CS5800: Algorithms — Iraklis Tsekourakis

Homework 4 Name: Avinash R. Arutla

1. (10 points)

Solution:

Bucket sorts worst case scenario $\theta(n)$ occurs when all the elements fall into a single bucket. If the sorting algorithm which is used here is a quadratic time sorting algorithm such as insertion sort, which is used in small sets due to its low overhead, the complexity increases to $\theta(n^2)$.

Here bucket sort assumes that the elements are divided into buckets uniformly, however when elements are skewed and have uninform distribution, the classic bucket sort assumption fails, leading to unevenly distributed buckets.

To preserve the linear average case running time of bucket sort and its worst case running time to $O(nlogn)$, we can use a different sorting algorithm other than the classic quadratic time sorting algorithm. We can use any sorting algorithm like merge sort or quick sort which have a better worst case running time than insertion sort.

In such a case, in the worst case running time, the complexity is reduced from $O(n^2)$ to $O(nlogn)$. This ensures the algorithm completes completes in linear time, In case of normal running scenario, the overhead introduced by the algorithm is reduced by evenly spread buckets with lesser number of elements, which makes the difference due to overhead negligible.

2. (15 points)

Solution:

Here N is the random variable, equal to the number of flips for two coins
So, there are total 4 possible outcomes

$$HHTTHTTH$$

So, the expectation of X can be written as

$$E[X] = \sum x_i P(x_i)$$

$$E[X] = 0\frac{1}{4} + 1\frac{1}{2} + 2\frac{1}{4}$$

This is because,

P(X=0) = both tails (1/4)

P(X=1) = one head, one tail (1/2)

P(X=2) = both heads (1/4)

$$E[X^2] = \sum x_i^2$$

$$E[X^2] = 0^2\frac{1}{4} + 1^2\frac{1}{2} + 2^2\frac{1}{4}$$
$$E[X^2] = 0 + \frac{1}{2} + 1$$
$$E[X^2] = \frac{3}{2}$$

$$E^2[X] = E[X] * E[X]$$
$$E^2[X] = 1 * 1$$
$$E^2[X] = 1$$

3. (10 points)

Solution:

The problem can be approached by finding the minimum and maximum numbers of the given set/array, and then finding any other number from the set which satisfies the minimum nor maximum condition. However this approach is not suitable even though we break it down into smaller components.

Instead we can use the tournament method, which aims to identify numbers the numbers which is not the smallest or largest with minimal comparisons.

Pairwise Comparisons

Divide the n numbers into pairs and compare each pair, which takes $\frac{n}{2}$ comparisons. This step effectively identifies candidates for being neither the minimum nor the maximum, as one number in each pair is not the smallest, and the other is not the largest in that pair.

Finding Potential Candidates

From each pair, select the ”loser” for the next round of comparisons to find the maximum of these ”losers” (or the minimum for the ”winners”). This step ensures that the number chosen after all comparisons is not the overall minimum or maximum.

Assuming n is even, you will have $\frac{n}{2}$ losers and $\frac{n}{2}$ winners. It takes $\frac{n}{2} - 1$ comparisons to find the maximum among the losers or the minimum among the winners.

Total Comparisons

The total number of comparisons for n numbers using this method is:

- $\frac{n}{2}$ comparisons for the initial pairing.
- $\frac{n}{2} - 1$ comparisons to find a non-minimum/maximum among the ”losers” or ”winners.”

So, the total number of comparisons is $\frac{n}{2} + \frac{n}{2} - 1 = n - 1$.

For $n > 2$ distinct numbers, to find a number that is neither the minimum nor the maximum, the smallest number of comparisons needed is $n - 1$. This method efficiently narrows down a candidate that satisfies the conditions without needing to explicitly find both the minimum and maximum values first.

4. (15 points)

Solution:

The concept of RANDOMZED-SELECT is to select an element at random, and divide the array into two subarrays, elements less than the pivot and greater than the pivot. Then it recursively calls itself on the subarray part where it has the ith smallest element.

RANDOMIZED-SELECT never makes a recursive call to a zero length array because of the following reasons

1. The base case here is, it checks if the length of array is 1, if so it returns the one element cause there are no comparisons to make.
2. When the array is partitioned, it makes two non overlapping subarrays according to the pivot, where one array is smaller than pivot and one is larger than the pivot.
3. The recursive calls are made on the conditions where, i¡k and i¿k, where k = 1-p+1
4. Here q = pivots position, p = starting index and k= length of left subarray
5. Left array case : When i¡k, the algorithm recursively parses the left subarray
6. Right array case : When i¿k, the algorithm parses the right subarray, adjusting i and i-k to accomodate new index.
7. When we term this mathematically, the algorithm works only when there are elements in the left or right subarrays to make the comparisons i¡k and i¿k. If there are no elements then there are no comparisons to make, and hence the algorithm doesnt work.

5. (20 points)

Solution:

```
RANDOMIZED-PARTITION(A,p,r):

    temp = RANDOM(p-1,r)
    swap(A[temp],A[r])
```

```
    return PARTITION(A,p,r)

RANDOMIZED-SELECT(A,p,r,i)

    while p<r:
        q = RANDOMIZED-PARTITION(A,p,r)
        holder = q-p+1

        if i== holder:
            return A[q]

        else if i<holder:
            r = q-1

        else:
            i-=hold
            p=q+1
        return A[p]

RANDOMIZED-SELECT(A,p,r,i)
```

6. (20 points)

Solution:

To implement a queue using two stacks, we utilize the fundamental operations of stacks—push (to add an item) and pop (to remove the top item)—to mimic the enqueue (add to the back) and dequeue (remove from the front) operations of a queue. The key idea is to use one stack for enqueuing elements and another stack for dequeuing elements. When we need to dequeue an element, we reverse the order of the enqueued elements by moving them to the second stack, making the front element accessible (due to the Last-In-First-Out property of stacks).

```
Stack EnqueueStack, DequeueStack

Function Enqueue(element)
    Push(EnqueueStack, element)

Function Dequeue()
    If IsEmpty(DequeueStack)
        While Not IsEmpty(EnqueueStack)
            Push(DequeueStack, Pop(EnqueueStack))
    If IsEmpty(DequeueStack)
        Error "Queue is empty"
```

```
Return Pop(DequeueStack)
```

Enqueue Operation : The enqueue operation takes a single push operation which takes $O(1)$ time.

Dequeue Operation : The dequeue operation usually takes $O(1)$ time, but in worst case scenario where the dequeue stack is empty, it might take $O(n)$ time because every element from the enqueue stack needs to be moved to the dequeue stack. However this overhead is amortized because for each operation only one element is moved before its dequeued. so technically the time complexity still stays at O(1).

In summary, using two stacks to implement a queue results in $O(1)$ amortized time per enqueue and dequeue operation, making it an efficient data structure for queue operations over a long sequence of actions.

7. (20 points)

Solution:

```
REVERSE-LL(head):

    prev = NULL
    next = NULL
    current = head

    while current!= NULL
        next = current.next
        current.next = prev
        prev = current
        current = next

    head = prev
    return head
```

The general algorithm of working is, the loop will iterate till the current element is equal to NULL, where the current element will be linked to the previous elemenet and the next element is similarly linked to the previous element which was the current element before. The loop then runs till the final element is reached and by that time the LL is reveresed and the final element is assigned as the HEAD of the Linked List