CS5800: Algorithms — Iraklis Tsekourakis

Homework 3 Name: Avinash R. Arutla

1. (15 points)

Solution:

```
MAX-HEAP-DELETE(A,x):

    index = MAP_TO_INDEX(x) // This function maps every element with an index number

    if index == A.heap_size:
        A.heap_size -=1
        return x

    swap A[index], A[A.heap-size] //swap x with the last element
    A.heap-size -=1

    if A[index] > A[parent[index]]:
        MAX-HEAPIFY-UP(A,index)
    else:
        MAX-HEAPIFY(A, index)
    return x

    MAX-HEAPIFY(A,i) :
    l = LEFT(i) // returns the index of the left child
    r = RIGHT(i) // returns the index of the right child

    l $\leq$ heap-size(A) and A[l] $>$ A[i]
        largest = l

    r $\leq$ heap-size(A) and A[r] $>$ A[i]
        largest = r

    if largest!=i
        swap A[i] with A[largest]
        MAX-HEAPIFY(A,largest)

    MAX-HEAPIFY-UP(A,i) :
        while i>1 and A[PARENT(i)] < A[i] :
            swap A[i] with A{PARENT(i)}
            i = PARENT(i)
```

```
PARENT(i):
    return i/2

LEFT(i):
    return 2*i

RIGHT(i):
    return 2*i+1
```

Here in the above problem, the max time taken by the algorithm to max-heapify is to replace the element x with an appropriate element, so that it satisfies the max-heap property.

So since we are traversing through the tree, the children are generated in $log_2 n$, the base 2 is because every node has 2 children, so we have $O(log_2)n$

2. (15 points)

Solution:
The given recurrence equation is

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

$\theta(n)$ can be written as a(n)

$$T(n) = T(n-1) + a(n)$$

To prove that the worst case running time is $\theta(n)$, we can write it as

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

So we can write $T(n) = \theta(n^2)$ as

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

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

To prove $T(n) = O(n^2)$,

So here $T(n) \leq cn^2$, where $c \geq 0$

So from $T(n) = T(n-1) + \theta(n)$, we replace T(n) with T(n-1)

$T(n-1) \leq c(n-1)^2$

Substituting the above equation in $T(n) = T(n-1) + an$

So we get,

$$T(n) \leq c(n-1)^2 + an \leq cn^2$$

$$T(n) \le c(n^2 - 2n + 1) + an \le cn^2$$
$$T(n) \le cn^2 - 2nc + c + an \le cn^2$$
$$T(n) \le cn^2 - (a - 2c)n + c \le cn^2$$
$$T(n) \le (a - 2c)n + c \le cn^2$$
$$n \ge \frac{c}{a - 2c}$$

Here we clearly know that, n needs to have values which are greater than 0.

So we can write,

$$n \ge \frac{c}{a - 2c} \ge 0$$

, here the denominator needs to be positive for the value to hold.

So we can write it as,

$$a - 2c > 0$$
$$a > 2c$$
$$\frac{a}{2} > c$$

Thus, for $T(n) = O(n^2)$, the conditions should be $\frac{a}{2} > c$ and $n \ge \frac{c}{a-2c}$ for it to stay valid.

Now we need to prove $T(n) = \Omega(n^2)$, for $T(n) \ge cn^2$ and $c > 0$

So similar to the above equation, we can replace n with n-1

$$T(n) \ge c(n-1)^2$$

So we can write

$$T(n) \ge c(n-1)^2 + an \ge cn^2$$
$$T(n) \ge c(n^2 - 2n + 1) + an \ge cn^2$$
$$T(n) \ge cn^2 - 2nc + c + an \ge cn^2$$
$$T(n) \ge cn^2 - (a - 2c)n + c \ge cn^2$$
$$T(n) \ge (a - 2c)n + c \ge 0$$
$$n(a - 2c) \le c$$
$$n \le \frac{c}{a - 2c}$$

Here c is always positive, since we are aware of the condition that $c \ge 0$

So, we need to prove that $a - 2c \ge 0$

$$a \ge 2c$$

Thus, $T(n) = \Omega(n^2)$ only when $a \ge 2c$

Thus, from both the above equations, we can prove that
$T(n) = O(n^2) and T(n) = \Omega(n^2)$

So hence, $T(n) = \theta(n^2)$

3. (20 points)

Solution:

```
PARITION(A,p,r):

    // A -> array A which needs to be sorted
    // p,r -> satarting index, and ending index

    pivot = A[r] // last element as pivot
    index = p-1 // elements before index are elements which are less than pivot

    p_count = 0

    p_end = p-1 // tracking end of section which is less than pivot
    p_start = p -1 // tracking start of section whcih is more than pivot

    for(p<=j<=r-1):

        if(A[j] < pivot):

        p_end +=1
        p_start +=1 //move elements to the start of array which are less than pivot

        swap(A[p_end], A[j])

        if p_end != p_start:
            swap(A[p_start], A[j]) #checks if there is a section of elements which are equa

        else if A[j] == pivot:
            p_start +=1
            swap(A[p_start], A[j])

    swap(A[p_start], A[r]) // placing all elements which are less than pivot, before it and

    return p_end+1, p_start+1
```

This algorithm takes $\theta(r - p)$ time to complete

4. (20 points)

Solution:

Given an arary n , if a decision tree were to be created, the decision tree would have l leaves.

So we can write $n! \leq l$

Here, from the rest of the tree, if we traversde to any leaf of the decision tree, since there are positive comparisons that can be made by a sorting algorithm.

So,

$$l \leq 2^d$$

(where d is depth of tree)

from the other equations, we can write

$$n! \leq l \leq 2^d$$

Now let us see if the running time is linear for atleast half of the inputs

In this equation, $n \leq l \leq 2^d$, we replace n! with $\frac{n!}{2}$

$$\frac{n!}{2} \leq l \leq 2^d$$

$$\frac{n!}{2} \leq 2^d$$

Applying log on both sides, we get

$$log\frac{n!}{2} \leq log(2^d)$$

$$n! - 1 \leq d$$

wkt $n! > (\frac{n}{e})^n$

So to further solve this, we write

$$(\frac{n}{e})^n - 1 \leq d$$

$$n\frac{n}{e} - 1 \leq d$$

$$nn - e - 1 \leq d$$

$$n^2 - e - 1 \leq d$$

As n is asymptotically greater than n, we can say that

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

So this means that elements can only be sorted in linear time.

WE have another case, where we need to check if the running time is linear for $\frac{1}{n}$ of inputs.

So from equation, $n! \leq l \leq 2^d$

we replace, $\frac{n!}{n}$

$$\frac{n!}{n} \leq l \leq 2^d$$

so, we can write

$$\frac{n!}{n} \leq 2^d$$

Applying log on both sides

$$log\frac{n!}{n} \leq log(2^d)$$

$$n! - n \leq d$$

Similar to the above problem, the sterlings approximation comes into place, $n! > (\frac{n}{e})^n$
So we can write,

$$(\frac{n}{e})^n \leq d$$

$$n - e - n \leq d$$

As the above equation, $\Omega(n^2)$, is greater than n, $d = \Omega(n^2)$, so elements cannot be sorted in linear time.

5. 10 points

Solution:
Step 1: Initialization

- **Input:** An array `arr` of n integers, where each integer is in the range 0 to k.
- **Output:** The number of integers within any query range $[a : b]$ in constant time.
- **Auxiliary Data Structure:** Create an auxiliary array `count` of size $k + 1$ initialized with zeros. This array will keep the count of each integer in the range 0 to k in the original array.

Step 2: Preprocessing - Building the Count Array

1. Iterate through the array `arr`, for each element `arr[i]`, increment `count[arr[i]]` by 1. After this step, `count[i]` will hold the total occurrences of the integer i in the original array.
2. Transform the count array into a prefix sum array: Create another array `prefixSum` of size $k+1$. Set `prefixSum[0] = count[0]`, then for each index i from 1 to k, calculate `prefixSum[i] = prefixSum[i-1] + count[i]`. This step aggregates the counts, and after its completion, `prefixSum[i]` will hold the total count of integers in the range 0 to i inclusive.

Step 3: Answering Queries

- If $a = 0$, then the answer is `prefixSum[b]` because it already represents the count of all numbers from 0 to b.
- If $a > 0$, the answer is `prefixSum[b] - prefixSum[a-1]`. This calculation gives the count of integers in the range $[a : b]$ by removing the count of integers that fall before a.

Time Complexity Analysis

- **Preprocessing Time:** The time to build the count array is $\Theta(n)$, as we go through each of the n elements once. Transforming the count array into the `prefixSum` array takes $\Theta(k)$ time, as we need to iterate through the $k+1$ elements of the count array. Therefore, the total preprocessing time is $\Theta(n+k)$.
- **Query Time:** Since the query involves only a constant number of arithmetic operations and array accesses, it can be answered in $\Theta(1)$ time.

Space Complexity

- The space complexity is $\Theta(k)$, due to the additional storage required for the count and `prefixSum` arrays.

6. (15 points)

Solution:

Stability in sorting algorithms is a property that ensures when two elements with equal keys are sorted, their original relative order is preserved.
Insertion Sort: This algorithm is stable. It compares each new element to the already sorted ones, inserting it into its correct position. Equal elements are not reordered, preserving their relative order.
Merge Sort: This algorithm is also stable. During the merge process, when two elements from different halves compare equal, the element from the left half (which comes first in the original array) is always chosen first, preserving the order of equal elements.
Heapsort: This algorithm is not stable. The process of building and adjusting the heap does not preserve the original order of equal elements because it prioritizes heap properties over the original ordering.
Quicksort: By default, quicksort is not stable. The partitioning step can rearrange equal elements, not preserving their original order. However, with careful implementation, it can be made stable, but that often involves sacrificing some of its in-place sorting benefits.

Time Cost: The augmentation and de-augmentation steps each require O(n) time, where n is the number of elements. The sorting itself depends on the algorithm used, but the comparison between augmented elements only adds a constant time overhead to each comparison operation. Thus, the total additional time complexity is O(n).

Space Cost: Augmenting the elements requires storing an additional piece of information (the original index) for each element. This means an additional O(n) space is needed, assuming the index can be stored in a space comparable to the size of the elements being sorted.

7. (10 points)

Solution:

To sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time, a specific approach utilizing Radix Sort is employed. This method capitalizes on the observation that when numbers are represented in base n, each integer within our range can have at most three digits.

1. Initially, we represent each integer within the specified range in base n. This transformation is important because it ensures that any number from 0 to $n^3 - 1$ will have no more than three digits in this base, setting the stage for Radix Sort.

2. Proceed to apply Radix Sort, focusing on one digit at a time, starting from the least significant digit (LSD) to the most significant digit (MSD). For each digit's sorting process:

   - We employ a stable, linear-time sorting algorithm like Counting Sort. The reason behind selecting Counting Sort lies in its capability to efficiently sort numbers based on their digit values in $O(n)$ time, which is crucial for maintaining the overall time complexity of $O(n)$.
   - Given that the numbers are represented in base n, and considering the defined range, this digit-based sorting operation is performed precisely three times, because of the three digits in the base n representation.

3. Upon sorting based on the third and most significant digit, the sorting process concludes with the numbers being fully ordered from 0 to $n^3 - 1$. Through its systematic digit-by-digit sorting, Radix Sort ensures the sequence is correctly arranged in the desired order.