CS-521-900, Assignment 2

Yiyun Zhang

July 21, 2020

1. **a)** Define k as error distance because it means how much steps the elements are apart. Then in original Quicksort algorithm, we can consider the *while* statement $while(p < r)$ means the error distance is 0. Therefore, in this case we can modify the *while* statement in order to modify the Quicksort procedure.

```
1 Quicksort(A,p,r,k){
2   while(p<r-k)
3     q=partition(A,p,r)
4     Quicksort(A,p,q-1,k)
5     Quicksort(A,q+1,r,k)
6   end
7 }
```

   In this case, the Quicksort function take variable k as error distance, the $while(p < r - k)$ statement means changing the recursion to have k error distance. The Quicksort's best case scenario for Quicksort call is $O(\log_2 n)$, similarly, since we have error distance k in this modified algorithm, the recursion stops when it equal to the error distance. In other words, there are $O(\log_2 k)$ Quicksort calls being stopped compared to the original Quicksort calls. Therefore, subtract $\log_2 k$ from $\log_2 n$: $O(\log_2 n - \log_2 k)$. Since the length is n and Quicksort go through the entire input array, therefore the overall best case running time of this modified Quicksort is $O(n \cdot (\log_2 n - \log_2 k))$.

   b) Since the error distance is k, the Insertion sort needs to swap all element pairs that have error distance k, so there are k swap calls in each loop. While the Insertion sort also go through the entire array, therefore the overall running time is $O(n \cdot k)$.

2. **a)** If an element y being swapped in every partition loop, then the total swap number is $n - 1$, therefore the maximum value of $N(y) = O(n - 1)$. For example, if an array's first element is the largest, the last element is the second largest and the rest elements between them are smaller. Choose the last element as pivot, then for all partition loop the first(largest) element will be swapped, which is $n - 1$ times.

b) If an array's first element is the largest, the last element is the second largest and the rest elements between them are smaller. Then the maximum value of $N(y) = O(n-1)$, while the other $n-1$ elements in the array will be swapped once. Therefore the total swap time will be $2 \cdot (n-1)$, on the average the worst-case upper-bound is $2 \cdot (n-1)/n$.

3. **a)**

```
1 Heap-Extract-Max(A)
2   Remove A[1]
3   A[1]=A[n]
4   n=n-1
5   Heapify(a,1,n)
6 End Heap-Extract-Max
```

Although it is called $Heap - Extract - Max()$, since both heaps are min heap, the function actually remove the minimum value.

```
1 For n = 0 to 1:
2   x_{0} = Heap-Extract-Max(a), y_{0} = Heap-Extract-Max(b)
3   x_{1} = Heap-Extract-Max(a), y_{1} = Heap-Extract-Max(b)
4 End
5 result = Quicksort([x_{0},x_{1},y_{0},y_{1}], p, r)
6 Return result[0], result[1]
```

Call $Heap - Extract - Max()$ function for each heap twice, record the two smallest keys from both heaps and use Quicksort to sort the 4 values. Return the two smallest values which are the smallest and the second smallest of $2n$ keys in both heaps. The complexity for this function is $Heap - Extrac - Max()$ function calls $Heapify()$ which is $O(\log n)$.

b) Using the Heap-Extract-Max() function above.

```
1 While !(a and b are both empty):
2   x = Heap-Extract-Max(a)
3   y = Heap-Extract-Max(b)
4 End
5 If x>y
6   Return x
7 else
8   Return y
```

The x and y store the last value of heap a and b before they go empty, return the larger value of x and y which is the largest of $2n$ keys in both heaps. The complexity for this function is $Heap-Extrac-Max()$ function calls $Heapify()$ which is $O(2 \cdot \log n)$, and it go through all n, therefore it is $O(n \log n)$

c)

```
1 Let newRoot = -infinity
2 Let newHeap[0] = newRoot
3 For each key level:
4   newHeap.add(a.currentLevelElements)
5   newHead.add(b.currentLevelElements)
6 End
7 Heap-Extract-Max(newHeap)
```

Define $-\infty$ as the new root of the new heap. Therefore a and b are the children of the root, left and right sub-trees are heaps. To construct the heap, copy and paste and keys in the same level of a and b to the new heap, it takes the constant time. After that, call Heap-Extract-Max(newHeap) to remove the $-\infty$ root, and the $Heap-Extract-Max(newHeap)$ function calls $Heapify()$, which is $O(\log n)$ complexity. Therefore the overall complexity is $O(\log n)$.

d)

```
1 Use the algorithm in previous question to merge both heaps
2 For i = 0 to 2n:
3   result.add(Heap-Extrac-Max(A))
4 End
5 Return result
```

Merge both heaps to one heap in order to use $Heap-Extrac-Max()$ function, record the root each time to generate the sorted 2n keys in a list. The merging process takes $O(\log n)$, the for loop and Heap-Extrac-Max() function takes $O(2n \cdot \log n)$, therefore the overall complexity is $O(\log n + 2n \cdot \log n) = O(n \log n)$

4.
```
1 Select(A,p,q,i){
2   Divide A to n/5 groups of size 5
3   Find the median of each group of 5 by brute force
      and store them in a set A' of size n/5
4   Use Select(A',1,n/5,n/10) to find the median x of n/5 medians
5   Partition the n elements around x
6   y.add([i,x])
7   if(i==1):
8     return y
9   Select(A,p,(q/2)-1,(i/2))
10 }
11 y.add(max(A))
```

Since $n = 2^k$, we can let $i = i/2$ because in each recursion $2^k/2 = 2^{k-1}$ which is the order statistics we need. In each recursion, the size of the queue becomes half, x is the median of n/5 medians, which is the order statistics we are looking for. Therefore, save the data sets $[i, x]$ in each recursion. Since the resursion applied from the beginning, the 2^k order

statistic is not contained in the result list y, therefore use a function $max()$ to search forthe largest number in A because $n = 2^k$. Since the recursion go through all elements in array, and the $max()$ function takes constant time, the overall complexity is $\theta(n)$.

5.
```
1 Select(A,p,q,i,k){
2   Divide A to n/5 groups of size 5
3   Find the median of each group of 5 by brute force
      and store them in a set A' of size n/5
4   Use Select(A',1,n/5,n/10) to find the median x of n/5 medians
5   Partition the n elements around x
7   if(i==(n-k)):
8     return rightside
9   if(i>(n-k)):
11    Select(A,k,q,i-(n-k),k)
12 }
13 Quicksort(rightside,p,r)
```

After the $Select()$ function, we find the the $n - k$ order statistic, which means the right side of the queue has k elements larger than $n-k$. Therefore when $i == (n - k)$, return the right side of the array and then use Quicksort to sort the k largest elements in the list. The worst case scenario for $Select()$ is $\theta(n)$, the complexity of Quicksort is $\theta(n \log n)$, since the n for Quicksort is actually k here, therefore the overall worst-case complexity is $\theta(n + k \log k)$.