys that hash to slot i in T. By using n_i^2 slots for the level-2 hash table S_i, the expected total storage required for the two-level scheme is therefore

$$E\left[\sum_{i=1}^{m} \Theta(n_i^2)\right] = \Theta(n)$$

Since the analysis is identical to the analysis from recitation of the expected running time of bucket sort.

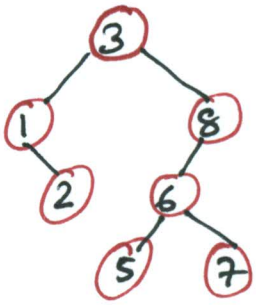
For a probability bound, apply Markov.

Binary-Search-Tree Sort

```
T ← φ        ▸ Create an empty BST
for i=1 to n
    do TREE-INSERT (T, A[i])
Perform an inorder tree walk of T.
```

Example:

A = [3 1 8 2 6 9 5]

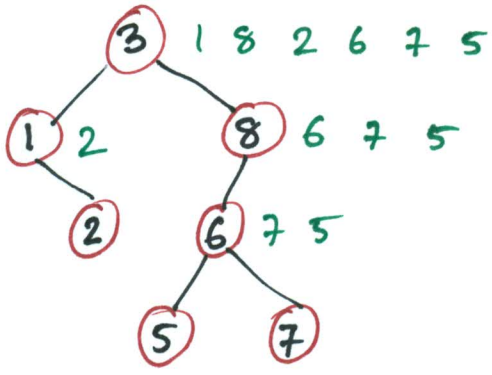


* Tree-walk time = $O(n)$

but how long does it take to build the BST?

Analysis of BST sort

BST sort performs the same comparisions as quicksort but in a different order



The ~~exposed~~ expected time to build the tree is asymptotically the same as the running time of quicksort.

Node depth

The depth of a node = the number of comparisions made during TREE-INSERT

Assuming all input permutations are equally-likely, we have

$$\text{average node depth} = \frac{1}{n} E\left[\sum_{i=1}^{n} [\#\text{comparisons to insert node } i]\right]$$

$$= \frac{1}{n} O(n \lg n) \quad \text{(quick sort analysis)}$$

$$= O(\lg n)$$

Week 6. Lecture Notes

Topics: Randomly built BST.
Red Black Tree.
Augmentation of data structure.
Interval trees.

Randomized BST sort

Rand. BST_Sort (A[1,...n])

1. Random permutation on A
2. BST_Sort (A)

The expected time to build the tree is asymptotically the same as the running time of Randomized Quicksort, which is $O(n \log n)$.

Node depth

The depth of a node = the number of comparisions made during TREE-INSERT

Assuming all input permutations are equally-likely, we have

$$\text{average node depth} = \frac{1}{n} E\left[\sum_{i=1}^{n} [\#\text{comparisons to insert node } i]\right]$$

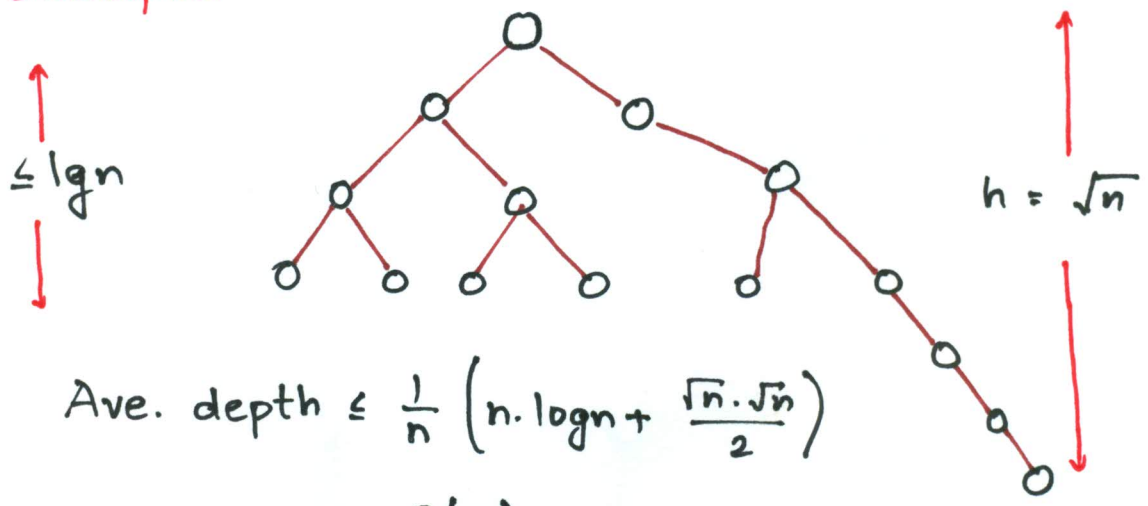
$$= \frac{1}{n} O(n \lg n) \quad \text{(quick sort analysis)}$$

$$= O(\lg n)$$

Expected Tree height.

Average node depth of a randomly built BST = $O(\log n)$, does not necessarily mean that its expected height is also $O(\log n)$ (although it is)

Example:



$$\text{Ave. depth} \le \frac{1}{n}\left(n \cdot \log n + \frac{\sqrt{n} \cdot \sqrt{n}}{2}\right)$$

$$= O(n)$$

Height of a randomly built binary search tree

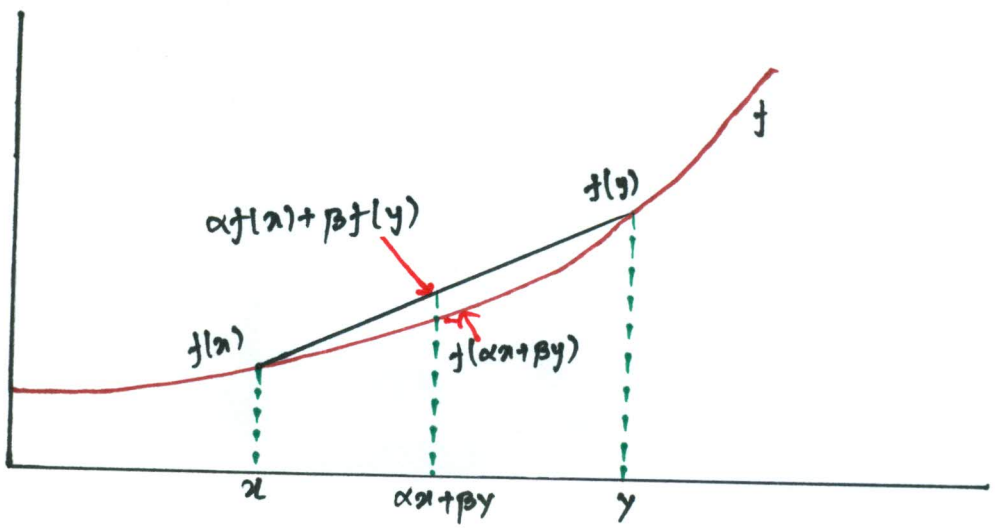
Outline of the analysis:

- Prove Jensen's inequality, which says that $f(E[x]) \leq E[f(x)]$, for any convex function f and random variable X.
- Analyze the exponential height of a randomly built BST on n nodes, which is the random variable $Y_n = 2^{X_n}$, where X_n is the random variable denoting the height of the BST.
- Prove that $2^{E[X_n]} \leq E[2^{X_n}] = E[Y_n] = O(n^3)$ and hence that $E[X_n] = O(\log n)$.

Convex functions

A function $f: R \to R$ is convex if for all $\alpha, \beta \geq 0$ such that $\alpha + \beta = 1$, we have

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y) \quad \text{for all } x, y \in R$$



Convexity Lemma

Lemma: Let $f: R \to R$ be a convex function, and let $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ be a set of non-negative constants such that $\sum_k \alpha_k = 1$. Then, for any set $\{x_1, x_2, \ldots, x_n\}$ of real numbers, we have

$$f\left(\sum_{k=1}^{n} \alpha_k x_k\right) \leq \sum_{k=1}^{n} \alpha_k f(x_k)$$

Proof: We prove by induction on n.

For $n=1$, we have $\alpha_1 = 1$, and hence $f(\alpha_1 x_1) \leq \alpha_1 f(x_1)$

Inductive step:

$$f\left(\sum_{k=1}^{n} \alpha_k x_k\right) = f\left(\alpha_n x_n + (1-\alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} x_k\right)$$

$$\leq \alpha_n f(x_n) + (1-\alpha_n) f\left(\sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} x_k\right)$$

$$\leq \alpha_n f(x_n) + (1-\alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} f(x_k)$$

$$= \sum_{k=1}^{n} \alpha_k f(x_k).$$

$$\therefore \quad f\left(\sum_{k=1}^{n} \alpha_k x_k\right) \leq \sum_{k=1}^{n} \alpha_k f(x_k)$$

Jensen's Inequality

Lemma: Let f be a convex function, and let X be a random variable. Then

$$f(E[X]) \leq E[f(X)]$$

Proof:

$$f(E[X]) = f\left(\sum_{k=-\infty}^{\infty} k \cdot Pr\{X = k\}\right)$$

↳ Definition of expectation

$$\leq \sum_{k=-\infty}^{\infty} f(k) \cdot Pr\{X = k\}$$

↳ Convexity lemma (generalized)

$$= E[f(X)]$$

Analysis of BST height

Let X_n be the random variable denoting the height of a randomly built binary search tree on n nodes, and let $Y_n = 2^{X_n}$ be its exponential height.

If the root of the tree has rank k, then

$$X_n = 1 + \max\{X_{k-1}, X_{n-k}\}$$

since each of the left and right subtrees of the root are randomly built.

Hence, we have

$$Y_n = 2 \cdot \max\{Y_{k-1}, Y_{n-k}\}$$

Define the indicator random variable Z_{nk} as

$$Z_{nk} = \begin{cases} 1, & \text{if root has rank } k; \\ 0, & \text{otherwise.} \end{cases}$$

Thus

$$Pr\{Z_{nk} = 1\} = E[Z_{nk}] = 1/n$$

and

$$Y_n = \sum_{k=1}^{n} Z_{nk} \left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right).$$

Exponential Height Recurrence

We have.

$$T_n = \sum_{k=1}^{n} Z_{nk} \left(2 \cdot \max\{T_{k-1}, T_{n-k}\}\right)$$

$$\Rightarrow E[T_n] = E\left[\sum_{k=1}^{n} Z_{nk} \left(2 \cdot \max\{T_{k-1}, T_{n-k}\}\right)\right]$$

↳ Taking expectations of both sid

$$= \sum_{k=1}^{n} E\left[Z_{nk} \left(2 \cdot \max\{T_{k-1}, T_{n-k}\}\right)\right]$$

↳ Linearity of expectation

$$= 2 \sum_{k=1}^{n} E[Z_{nk}] \cdot E[\max\{T_{k-1}, T_{n-k}\}]$$

↳ Independence of the rank of root from ranks of subtree roots.

$$\leq \frac{2}{n} \sum_{k=1}^{n} E[T_{k-1} + T_{n-k}]$$

↳ The max of two non-negative numbers is atmost their sum. and $E[Z_{nk}] = 1/n$

$$= \frac{4}{n} \sum_{k=0}^{n-1} E[T_k]$$

↳ Each term appears twice and re-index.

Solving the recurrence

Use substitution to show that $E[Y_n] \leq cn^3$ for some positive constants c, which we can pick sufficiently large to handle the initial conditions.

We have,

$$E[Y_n] = \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k]$$

$$\leq \frac{4}{n} \sum_{k=0}^{n-1} ck^3 \quad \rightarrow \text{Substitution}$$

$$\leq \frac{4c}{n} \int_0^n x^3 dx \quad \rightarrow \text{Integral method}$$

$$= \frac{4c}{n} \left(\frac{n^4}{4}\right) \quad \rightarrow \text{Solving the integral}$$

$$= cn^3$$

Putting it all together, we have

$$2^{E[X_n]} \leq E[2^{X_n}]$$

Jensen's inequality, since $f(x) = 2^x$ is convex

So,

$$2^{E[X_n]} \leq E[2^{X_n}]$$

$$= E[Y_n] \quad \rightarrow \text{Definition}$$

$$\leq cn^3 \quad \rightarrow \text{just showed above}$$

$$E[X_n] \leq 3 \log n + O(1) \quad \rightarrow \text{Taking log both sides}$$

Hence,

$$\boxed{E[X_n] \leq 3 \log n + O(1)}$$

Post Mortem

Q. Does the analysis have to be this hard?

Q. Why bother with analyzing exponential height?

Q. Why not just develop the recurrence on

$$X_n = 1 + \max\{X_{k-1}, X_{n-k}\}$$

directly.

Answer:

The inequality $\max\{a,b\} \leq a+b$, provides a poor upper bound, since the R.H.S. approaches the L.H.S. slowly as $|a-b|$ increases.

The bound

$$\max\{2^a, 2^b\} \leq 2^a + 2^b$$

allows the R.H.S. to approach the L.H.S. far more quickly as $|a-b|$ increases.

By using the convexity of $f(x) = 2^x$ via Jensen's inequality, we can manipulate the sum of exponentials, resulting in a tight analysis.

Balanced Search Trees

Balanced search tree: A search-tree data structure for which a height of $O(\log n)$ is guaranteed when implementing a dynamic set on n items.

Examples: AVL Trees, 2-3 trees
2-3-4 trees, B-trees.
Red-Black Trees

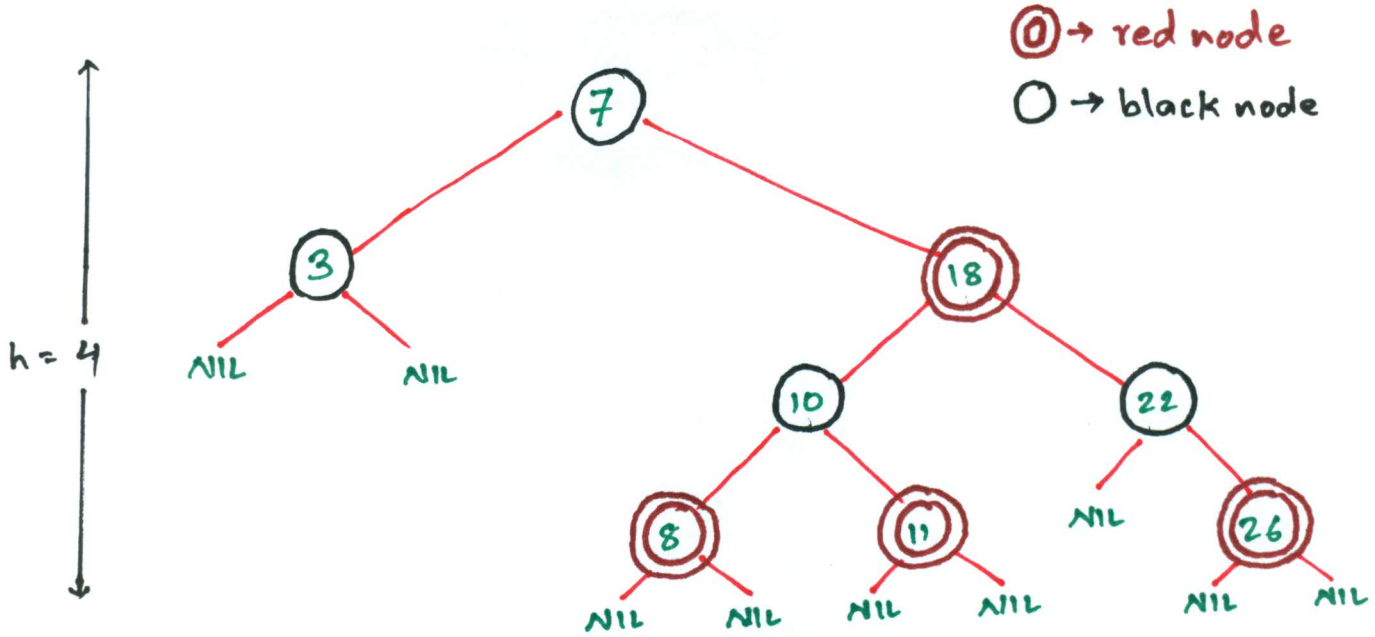
Red-Black Trees

This data structure requires an extra one-bit field - color, in each node.

Red-black properties:

1. Every node is either red or black
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height (x).

Example of a red-black tree



1. Every node is either red or black
2. The root and leaves (NIL's) are black
3. If a node is red, then it's parent is black
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black height (x)

Height of a red black tree

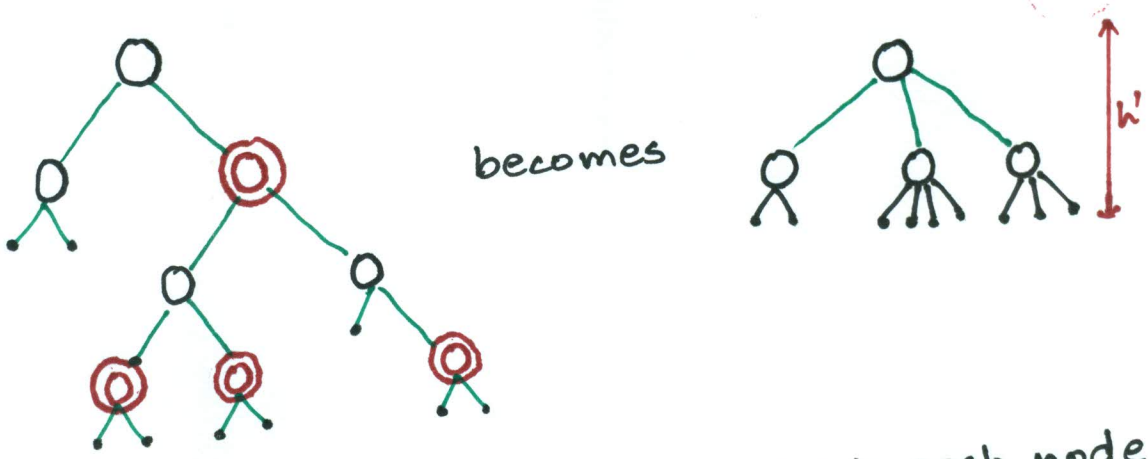
Theorem:
A red black tree with n keys has height

$$h \leq 2 \log(n+1).$$

Proof:

- Intituition: Merge red nodes into their black parents.

So.



- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth h' of leaves
- We have $h' \geq h/2$, since atmost half the leaves on any path are red
- The number of leaves in each tree is $n+1$

$$\Rightarrow n+1 \geq 2^{h'}$$

$$\Rightarrow \log(n+1) \geq h' \geq h/2$$

$$\Rightarrow \boxed{h \leq 2 \log(n+1)}$$

Query Operations

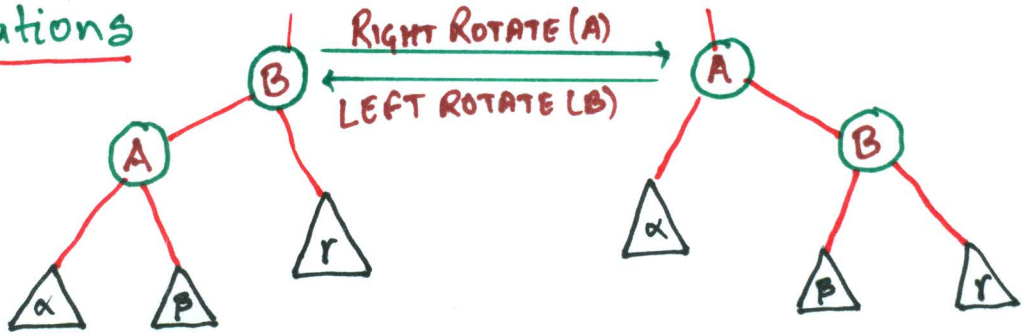
Corollary: The queries SEARCH, MIN, MAX, SUCCESSOR and PREDECESSOR all run in $O(\log n)$ time on a red-black tree with n-nodes.

Modifying Operations

The operations INSERT and DELETE cause modifications to the red black tree:

- The operation itself
- color changes
- restructuring the links of the tree:
  "rotation"

Rotations



Rotations maintain the inorder ordering of keys:

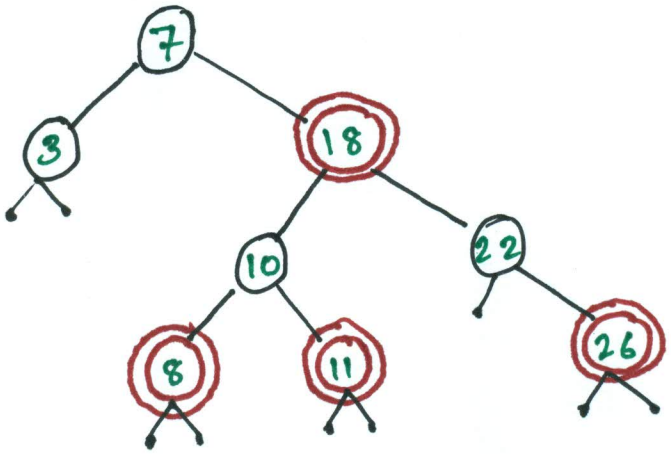
$a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c$

A rotation can be performed in $O(1)$ time.

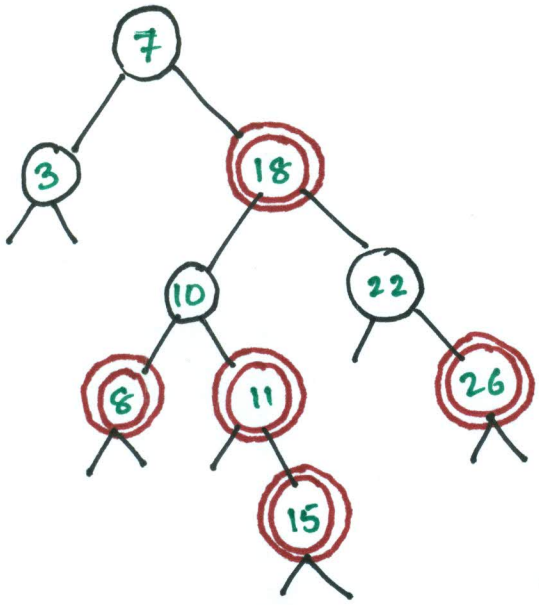
Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring

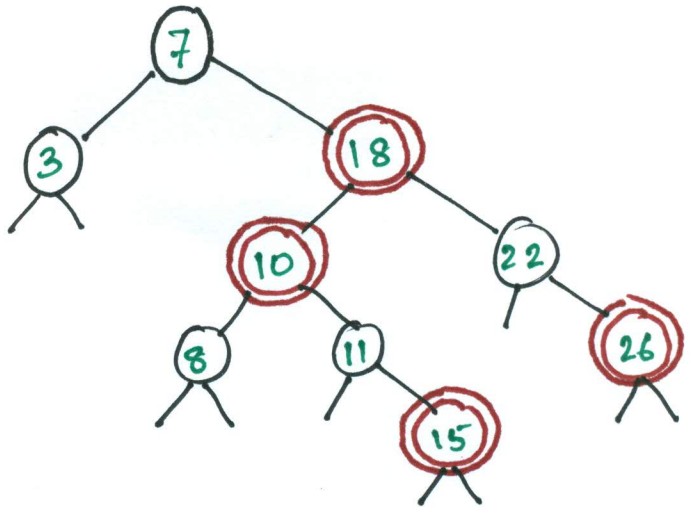
Example:



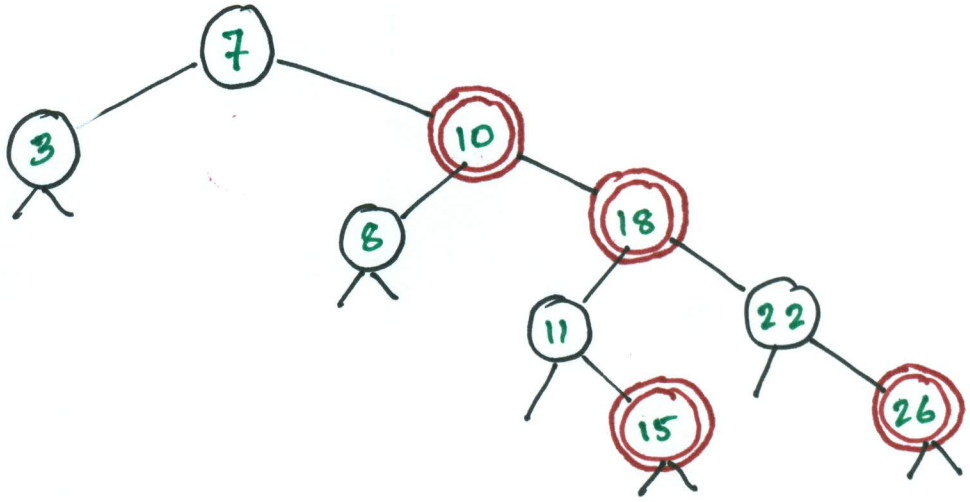
1. Insert $x = 15$.



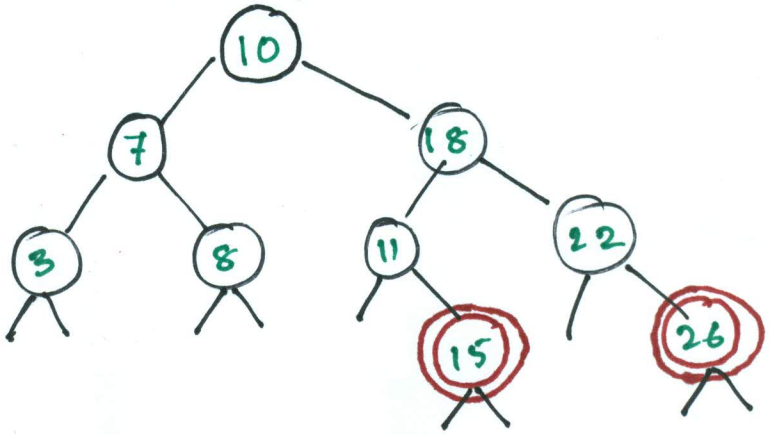
2. Recolor, moving the violation up the tree



3. RIGHT-ROTATE (10)



4. LEFT-ROTATE (7) and recolor



Pseudo code

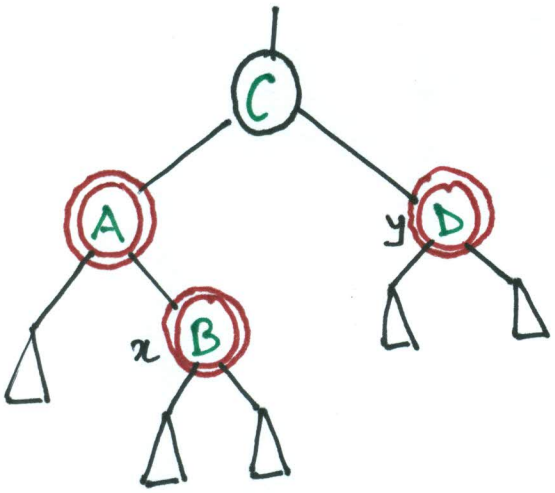
```
1.  RB-INSERT(T, x)
2.      TREE-INSERT(T, x)
3.      color[x] ← RED          ▸ only RB property 3 can be violated
4.      while x ≠ root[T] and color[p[x]] = RED
5.          do if p[x] = left[p[p[x]]]
6.              then y ← right[p[p[x]]]      ▸ y = aunt/uncle of x
7.                   if color[y] = RED
8.                   then ⟨Case 1⟩
9.                   else if x = right[p[x]]
10.                       then ⟨case 2⟩      ▸ case 2 falls into case 3
11.                      ⟨case 3⟩
12.             else ⟨"then" clause with "left" and
                      "right" swapped⟩
13.     color[root[T]] ← BLACK
```

Graphical Notation

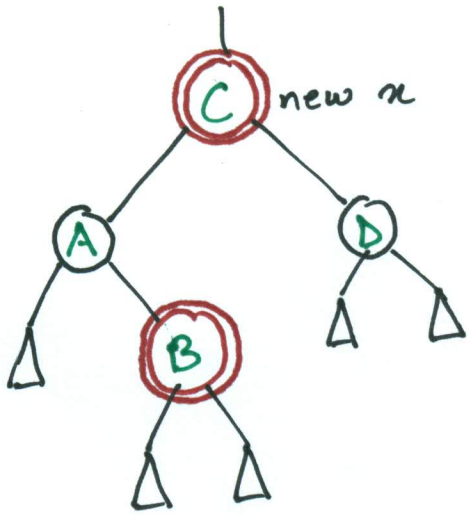
Let △ denote a subtree with a black root

All △'s have the same black height

Case 1



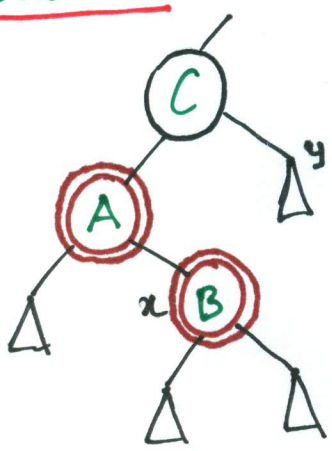
Recolor



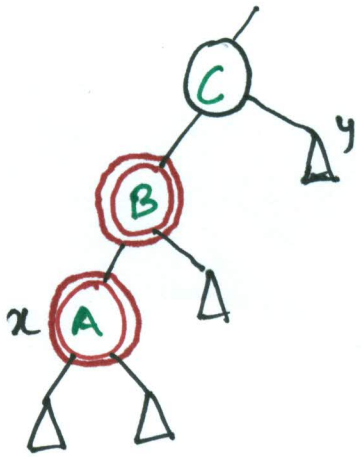
(Or, children of A are swapped)

Push C's black onto A and D, and recurse since C's parent may be red.

Case 2

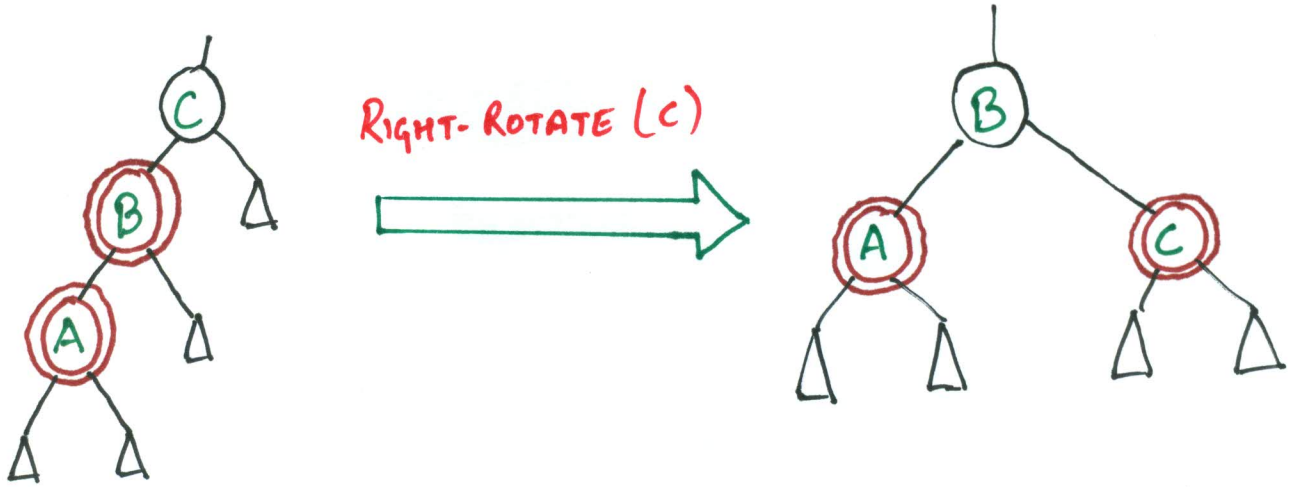


LEFT-ROTATE A



Transform to Case 3

Case 3



Done! No more violations of RB property 3 are possible.

Analysis

- Go up the tree performing Case 1, which only recolor nodes
- If case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

Running Time:

$O(\log n)$ with $O(1)$ rotations

Note:

RB-DELETE - takes same asymptotic running time.

Dynamic Order Statistics

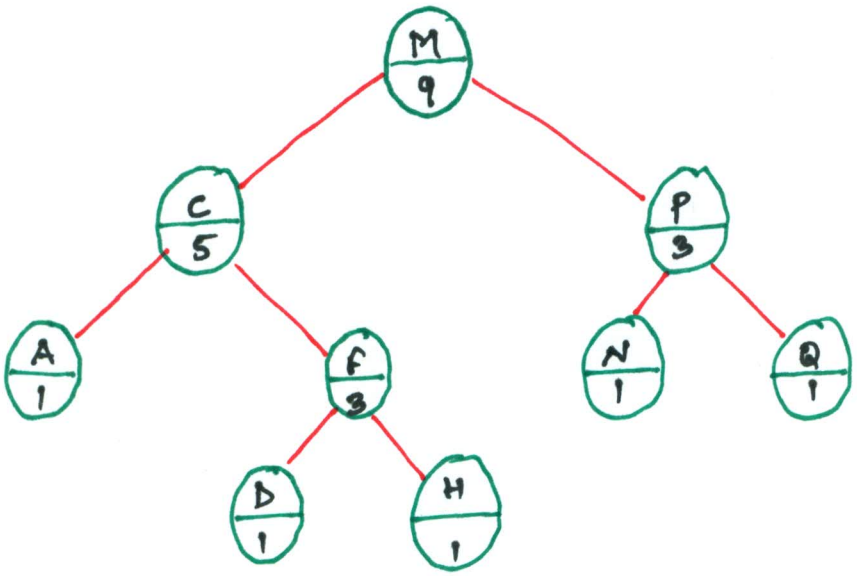
OS-SELECT (i, S) : returns the i^{th} smallest element in the dynamic set S

OS-RANK (x, S): returns the rank of $x \in S$ in the sorted order of S's elements.

IDEA: Use a red-black tree for the set S, but keep subtree sizes in the node.

Notation for nodes:

Example of an OS tree



size[x] = size[left[x]] + size[right[x]] + 1

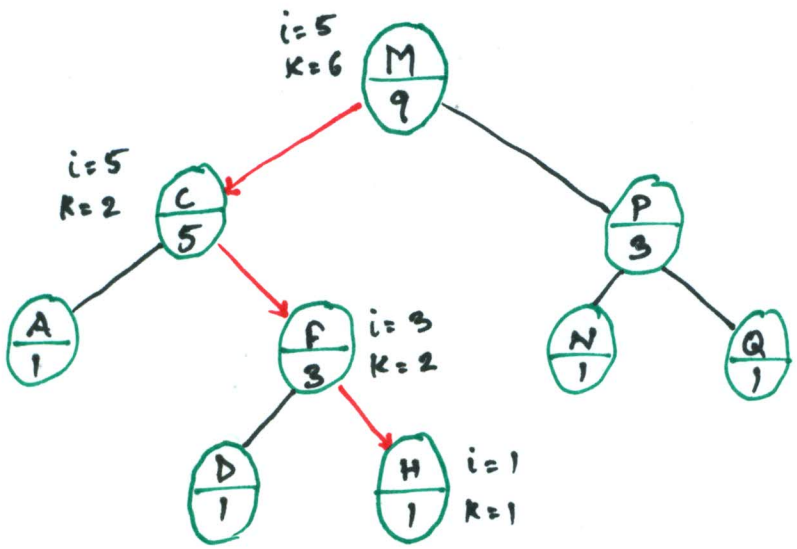
SELECTION

Implementation Trick: Use a sentinel (dummy record) for NIL such that size[NIL] = 0.

```
1.  OS-SELECT(x, i)
2.      k ← size[left[x]] + 1        ▷ k = rank(x)
3.      if i = k then return x
4.      if i < k
5.          then return OS-SELECT(left[x], i)
6.          else return OS-SELECT(right[x], i-k)
```

Example:

OS-SELECT(root, 5)



Running Time: $O(h) = O(\log n)$ for red-black trees.

Data Structure Maintenance

Q. Why not keep the ranks themselves in the nodes instead of subtree sizes?

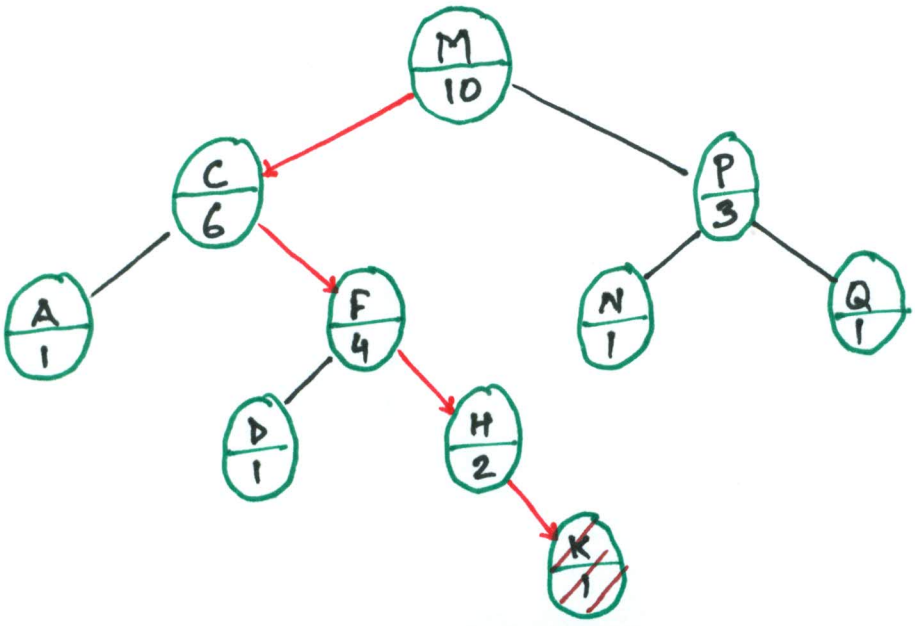
A. They are hard to maintain when the red-black tree is modified.

Modifying operations: INSERT and DELETE

Strategy: Update subtree sizes when inserting or deleting.

Example of insertion

INSERT ("K")



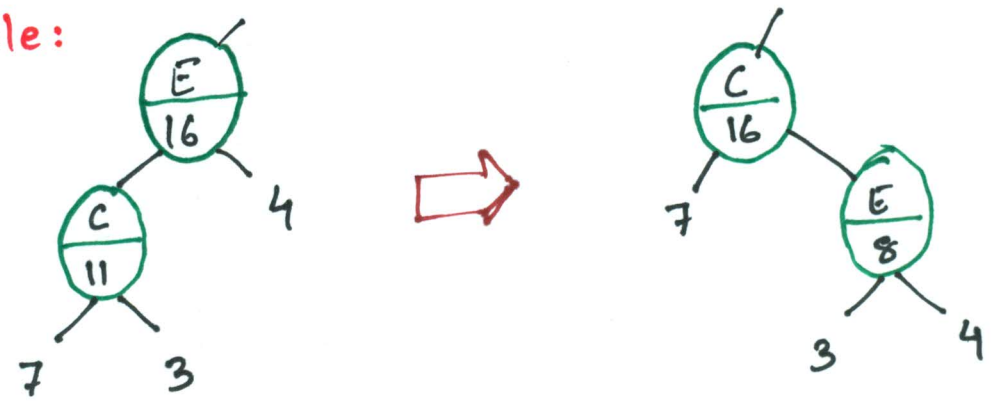
Handling rebalancing

Don't forget that RB-INSERT and RB-DELETE may also need to modify the red-black tree in order to maintain balance.

Recolorings: no effect on subtree sizes.

Rotations: fix up subtree sizes in $O(1)$ time

Example:



∴ RB-INSERT and R.B. DELETE run in $O(\log n)$ time.

Data-structure augmentation

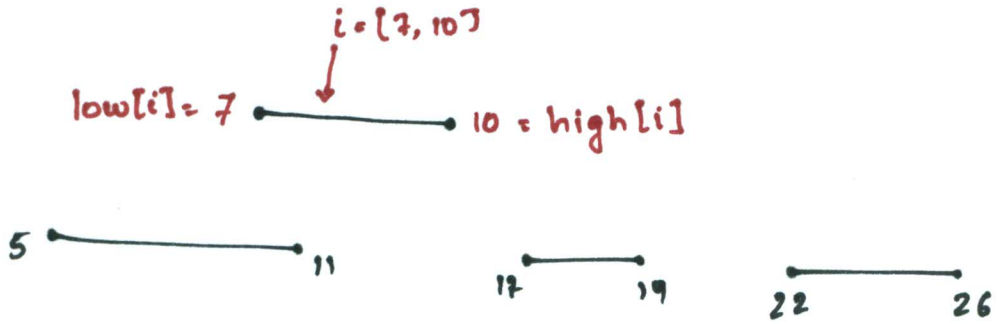
Methodology: (eg., order-statistics trees)

1. Choose an underlying data structure (red. black trees)
2. Determine additional information to be stored in the data structure (subtree sizes)
3. Verify that this information can be maintained for modifying operations (RB-INSERT, RB-DELETE - don't forget rotations)
4. Develop new dynamic-set operations that use the information (OS-SELECT and OS-RANK)

These steps are guidelines, not rigid rules.

Interval Trees

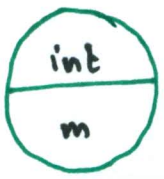
Goal: To maintain a dynamic set of intervals, such as time intervals.



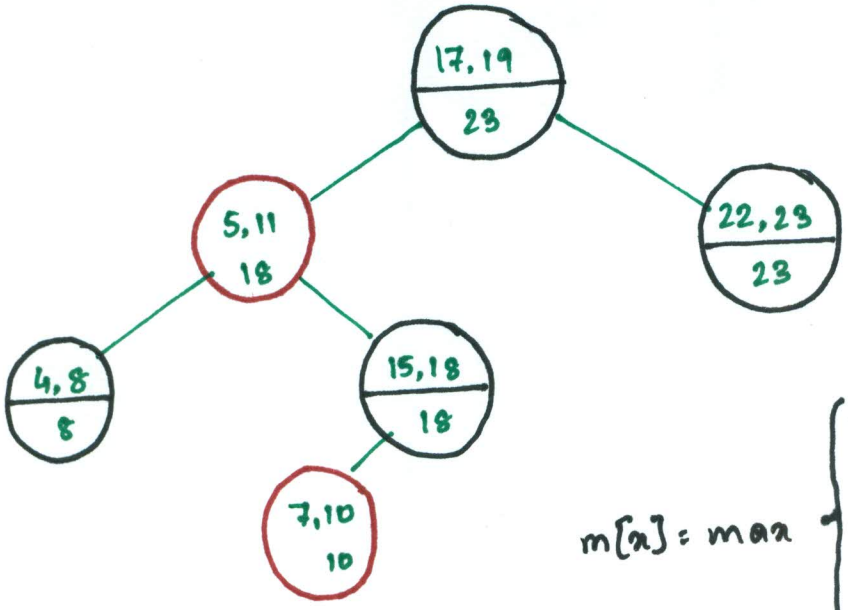
Query: For a given query interval i, find an interval in the set that overlaps i.

Following the methodology

1. Choose an underlying data structure.
   - Red black tree keyed on low(left) endpoint.
2. Determine additional information to be stored in the data structure.
   - Store in each node x the largest value $m[x]$ in the subtree rooted at x, as well as the interval $int[x]$ corresponding to the key.



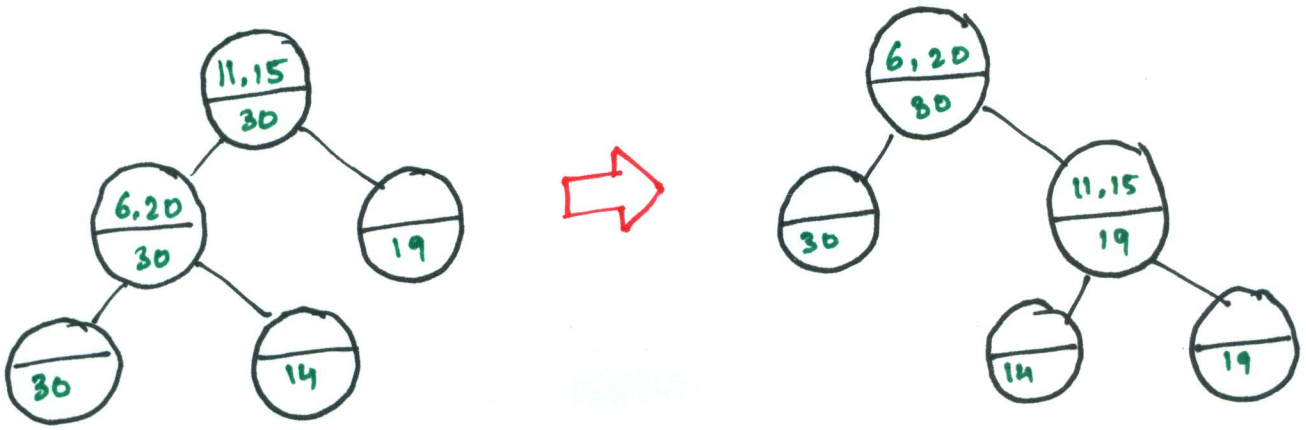
Example interval tree



$$m[x] = \max \begin{cases} high[int[x]] \\ m[left[x]] \\ m[right[x]] \end{cases}$$

Modifying operations

3. Verify that this information can be maintained for modifying operations.
   - INSERT : fix m's on the way down
   - ROTATION: fixup : O(1) time per rotation.



Total insert time: O(log n);

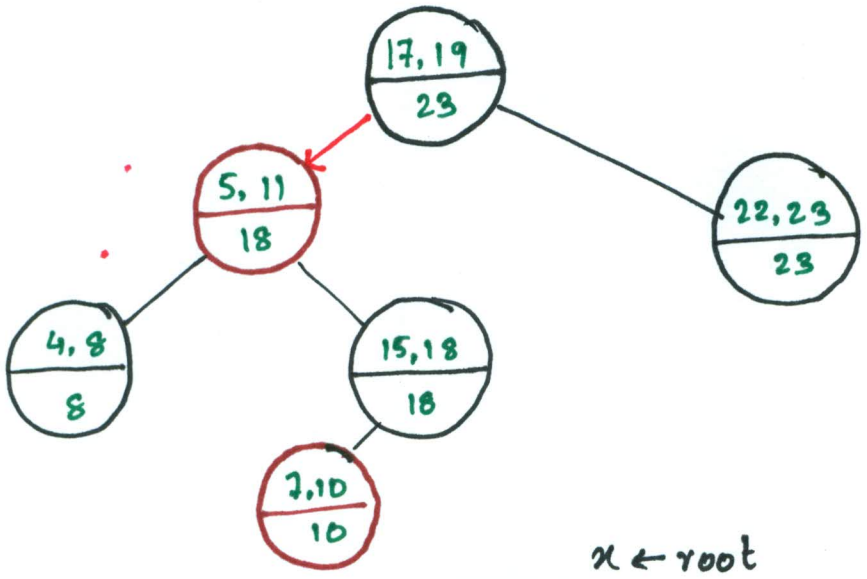
Delete similar.

New operations

4. Develop new dynamic-set operations that use the information.

```
INTERVAL-SEARCH (i)
1.  x ← root
2.  while x ≠ NIL and (low[i] > high[int[x]]
                       or low[int[x]] > high[i])
3.  do ▸ i and int[x] don't overlap
4.     if left[x] ≠ NIL and Low[i] ≤ m[left[x]]
5.        then x ← left[x]
6.        else x ← right[x]
7.  return x
```

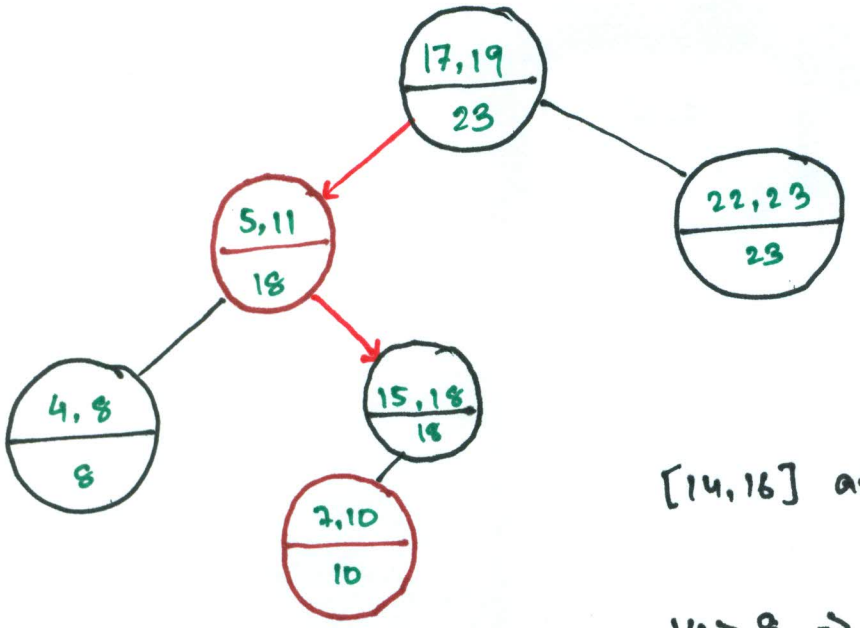
Example 1: INTERVAL-SEARCH ([14,16])



$x \leftarrow root$

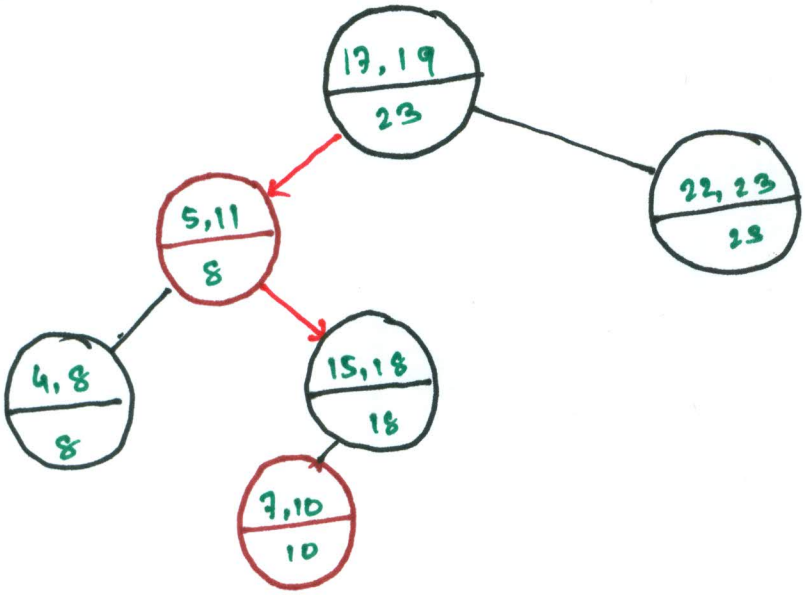
[14,16] and [17,19] don't overlap

$14 \leq 18 \Rightarrow x \leftarrow left[x]$



[14,16] and [5,11] don't overlap

14 > 8 → x ← right [x]

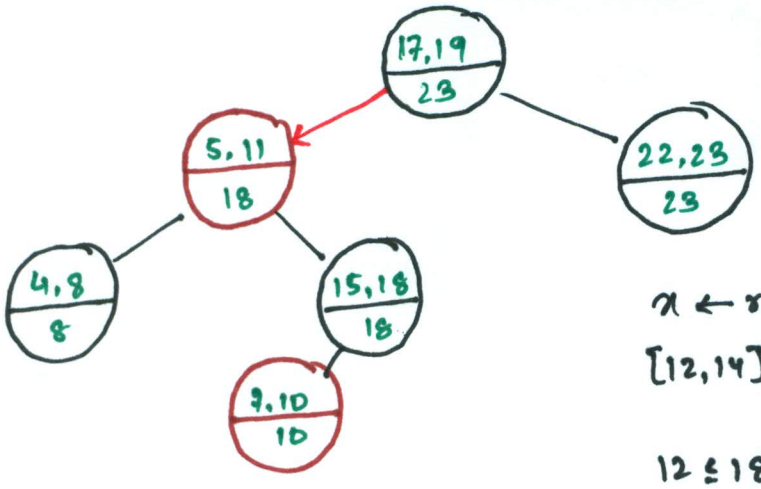


[14,16], and [15,18] overlap

return [15,18]

Example 2: INTERVAL-SEARCH([12,14])

1.

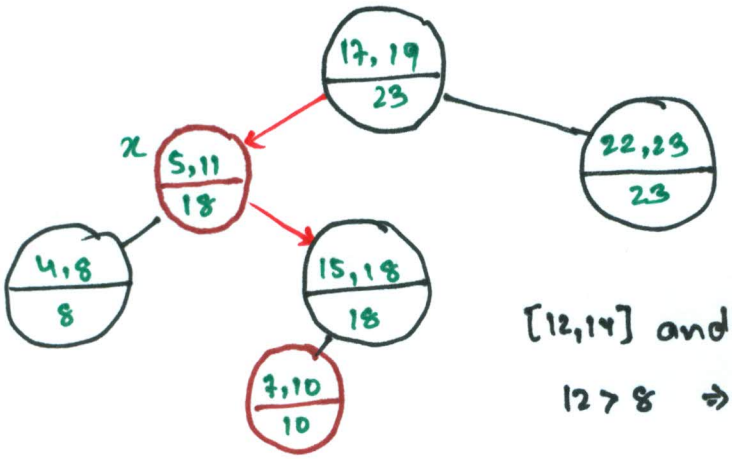


$x \leftarrow$ root

[12,14] and [17,19] don't overlap

$12 \leq 18 \Rightarrow x \leftarrow$ left[x]

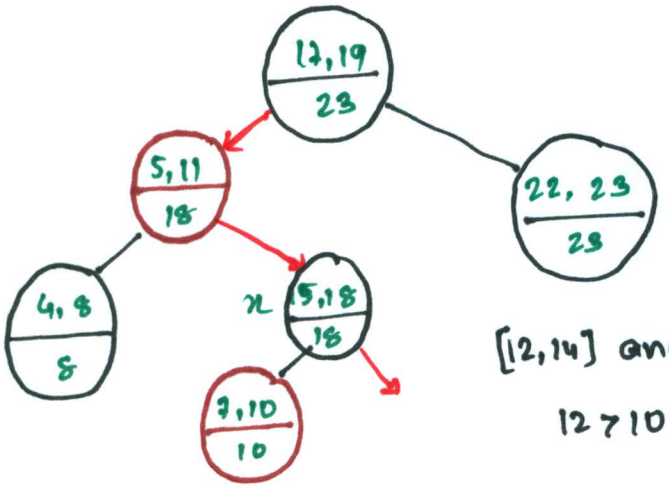
2.



[12,14] and [5,11] don't overlap

$12 > 8 \Rightarrow x \leftarrow$ right[x]

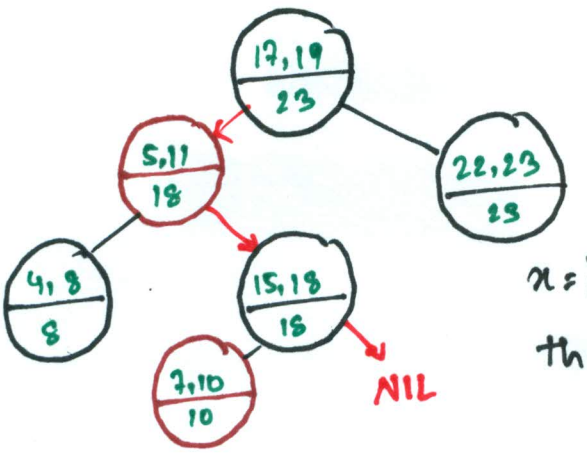
3.



[12,14] and [15,18] don't overlap

$12 < 10 \Rightarrow x \rightarrow$ right[x]

4.



$x = [NIL] \Rightarrow$ no interval that overlaps [12,14] exist.

Analysis

Time = $O(h) = O(\log n)$, since INTERVAL-SEARCH does constant work at each level as it follows a simple path down the tree.

List all overlapping intervals:

- Search, list, delete, repeat
- Insert them all again at the end

Time = $O(K \log n)$, where K is the total number of overlapping intervals.

This is an output-sensitive bound.

Best algorithm to date: $O(K + \log n)$

Correctness

Theorem: Let L be the set of intervals in the left subtree of node x, and let R be the set of intervals in x's right subtree.

if the search goes right, then

$$\{i' \in L : i' \text{ overlaps } i\} = \phi$$

if the search goes left, then

$$\{i' \in L : i' \text{ overlaps } i\} = \phi$$

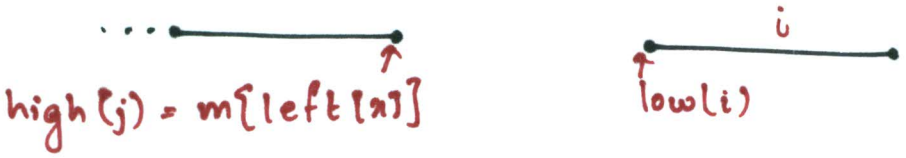
$$\Rightarrow \{i' \in R : i' \text{ overlaps } i\} = \phi$$

In other words, it's always safe to take ~~any~~ only 1 of the two children: we will either find something or nothing was to be found.

Correctness Proof

Proof: Suppose first that search goes right.

- If left[x] = NIL, then we are done, since $L = \phi$
- Otherwise, the code dictates that we must have low[i] > m[left[x]]. The value m[left[x]] corresponds to the right endpoint of some interval $j \in L$, and no other interval in L can have a larger right endpoint then high(j)



- Therefore, $\{i' \in L : i' \text{ overlaps } i\} = \phi$.

Next suppose that the search goes left, and assume that

$$\{i' \in L : i' \text{ overlaps } i\} = \phi$$

- Then, the code dictates that low[i] $\leq$ m[left[x]] = high[j] for some $j \in L$
- Since $j \in L$, it doesnot overlap i, and hence
  high[i] < low[j]
- But, the binary-search tree property implies that for all $i' \in R$, we have low[j] $\leq$ low[i']
- But then $\{i' \in R;\ i' \text{ overlaps } i\} = \phi$.

Week 7 Lecture Notes

Topics: Fixed universe successor
Van Emde Boas data structure
Amortized analysis
Computation Geometry

Fixed-universe successor problem

Goal: Maintain a dynamic subset S of size n of the universe $U = \{0, 1, \ldots, u-1\}$ of size u subject to these operations.

INSERT ($x \in U \setminus S$): Add x to S

DELETE ($x \in S$): Remove x from S

SUCCESSOR ($x \in U$): Find the next element in S larger than any element x of the universe U

PREDECESSOR ($x \in U$): Find the previous element in S smaller than x.

Solutions to fixed-universe successor problem

Goal: Maintain a dynamic subset S of size n of the universe $U = \{0, 1, \ldots, u-1\}$ of size u, subject to

INSERT, DELETE, SUCCESSOR, PREDECESSOR

- Balanced search trees can implement operations in $O(\lg n)$ time, without fixed-universe assumptions.
- In 1975, Peter van Emde Boas solved this problem in $O(\lg \lg u)$ time per operation.
  - If u is only polynomial in n, that is, $u = O(n^c)$, then $O(\lg \lg n)$ time per operation - exponential speedup

$O(\lg \lg u)$?!

Where could a bound of $O(\lg \lg u)$ arise?

- Binary search over $O(\lg u)$ things
- $T(u) = T(\sqrt{u}) + O(1)$

$$T'(\lg u) = T'((\lg u)/2) + O(1)$$

$$= O(\lg \lg u).$$

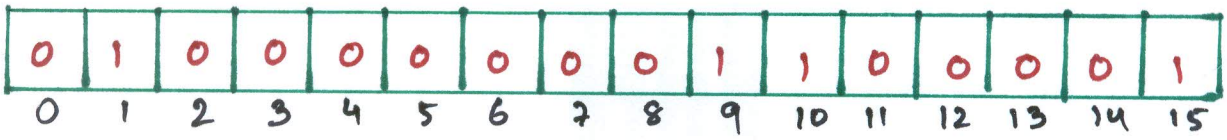
1). Starting point: Bit Vector

Bit vector v stores, for each $x \in U$

$$v_x = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

Example:

$u = 16,\ n = 4,\ S = \{1, 9, 10, 15\}$



Insert / Delete run in $O(1)$ time

Successor / Predecessor run in $O(u)$ worst-case time

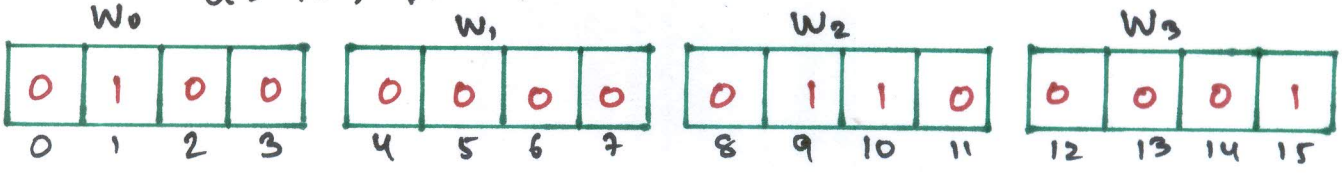
2. Split universe into widgets

Carve universe of size u into $\sqrt{u}$ widgets.

$W_0, W_1, \ldots W_{\sqrt{u}-1}$ each of size $\sqrt{u}$

Example:

$u = 16,\ \sqrt{u} = 4$



W_0 represents $0, 1, \ldots, \sqrt{u}-1 \in U$;

W_1 represents $\sqrt{u}, \sqrt{u}+1, \ldots, 2\sqrt{u}-1 \in U$;

$\vdots$

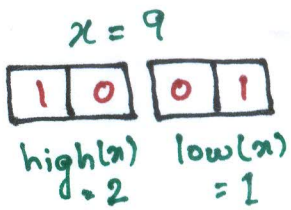
W_i represents $i\sqrt{u}, i\sqrt{u}+1, \ldots, (i+1)\sqrt{u}-1 \in U$;

$\vdots$

$W_{\sqrt{u}-1}$ represents $u-\sqrt{u}, u-\sqrt{u}+1, \ldots, u-1 \in U$.

Define $high(x) \geq 0$ and $low(x) \geq 0$

so that $x = high(x)\sqrt{u} + low(x)$.

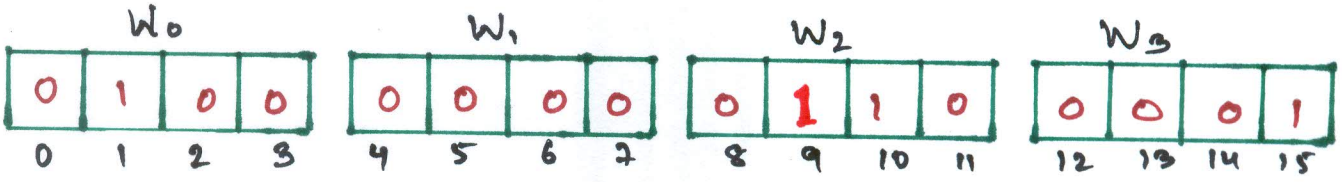


That is, if we write $x \in U$ in binary,

$high(x)$ is the high-order half of the bits, and

$low(x)$ is the low-order half of the bits.

For $x \in U$, $high(x)$ is the index of widget containing x and $low(x)$ is the index of x within that widget.



INSERT(x)

1. insert x into widget $W_{high(x)}$ at position $low(x)$
2. mark $W_{high(x)}$ as non empty

Running time: $T(n) = O(1)$

SUCCESSOR(x)

1. look for successor of x within widget $W_{high(x)}$ } $O(\sqrt{u})$
2. starting after position $low(x)$
3. if successor found
4. then return it
5. else find smallest $i > high(x)$ } $O(\sqrt{u})$
6. for which W_i is non-empty
7. return smallest element in W_i } $O(\sqrt{u})$

Running time: $T(u) = O(\sqrt{u})$

Revelation

SUCCESSOR (x)

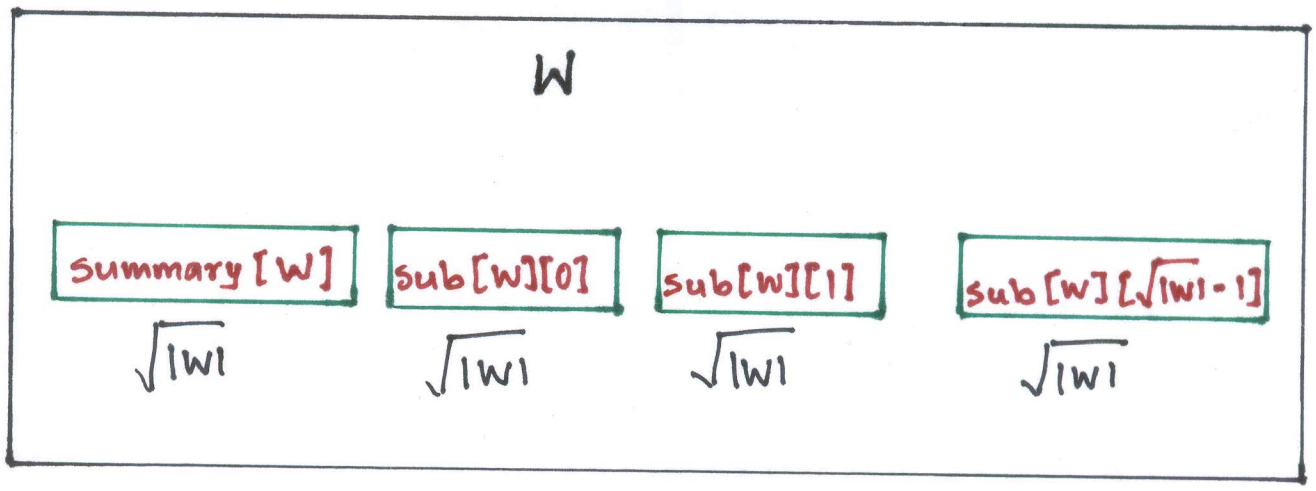
1. look for successor of x within widget $W_{high(x)}$ } recursive successor
2. starting after position low(x)
3. if successor found
4. then return it
5. else find smallest $i > high(x)$ } recursive successor
6. for which W_i is nonempty
7. return smallest element in W_i } recursive successor

3. Recursion

Represent universe by widget of size u

Recursively split each widget W of size $|W|$ into $\sqrt{|W|}$ subwidgets sub[W][0], sub[W][1], ..., sub[W][$\sqrt{|W|}-1$] each of size $\sqrt{|W|}$

Store a summary widget summary[W] of size $\sqrt{|W|}$ representing which subwidgets are non-empty.



Define $high(x) \geq 0$ and $low(x) \geq 0$

so that $x = high(x)\sqrt{|W|} + low(x)$

INSERT (x, W)

1. if sub[W][high(x)] is empty
2. then INSERT(high(x), summary[W])
3. INSERT(low(x), sub[W][high(x)])

Running time: $T(u) = 2T(\sqrt{u}) + O(1)$

$T'(\log u) = 2T'((\lg u)/2) + O(1)$

$= O(\lg u)$

SUCCESSOR (x, W)

$j \leftarrow$ SUCCESSOR(low(x), sub[W][high(x)]) } $T(\sqrt{u})$

if $j < \infty$

then return $high(x)\sqrt{|W|} + j$

else $i \leftarrow$ SUCCESSOR(high(x), summary[W]) } $T(\sqrt{u})$

$j \leftarrow$ SUCCESSOR$(-\infty$, sub[W][i]) } $T(\sqrt{u})$

return $i\sqrt{|W|} + j$

Running time: $T(u) = 3T(\sqrt{u}) + O(1)$

$T'(\lg u) = 3T'((\lg(u))/2) + O(1)$

$= O((\lg u)^{\lg 3})$.

Improvements

Need to reduce INSERT and SUCCESSOR down to 1 recursive call each.

- 1 call: $T(u) = 1\,T(\sqrt{u}) + O(1)$
  $= O(\lg \lg n)$
- 2 calls: $T(u) = 2\,T(\sqrt{u}) + O(1)$
  $= O(\log n)$
- 3 calls: $T(u) = 3\,T(\sqrt{u}) + O(1)$
  $= O((\lg n)^{\lg 3})$

We are closer to this goal than it may seem!

Recursive calls in successor

If x has a successor within sub[w][high(x)], then there is only 1 recursive call to SUCCESSOR.

Otherwise, there are 3 recursive calls.

- SUCCESSOR (low(x), sub[w][high(x)])
  discovers that sub[w][high(x)] has no successor
- SUCCESSOR (high(x), summary[w])
  finds next non-empty subwidget sub[w][i]
- SUCCESSOR ($-\infty$, sub[w][i])
  finds smallest element in subwidget sub[w][i]

Reducing recursive calls in successor

If x has no successor within sub[W][high(x)], there are 3 recursive calls:

- SUCCESSOR (low(x), sub[W][high(x)]) discovers that sub[W][high(x)] hasn't successor
  - Could be determined using the maximum value in the subwidget sub[W][high(x)]
- SUCCESSOR (high(x), summary[W]) finds next nonempty subwidget sub[W][i]
- SUCCESSOR ($-\infty$, sub[W][i])

  finds minimum element in subwidget sub[W][i].

Improved Successor

INSERT (x, W)

if sub[W][high(x)] is empty

then INSERT (high(x), summary [W])

INSERT (low(x), sub[W][high(x)])

if $x <$ min [W], then min [W] $\leftarrow x$

if $x >$ max[W], then max[W] $\leftarrow x$

} new (augmentation)

Running Time: $T(u) = 2T(\sqrt{u}) + O(1)$

$T'(\lg u) = 2T'((\lg u)/2) + O(1)$

$= O(\lg u)$

SUCCESSOR (x, W)

if low(x) < max[sub[W][high(x)]]

then $j \leftarrow$ SUCCESSOR (low(x), sub[W][high(x)]) } $T(\sqrt{u})$

return high(x) $\sqrt{|W|}$ + j

else $i \leftarrow$ SUCCESSOR (high(x), summary [W]) } $T(\sqrt{u})$

$j \leftarrow$ min [sub[W][i]]

return $i \sqrt{|W|} + j$

Running Time: $T(u) = T(\sqrt{u}) + O(1)$

$= O(\lg \lg u)$

Recursive calls in insert

If sub[w][high(x)] is already in summary[w], then there is only 1 recursive call to INSERT. Otherwise, there are 2 recursive calls:

- INSERT (high(x), summary [w])
- INSERT (low(x), sub[w][high(x)])

Idea:

We know that sub[w][high(x)]) is empty.

Avoid second recursive call by specially storing a widget containing just 1 element.

Specifically, do not store min recursively.

Improved insert

INSERT (x, W)

1. if $x <$ min[W] then exchange $x \leftrightarrow$ min[W]
2. if sub[W][high(x)] is nonempty, that is
3. min[sub[W][high(x)]] $\neq$ NIL
4. then INSERT(low(x), sub[W][high(x)])
5. else min[sub[W][high(x)]] $\leftarrow$ low(x)
6. INSERT(high(x), summary[W])
7. if $x >$ max[W] then max[W] $\leftarrow x$

Running Time: $T(u) = 1\,T(\sqrt{u}) + O(1)$

$= O(\lg \lg u)$

SUCCESSOR (x, W)

1. if $x <$ min[W] then return min[W]
2. if low(x) < max[sub[W][high(x)]]
3. then $j \leftarrow$ SUCCESSOR(low(x), sub[W][high(x)])
4. return high(x)$\sqrt{|W|} + j$
5. else $i \leftarrow$ SUCCESSOR(hig(x), summary[W])
6. $j \leftarrow$ min[sub[W][i]]
7. return $i\sqrt{|W|} + j$

Running Time: $T(u) = 1\,T(\sqrt{u}) + O(1)$

$= O(\lg \lg u)$

Deletion

DELETE (x, W)

1. if min [W] = NIL or $x <$ min [W] then return
2. if x = min[W]
3. then $i \leftarrow$ min [summary [W]]
4. $x \leftarrow i\sqrt{|W|}$ + min [sub[W][i]]
5. min [W] $\leftarrow x$
6. DELETE (low(x), sub [W][high(x)])
7. if sub [W][high(x)] is now empty, that is
   min [sub[W][high(x)]] = NIL
8. then DELETE (high(x), summary [W])

(in this case, the first recursive call was cheap)

Amortized Algorithms

How large should a hash table be?

Goal:

Make the table as small as possible, but large enough so that it won't overflow (or otherwise become inefficient).

Problem:

What if we don't know the proper size in advance?

Solution:

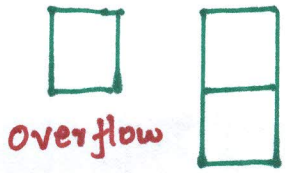
Dynamic Tables

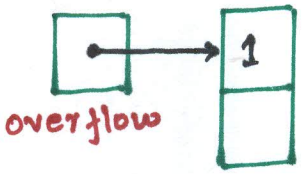
Idea:

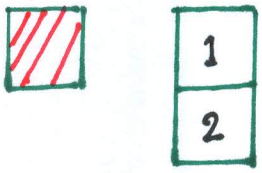
Whenever the table overflows, "grow" it by allocating (via `malloc` or `new`) a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.

Example of a dynamic table

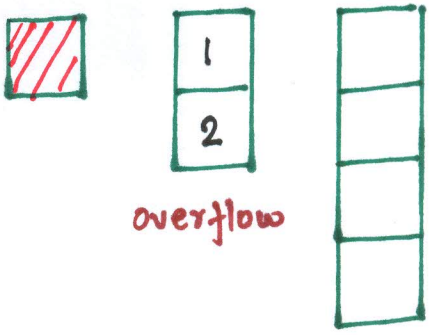
1. INSERT
2. INSERT

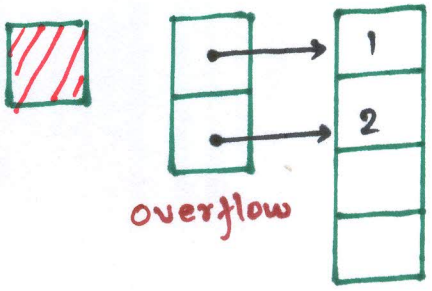


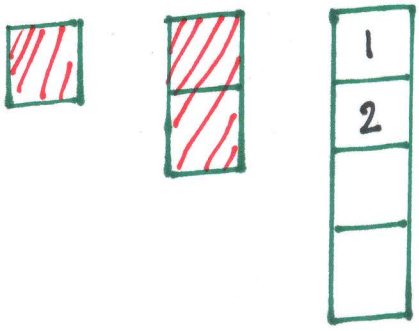




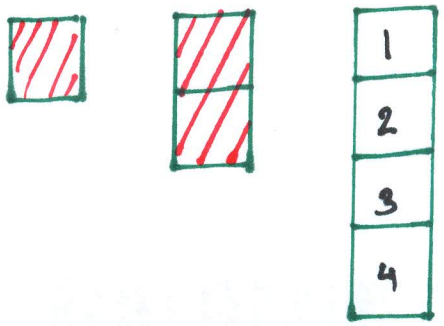
1. INSERT
2. INSERT
3. INSERT



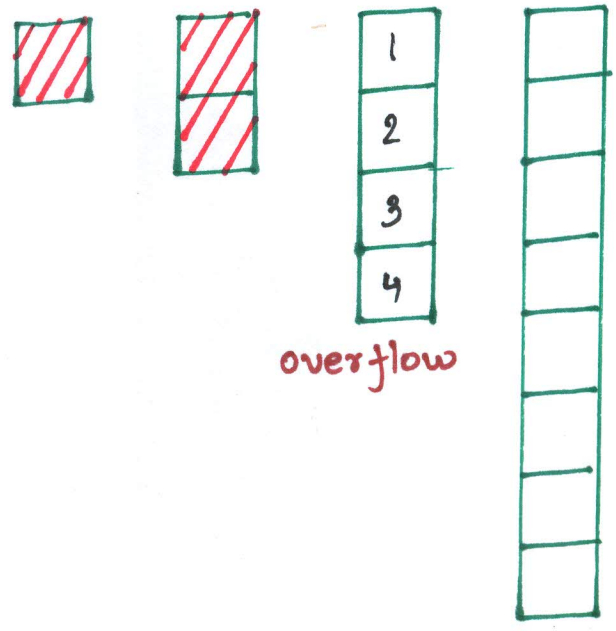


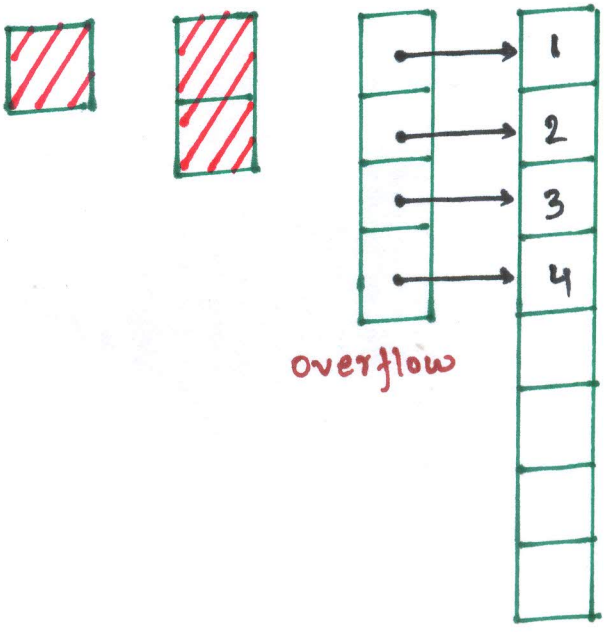


1. INSERT
2. INSERT
3. INSERT
4. INSERT

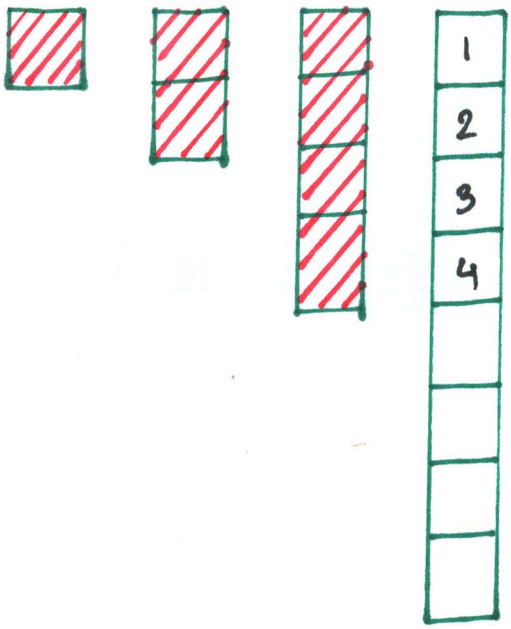


1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT

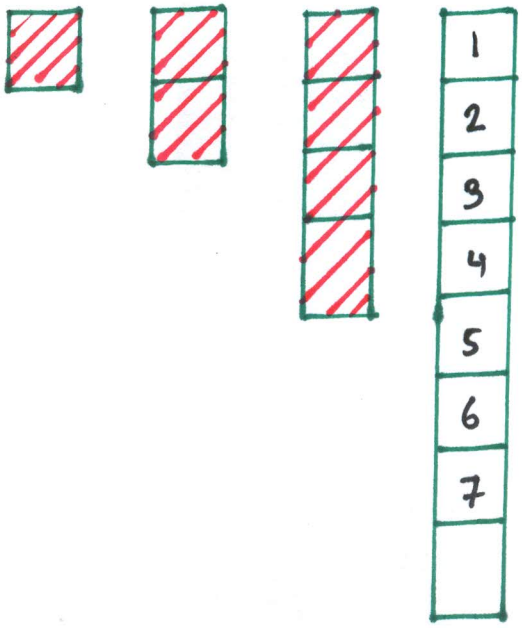




1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Worst-case analysis

Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$.

Therefore, the worst case time for n insertions is:

$$n \cdot \Theta(n) = \Theta(n^2).$$

WRONG! Infact, the worst case cost for n insertions is only $\Theta(n) << \Theta(n^2)$

Let us see why?

Tighter analysis

Let c_i = the cost of i^{th} insertion

$$= \begin{cases} i, & \text{if } i-1 \text{ is an exact power of 2;} \\ 1, & \text{otherwise.} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

Tighter analysis (continued)

i	1	2	3	4	5	6	7	8	9	10
size$_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1	1	1	1	1	1	1	1	1
		1	2		4				8	

Cost of n operations $= \sum_{i=1}^{n} c_i$

$$\leq n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j$$

$$\leq 3n$$

$$= \theta(n)$$

Thus, the average cost of each dynamic-table operation is:

$$\frac{\theta(n)}{n} = \theta(1)$$

Amortized Analysis

An amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be eapensive.

Even though we are taking averages, however, probability is not involved!

- An amortized analysis guarantees the average performance of each operation in the worst case.

Types of amortized analyses

Three common amortization arguments:

the aggregate method,
the accounting method,
the potential method.

We have just seen an aggregate analysis.

The aggregate method, though simple, lacks the precision of other two methods.

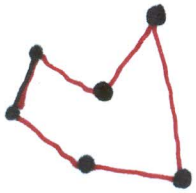
In particular, the accounting and potential methods allow a specific amortized cost to be allocated to each operation.

Computational Geometry

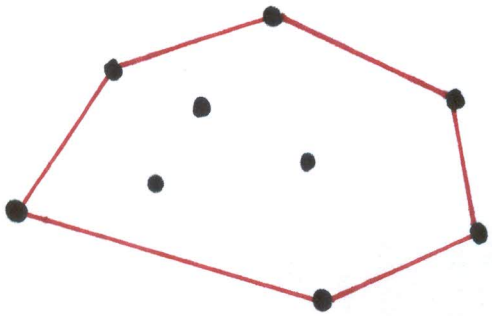
- Algorithms for solving "geometric problems" in 2D and higher.
- Fundamental Objects

- Basic Structures

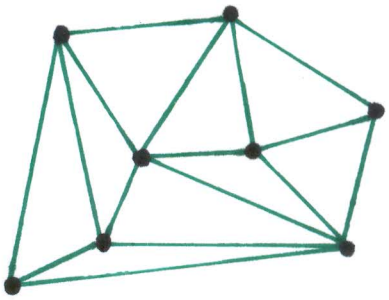
point set



polygon



Convex hull



triangulation.

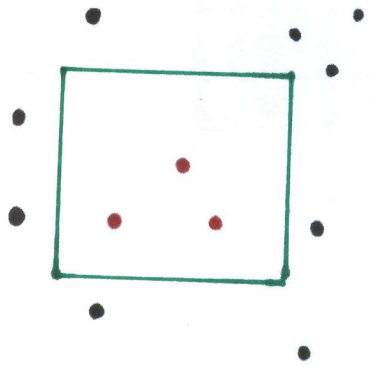
Orthogonal Range Searching

Input: n points in d dimensions.

- E.g. representing a database of n records each with d numeric fields.

Query: Axis aligned box (in 2D, a rectangle)

- Reports on the points inside the box:
  - Are there any points?
  - How many are there?
  - List the points.



Goal: Preprocessing points into data structure to support fast queries.

- Primary goal: Static data structure
- In 1D we will also obtain a dynamic data structure supporting insert and delete.

1D range searching

In 1D, the query is an interval

First solution using ideas we know:

- Interval trees
  - Represent each point x by the interval $[x,x]$.
  - Obtain a dynamic structure that can list K answers in a query in $O(K \lg n)$ time.

Second solution using ideas we know:

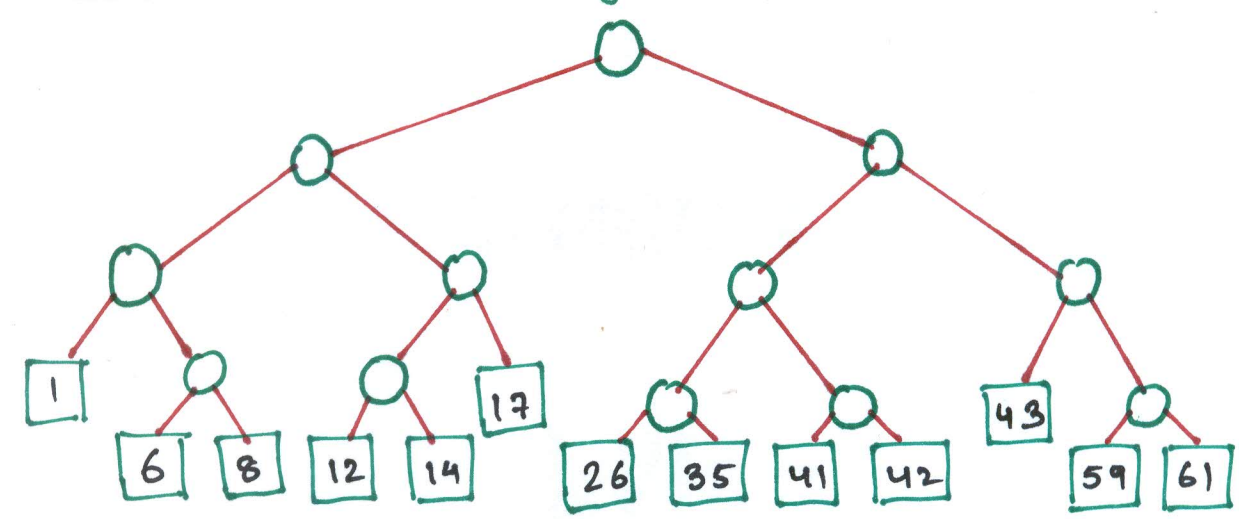
- Sort the points and store them in an array
  - Solve query by binary search on end points
  - Obtain a static structure that can list K answers in a query in $O(K+\log n)$ time.

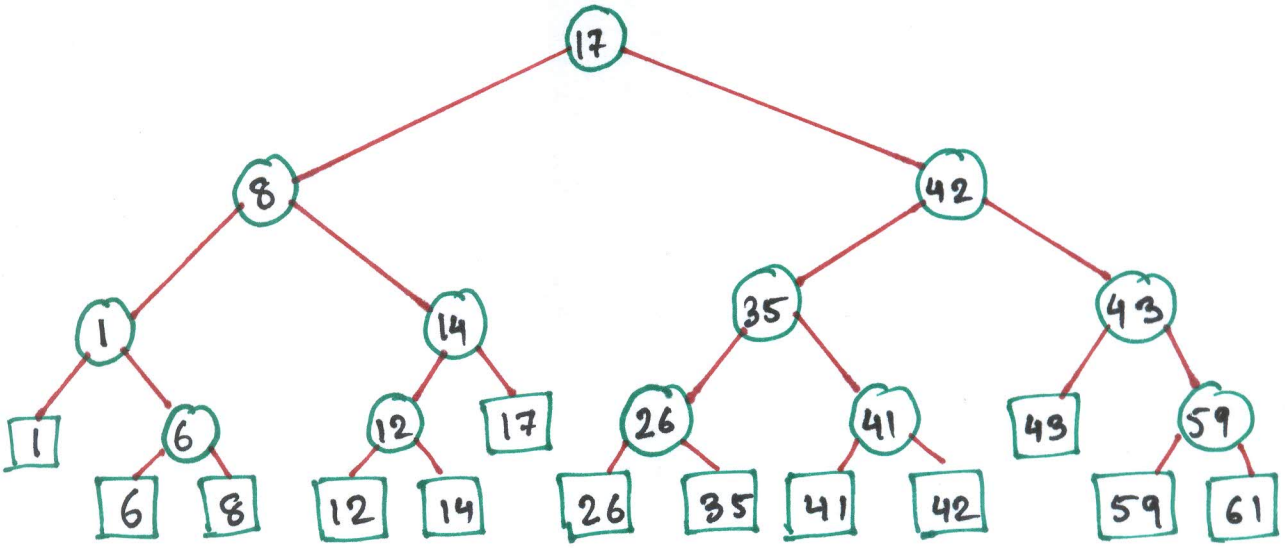
Goal: Obtain a dynamic structure that can list K answers in a query in $O(K+\log n)$ time.

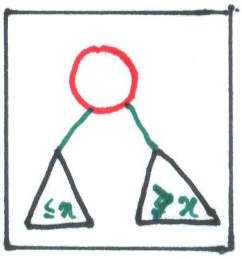
New solution that extends to higher dimensions:

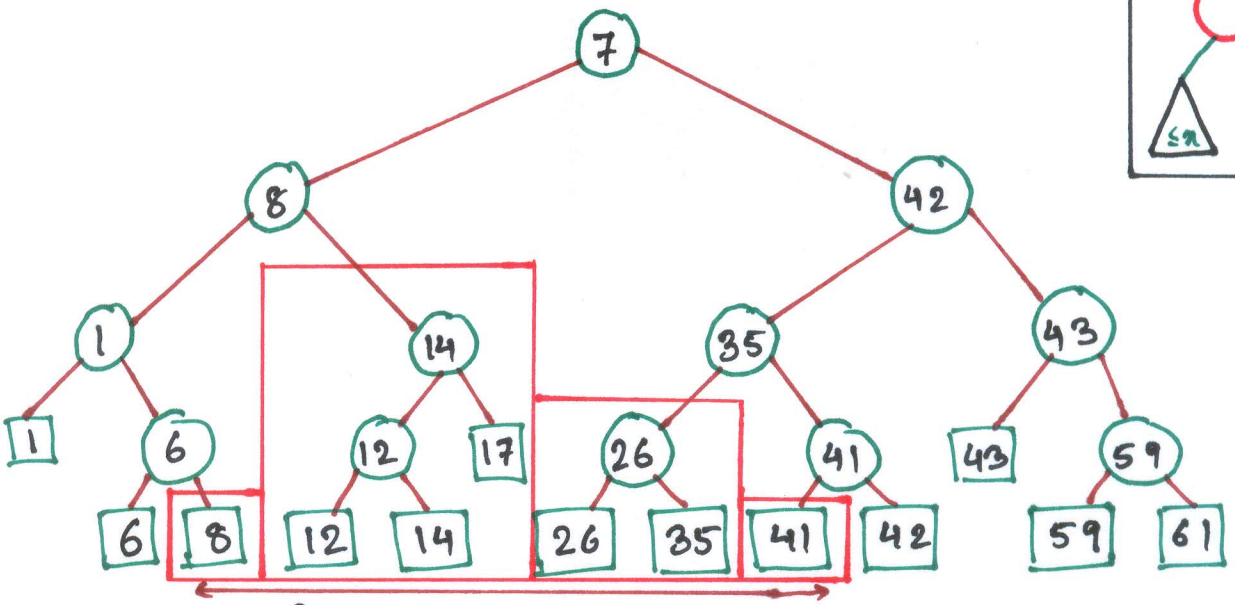
- Balanced binary search tree
  - New organization principle:
    - store points in the leaves of the tree
  - Internal nodes store copies of the leaves to satisfy binary search property.
    - Node x stores in $key[x]$ the maximum key of any leaf in the left subtree of x.

Example of a 1D range tree



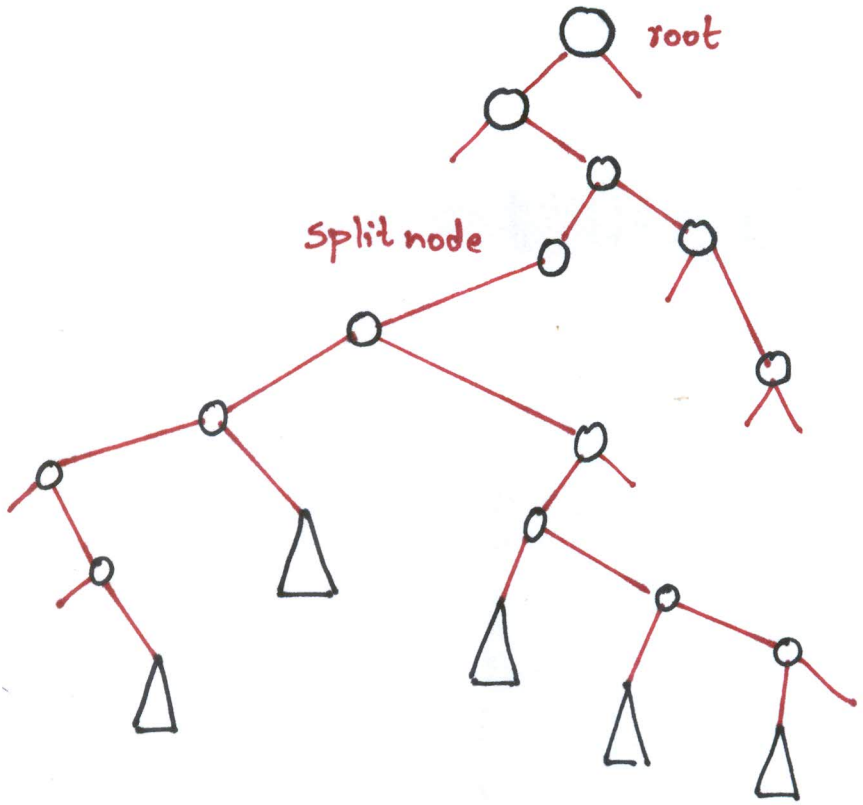






RANGE-QUERY ([7,41])

General 1D range query



Pseudocode, part 1: Find the split node

```
1D-RANGE-QUERY (T, [x1, x2])
1.  w ← root [T]
2.  while w is not a leaf and (x2 ≤ key [w] or key[w] < x1)
3.        do if x2 ≤ key[w]
4.              then w ← left [w]
5.              else w ← right [w]
```

▶ w is now the split node

traverse left and right from 'w' and report relevant subtrees.

Pseudo code, part 2: Traverse left and right from Split Node.

```
1D-RANGE-QUERY (T, [x1, x2])
  [find the split node]
  ▶ w is now the split node
1. if w is leaf
2.    then output leaf w if x1 ≤ key[w] ≤ x2
3. else v ← left[w]          ▶ left traversal
4.      while v is not a leaf
5.          do if x1 ≤ key[v]
6.              then output the subtree rooted at right[v]
7.                   v ← left[v]
8.              else v ← right[v]
9. output the leaf v if x1 ≤ key[v] ≤ x2
  [symmetrically for right travelsal
```

Analysis of 1D-range query

Query Time: Answer to range query represented by $O(\log n)$ subtrees found in $O(\log n)$ time.

Thus

- Can test for points in interval in $O(\log n)$ time
- Can count points in interval in $O(\log n)$ time if we augment the tree with subtree sizes.
- Can report the first k points in $O(k+\lg n)$ time.

Space: $O(n)$

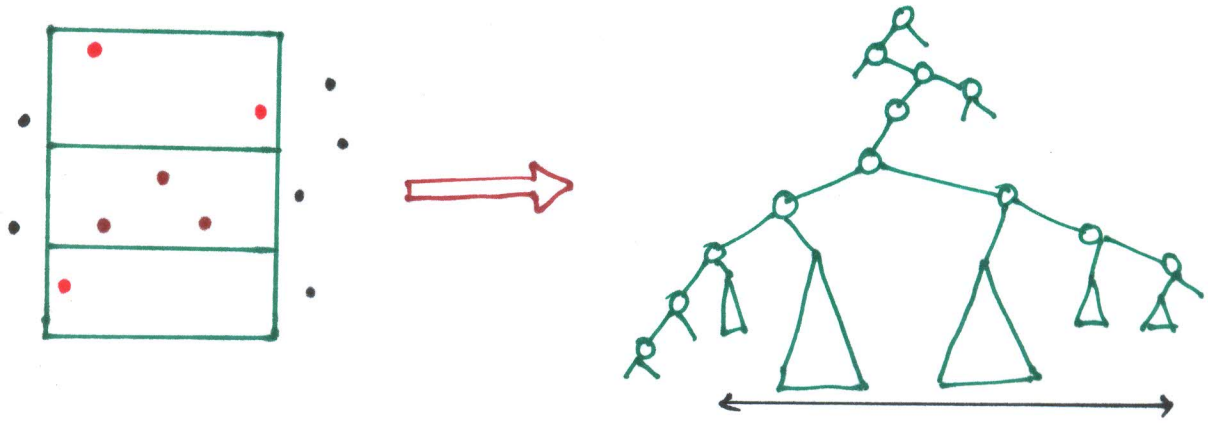
Preprocessing time: $O(n \log n)$

2D range trees

Store a primary 1D range tree for all the points based on x-coordinate

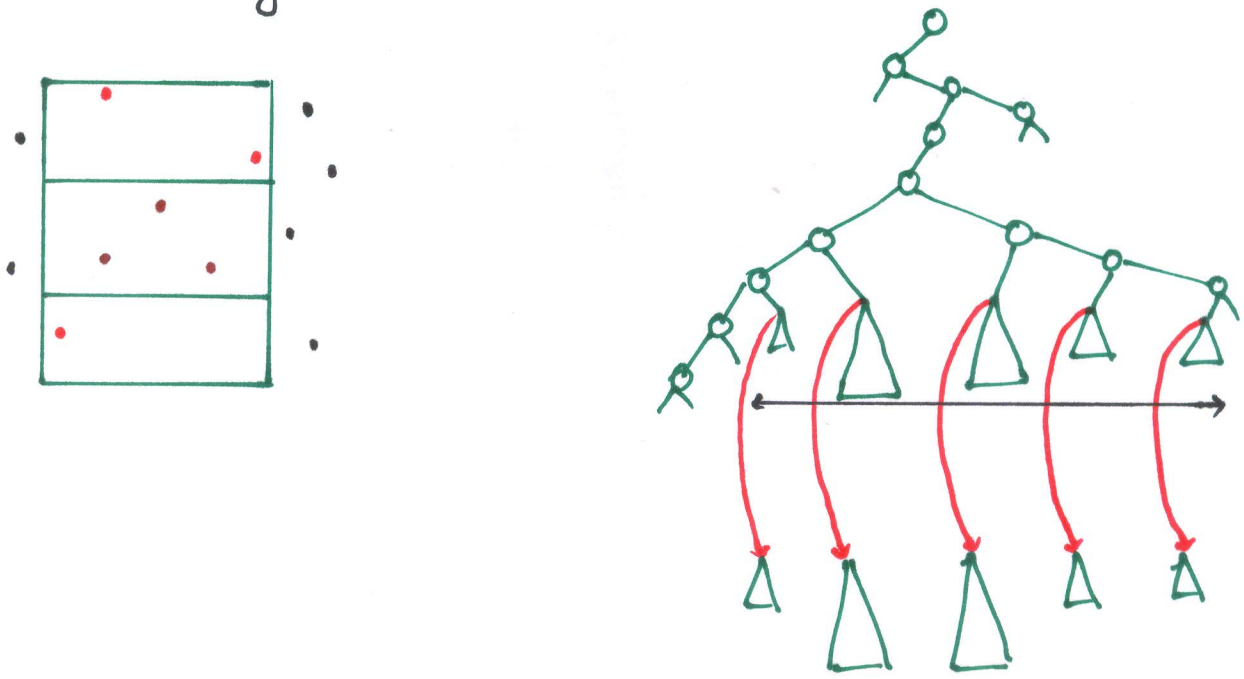
Thus in $O(\log n)$ time we can find $O(\log n)$ subtrees representing the points with proper x-cordinates.

How to restrict to points with proper y-co-ordinates?



Idea: In primary 1D range tree of x-coordinate, every node stores a secondary 1D range tree based on y-co-ordinate for all points in the subtree of the node

Recursively search within each.



Analysis of 2D range tree

Query time: In $O(\lg^2 n) = O((\lg n)^2)$ time, we can represent answer to range query by $O(\lg^2 n)$ subtrees.
Total cost for reporting K points: $O(K+(\lg n)^2)$.

Space: The secondary trees at each level of the primary tree together store a copy of the points. Also, each point is present in each secondary tree along the path from the leaf to the root.
Either way, we obtain that space is $O(n \log n)$.

Preprocessing time: $O(n \log n)$

d-dimensional range trees

Each node of secondary y-structure stores a tertiary z-structure representing the points in the subtree rooted at the node, etc.

Query time: $O(K + \log^d n)$ to report K points
Space: $O(n \log^{d-1} n)$
Preprocessing time: $O(n \log^{d-1} n)$

Best Data Structure to date:

Query time: $O(K + \log^{d-1} n)$ to report K points.

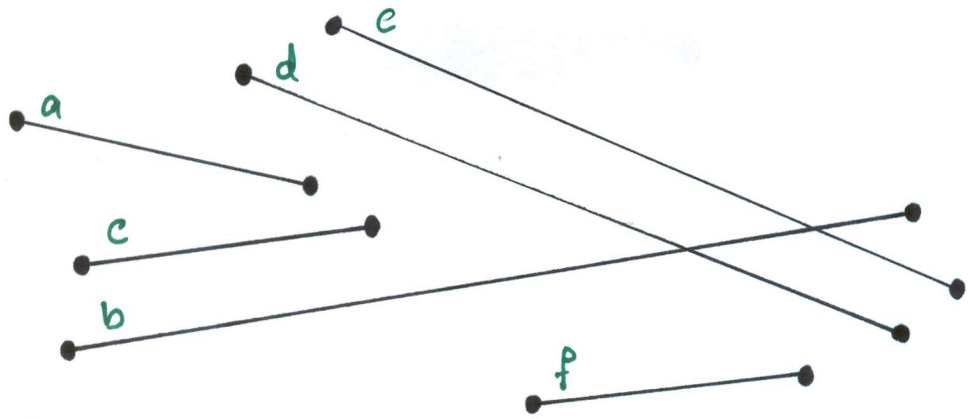
Space: $O(n(\log n / \log\log n)^{d-1})$

Preprocessing time: $O(n \log^{d-1} n)$

Line-segment intersection

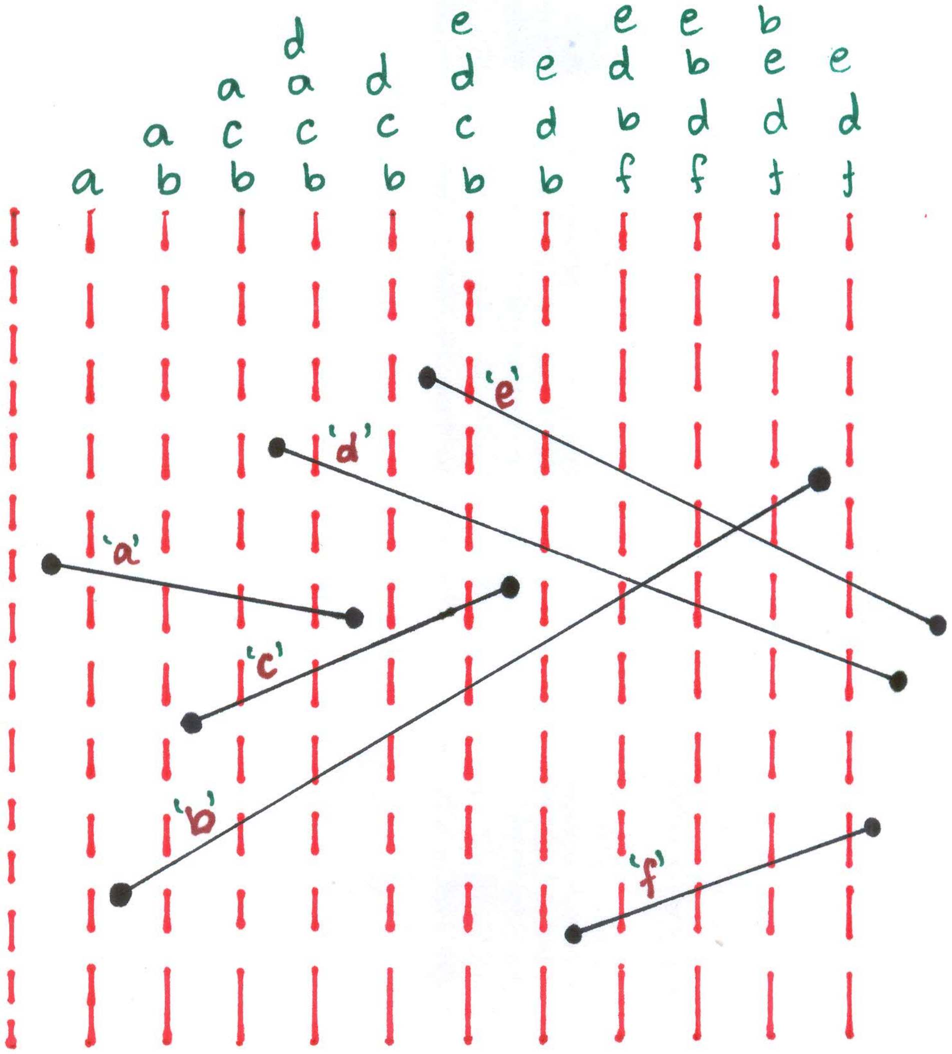
Given n line segments, does any pair intersect?

Obvious algorithm: $O(n^2)$



Sweep-line algorithm

- Sweep a vertical line from left to right
  (conceptually replacing x-cordinate with time)
- Maintain dynamic set S of segments that intersect the sweep line, ordered (tentatively) by y-co-ordinate of intersection.
- Order changes when
  - new segment is encountered, } segment endpoints
  - existing segment finishes, or } segment endpoints
  - two segments cross
- Key eventpoints are therefore segment endpoints.


a
a b
a c b
d a c b
d c b
e d c b
e d b
e d b f
e b d f
b e d f
e d f
'a'
'b'
'c'
'd'
'e'
'f'

Sweep-line algorithm

Process event points in order by sorting segment endpoints by x-coordinates and looping through:

- For a left endpoint of segment s:
  - Add segment s to dynamic set S
  - Check for intersection between s and its neighbours in S
- For a right endpoint of segment s:
  - Remove segment s from dynamic set S
  - Check for intersection between s and the neighbours of s in S.

Analysis

Use red black tree to store dynamic set S.

Total running time: $O(n \log n)$

Correctness

Theorem: If there is an intersection, the algorithm finds it.

Proof: Let X be the leftmost intersection point.

Assume for simplicity that

- only two segments s_1, s_2 pass through X, and
- no two points have the same x-co-ordinates.

At some point before we reach X, s_1 and s_2 become consecutive in order of S.

Either initially consecutive when s_1 and s_2 inserted or became consecutive when another deleted.

Week 8: Lecture Notes

Topics: Dynamic programming
Longest Common Subsequence
Graphs
Prim's Algorithms
Graph Search.

Dynamic Programming

Design technique:

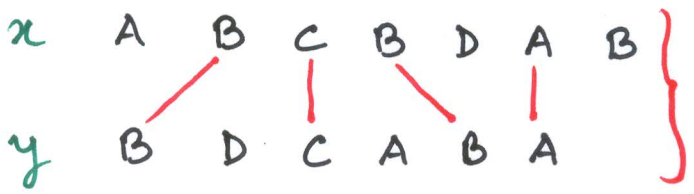
Like divide-and-conquer

Example:

Longest Common Subsequence (LCS)

Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$, find a longest subsequence common to them both.

'a' not 'the'



BCBA = LCS(x,y)

functional notation
but not a function

Brute-Force LCS Algorithm

Check every subsequence of $x[1 \ldots m]$ to see if it is also a subsequence of $y[1 \ldots n]$.

Analysis:

- Checking = $O(n)$ time per subsequence
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst case running time = $O(n 2^m)$
= exponential time

Towards a better algorithm

Simplification:

1. Look at the length of the longest-common subsequence.
2. Extend the algorithm to find the LCS of sequence s.

Notation:

- Denote the length of a sequence s by $|s|$

Strategy:

- Consider prefixes of x and y.
- Define $c[i,j] = |LCS(x[1 \ldots i], y[1 \ldots j])|$
- Then $c[m,n] = |LCS(x,y)|$.

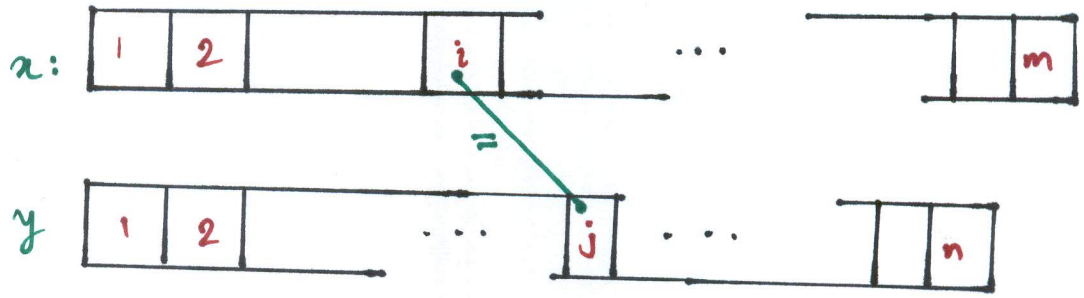
Recursive Formulation

Theorem:

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j] \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

Proof:

Case $x[i] = y[j]$:



Let $z[1 \ldots k] = LCS(x[1 \ldots i], y[1 \ldots j])$, where $c[i,j] = k$.

Then, $z[k] = x[i]$, or else z could be extended.

Thus, $z[i \ldots k-1]$ is CS of $x[1 \ldots i-1]$ and $y[1 \ldots j-1]$

Claim: $z[1 \ldots k-1] = LCS(x[1 \ldots i-1], y[1 \ldots j-1])$.

Suppose w is longer CS of $x[1 \ldots i-1]$ and $y[1 \ldots j-1]$, that is $|w| > k-1$.

Then cut and paste: $w \| z[k]$ (w concatenated with $z[k]$) is a common subsequence of x and y with

$$|w \| z[k]| > k.$$

Contradiction, proving claim.

Thus $c[i-1, j-1] = k-1$, which implies that $c[i,j] = c[i-1, j-1] + 1$.

Other cases are similar.

Dynamic Programming hallmark #1

Optimal Substructure:

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

- If $z = LCS(x,y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

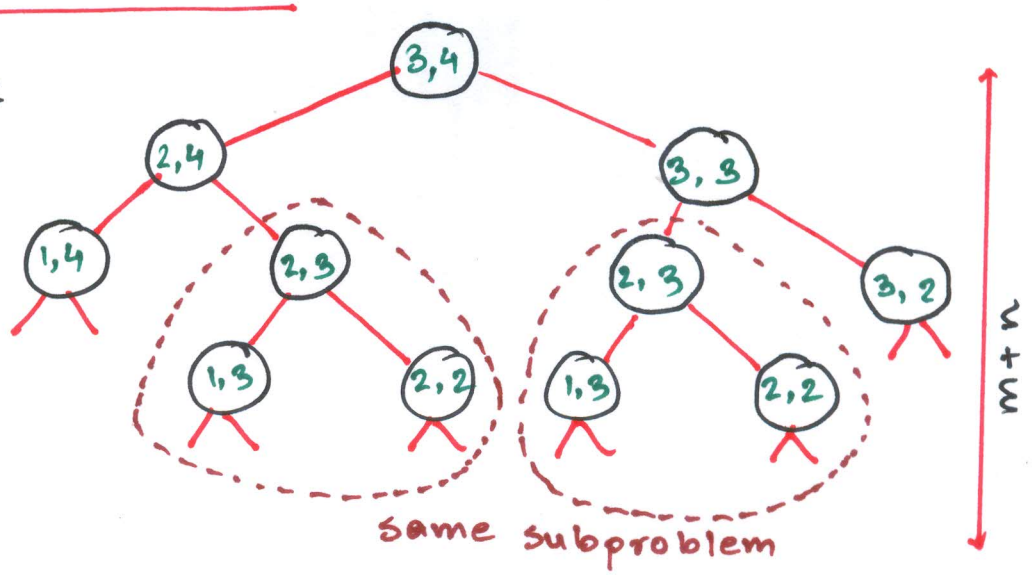
Recursive Algorithm for LCS

```
LCS (x, y, i, j)
1. if x[i] = y[j]
2.     then c[i,j] ← LCS(x,y,i-1,j-1) + 1
3. else c[i,j] ← max{ LCS(x,y,i-1,j), LCS(x,y,i,j-1) }
```

Worst case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion Tree

$m = 3$
$n = 4$:



Height = $m+n$ ⇒ work potentially exponential, but we are solving subproblems already solved.

Dynamic Programing hallmark #2

Overlapping subproblems:

A recursive solution contains a "small" number of distinct subproblems repeated many times.

- The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

Memoization Algorithm

Memoization:

After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

```
LCS(x,y,i,j)
    if c[i,j] = NIL
       then if x[i] = y[j]
               then c[i,j] ← LCS(x,y,i-1,j-1) + 1
               else c[i,j] ← {LCS(x,y,i-1,j),
                              LCS(x,y,i,j-1)}
```

} same as before

Time: $\Theta(mn)$ = constant work per table entry

Space: $\Theta(mn)$

Graphs (review)

Definition

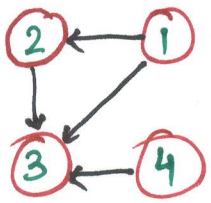
- A directed graph (digraph), $G = (V, E)$ is an ordered pair consisting of:
  - a set V of vertices (singular vertex)
  - a set $E \subseteq V \times V$ of edges.
- In an undirected graph, $G = (V, E)$, the edge set E consists of unordered pair of vertices.
- In either case, we have $|E| = O(V^2)$
- Moreover if G is connected, then $|E| \geq |V| - 1$, which implies that $\log |E| = \Theta(\log V)$

Adjacency matrix representation

The adjacency matrix of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \ldots n, 1 \ldots n]$ given by

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases}$$

Example.



A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

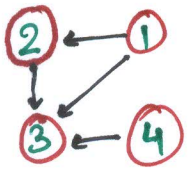
$O(V^2)$ storage

$\Rightarrow$ dense representation.

Adjacency-list representation

An adjacency list of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.

Example:



$Adj[1] = \{2, 3\}$

$Adj[2] = \{3\}$

$Adj[3] = \{\}$

$Adj[4] = \{3\}$

For undirected graphs:

$|Adj[v]| = \text{degree}(v)$

For digraphs, $|Adj[v]| = \text{out-degree}(v)$

Handshaking Lemma:

$\sum_{v \in V} = 2|E|$ for undirected graphs

$\Rightarrow$ adjacency list • use $\Theta(V+E)$ storage

(a sparse representation for either type of graph)

Minimum Spanning Tree (MST)

Input: A connected, undirected graph $G = (V, E)$ with weight function $w: E \to R$.

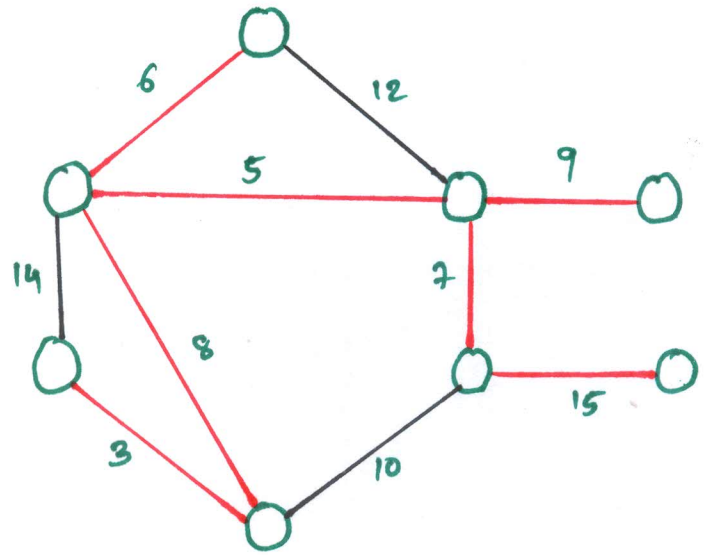
- for simplicity, assume that all edge weights are distinct.

Output:

A spanning tree T - a tree that connects all vertices - of minimum weight.

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

Example of MST

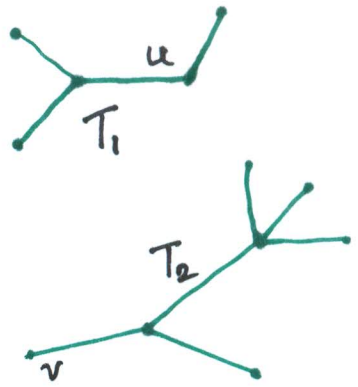


Optimal Substructure

Consider MST T of graph G.

Remove any edge $(u,v) \in T$.

Then T is partitioned into two subtrees T_1 and T_2. (other edges of G are not shown)



Theorem:

The subtree T_1 is an MST of $G_1 = (V_1, E_1)$, the graph induced by the vertices of T_1:

$$V_1 = \text{vertices of } T_1$$
$$E_1 = \{(x,y) \in E : x,y \in V_1\}$$

Similarly for T_2.

Proof:

Cut and paste:

$$w(T) = w(u,v) + w(T_1) + w(T_2)$$

If T_1' were a lower-weight spanning tree than T_1 for G_1, then $T' = \{(u,v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G.

Do we also have overlapping subproblems?

- Yes
  - Great, then dynamic programming may work
- Yes, but MST exhibits another powerful property which leads to an even more efficient algorithm.

Hallmark for Greedy Algorithm

Greedy Choice Property:

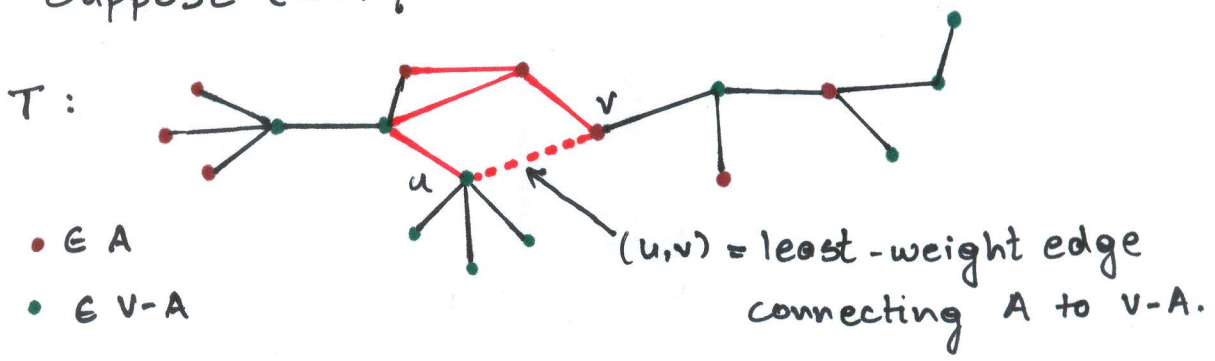
A locally optimal choice is globally optimal

Theorem:

Let T be the MST of $G=(V,E)$, and let $A \subseteq V$. Suppose that $(u,v) \in E$ is the least-weight edge connecting A to $V-A$. Then $(u,v) \in T$.

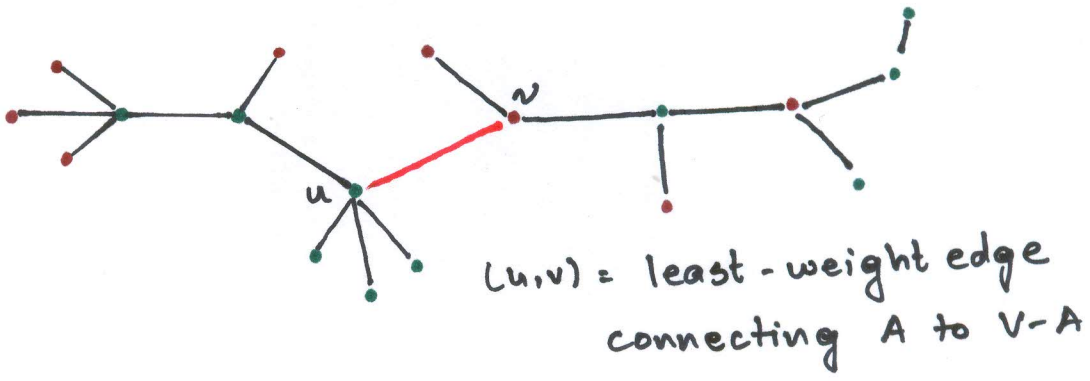
Proof:

Suppose $(u,v) \notin T$.



Consider the unique simple path from u to v in T

Swap (u,v) with the first edge on this path that connects a vertex in A to a vertex in $V-A$.



A lighter-weight spanning tree than T results.

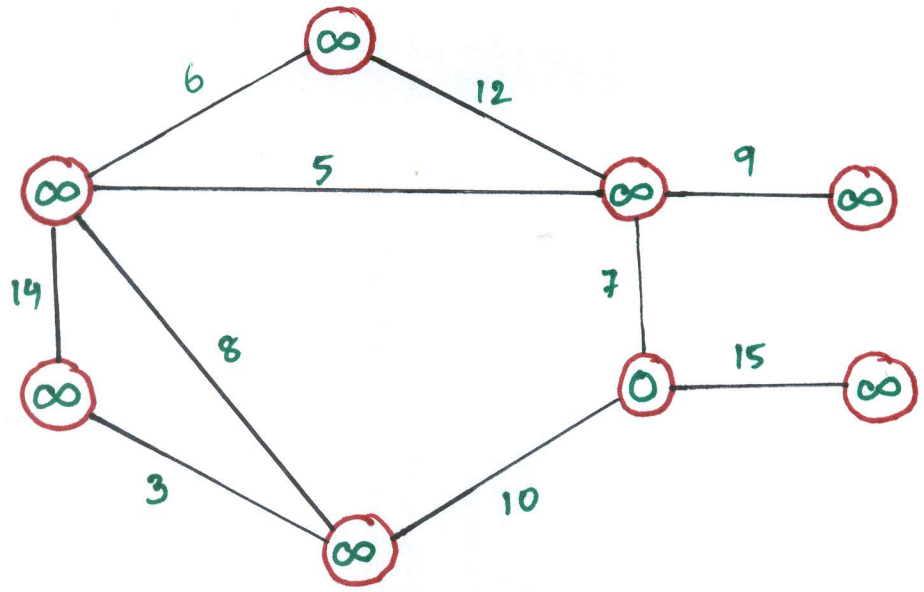
Prim's Algorithm

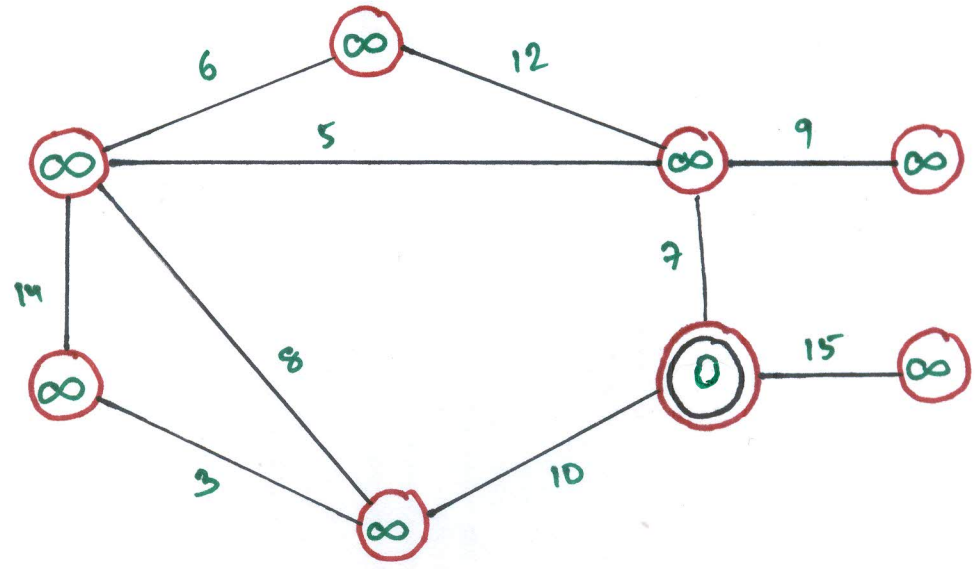
Idea: Maintain $V - A$ as a priority queue Q. Key each vertex in Q with the weight of least-weight edge connecting it to a vector in A.

```
1.  Q ← V
2.  key[v] ← ∞ for all v ∈ V
3.  key [s] ← 0 for some arbitrary s ∈ V
4.  while Q ≠ ϕ
5.        do u ← EXTRACT-MIN (Q)
6.           for each v ∈ Adj [u]
7.              do if v ∈ Q and w(u,v) < key [u]
8.                 then key [v] ← w(u,v)
9.                      π [v] ← u
```

At the end $\{v, \pi[v]\}$ forms the MST.

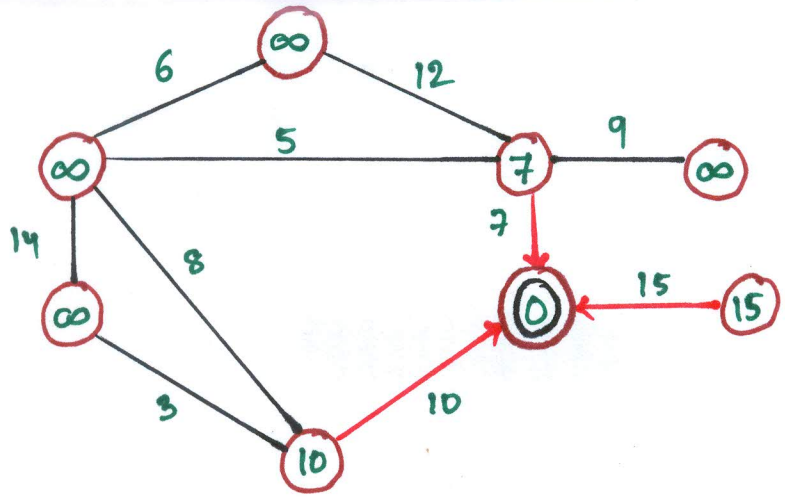
Example of Prim's Algorithm

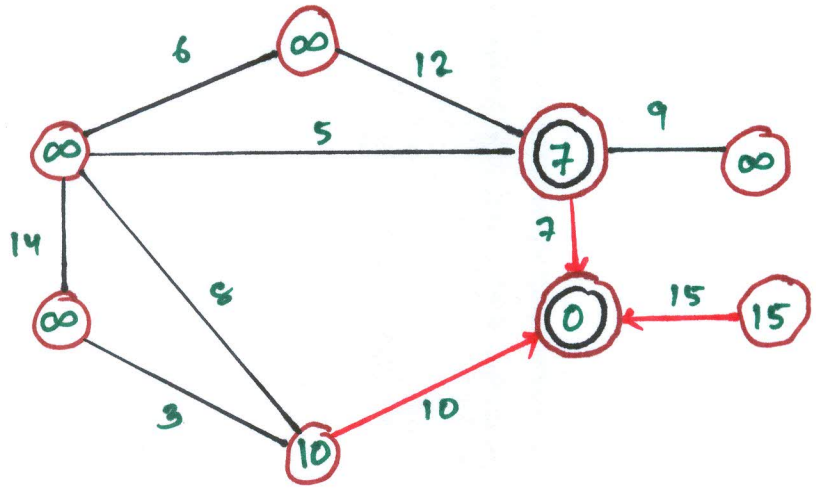


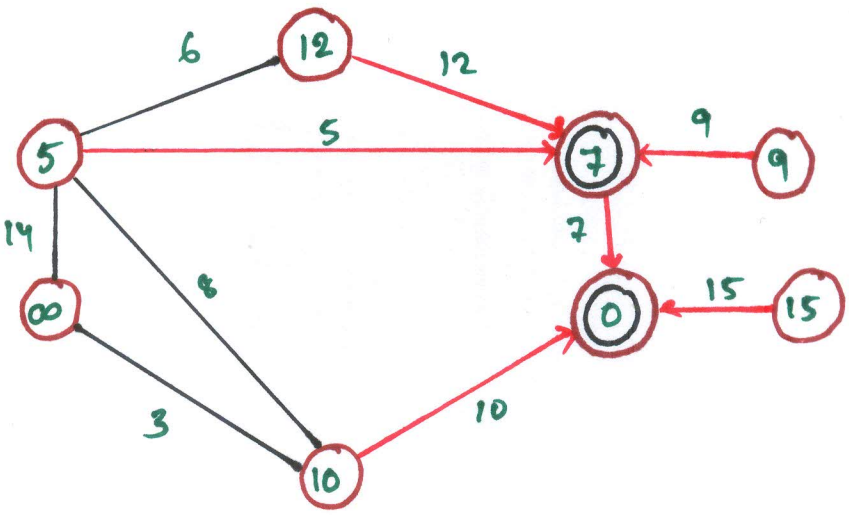


 $\in A$

 $\in V-A$

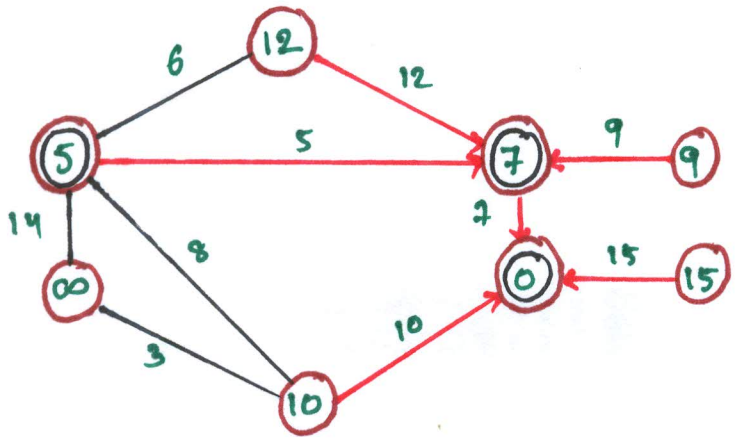

∞
6
12
∞
5
7
9
∞
14
8
7
∞
0
15
15
3
10
10

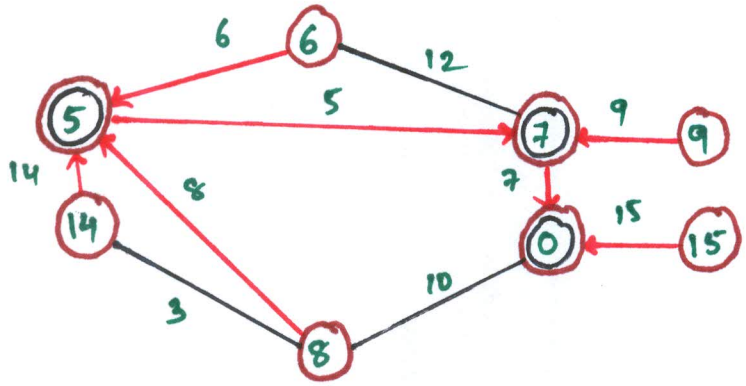

∞
6
12
∞
5
7
9
∞
14
8
7
∞
0
15
15
3
10
10

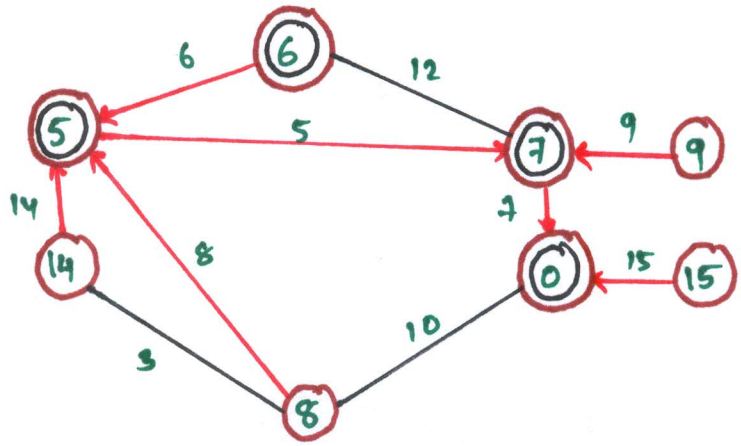

12
6
12
5
5
7
9
9
14
8
7
∞
0
15
15
3
10
10

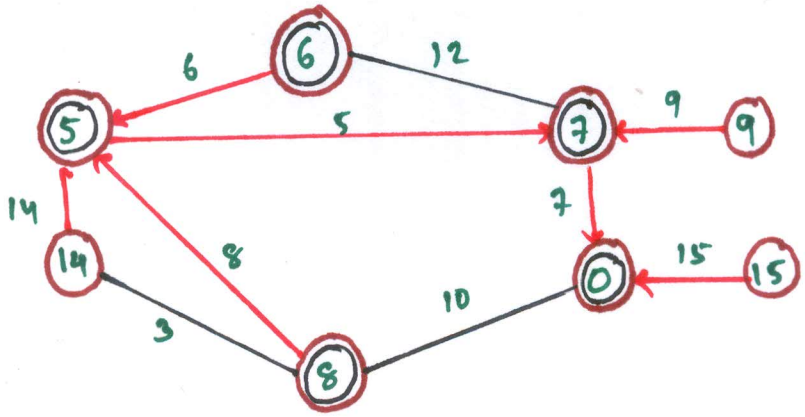
$\in A$

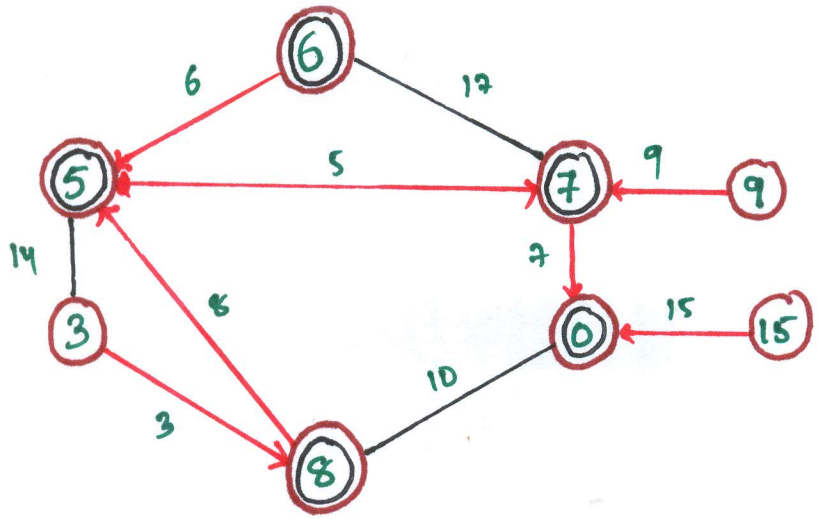
$\in V-A$

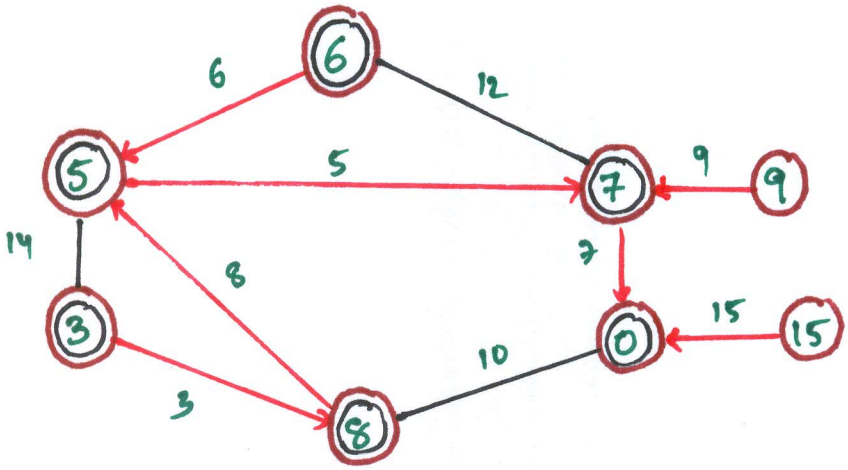

12
6
12
5
5
7
9
9
14
7
15
15
8
00
0
10
3
10

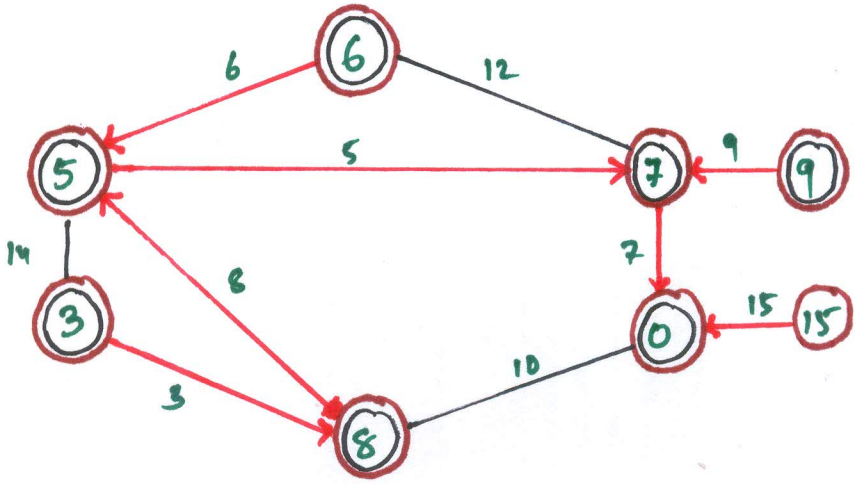

6
6
12
5
5
7
9
9
14
7
8
15
15
14
0
10
3
8

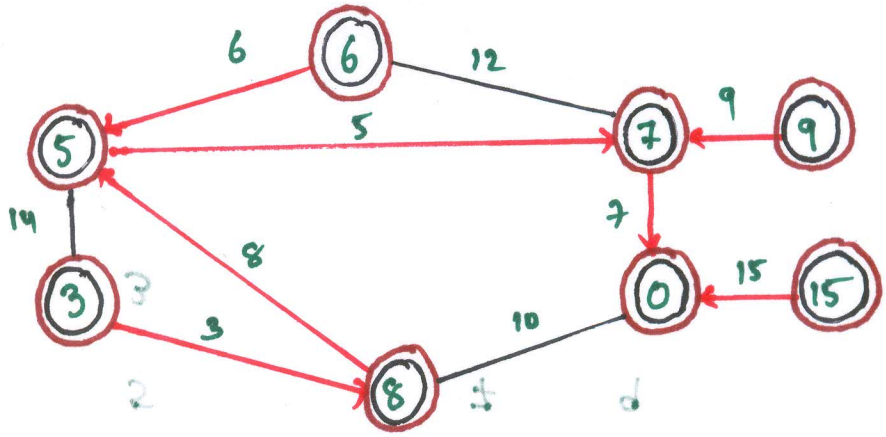

6
6
12
5
5
7
9
9
14
7
8
15
15
14
0
10
3
8


6
6
12
5
5
7
9
9
14
7
8
15
15
14
0
10
3
8


6
6
17
5
5
7
9
9
14
7
8
3
0
15
15
10
3
8


6
6
12
5
5
7
9
9
14
7
8
3
0
15
15
10
3
8


6
6
12
5
5
7
9
9
14
7
8
3
0
15
15
10
3
8


6
6
12
5
5
7
9
9
14
7
8
3
0
15
15
10
3
8

Analysis of Prim

$O(V)$ total:
- $Q \leftarrow V$
- $key[v] \leftarrow \infty$ for all $v \in V$
- $key[s] \leftarrow 0$ for some arbitrary $s \in V$

$|V|$ times:
- while $Q \neq \phi$
- do $u \leftarrow$ EXTRACT-MIN (Q)

degree (u) times:
- for each $v \in Adj[u]$
- do if $v \in Q$ and $w(u,v) < key[v]$
- then $key[v] \leftarrow w(u,v)$
- $\pi[v] \leftarrow u$

Handshaking lemma $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's

Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\log V)$	$O(\log V)$	$O(E \log V)$
Fibonacci heap	$O(\log V)$ amortized	$O(1)$ amortized	$O(E + V \log V)$ worst case

MST Algorithms

Kruskal's algorithm:

- Uses the disjoint-set data structure
- Running time = $O(E \log V)$

Best to date:

- Karger, Klein, and Tarjan [1993]
- Randomized algorithm
- $O(V+E)$ expected time.

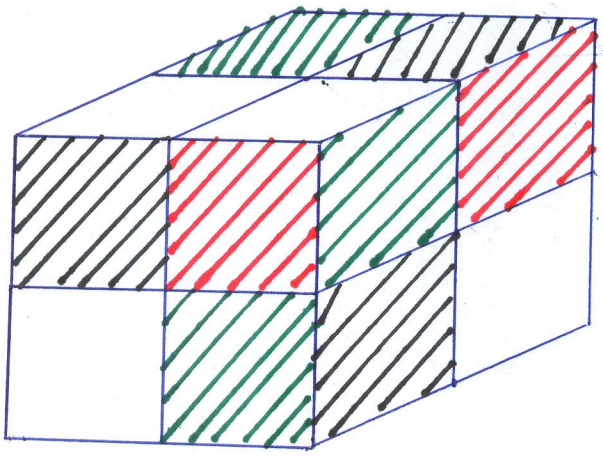
Graph Search

"Explore a graph", e.g.:

- find a path from start vertex s to a desired vertex
- visit all vertices or edges of graph, or only those reachable from s.

Pocket Cube

Consider a 2×2×2 Rubik's cube

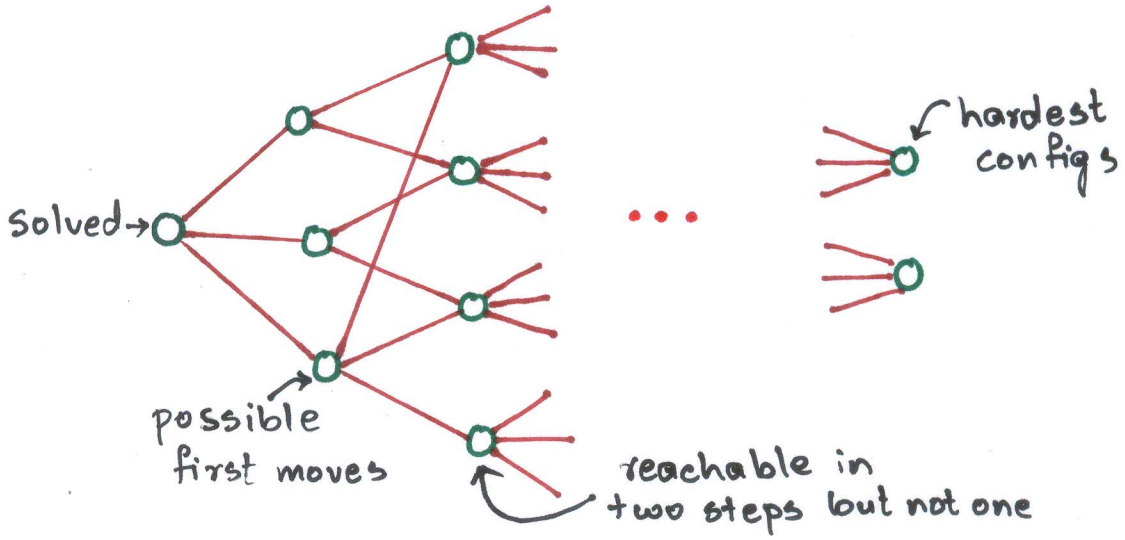


Configuration graph:

vertex for each possible state

edge for each basic move (e.g. 90° turn) from one state to another.

undirected: moves are reversible.

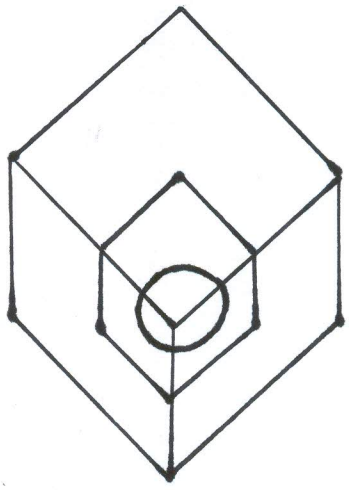


Diameter ("God's Number")

- 11 for 2×2×2
- 20 for 3×3×3
- $\Theta(n^2/\log n)$ for n×n×n [Demaine, Demaine, Eisenstat, Lubiw Winslow 2011]

- Number of vertices = $8! \cdot 3^8 = 264,539,520$

  where 8! comes from having 8 cubelets in abitrary positions and 3^8 comes ~~from~~ as each cubelethas 3 possible twists.



This can be divided by 24 if we remove cube symmetries and further divided by 3 to account for actually reachable configurations (there are 3 connected components).

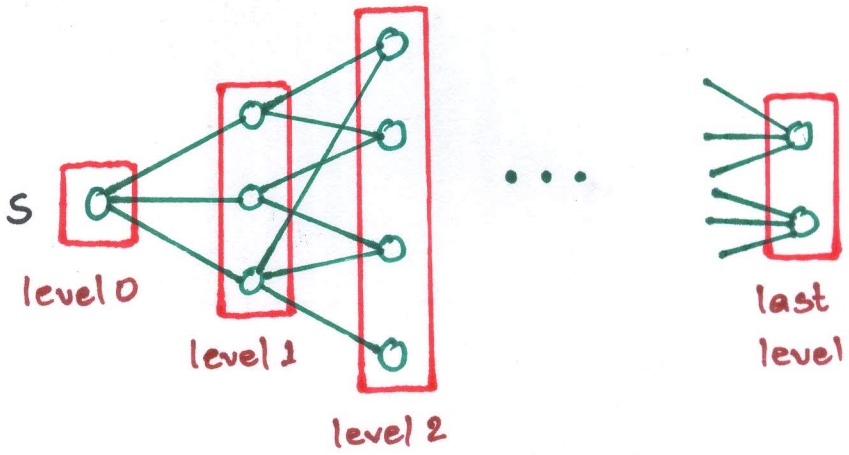
Week 9: Lecture Notes

Topics: BFS and DFS
Shortest path problem
Dijktra's Algorithm
Example of Dijktra
Bellman Ford

Breadth-First Search

Explore graph level by level from s (start vertex)

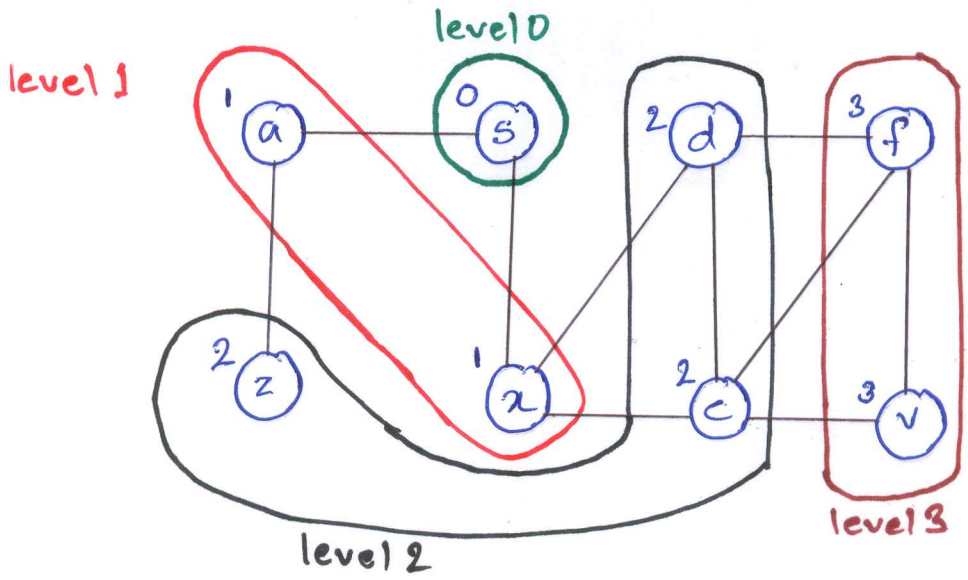
- level 0 = {s}
- level i = vertices reachable by path of i edges but not fewer.
- build level i>0 from i-1 by trying all outgoing edges, but ignoring vertices from previous levels



Breadth-First-Search Algorithm

```
BFS(V, Adj, s):
1.  level = {s: 0}
2.  parent = {s: None}
3.  i = 1
4.  frontier = [s]
5.  while frontier
6.      next = []
7.      for u in frontier:
8.          for v in Adj[u]:
9.              if u not in level:
10.                 level[v] = i
11.                 parent[v] = u
12.                 next.append(v)
13.     frontier = next
14.     i = i+1
```

Example



$frontier_0 = \{s\}$

$frontier_1 = \{a, x\}$

$frontier_2 = \{z, d, c\}$

$frontier_3 = \{f, v\}$ (not x, c, d)

Analysis

- vertex v enters next (and then frontier) only once
  (because level[s] then set)
  base case: $v = s$
- $\Rightarrow$ Adj[v] looped through only once.

$$\text{time} = \sum_{v \in V} |\text{Adj}[v]| = \begin{cases} |E| & \text{for directed graphs} \\ 2|E| & \text{for undirected graphs} \end{cases}$$

- $\Rightarrow O(E)$ time
- $O(V+E)$ ("linear time") to also list vertices unreachable from v (those still not assigned level)

Shortest Paths

- for every vertex v, fewest edges to get from s to v is

$$\begin{cases} \text{level}[v] & \text{if } v \text{ is assigned level} \\ \infty & \text{else (no path)} \end{cases}$$

- parent pointers from shortest-path tree
  = union of such shortest path for each v.

$\Rightarrow$ to find shortest path, take v, parent[v], parent[parent[v]], etc., until s (or None)

Depth - First Search (DFS)

This is like exploring a maze.

- follow a path until you get stuck.
- backtrack along breadcrumbs until reach unexplored neighbour
- recursively explore
- careful not to repeat a vertex.

Depth-First-Search Algorithm

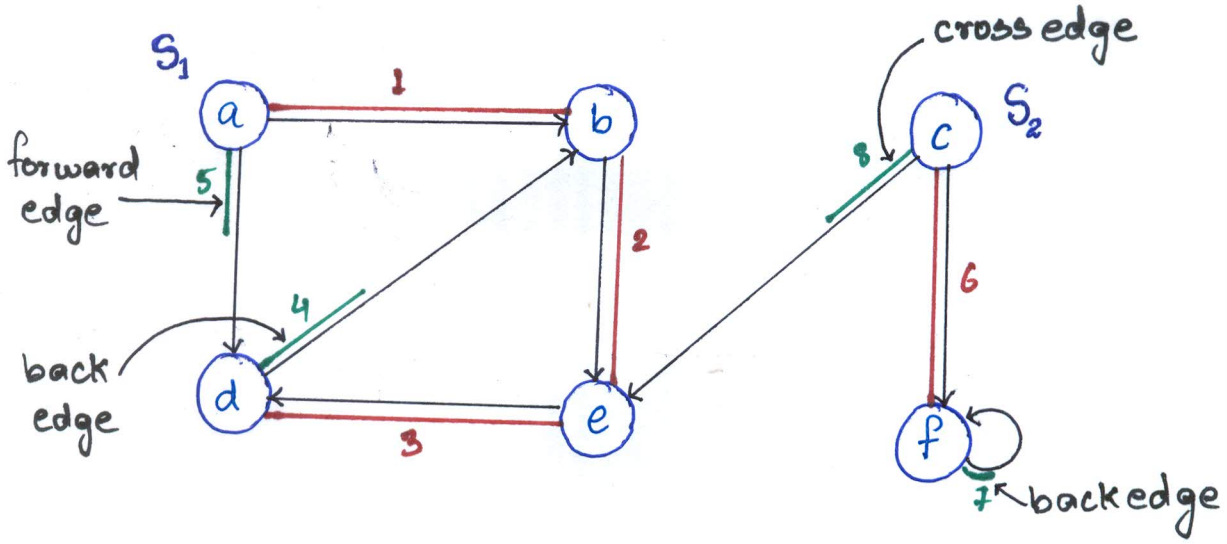
```
1.   parent = {s: None}
2.   DFS-visit (V, Adj, s):
3.       for v in Adj[s]:
4.           if v not in parent:
5.               parent[v] = s
6.               DFS-visit (V, Adj, v)
7.   DFS (V, Adj)
8.       parent = {}
9.       for s in V:
10.          if s not in parent:
11.              parent[s] = None
12.              DFS-visit (V, Adj, s)
```

search from start vertex s (only see stuff reachable from s)

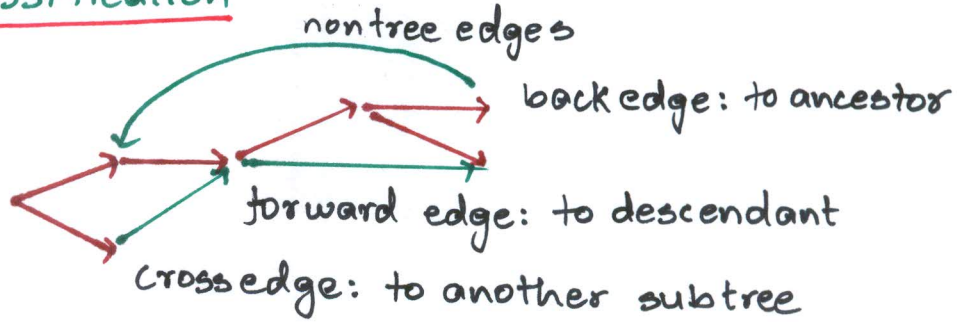
explore entire graph

(could do same to extend BFS)

Example of DFS



Edge Classification



- to compute this classification (back or not), mark nodes for duration they are "on the stack"
- only tree and back edges in undirected graph.

Analysis of DFS

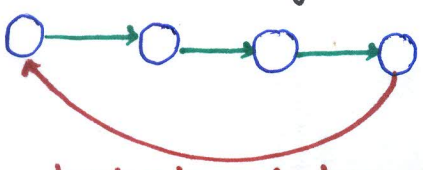
- DFS-visit gets called with a vertex s only once
  $\Rightarrow$ time in DFS-visit = $\sum_{s \in V} |Adj[s]| = O(E)$
- DFS outer loop adds just $O(V)$
  $\Rightarrow$ $O(V+E)$ time (linear time)

Cycle Detection

Graph G has a cycle $\Leftrightarrow$ DFS has a back edge

Proof:

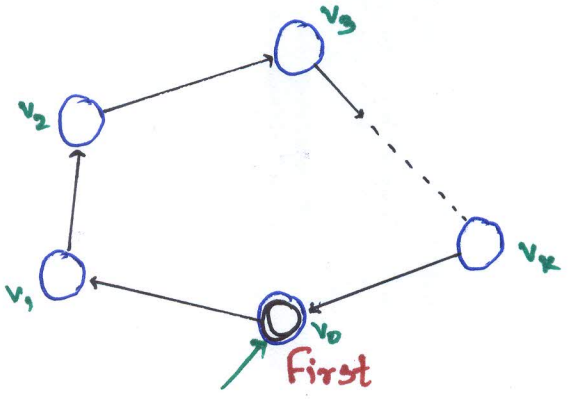
($\Leftarrow$) tree edges



is a cycle

back edge: to tree ancestor

($\Rightarrow$) consider first visit to cycle:



- before visit to v_i finishes,
  will visit v_{i+1} (and finish):
  will consider edge (v_i, v_{i+1})
  $\Rightarrow$ visit v_{i+1} now or already did

- $\Rightarrow$ before visit to v_0 finishes
  will visit v_k (and didn't before)

- $\Rightarrow$ before visit to v_k (or v_0) finishes
  will see (v_k, v_0) as back edge.

Job Scheduling

Given Directed Acyclic Graph (DAG), where vertices represents task and edges represent dependencies, order tasks without violating dependencies.

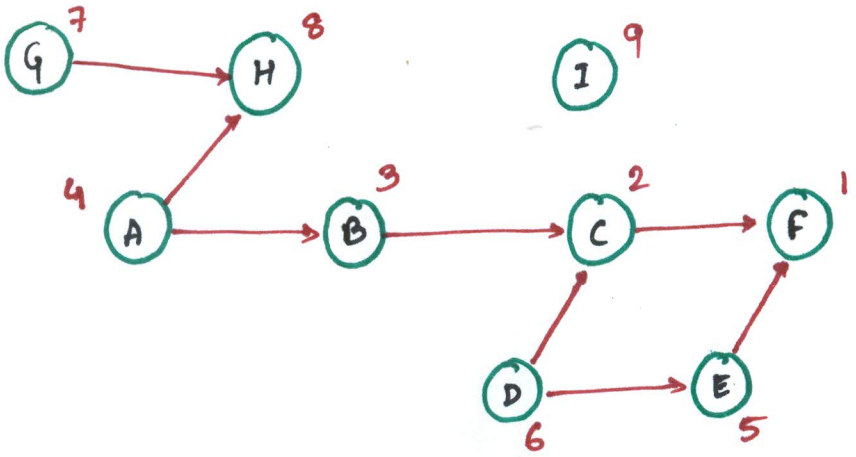


Fig: Dependence graph: DFS finishing times.

Source:

Source = vertex with no incoming edges
= scheduling at begining (A, G, I).

Attempt:

BFS from each source

- from A finds A, BH, C, F
- from D finds D, BE, CF ← slow... and wrong!.
- from G finds G, H
- from I finds I.

Topological Sort

Reverse of DFS finishing times
(time at which DFS-visit(v) finishes)

```
DFS-visit(v)
...
    order.append(v)
order.reverse()
```

Correctness

For any edge (u,v) - u ordered before v, i.e. v finished before u

- if u visited before v:
  - before visit to u finishes, will visit v (via (u,v) or otherwise)

    $\Rightarrow$ v finishes before u

if v visited before u

graph is cyclic

$\Rightarrow$ u cannot be reached from v

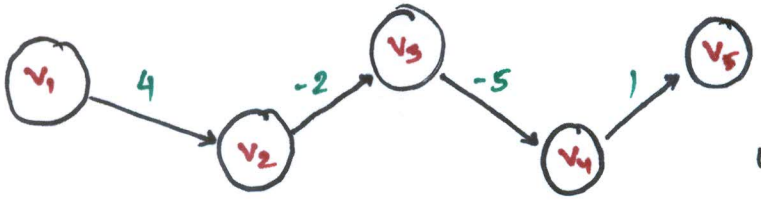
$\Rightarrow$ visit to v finishes before visiting u.

Paths in graphs

Consider a diagraph $G=(V, E)$ with edge-weight function $w: E \to R$. The weight of path $p = v_1 \to v_2 \to \cdots \to v_k$ is defined to be

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

Example:



$w(p) = -2$

Shortest Paths

A "shortest path" from u to v is a path of minimum weight from u to v.

The "shortest path weight" from u to v is defined as:

$$\delta(u,v) = \min \{w(p): p \text{ is a path from } u \text{ to } v\}$$

Note: $\delta(u,v) = \infty$ if no path from u to v exists.

Optimal Substructure

Theorem:

A subpath of a shortest path is a shortest path.

Triangle Inequality

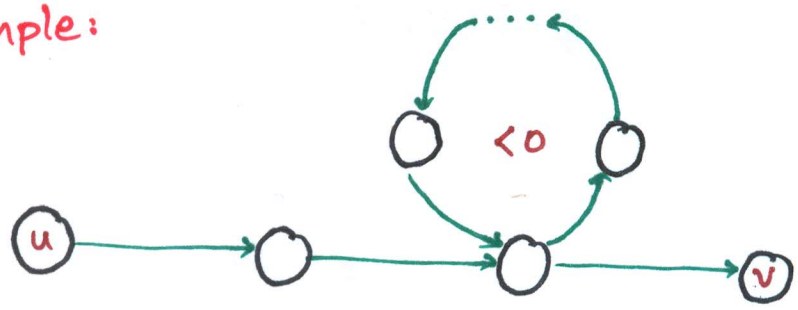
For all $u, v, x \in V$, we have

$$\delta(u,v) \leq \delta(u,x) + \delta(x,v)$$

Well-definedness of shortest paths

If a graph G contains a negative-weight cycle, then some shortest path may not exist.

Example:



Single-source shortest paths

Problem: From a given source vertex $s \in V$, find the shortest-path weights $\delta(u,v)$ for all $v \in V$

If all edge weights $w(u,v)$ are non-negative, all shortest-path weights must exist.

Idea: Greedy

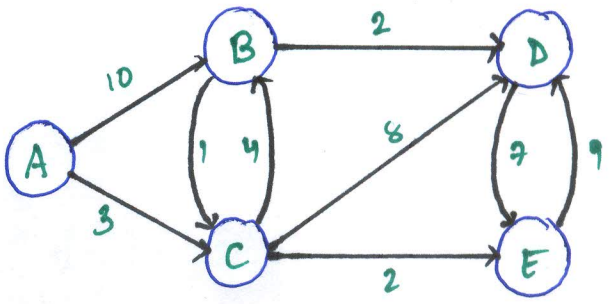
1. Maintain a set S of vertices whose shortest-path distance from s are known.
2. At each step add to S the vertex $v \in V-S$ whose distance estimate from s is minimal.
3. Update the distance estimates of vertices adjacent to v.

Dijkstra's Algorithm

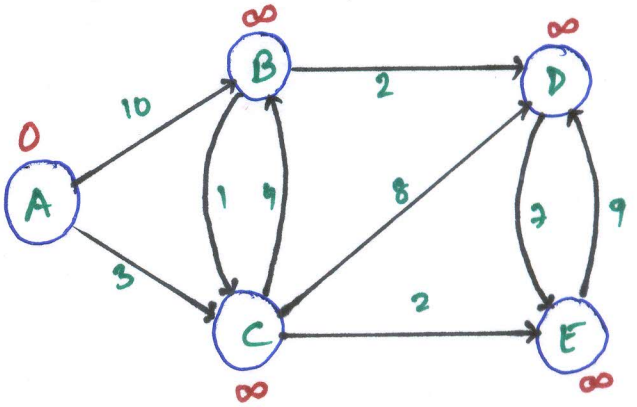
```
d[s] ← 0
for each v ∈ V-{s}
    do d[v] ← ∞
s ← ϕ
Q ← V        ▶ Q is a priority queue maintaining V-S
while Q ≠ ϕ
    do u ← EXTRACT-MIN(Q)
       s ← s ∪ {u}
       for each v ∈ Adj[u]
           do if d[v] > d[u] + w(u,v)        } relaxation
              then d[v] ← d[u] + w(u,v)       } step
                   ↑
                   Implicit DECREASE-KEY
```

Example of Dijkstra's Algorithm

Graph with non-negative edge weights

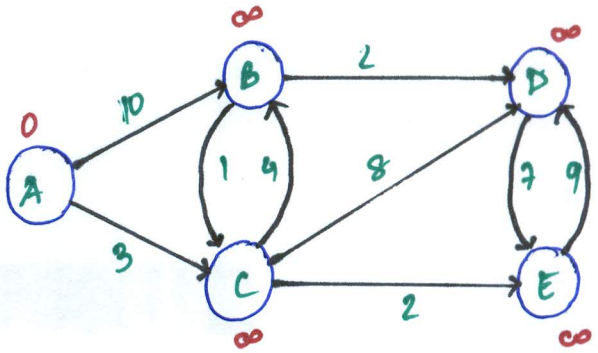


Initialize



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

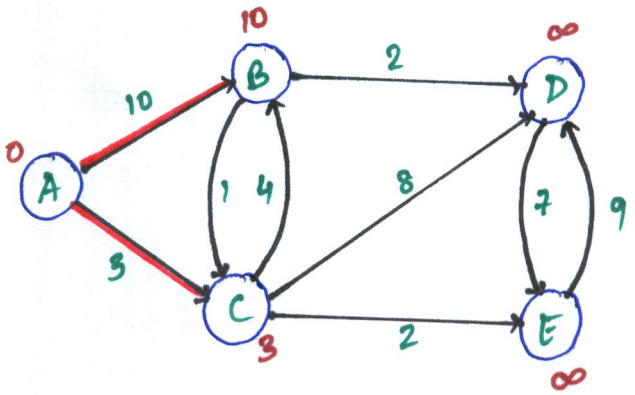
A ← EXTRACT-MIN(Q)



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {A}

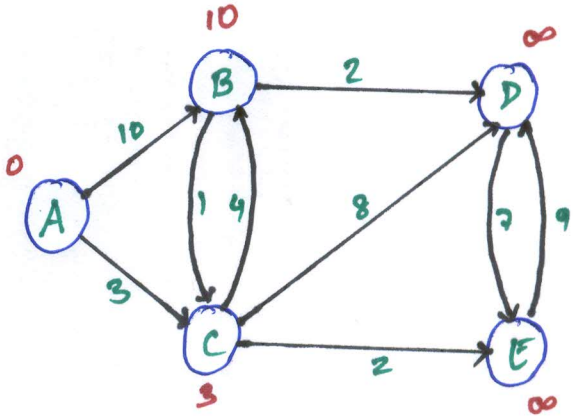
Relax all edges leaving A



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	-	-

S: {A}

C ← EXTRACT-MIN(Q)

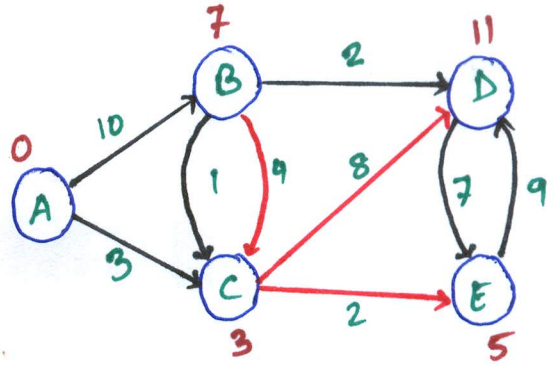


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	-	-

S: {A, C}

Relax all edges leaving C

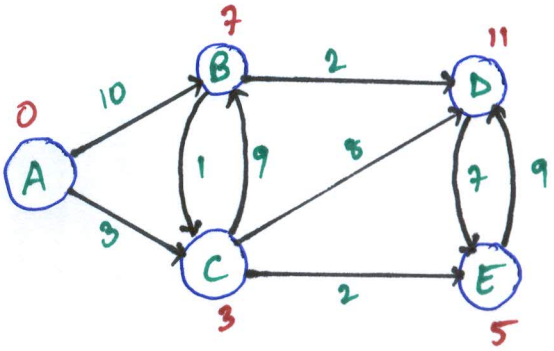
Q:	A	B	C	D	E
	[0]	∞	∞	∞	∞
		10	[3]	-	-
		7		11	5



S: {A, C}

E ← EXTRACT-MIN(Q)

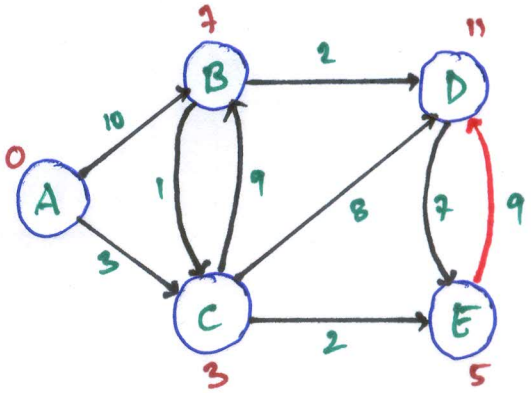
Q:	A	B	C	D	E
	[0]	∞	∞	∞	∞
		10	[3]	-	-
		7		11	[5]



S: {A, C, E}

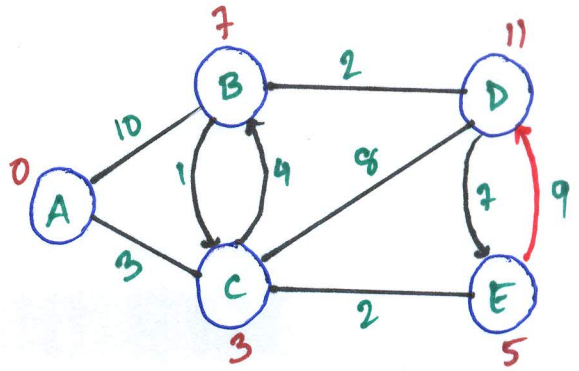
Relax all edges leaving E

Q	A	B	C	D	E
	[0]	∞	∞	∞	∞
		10	[3]	-	-
		7		11	[5]
		7		11	



S: {A, C, E}

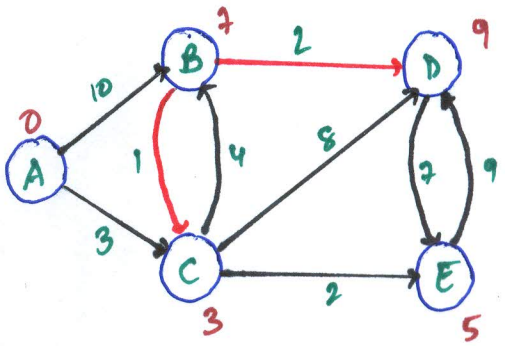
B ← EXTRACT-MIN(Q):



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: {A, C, E, B}

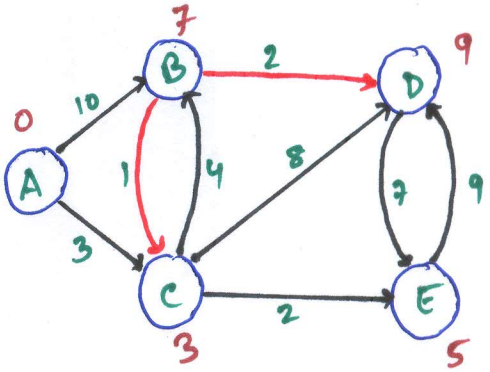
Relax all edges leaving B



Q:	A	B	C		E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: {A, C, E, B}

D ← EXTRACT-MIN(Q)



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: {A, C, E, B, D}

Correctness - Part I

Lemma: Initializing $d[s] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all $v \in V - \{s\}$ establishes $d[v] \geq \delta(s,v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps.

Proof:

Suppose not. Let v be the first vertex for which $d[v] < \delta(s,v)$, and let u be the vertex that caused $d[v]$ to change: $d[v] = d[u] + w(u,v)$. Then

$$\begin{aligned} d[v] &< \delta(s,v) && \text{supposition} \\ &\leq \delta(s,u) + \delta(u,v) && \text{triangle inequality} \\ &\leq \delta(s,u) + w(u,v) && \text{sh. path} \leq \text{specific path} \\ &\leq d[u] + w(u,v) && v \text{ is first violation.} \end{aligned}$$

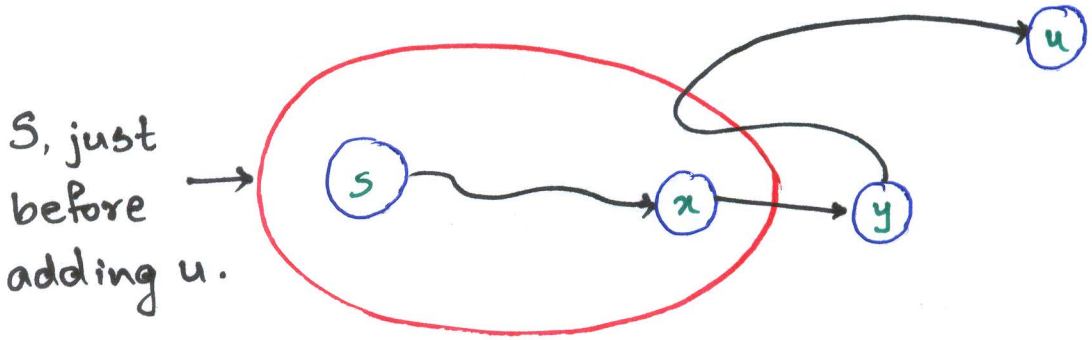
which is a contradiction.

Correctness - Part II

Theorem: Dijkstra's algorithm terminates with $d[v] = \delta(s,v)$ for all $v \in V$

Proof:

It suffices to show that $d[v] = \delta(s,v)$ for every $v \in V$ when v is added to S. Suppose u is the first vertex added to S for which $d[u] \neq \delta(s,u)$. Let y be the first vertex in $V - S$ along a shortest path from s to u, and x be it's predecessor:



Since u is the first vertex violating the claimed invariant, we have $d[x] = \delta(s,x)$. Since subpaths of shortest paths are shortest paths, it follows that $d[y]$ was set to $\delta(s,x) + w(x,y) = \delta(s,y)$ when (x,y) was relaxed just after x was added to S.

Consequently, we have $d[y] = \delta(x,y) \leq \delta(s,u) \leq d[u]$.

But $d[u] \leq d[y]$ by our choice of u, and hence

$$d[y] = \delta(x,y) = \delta(s,u) = d[u]$$

which is a contradiction.

Analysis of Dijkstra

```
while Q ≠ ∅
  do u ← EXTRACT-MIN(Q)
     S ← S ∪ {u}
     for each v ∈ Adj[u]
        do if d[v] > d[u] + w(u,v)
           then d[v] ← d[u] + w(u,v)
```

Handshaking Lemma

$\Rightarrow \Theta(E)$ implicit DECREASE-KEY's

Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Note: Same formula as in the analysis of Prim's minimum spanning tree algorithm.

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\log V)$	$O(\log V)$	$O(E \log V)$
Fibonacci heap	$O(\log V)$ amortized	$O(1)$ amortized	$O(E + V \log V)$ worst case

Negative Weight Cycle

Recall: If a graph $G = (V, E)$ contains a negative-weight cycle, then some shortest paths may not exist.

Bellman-Ford Algorithm:

Finds all shortest-path lengths from a source $s \in V$ to all $v \in V$ or determines that a negative cycle exists.

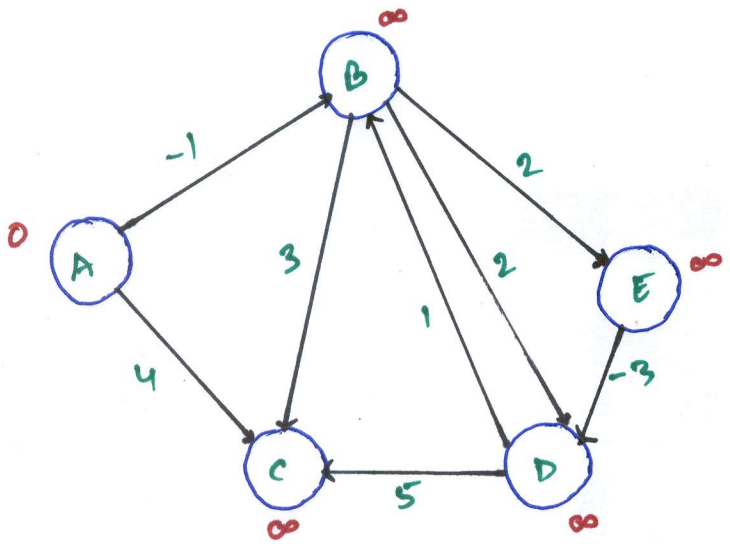
Bellman-Ford Algorithm

```
1.  d[s] ← 0                                          } initialization
2.  for each v ∈ V - {s}                              }
3.      do d[v] ← ∞                                   }
4.  for i ← 1 to |V| - 1
5.      do for each edge (u,v) ∈ E
6.          do if d[v] > d[u] + w[u,v]                } relaxation
7.              then d[v] ← d[u] + w(u,v)             } step
8.  for each edge (u,v) ∈ E
9.      do if d[v] > d[u] + w(u,v)
10.             then report that a negative-weight
                     cycle exists.
```

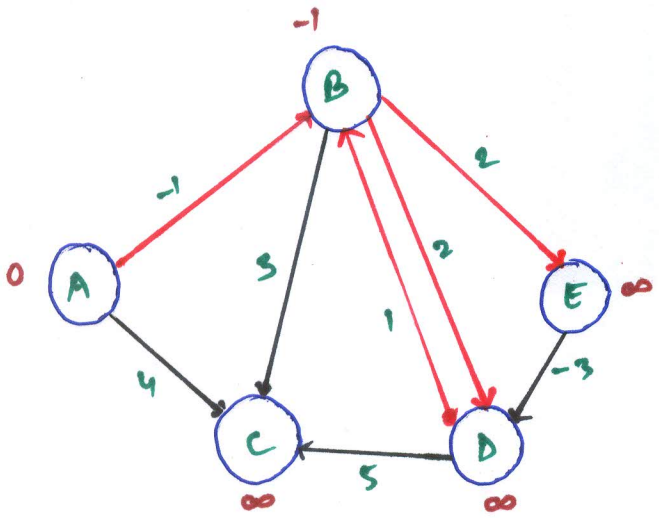
At the end, $d[v] = \delta(s,v)$

Time = $O(VE)$

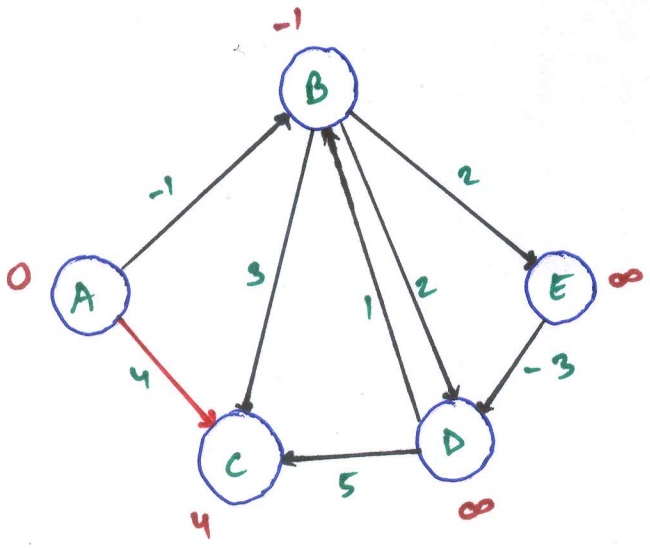
Example of Bellman-Ford



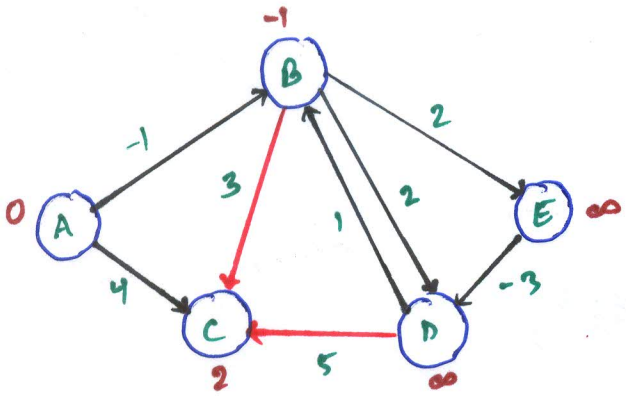
A	B	C	D	E
0	∞	∞	∞	∞



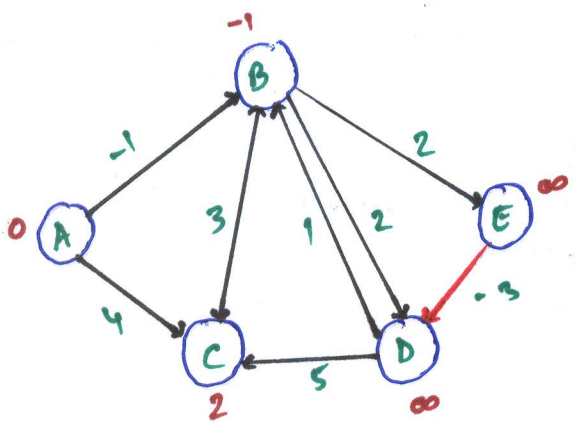
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞



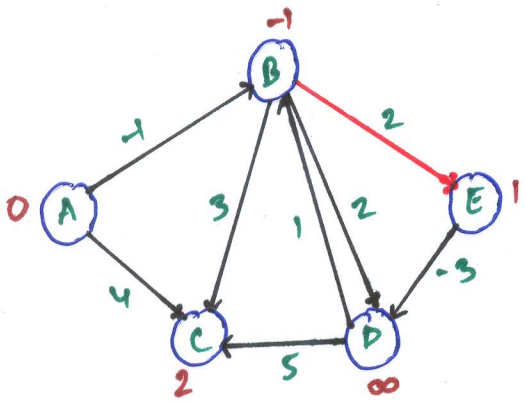
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞



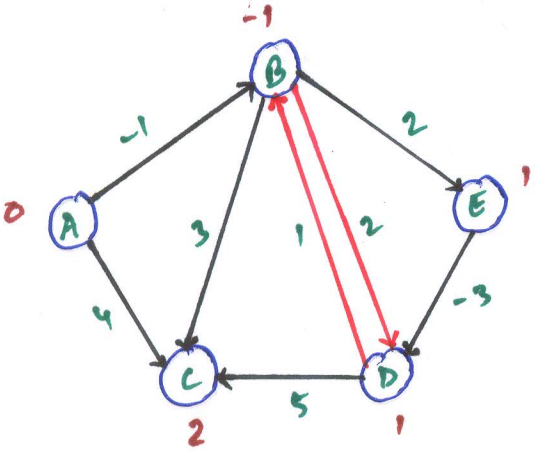
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



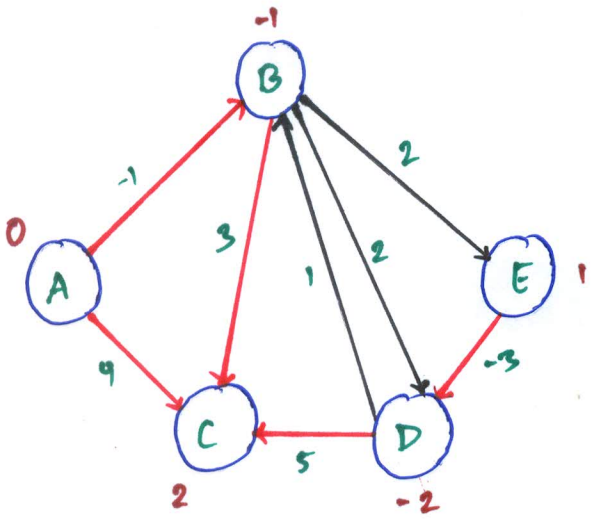
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



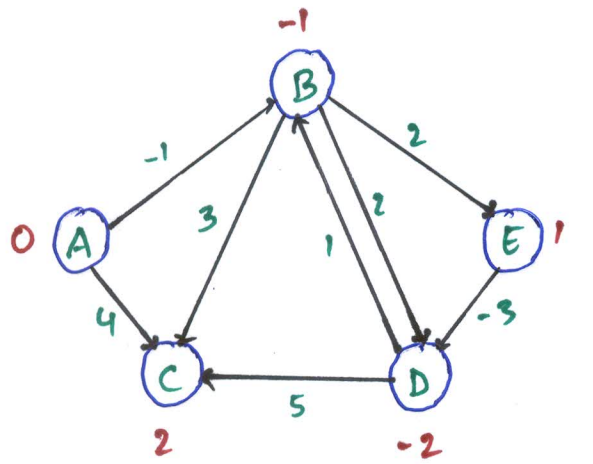
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

Week 10: Lecture Notes

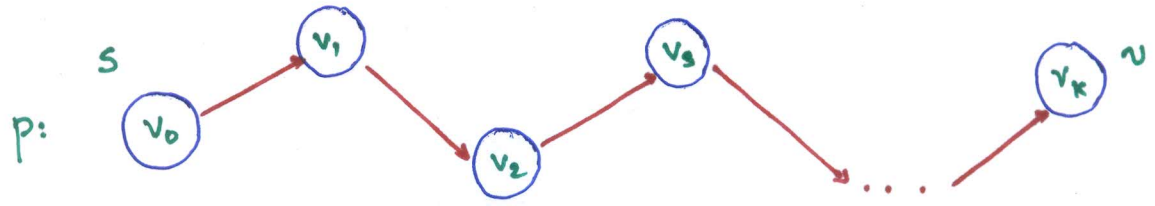
Topics: Correctness of Bellman Ford
Application of Bellman Ford
All pairs shortest path
Floyd-Warshall
Johnson Algorithm

Correctness of Bellman Ford

Theorem: If $G = (V, E)$ contains no negative-weight cycles then after the Bellman-Ford algorithm executes, $d[v] = \delta(s, v)$ for all $v \in V$

Proof:

Let $v \in V$ be any vertex, and consider a shortest path p from s to v within the minimum number of edges.



Since p is a shortest path, we have

$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i)$$

Initially, $d[v_0] = 0 = \delta(s, v_0)$, and $d[s]$ is unchanged by subsequent relaxations.

- After 1 pass through E, we have $d[v_1] = \delta(s, v_1)$
- After 2 passes through E, we have $d[v_2] = \delta(s, v_2)$
- ⋮
- After k passes, we have $d[v_k] = \delta(s, v_k)$

Since G contains no negative-weight cycles, p is simple.

Longest simple path has $\leq |V| - 1$ edges.

Detection of negative-weight cycles

Corollary: If a value $d[v]$ fails to converge after $|V|-1$ passes, there exists a negative-weight cycle in G reachable from s.

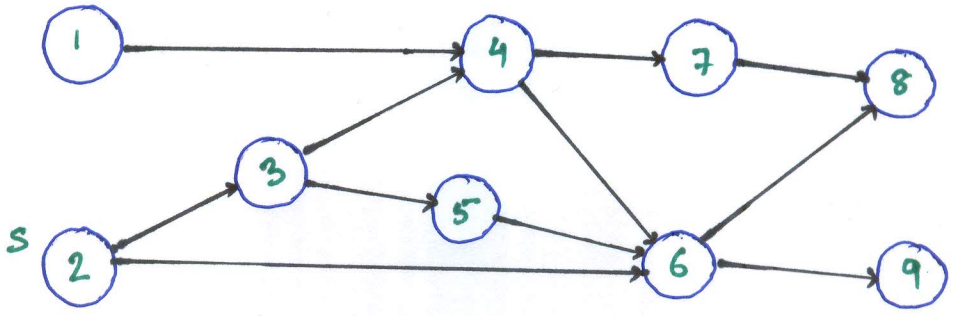
DAG Shortest paths

If the graph is a directed acyclic graph (DAG), we first topologically sort the vertices.

Determine $f: V \to \{1, 2, \ldots, |V|\}$ such that $(u,v) \in E$

$\Rightarrow f(u) < f(v)$

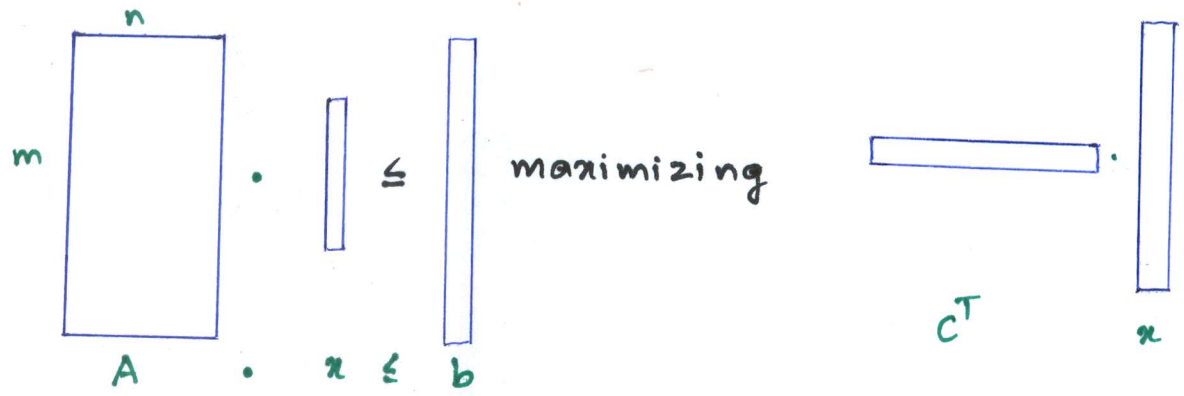
$O(V+E)$ time using depth-first search.



Walk through the vertices $u \in V$ in this order, relaxing the edges in $Adj[u]$, thereby obtaining the shortest paths s in a total of $O(V+E)$ time.

Linear Programming

Let "A" be an mxn matrix, "b" be an m-vector and "c" be an n-vector. Find an n-vector "x" that maximizes $c^T x$ subject to $Ax \leq b$, or determines that no such solution exists.



Linear Programming Algorithms

Algorithms for the general problem

- Simplex methods – practical but worst case exponential time
- Ellipsoid algorithm - polynomial time, but slow in practive
- Interior point methods - polynomial time and competes with simplex

Feasibility problem:

No optimization criterian.

Just find x such that $Ax \leq b$.

- In general, just as hard as ordinary LP.

Solving a system of difference constraints

Linear programming where each row of "A" contains exactly one 1, one -1 and the rest 0's.

Example:

$$\left.\begin{aligned} x_1 - x_2 &\le 3 \\ x_2 - x_3 &\le -2 \\ x_1 - x_3 &\le 2 \end{aligned}\right\} \; x_j - x_i \le w_{ij}$$

Solution:

$x_1 = 3$

$x_2 = 0$

$x_3 = 2$

Constraint Graph:

$x_j - x_i \le w_{ij}$ ⇒ $v_i \xrightarrow{w_{ij}} v_j$

(The A matrix has dimensions $|E| \times |V|$)

Unsatisfiable Constraints

Theorem: If the constraint graph contains a negative-weight cycle, then the system of differences is unsatisfiable.

Proof:

Suppose that the negative weight cycle is $v_1 \to v_2 \to \cdots \to v_k \to v_1$. Then, we have

$$\begin{aligned} x_2 - x_1 &\le w_{12} \\ x_3 - x_2 &\le w_{23} \\ &\vdots \\ x_k - x_{k-1} &\le w_{k-1,k} \\ x_1 - x_k &\le w_{k1} \\ \hline 0 &\le \text{weight of cycle} \\ &< 0. \end{aligned}$$

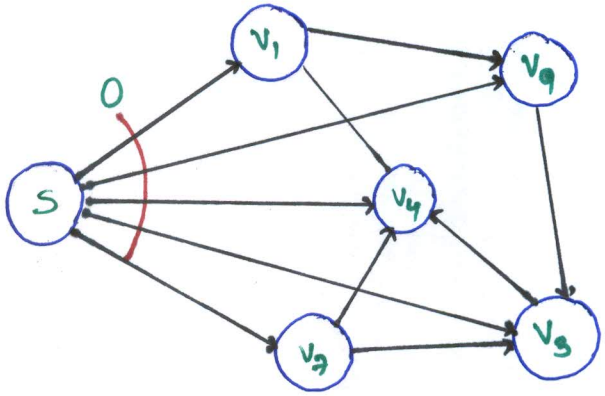
Therefore, no values for x_i can satisfy the constraints.

Satisfying the constraints

Theorem: Suppose no negative weight cycle exists in the constraint graph. Then the constraints are satisfiable.

Proof:

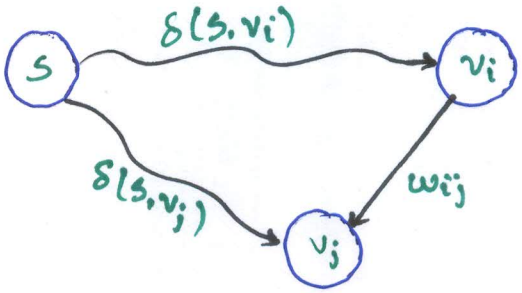
Add a new vertex s to V with a zero weight edge to each vertex $v_i \in V$



Note:
No negative-weight cycles introduced
$\Rightarrow$ shortest paths exist.

Claim:

The assignment $x_i = \delta(s, v_i)$ solves the constraints. Consider any constraint $x_j - x_i \leq w_{ij}$ and consider the shortest paths from s to v_j and v_i.



The triangle inequality gives us $\delta(s, v_j) \leq \delta(s, v_i) + w_{ij}$. Since $x_i = \delta(s, v_i)$ and $x_j = \delta(s, v_j)$, the constraint $x_j - x_i \leq w_{ij}$ is satisfied.

Bellman-Ford and linear programming

Corollary: The Bellman-Ford algorithm can solve a system of m difference constraints on n variables in $O(mn)$ time.

- Single-source shortest paths is a simple LP problem.
- In fact, Bellman-Ford maximizes $x_1 + x_2 + \cdots + x_n$ subject to the constraints $x_j - x_i \leq w_{ij}$ and $x_i \leq 0$
- Bellman-Ford also minimizes $\max_i \{x_i\} - \min_i \{x_i\}$

Shortest Paths

Single-source shortest paths:

- Non-negative edge weights
  - Dijkstra's algorithm - $O(E + V \log V)$
- General
  - Bellman Ford - $O(VE)$
- DAG
  - One pass of Bellman ford $O(V+E)$

All-pairs shortest paths

- Non-negative edge weights
  - Dijkstra's algorithm $|V|$ times - $O(VE + V^2 \log V)$

All-pairs Shortest Paths

Input: Digraph $G = (V, E)$, where $|V| = n$, with edge-weight function $w: V \to R$

Output: $n \times n$ matrix of shortest-path lengths $\delta(i,j)$ for all $i, j \in V$.

Idea #1.

- Run Bellman Ford once from each vertex
- Time = $O(V^2 E)$.
- Dense graph $\Rightarrow$ $O(V^4)$ time

"Good first try"

Dynamic Programming

Consider the $n \times n$ adjacency matrix $A = (a_{ij})$ of the digraph, and define

$d_{ij}^{(m)}$ = weight of a shortest path from i to j that uses at most m edges.

Claim: We have

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

and for $m = 1, 2, \ldots, n-1$,

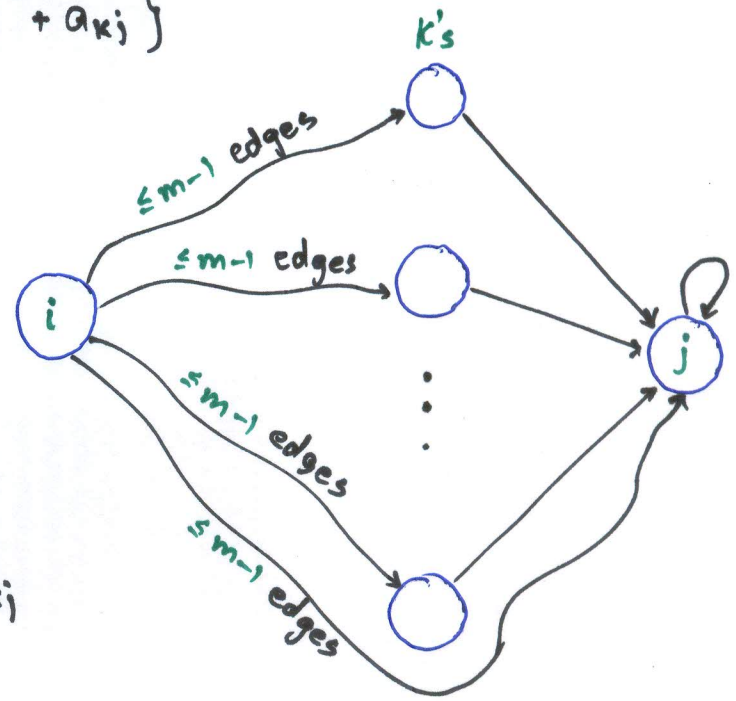
$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$

Proof of claim:

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$

Relaxation!

for $k \leftarrow 1$ to n
  do if $d_{ij} > d_{ik} + a_{kj}$
    then $d_{ij} \leftarrow d_{ik} + a_{kj}$



Note: No-negative weight cycle implies

$$\delta(i,j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \cdots$$

Matrix Multiplication

Compute $C = A \cdot B$, where C, A and B are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} .$$

Time = $\Theta(n^3)$ using the standard algorithm.

What if we map "+" → "min" and "·" → "+" ?

$$c_{ij} = \min_k \{a_{ik} + b_{kj}\}$$

Thus, $D^{(m)} = D^{(m-1)} \text{ "}\times\text{" } A$

Identity matrix = $I = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix} = D^0 = (d_{ij}^{(0)})$.

The (min, +) multiplication is associative, and with the real numbers, it forms an algebraic structure called a closed semiring

Consequently, we can compute

$$D^{(1)} = D^{(0)} \cdot A = A^1$$

$$D^{(2)} = D^{(1)} \cdot A = A^2$$

$$\vdots$$

$$D^{(n-1)} = D^{(n-2)} \cdot A = A^{n-1}$$

yielding $D^{(n-1)} = (\delta(i,j))$

Time = $\Theta(n \cdot n^3) = \Theta(n^4)$

No better than $n \times$ B-F.

Improved matrix multiplication algorithm

Repeated squaring:

$$A^{2k} = A^k \times A^k$$

Compute $\underbrace{A^2, A^4, \ldots, A^{2^{\lceil \lg(n-1) \rceil}}}_{O(\log n) \text{ squarings.}}$

Note:

$$A^{n-1} = A^n = A^{n+1} = \ldots$$

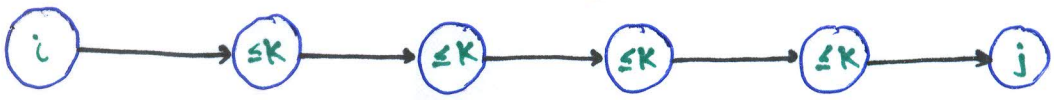
Time = $\theta(n^3 \lg n)$

To detect negative weight cycles, check the diagonal for negative values in $O(n)$ additional time.

Floyd-Warshall Algorithm

Also dynamic programming, but faster!

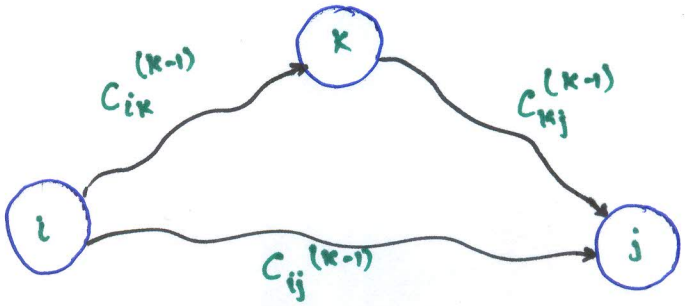
Define $c_{ij}^{(k)}$ = weight of a shortest path from i to j with intermediate vertices belonging to the set $\{1, 2, \ldots, k\}$



Thus, $\delta(i,j) = c_{ij}^{(n)}$. Also, $c_{ij}^{(0)} = a_{ij}$.

Floyd-Warshall Recurrence

$$c_{ij}^{(k)} = \min_k \left\{ c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)} \right\}$$



intermediate vertices in $\{1, 2, \ldots, k\}$

Pseudocode for Floyd-Warshall

```
1.  for k ← 1 to n
2.      do for i ← 1 to n
3.          do for j ← 1 to n
4.              do if c_ij > c_ik + c_kj          } relaxation
5.                  then c_ij ← c_ik + c_kj       }
```

Notes:

Okay to omit superscripts, since extra relaxations can't hurt

Runs in $\Theta(n^3)$ time

Simple to code

Efficient in practice.

Transitive Closure of a directed graph

Compute $t_{ij} = \begin{cases} 1, & \text{if there exists a path from } i \text{ to } j \\ 0, & \text{otherwise} \end{cases}$

IDEA:

Use Floyd-Warshall, but with $(\vee, \wedge)$ instead of $(\min, +)$:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

Time = $\Theta(n^3)$

Graph reweighting

Theorem:

Given a label $h(v)$ for each $v \in V$, reweight each edge $(u,v) \in E$ by

$$\hat{w}(u,v) = w(u,v) + h(u) - h(v)$$

Then all paths between the same two vertices are reweighted by the same amount.

Proof:

Let $p = v_1 \to v_2 \to \cdots \to v_k$ be a path in the graph.

Then we have,

$$\begin{aligned}\hat{w}(p) &= \sum_{i=1}^{k-1} \hat{w}(v_i, v_{i+1}) \\ &= \sum_{i=1}^{k-1} \left(w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1})\right) \\ &= w(p) + h(v_k) - h(v_1)\end{aligned}$$

Johnson's Algorithm

1. Find a vertex labeling h such that $\hat{w}(u,v) \geq 0$ for all $(u,v) \in E$ by using Bellman-Ford to solve the difference constraints: $h(v) - h(u) \leq w(u,v)$ or determining that a negative weight cycle exists.
   - Time = $O(VE)$
2. Run Dijkstra's algorithm from each vertex using $\hat{w}$.
   - Time = $O(VE + V^2 \log V)$
3. Reweight each shortest-path length $\hat{w}(p)$ to produce the shortest-path lengths $w(p)$ of the original graph
   - Time = $O(V^2)$

Total time = $O(VE + V^2 \log V)$

Week 11. Lecture Notes

Topics: Disjoint set data structure
Union-Find
Augmented disjoint set data structure
Network flow

Disjoint-set data structure (Union-Find)

Problem:

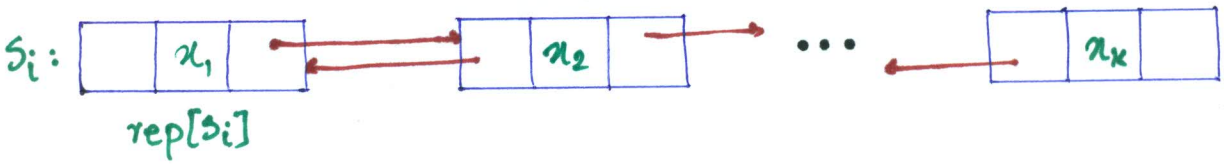
Maintain a dynamic collection of pairwise-disjoint sets $S = \{S_1, S_2, \ldots, S_r\}$. Each set S_i has one element distinguished as the representative element, $rep[S_i]$

Must support 3 operations:

- MAKE-SET(x): adds new set $\{x\}$ to S with $rep[\{x\}] = x$ (for any $x \notin S_i$ for all i)
- UNION(x,y): replaces sets S_x, S_y with $S_x \cup S_y$ in S for any x,y in distinct sets S_x, S_y.
- FIND-SET(x): returns representative $rep[S_x]$ of set S_x containing element x.

Simple linked-list solution

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as an (unordered) doubly linked list. Define representative element $rep[S_i]$ to be the front of the list, x_1.



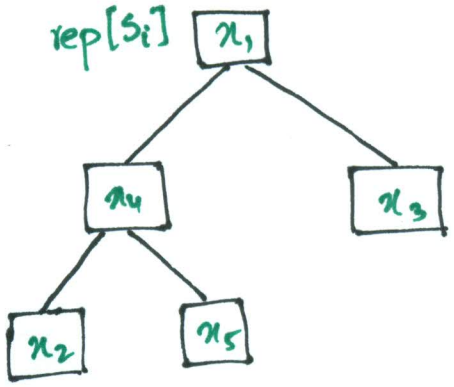
- MAKE-SET (x) initializes x as a lone node — $\Theta(1)$
- FIND-SET (x) walks left in the list containing x until it reaches the front of the list — $\Theta(n)$
- UNION (x,y) concatenates the lists containing x and y, leaving rep. as FIND-SET [x] — $\Theta(n)$

Simple balanced-tree solution

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as a balanced tree (ignoring keys). Define representative element rep[S_i] to be the root of the tree.

- MAKE-SET (x) initializes x as a lone node — $\Theta(1)$
- FIND-SET (x) walks up the tree containing x until it reaches the root — $\Theta(\log n)$
- UNION (x,y) concatenates the trees containing x and y changing rep. — $\Theta(\log n)$

$S_i = \{x_1, x_2, x_3, x_4, x_5\}$



Plan of attack

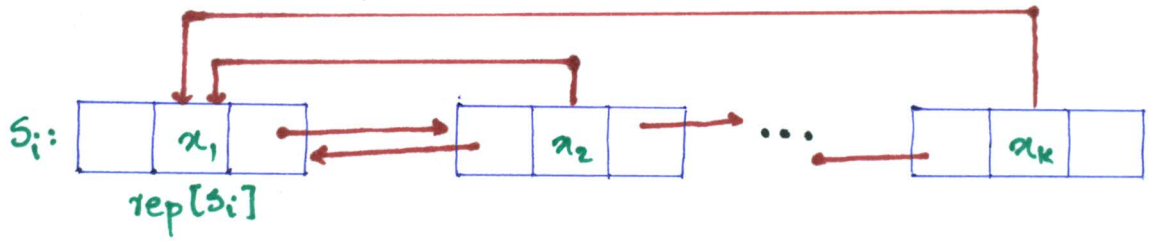
We will build a simple disjoint union data structure that, in an amortized sense, performs significantly better than $\Theta(\lg n)$ per operation, even better than $\Theta(\lg \lg n)$, $\Theta(\lg \lg \lg n)$, etc., but not quite $\Theta(1)$.

To reach this goal, we will introduce two key tricks Each trick converts a trivial $\Theta(n)$ solution into a simple $\Theta(\lg n)$ amortized solution. Together, the two tricks yield a much better solution.

First trick arises in an augmented linked list.
Second trick arises in a tree structure.

Augmented linked-list solution

Store set $S_i = \{x_1, x_2, \ldots, x_k\}$ as unordered doubly linked list. Define $rep[S_i]$ to be front of the list, x_1.
Each element x_j also stores pointer $rep[x_j]$ to $rep[S_i]$



- FIND-SET(x) returns $rep[x]$ — $\Theta(1)$
- UNION(x,y) concatenates the lists containing x and y and updates the rep pointers for all elements in the list containing y. — $\Theta(n)$

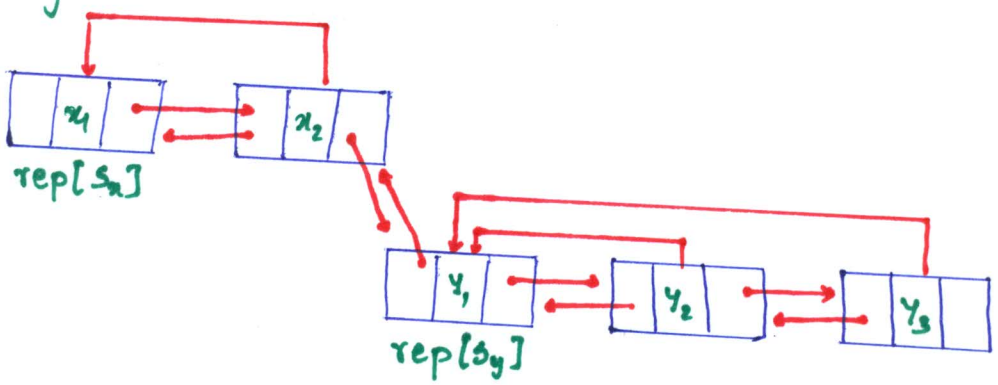
Example of augmented linked-list solution

Each element x_j stores pointer $rep[x_j]$ to $rep[S_i]$.

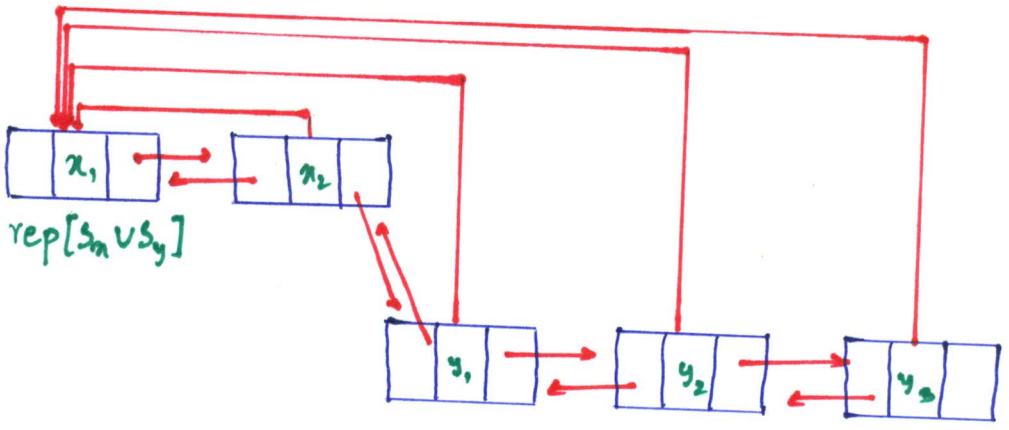
UNION (x, y)

- concatenates the lists containing x and y, and
- updates the rep pointers for all elements in the list containing y.

$S_x \cup S_y$:



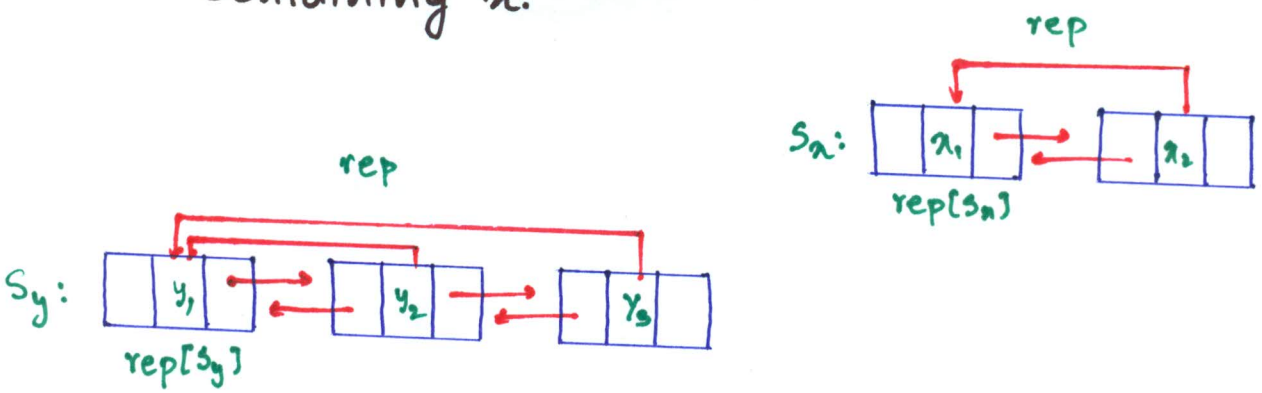
$S_x \cup S_y$:

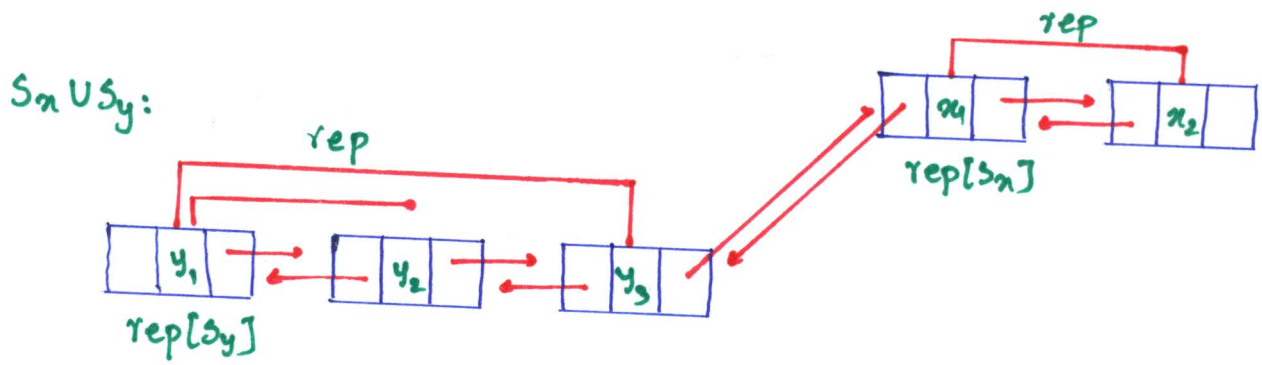


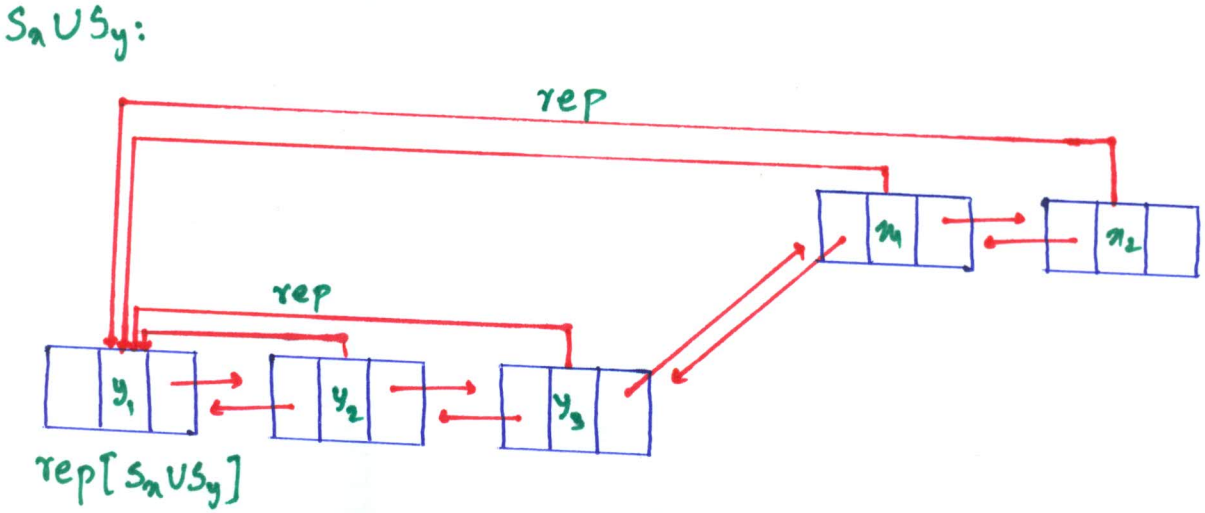
Alternative Concatenation

UNION(x,y) could instead

- concatenate the lists containing y and x, and
- update the rep pointers for all elements in the list containing x.







Trick 1: Smaller into larger

To save work, concatenate smaller list into the end of the larger list. Cost = Θ(length of smaller list)

Augment list to store its weight (# elements)

Let "n" denote the overall number of elements (equivalently, the number of MAKE-SET operations).

Let "m" denote the total number of operations

Let "f" denote the number of FIND-SET operations.

THEOREM: Cost of all UNION's is $O(n \lg n)$.

Corollary: Total cost is $O(m + n \lg n)$.

Analysis of Trick 1

To save work, concatenate smaller list into the end of the larger list. Cost = Θ(1 + length of smaller list)

Theorem: Total cost of UNION's is $O(n \lg n)$

Proof:

Monitor an element x and set S_x containing it. After initial MAKE-SET(x), weight $[S_x] = 1$. Each time S_x is united with S_y, weight $[S_y] \geq$ weight $[S_x]$, pay 1 to update rep[x], and weight $[S_x]$ at least doubles (increasing by weight $[S_y]$).

Each time S_y is united with smaller set S_x, pay nothing, and weight $[S_x]$ only increases.

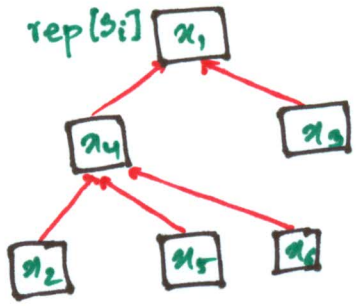
Thus pay $\leq \lg n$ for x

Representing sets as trees

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as an unordered, potentially unbalanced, not necessarily binary tree, storing only parent pointers. rep[S_i] is the tree root.

$S_i = [x_1, x_2, x_3, x_4, x_5, x_6]$

- MAKE-SET(x) initializes x as a lone node — $\Theta(1)$
- FIND-SET(x) walks up the tree containing x until it reaches the root - $\Theta(\text{depth}[x])$
- UNION(x,y) concatenates the trees containing x and y

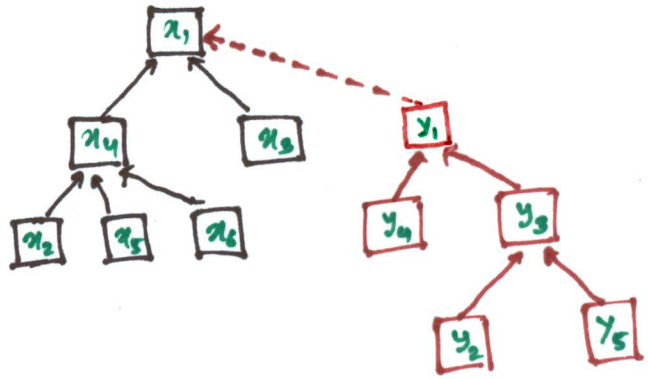


Trick 1 adapted to trees

UNION(x,y) can use a simple concatenation strategy:
Make root FIND-SET(y) a child of root FIND-SET(x).

$\Rightarrow$ FIND-SET(y) = FIND-SET(x)

We can adapt Trick 1 to this content also:
Merge tree with smaller weight into tree with larger weight.



Height of tree increases only when it's size doubles, so height is logarithmic in weight.

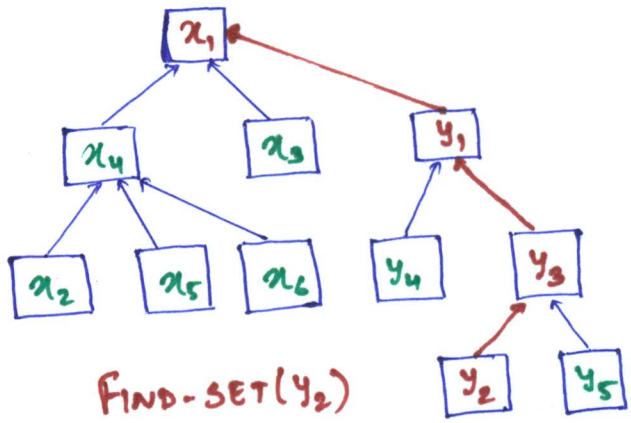
Thus total cost is $O(m + f \lg n)$

Trick 2: Path compression

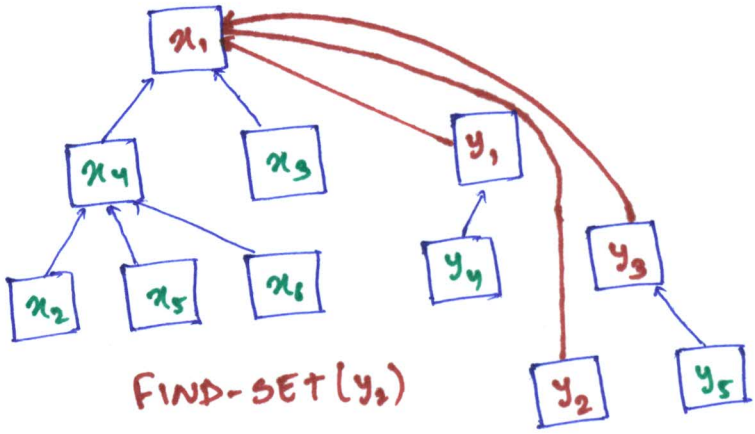
When we execute a FIND-SET operation and walk up a path p to the root, we know the representatives for all nodes on path p.

Path compression makes all of those nodes direct children of the root.

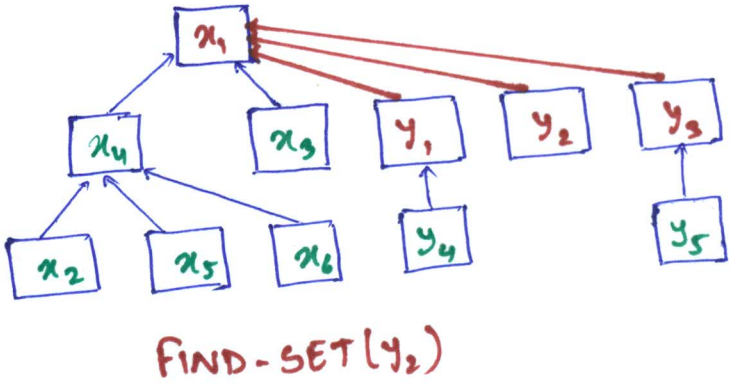
Cost of FIND-SET(x) is still $\Theta(\text{depth}[x])$



cost of FIND-SET(x) is still $\Theta(\text{depth}[x])$



Cost of FIND-SET(x) is still $\Theta(\text{depth}[x])$



Analysis of Trick 2 alone

Theorem:

Total cost of FIND-SET is $O(m \lg n)$

Proof:

Amortization by potential function.

The weight of a node x is # nodes in it's subtree.

Define $\Phi(x_1, \ldots, x_n) = \sum_i \lg \text{weight}[x_i]$

UNION (x_i, x_j) increases potential of root FIND-SET(x_i), by at most $\lg \text{weight}[\text{root FIND-SET}(x_j)] \leq \lg n$.

Each step down $p \to c$ made by FIND-SET(x_i), except the first, moves c's subtree out of p's subtree. Thus if $\text{weight}[c] \geq \frac{1}{2}\,\text{weight}[p]$, Φ decreases by ≥ 1. paying for the step down. There can be at most $\lg n$ steps $p \to c$ for which $\text{weight}[c] < \frac{1}{2}\,\text{weight}[p]$.

Theorem:

If all UNION operations occur before all FIND-SET operations, then total cost is $O(mn)$

Proof:

If a FIND-SET operation traverses a path with k nodes, costing $O(k)$ time, then $k-2$ nodes are made new children of the root.

This change can happen only once for each of the n elements.

So, the total cost of FIND-SET is $O(f+n)$.

Ackermann's function A

Define $A_k(j) = \begin{cases} j+1, & \text{if } k=0 \\ A_{k-1}^{(j+1)}(j), & \text{if } k \geq 1 \end{cases}$ — iterate $j+1$ times

$A_0(j) = j+1$ $\qquad$ $A_0(1) = 2$

$A_1(j) \sim 2j$ $\qquad$ $A_1(1) = 3$

$A_2(j) \sim 2j\,2^j > 2^j$ $\qquad$ $A_2(1) = 7$

$A_3(j) > 2^{2^{\cdot^{\cdot^{\cdot^{2^j}}}}}$ (j twos) $\qquad$ $A_3(1) = 2047$

$A_4(1)$ is a lot bigger. $A_4(1) > 2^{2^{\cdot^{\cdot^{\cdot^{2^{2047}}}}}}$ (2048 twos)

Define $\alpha(n) = \min\{k : A_k(1) \geq n\} \leq 4$ for practical n

Analysis of Tricks 1 + 2

Theorem:

In general, total cost is $O(m\,\alpha(n))$.

Application: Dynamic Connectivity

Suppose a graph is given to us incrementally by

ADD-VERTEX (v)

ADD-EDGE(u,v)

and we want to support connectivity queries:

CONNECTED (u,v):

Are u and v in the same connected component?

for example, we want to maintain a spanning forest, so we check whether each new edge connects a previously disconnected pair of vertices.

Sets of vertices represent connected components.

Suppose a graph is given to us incrementally by

ADD-VERTEX (v) - MAKE-SET(v)

ADD-EDGE(u,v) - if not CONNECTED(u,v)
then UNION(v,w)

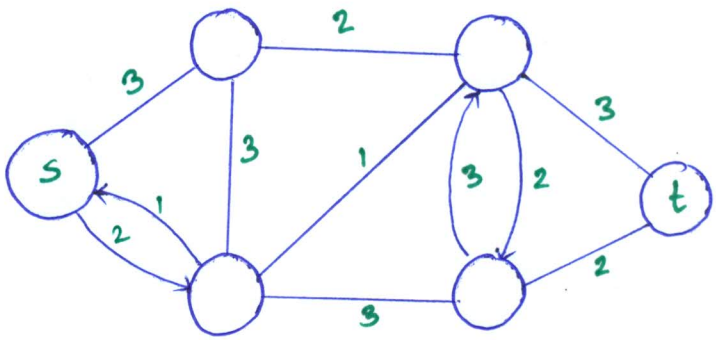
and

CONNECTED(u,v): FIND-SET(u) = FIND-SET(v)

Flow networks

Definition: A flow network is a directed graph $G=(V,E)$ with two distinguished vertices: a source s and a sink t. Each edge $(u,v) \in E$ has a non-negative capacity $c(u,v)$. If $(u,v) \notin E$, then $c(u,v) = 0$.

Example:



Definition: A positive flow on G is a function $p: V \times V \to R$ satisfying the following:

- Capacity constraint: for all $u, v \in V$

$$0 \leq p(u,v) \leq c(u,v)$$

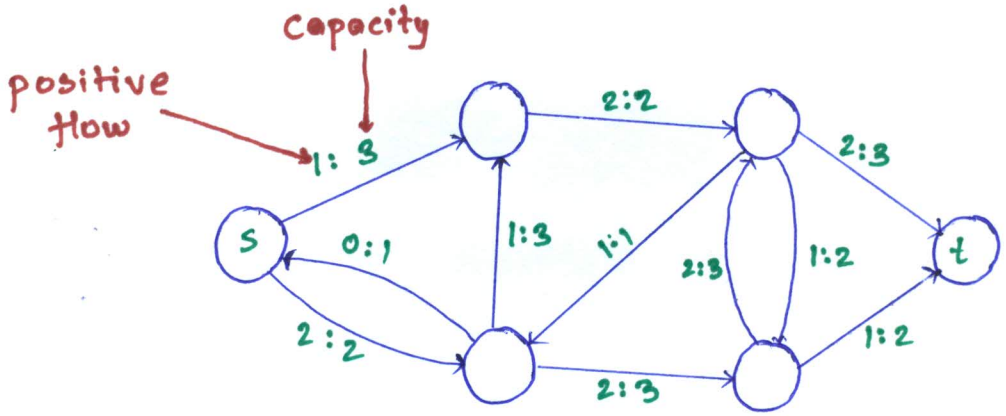
- Flow conservation: for all $u \in V - \{s,t\}$,

$$\sum_{v \in V} p(u,v) - \sum_{v \in V} p(v,u) = 0$$

The value of a flow is the net flow out of the source:

$$\sum_{v \in V} p(s,v) - \sum_{v \in V} p(v,s)$$

A flow on a network



Flow conservation (like Kirchoff's current law):

- Flow into u is 2+1=3
- Flow out of u is 0+1+2=3

The value of this flow is 1-0+2=3.

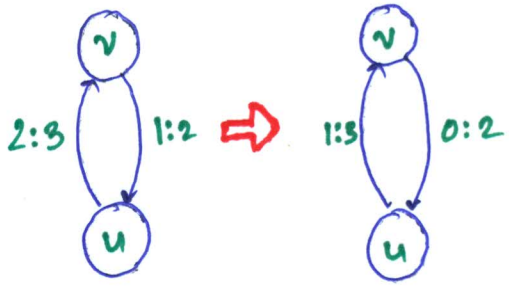
Maximum-flow problem: Given a flow network G, find a flow of maximum value in G.

In the above figure, value of maximum flow = 4.

Flow Cancellation

Without loss of generality, positive flow goes either from u to v, or from v to u, but not both.

The capacity constraint and flow conservation are preserved by this transformation



Net flow from u to v in both cases is 1

INTUITION:

View flow as a rate, not a quantity.

A notational simplification

IDEA: Work with the net flow between two vertices, rather than with the positive flow.

Definition: A (net) flow on G is a function $f: V \times V \to R$ satisfying the following:

- Capacity constraint: For all $u, v \in V$,
$$f(u,v) \leq c(u,v)$$
- Flow conservation: For all $u \in V - \{s,t\}$
$$\sum_{v \in V} f(u,v) = 0$$
} one assumption instead of two
- Skew-symmetry: for all $u, v \in V$
$$f(u,v) = -f(v,u)$$

Equivalence of Definitions

Theorem: The two definitions are equivalent

Proof:

Let $f(u,v) = p(u,v) - p(v,u)$

Capacity constraint: Since $p(u,v) \leq c(u,v)$ and $p(v,u) \geq 0$, we have $f(u,v) \leq c(u,v)$.

Flow conservation:

$$\sum_{v \in V} f(u,v) = \sum_{v \in V} (p(u,v) - p(v,u))$$

$$= \sum_{v \in V} p(u,v) - \sum_{v \in V} p(v,u)$$

Skew symmetry: $f(u,v) = p(u,v) - p(v,u)$

$= -(p(v,u) - p(u,v))$

$= -f(v,u)$

Next, consider $p(u,v) = \begin{cases} f(u,v), & \text{if } f(u,v) > 0 \\ 0, & \text{if } f(u,v) \leq 0 \end{cases}$

Capacity constraint: By definition $p(u,v) \geq 0$. Since $f(u,v) \leq c(u,v)$, it follows that $p(u,v) \leq c(u,v)$

Flow conservation: If $f(u,v) > 0$, then $p(u,v) - p(v,u) = f(u,v)$.

If $f(u,v) \leq 0$, then $p(u,v) - p(v,u) = -f(v,u) = f(u,v)$ (by skew symmetry)

Therefore,

$$\sum_{v \in V} p(u,v) - \sum_{v \in V} p(v,u) = \sum_{v \in V} f(u,v)$$

Notation

Definition: The value of a flow f, denoted by $|f|$ is given by

$$|f| = \sum_{v \in V} f(s,v) = f(s,V)$$

Implicit summation notation:

A set used in an arithmetic formula represents a sum over the elements of the set.

Example:

flow conservation:

$f(u,V) = 0$ for all $u \in V - \{s,t\}$.

Simple properties of flow

Lemma:

$f(X, X) = 0$

$f(X, Y) = -f(Y, X)$

$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ if $X \cap Y = \phi$

Theorem

$|f| = f(V, t)$

Proof: $|f| = f(s, V)$

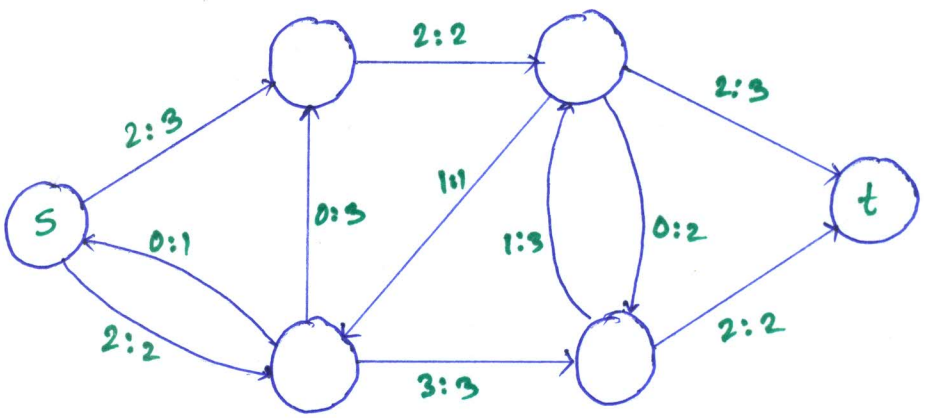
$= f(V, V) - f(V - s, V)$ Omit braces

$= f(V, V - s)$

$= f(V, t) + f(V, V - s - t)$

$= f(V, t)$

Flow into the sink



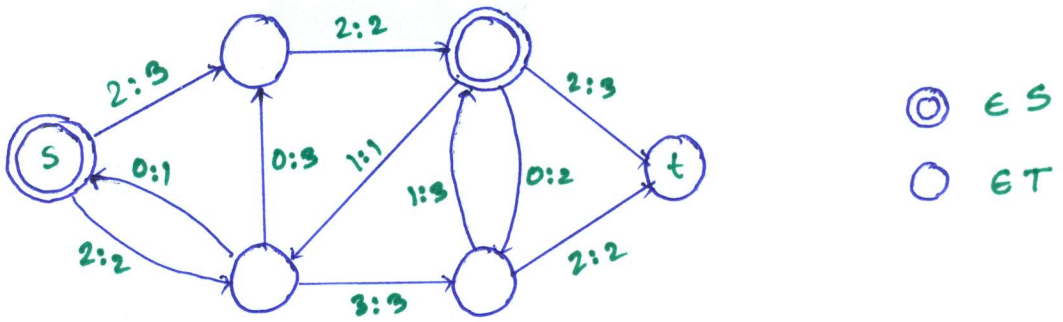
$|f| = f(s, V) = 4$

$f(V, t) = 4.$

Cuts

A cut (S,T) of a flow network $G=(V,E)$ is a partition of V such that $s \in S$ and $t \in T$.

If f is a flow on G, then the flow across the cut is $f(S,T)$



$$f(S,T) = (2+2) + (-2+1-1+2) = 4$$

Another characterization of flow value

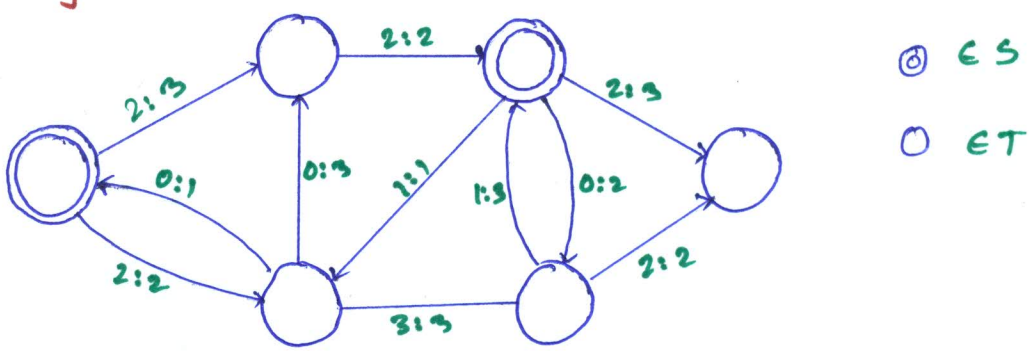
Lemma: For any flow f and any cut (S,T) $|f| = f(S,T)$.

Proof:

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \\ &= f(S,V) \\ &= f(s,V) - f(S-s,V) \\ &= f(s,V) = |f| \end{aligned}$$

Capacity of a cut

capacity of a cut (S,T) is $c(S,T)$



$$\begin{aligned} c(S,T) &= (3+2) + (1+2+3) \\ &= 11 \end{aligned}$$

Upper bound on the maximum flow value

Theorem:

The value of any flow is bounded above by the capacity of any cut.

Proof:

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u,v) \\ &= c(S,T) \end{aligned}$$

Residual Network

Definition:

Let f be a flow on $G = (V,E)$. The residual network $G_f(V, E_f)$ is the graph with strictly positive residual capacities:

$$c_f(u,v) = c(u,v) - f(u,v) > 0.$$

Edges in E_f admit more flow.

Example:

Lemma:

$$|E_f| \leq 2|E|$$

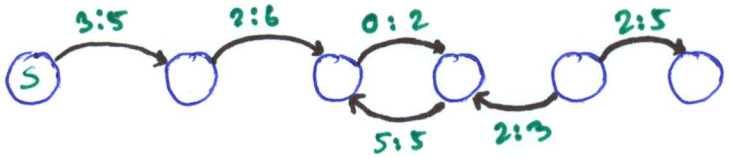
Augmenting Paths

Definition: Any path from s to t in G_f is an augmenting path in G with respect f. The flow value can be increased along an augmenting path p by

$$c_f(p) = \min_{(u,v) \in p} \{c_f(u,v)\}$$

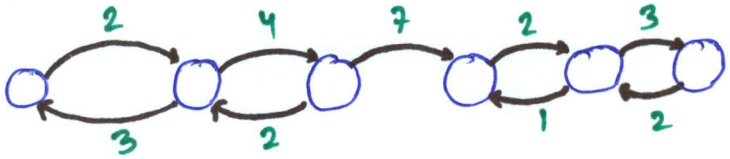
Ex:

G:



$c_f(p) = 2$

G_f:



Max-flow, min cut theorem

Theorem:

The following are equivalent:

1. f is a maximum flow
2. f admits no augmenting paths
3. $|f| = c(S,T)$ for some cut $[S,T)$

Week 12 - Lecture Notes

Topics: Dynamic Programming
- memoization and subproblems
- Fibonacci } Examples
- Shortest paths } Examples
- guessing and DAG views

Computational Complexity

Dynamic Programming (DP)

- Big idea, hard, yet simple
- Powerful algorithmic design technique
- Large class of seemingly exponential problems have a polynomial solution ("only") via DP.
- Particularly for optimization problems (min/max)
  - Example: Shortest paths.

A dynamic programming is a controlled brute-force method.

It uses recursion and re-use.

i.e.

DP ≈ "controlled-brute-force"

DP ≈ "recursion and re-use"

Fibonacci Numbers

Fibonacci numbers are of the form

$$F_1 = F_2 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

Goal: Compute F_n

Naive Algorithm

follows recursive definition.

fib(n):

1. if $n \leq 2$ return $f = 1$
2. else return f: fib(n-1) + fib(n-2)

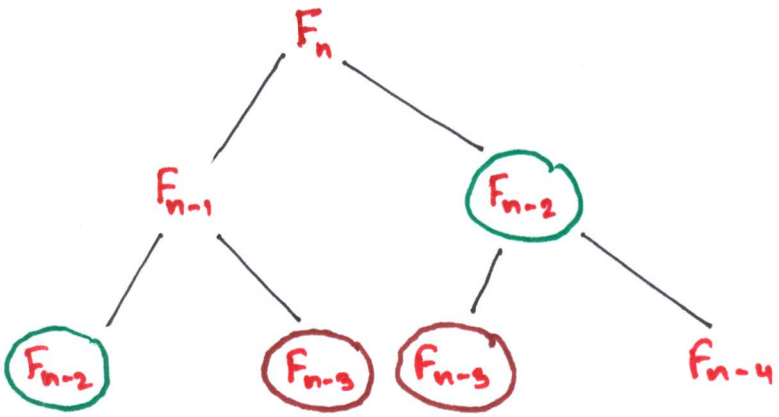
$$\Rightarrow T(n) = T(n-1) + T(n-2) + O(1)$$

$$\geq F_n \approx \phi^n$$

$$\geq 2T(n-2) + O(1)$$

$$\geq 2^{n/2}$$

Exponential - BAD!



Memoized DP Algorithm

```
1.  memo = {}
    fib(n):
2.      if n is in memo : return memo[n]
3.      else: if n ≤ 2 : f = 1
4.            else f = fib(n-1) + fib(n-2)
5.            memo[n] = f
6.            return f
```

- fib(k) only recurses first time called ∀ k
- only n nonmemoized cells : k = 1, 2, ..., n
- memoized calls free ($\Theta(1)$ time)
- $\Theta(1)$ time per call (ignoring recursion)

Polynomial - GOOD!

- DP ≈ "recursion + memoization"
  - memoize (remember) and re-use solutions to subproblems that help solve problem
    - in Fibonacci, subproblems are $F_1, F_2, \ldots, F_n$

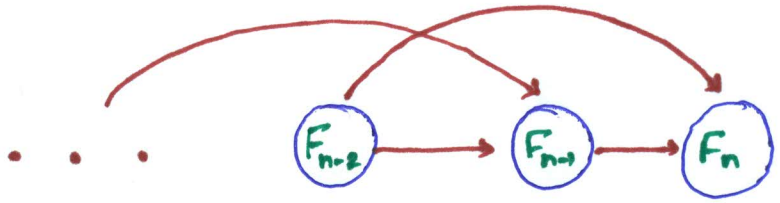
⇒ time = # subproblems · (time per subproblem)

- Fibonacci : # subproblems = n
  time per subproblem = $\Theta(1)$
  ∴ time = $\Theta(n)$ (ignoring recursions)

Bottom-up DP Algorithm

```
1.  fib = {}
2.  for K in [1,2,...,n]:
3.      if K ≤ 2: f = 1
4.      else: f = fib[K-1] + fib[K-2]      } Θ(1)
5.      fib[K] = f                         } Θ(n)
6.  return fib[n]
```

- exactly the same computation as memoized DP (recursion "unrolled")
- in general: topological sort of subproblem dependency DAG.



- practically faster: no recursion
- analysis more obvious
- can save space: last 2 fibs $\Rightarrow \Theta(1)$

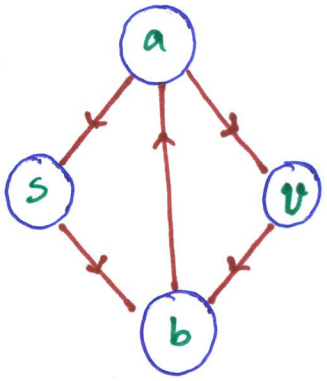
Shortest Paths

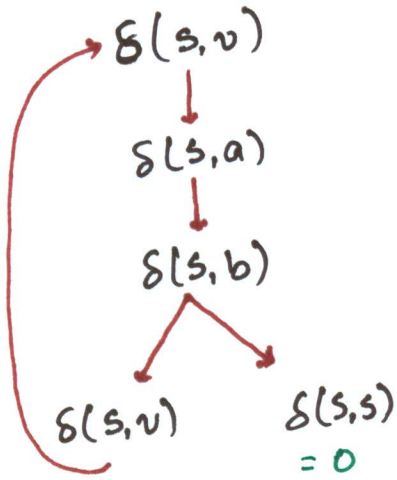
- Recursive formulation

  $\delta(u,v) = \min \{ w(u,v) + \delta(s,u) \mid (u,v) \in E \}$

- Memoized DP algorithm: takes infinite time if cycles.
  (necessary to handle negative cycles)

- Works for directed acyclic graphs in $O(V+E)$
  ~(effectively DFS/ topological sort + Bellman ford rolled into single recursion)





- Subproblem dependency should be acyclic.
  - more subproblems remove cyclic dependence

    $\delta_k(s,v)$ = shortest $s \to v$ path using $\leq k$ edges

  - recurrence:

    $\delta_k(s,v) = \min \{ \delta_{k-1}(s,u) + w(u,v) \mid (u,v) \in E \}$

    $\delta_0(s,v) = \infty$ for $s \neq v$ } base case

    $\delta_k(s,s) = 0$ for any k }

  - goal: $\delta(s,v) = \delta_{|V|-1}(s,v)$ } if no negative cycle exists

  - memoize

  - time: $\underbrace{\#\text{ subproblems}}_{|V||V|} \cdot \underbrace{(\text{time per subproblems})}_{O(V)} = O(V^3)$

    - actually $\Theta(\text{indegree}(v))$ for $\delta_k(s,v)$

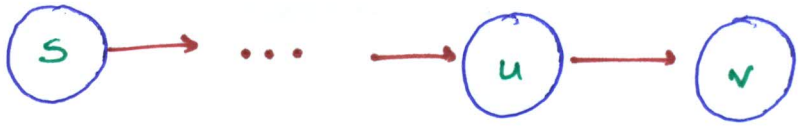
    $\Rightarrow$ time $\Theta\left(V \sum_{v \in V} \text{indegree}(v)\right) = \Theta(VE)$

BELLMAN FORD!

Guessing

How to design recurrence

- want shortest $s \to v$ path

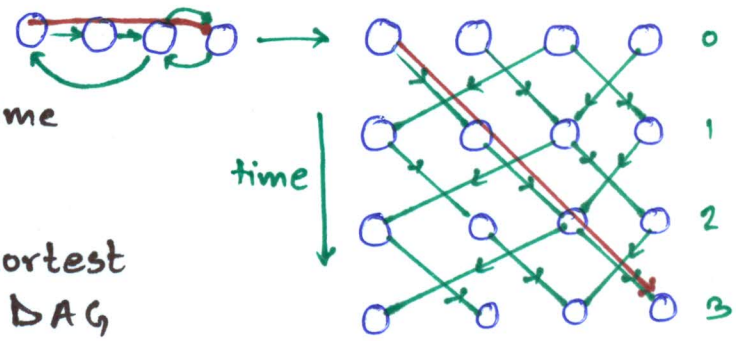


- what is the last edge in path? don't know
- guess it is (u,v)
- path is shortest $s \to u$ path + edge (u,v)
  by optimal substructure
- cost is $\delta_{k-1}(s,u) + w(u,v)$
  another subproblem
- to find best guess, try all ($|V|$ choices) and use best.
- *Key: small (polynomial) # possible guesses per subproblem
  - typically this dominates time/subproblem.

* DP ≈ recursion + memoization + guessing

DAG view

- like replicating graph to represent time
- converting shortest paths in graph to shortest paths in DAG



* DP ≈ shortest paths in some DAG

Summary

- DP ≈ careful brute force
- ≈ guessing + recursion + memoization
- ≈ dividing into reasonable # subproblems whose solution relate - acyclicly - usually via guessing parts of solution

- time = # subproblems × (time per subproblem)

  treating recursive calls as O(1)
  (usually mainly guessing)

  - essentially an amortization
  - count each subproblem only once;
    after first time, costs O(1) via memoization

- DP ≈ shortest paths in some DAG.

5 easy steps to Dynamic Programming

define subproblems — count # subproblems

guess (part of solution) — count # choices

relate subproblem solutions — compute time per subproblem

recurse + memoize problems — time = (time per subproble) × # subproblems

OR

build DP table bottom-up

check subproblems acyclic/topological order.

Solve original problem: $\Longrightarrow$ extra time

= a subproblem OR by counting subproblem solutions.

Examples	Fibonacci	Shortest paths
subproblems	F_K for $1 \le K \le n$	$\delta_K(s,v)$ for $v \in V, 0 \le K \le \lvert V \rvert$ = min $s \to v$ path using $\le K$ edges
# subproblems	n	V^2
guess	nothing	edge into v (if any)
# choices	1	indegree(v) + 1
recurrence	$F_K = F_{K-1} + F_{K+2}$	$\delta_K(s,v) = \min \{\delta_{K-1}(s,u) + w(u,v) \mid (u,v) \in E\}$
time per subproblem	$\Theta(1)$	$\Theta(1 + \text{indegree}(v))$
topological order	for $K = 1, \ldots, n$	for $K = 0, 1, \ldots \lvert V \rvert - 1$ for $v \in V$
total time	$\Theta(n)$	$\Theta(VE)$ + $\Theta(V^2)$ unless efficient about indegree
original problem	f_n	$\delta_{\lvert V \rvert - 1}(s,v)$ for $v \in V$
extra time	$\Theta(1)$	$\Theta(V)$

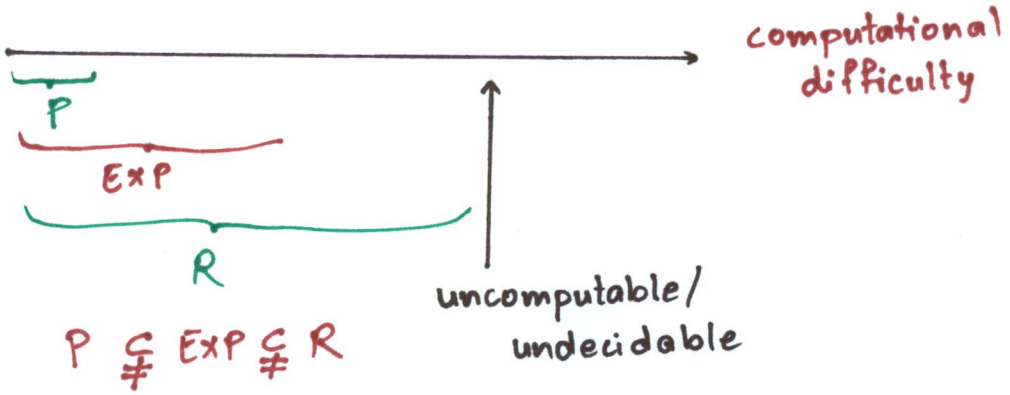
Computational Complexity

Definitions:

P = { problems solvable in (n^c) time } (polynomial)

EXP = { problems solvable in (2^{n^c}) time } (exponential)

R = { problems solvable in finite time } "recursive"



Examples:

negative-weight cycles detection $\in P$

nxn Chess $\in EXP$ but $\notin P$
↳ who wins from given board configuration?

Tetris $\in EXP$ but don't know whether $\in P$
↳ survive given pieces from given board.

Halting Problem

Given a computer program, does it ever halt (stop)?

- uncomputable ($\notin R$): no algorithm solves it (correctly in finite time on all inputs)

- decision problem: answer is YES or NO

Most Decision Problems are Uncomputable

- program ≈ binary string ≈ nonnegative integer $\in \mathbb{N}$
- decision problem = a function from binary strings (≈ non-neg. intergers) to {YES (1), NO (0)}
- ≈ infinite sequence of bits ≈ real number $\in \mathbb{R}$

  $|\mathbb{N}| << |\mathbb{R}|$: no assignment of unique nonnegative integers to real numbers ($\mathbb{R}$ uncountable)
- $\Rightarrow$ not nearly enough programs for all problems
- each program solves only one problem
- $\Rightarrow$ almost all problems cannot be solved

NP

NP = {Decision problems solvable in polynomial time via a lucky algorithm} "The lucky algorithm can make lucky gueses, always "right" without trying all options"

- nondeterministic model : algorithm makes gueses and then says YES or NO
- guesses guaranteed to lead to YES outcome if possible

Example:

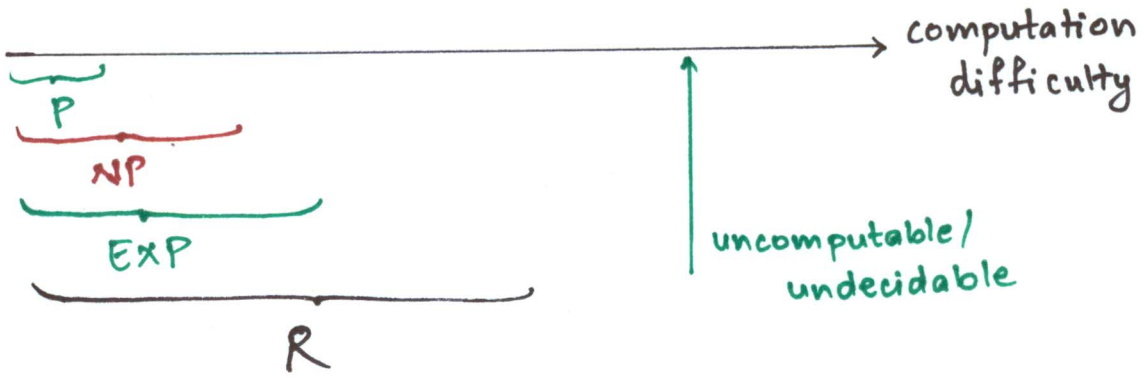
Tetris $\in$ NP

- nondeterministic algorithm : guess each move, did I survive?
- proof of YES: list what moves to make (rules of Tetris are easy)

NP

NP = {decision problems with solutions that can be "checked" in polynomial time}

$\Rightarrow$ when answer is YES,
it can be proved, and
polynomial-time algorithm can check proof.



P ≠ NP

It is a big conjecture (worth $1,000,000)

- ≈ cannot engineer luck
- ≈ generating (proofs of) solutions can be harder than checking them

Hardness and completeness

Claim:

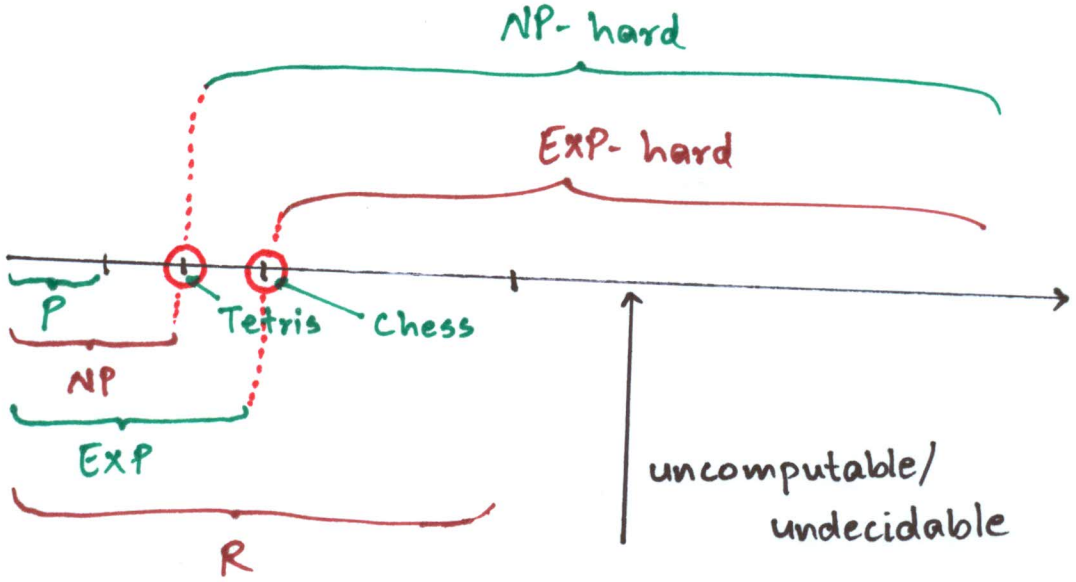
If $P \neq NP$, then Tetris $\in NP - P$

Proof:

- Tetris is NP-hard = "as hard as" every problem $\in NP$

Infact

- Tetris is NP-complete = $NP \cap$ (NP-hard)



- Chess is EXP-complete = EXP $\cap$ EXP-hard.

EXP-hard is as hard as every problem in EXP.

If $NP \neq EXP$, then Chess $\notin EXP \setminus NP$.

Whether $NP \neq EXP$ is also an open problem but "less" famous / "important".

Reductions

Convert the problem into a problem that is already known how to solve (instead of solving from scratch)

- most common algorithm design technique
- unweighted-shortest path → weighted (set weights = 1)
- min product path → shortest path (take logs)
- longest path → shortes path (negative weights
- shortest order tour → shortest path (K copies of the graph)
- cheapest leaky-tank path → shortest path (graph reduction)

All of the above are One-call reductions:

A problem → B problem → B solution → A solution

Multicall reductions:

- solve A using free calls to B,

"in this sense, every algorithm reduces problem → model of computation."

NP- Complete Problems

NP- Complete problems are all interreducible using polynomial time reductions (same difficulty)

We can use reductions to prove NP-hardness → Tetris.

Examples of NP- Complete Problems

- Knapsack
- 3- partition : given n integers, divide them into triples of equal sum?
- Travelling Salesman Problem:
  - → shortest path that visits all vertices of a given graph
  - → is minimum weight $< x$? (decision version)
- longest common subsequence of K strings
- Minesweeper, Soduku and most puzzles
- SAT : given a Boolean formula (and, or, not), is it ever true?
- Shortest paths amidst obstacles in 3D
- 3- coloring a given graph
- find largest clique in a given graph.