Design and Analysis of Algorithms

Assignment-2

Solutions (Hints)

REMARK: The codes given below are just a guide, this is not the only way to write programs for these algorithms. What is important is that your program should implement the algorithm in the correct manner (as efficient as possible).

1.a) **Quicksort**

```
 int partition(int arr[],int low,int high)
{
        int pivot=arr[low];// pivot is first element
        int i=low,j;
        for(j=low+1;j<=high;j++)
        {
                if(arr[j]<=pivot)
                {
                        i=i+1;
                        swap(&arr[i],&arr[j]);
                }
        }
        swap(&arr[i],&arr[low]);
        return i;

}
void quicksort(int arr[],int low,int high)
{
        if(low<high)
        {
                int pi=partition(arr,low,high);
                   quicksort(arr,low,pi-1); // once partition is called the pivot is in its correct position and
                                so there is no need to sort it anymore, and so we call upto pi-1 and from pi+1
                quicksort(arr,pi+1,high);
        }
}
```

```
b)  int partition(int arr[],int low,int high)
{       int pivot=arr[high];// last element is pivot
        i=low-1;// i starts from -1
        for(j=low;j<=high-1;j++)
        {
                If (arr[j]<=pivot)
                {
```

```
                    i++;
                    swap(&arr[i],&arr[j]);
                }
            }
            swap(&arr[i+1],&arr[high]);
            return i+1;
}
```

2. **Merge Sort**

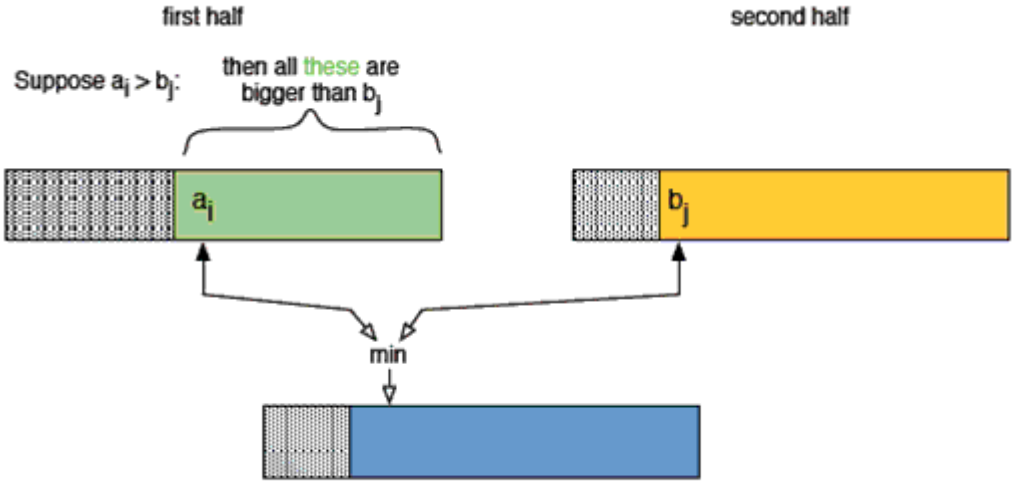
```
 void merge(int arr[], int l, int m, int r)
{
            int i, j, k;
            int n1 = m - l + 1;
            int n2 =  r - m;

             /* create temp arrays */
             int L[n1], R[n2];

             /* Copy data to temp arrays L[] and R[] */
             for (i = 0; i < n1; i++)
                        L[i] = arr[l + i];
             for (j = 0; j < n2; j++)
                        R[j] = arr[m + 1+ j];

              /* Merge the temp arrays back into arr[l..r]*/
            i = 0; // Initial index of first subarray
           j = 0; // Initial index of second subarray
           k = l; // Initial index of merged subarray
            while (i < n1 && j < n2)
            {
                        if (L[i] <= R[j])
                       {
                                    arr[k] = L[i];
                                    i++;
                        }
                        else
                        {
                                    arr[k] = R[j];
                                    j++;
                        }
                         k++;
            }

             /* Copy the remaining elements of L[], if there
              are any */
             while (i < n1)
             {
                         arr[k] = L[i];
```

```
                i++;
                k++;
        }

        /* Copy the remaining elements of R[], if there
          are any */
        while (j < n2)
        {
                arr[k] = R[j];
                j++;
                k++;
        }
}

/* l is for left index and r is right index of the
  sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
        if (l < r)
        {
                int m = l+r/2;

                // Sort first and second halves
                mergeSort(arr, l, m);
                mergeSort(arr, m+1, r);

                merge(arr, l, m, r);
        }
}
```

3. Inversion Count

Suppose we know the number of inversions in the left half and right half of the array (let be inv1 and inv2), what kinds of inversions are not accounted for in Inv1 + Inv2? The answer is – the inversions we have to count during the merge step. Therefore, to get number of inversions, we need to add number of inversions in left subarray, right subarray and merge().

How to get number of inversions in merge()?
In merge process, let i is used for indexing left sub-array and j for right sub-array. At any step in merge(), if a[i] is greater than a[j], then there are (mid – i) inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j]




```
int _mergeSort(int arr[], int left, int right)
{
        int mid, inv_count = 0;
        if (right > left)
        {
                /* Divide the array into two parts and call _mergeSortAndCountInv()
                for each of the parts */
                mid = (right + left)/2;

                 /* Inversion count will be sum of inversions in left-part, right-part
                   and number of inversions in merging */
                 inv_count  = _mergeSort(arr, left, mid);
                 inv_count += _mergeSort(arr, mid+1, right);

                /*Merge the two parts*/
                  inv_count += merge(arr,left, mid+1, right);
        }
        return inv_count;
}

/* This funt merges two sorted arrays and returns inversion count in
  the arrays.*/
int merge(int arr[],int left, int mid, int right)
{
        int i, j, k;
        int inv_count = 0;

        i = left; /* i is index for left subarray*/
        j = mid;  /* j is index for right subarray*/
       k = left; /* k is index for resultant merged subarray*/
        while ((i <= mid - 1) && (j <= right))
```

```
        {
                if (arr[i] <= arr[j])
                {
                        temp[k++] = arr[i++];
                }
                else
                {
                temp[k++] = arr[j++];

                /*this is tricky -- see above explanation/diagram for merge()*/
                inv_count = inv_count + (mid - i);
                }
        }

         /* Copy the remaining elements of left subarray
        (if there are any) to temp*/
         while (i <= mid - 1)
         temp[k++] = arr[i++];

        /* Copy the remaining elements of right subarray
         (if there are any) to temp*/
        while (j <= right)
        temp[k++] = arr[j++];

        /*Copy back the merged elements to original array*/
        for (i=left; i <= right; i++)
         arr[i] = temp[i];

         return inv_count;
}
```

4. **Quickselect**

```
 int partition(int arr[],int low,int high)
{
        int pivot=arr[low];
        int i=low,j;
        for(j=low+1;j<=high;j++)
        {
                if(arr[j]>pivot)// sorting in descending order
                {
                        i=i+1;
                        swap(&arr[i],&arr[j]);
                }
        }
        swap(&arr[i],&arr[low]);
        return i;
```

```
}
// the below function returns the position of the kth largest element
int quickselect(int arr[],int low,int high,int k)
{
	if(k>0 && k<=high-low+1)
	{
		int pi=partition(arr,low,high);
		if(pi-low>k-1)
		return quickselect(arr,low,pi-1,k);
		else if(pi-low==k-1)
		return pi;
		else return quickselect(arr,pi+1,high,k-pi+low-1);
	}
}
```