Algorithm assignment 3
Nisharg Gosai

Problems

1. (15pts) The operation MAX-HEAP-DELETE(A,x) of a max-heap priority queue deletes the object x from max-heap A. Give an implementation in pseudocode of MAX-HEAP-DELETE for an n-element max-heap that runs in O(lgn) time plus the overhead for mapping priority queue objects to array indices.

```
MAX-HEAP-DELETE(A, x)
1. index = MAP-TO-INDEX(x)  // Find the index of x in A
2. if index is NOT FOUND
3.     return ERROR "Element not found in the heap"
4. lastElement = A[HEAP-SIZE[A]]
5. A[index] = lastElement  // Replace x with the last element
6. HEAP-SIZE[A] = HEAP-SIZE[A] - 1  // Reduce the heap size
7. if lastElement > A[PARENT(index)]
8.     HEAP-INCREASE-KEY(A, index, lastElement)
9. else
10.    MAX-HEAPIFY(A, index)
11. return SUCCESS
```

2. (15pts) The worst-case running time for quicksort is given by the recurrence T(n) = T(n-1) +Θ(n). Use the substitution method to prove that this recurrence has the solution T(n) = Θ(n2), as claimed at the beginning of section 7.2.

To prove that the recurrence is $\Theta(n^2)$, we need to prove that the recurrence is $\Omega(n^2)$ and $O(n^2)$,

Upper bound

T(n) = $O(n^2)$

Let $\Theta(n) = dn$

For k<n,

T(k)<=c*k^2 (assumption)

T(n)<=cn^2 for n>k and c>0

Therefore, T(n-1)<=c$(n-1)^2$

T(n) = T(n-1) + $\Theta(n)$

<= c$(n-1)^2$ +dn

= c(n^2-2n+1)+dn

=cn^2-2cn+c+dn

Since, -2cn<dn, 2c>d and n>= $\frac{c}{2c-d}$

<=cn^2

Therefore, T(n)<=cn^2……..(A)

Lower bound

T(n) = $\Omega(n^2)$

For k<n,

T(k)>=c*k^2 (assumption)

T(n)>=cn^2 for n>k and c>0

Therefore, T(n-1)>=c$(n-1)^2$

T(n) = T(n-1) + $\Theta(n)$

>= c$(n-1)^2$ +dn

= c(n^2-2n+1)+dn

=cn^2-2cn+c+dn

Since, 2c<d and n<= $\frac{c}{2c-d}$

>=cn^2

Therefore, T(n)>=cn^2……..(B)

Using A and B, we proved that the recurrence is $\Theta(n^2)$

3. (20pts) In quicksort, the PARTITION procedure returns an index q such that each element of A[p:q-1] is less or equal to A[q] and each element of A[q+1 : r] is greater than A[q]. Modify the PARTITION procedure to produce a procedure PARTITION'(A,p,r), which permutes the elements of A[p : r] and returns indices q and t, where p <= q <= t <= r, such that

• all elements of A[q : t] are equal,
• each element of A[p : q-1] is less than A[q], and
• each element of A[t+1 : r] is greater than A[q].

Like PARTITION, your PARTITION' procedure should take Θ(r-p) time.

```
PARTITION'(A, p, r)
1. pivot = A[r]                  // Choose a pivot element, here we choose the last element
2. q = p
3. t = r - 1
4. i = p
5. while i <= t
6.     if A[i] < pivot
7.         swap A[i] and A[q]
8.         q = q + 1
9.         i = i + 1
10.    else if A[i] > pivot
11.        swap A[i] and A[t]
12.        t = t - 1
13.    else
14.        i = i + 1
15. swap A[t + 1] and A[r]  // Place pivot between less-than and greater-than partitions
16. t = t + 1                    // Adjust t to include pivot
17. return q, t
```

4. (20pts) Show that there is no comparison sort whose running time is linear for at least half of the n! inputs of length n. What about a fraction of 1/n of the inputs of length n? What about a fraction 1/2n?

From the discussion before,
A binary tree with n elements has height h has L<=2^h leaves,

Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements.

Because each of the n! permutations of the input appear as one or more leaves,
we have n!<= L .

Since a binary tree of height h has no more than 2^h leaves, we have
n!<=L<=2^h
which, by taking logarithms, implies h>=lg(n!)

1) For at least half of the n!. We have,

   $\frac{n!}{2} <= n! <= L <= 2^h$

   By taking logarithm on both sides,

   $h >= lg(\frac{n!}{2}) = lg(n!)/lg2 = lg(n!) - 1 = \Theta(nlgn) - 1 = \Theta(nlgn)$,

   Therefore, there is no comparison sort whose running time is linear for at least half of n! Inputs of length n.

2) For 1/n of the n! Inputs of length n. We have,

   $(\frac{1}{n})n! <= n! <= L <= 2^h$

   By taking logarithms on both sides,

   $h >= lg(\frac{n!}{n}) = lg(n!) - lgn = \Theta(nlgn) - lgn = \Theta(nlgn)$

   Therefore, there is no comparison sort whose running time is linear for 1/n of the n! Inputs of length n.

3) For $\frac{1}{2^n}$ of the n! Inputs of length n. We have,

$(\frac{1}{2^n})n! <= n! <= L = 2^h$

By taking logarithms on both sides,

$h >= lg(\frac{n!}{2^n}) = lg(n!) - n = \Theta(nlgn) - n = \Theta(nlgn)$

Therefore, there is no comparison sort whose running time is linear for $\frac{1}{2^n}$ of the n! Inputs of length n

5. (10pts) Describe (in English, not pseudocode) an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range [a:b] in O(1) time. Your algorithm should use Θ(n + k) preprocessing time.

So our algorithm will work similarly to how the Counting Sort algorithm works

Lines 7-8 of the Counting Sort algorithm determine for each i=0,1,..,k how many input elements are less than or equal to i by keeping a running sum of the array C.

We can simply get how many of the n integers fall into the range[a:b] by calculating C[b]-C[a-1] after lines 7-8.

This will take constant time.

6. (15pts) Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any comparison sort stable. How much additional time and space does your scheme entail?

Stable: Insertion and Merge sort
Not stable: Heap sort and Quicksort

To make any comparison sort stable, we can simply preprocess the input of elements with pairs of elements and their index, so the input will be (elements, index of that element)

For example if we have an array [7,3,2,1,2], it will become [(7,1),(3,2),(2,3),(1,4),(2,5)]

So if we have the same element, for example 2 we can compare their indices,
(2,3) and (2,5)
3<5 so (2,3) will come first

The running is asymptotically the same and the space is doubled for this scheme.

7. (10pts) Describe (in English, not pseudocode) how to sort n integers in the range 0 to n3 – 1 in O(n) time.

We can sort in O(n) using radix sort, since range is given for n integers we can sort from their least significant digit to the most significant digit.

Do for each digit i where i varies from from least significant digit to most significant digit,
-Sort input array on i using a stable sorting algortihm like counting sort