Name: Prasham Dhaneshbhai Sheth
UNI: pds2136
Session: 00x
Collaborators: -

Hw1—Theoretical part

1. **Solution to problem 1**

Algorithm 1

`Horner's rule`

1: $z = a_n$
2: **for** $i = n - 1$ down to 0 **do**
3: $\quad z = zx + a_i$
4: **end for**

(a) The idea behind the given algorithm is to split the given polynomial $a_n x^n + a_{n-1}x^{n-1} + ... + a_2x^2 + a_1x + a_0$ into $(((a_nx + a_{n-1})x + a_{n-2})x... + a_1)x + a_0$

For proving that the algorithm correctly evaluates the polynomial at given x; we will prove the algorithm by induction.

- Base: Considering n=0 as the base case. For this case the polynomial is a_0.
  Looking at the algorithm, the value of a_0 is stored in z. going to line 2 the for loop needs to run from -1 to 0. the for loop thus, won't run. and hence z at the end of algorithm would have a_0 in it.
  Hence the algorithm works for the base case.
- Step: In the given algorithm, inside the for loop we can conclude that the z variable is updated such that it takes the previous value of z and multiplies it with x and adds a_i to it.
  Now for a given, z is initialized with a_n and when it goes into the loop, after the first iteration $z = z * x + a_i$ that is $z = a_nx + an - 1$.
  Similarly after loop 2, $z = a_nx^2 + a_{n-1}x + a_{n-2}$ From this we can say that after the loop i ends, z has the value $\Sigma a_{i+j} * x^j$ such that $n - i >= j >= 0$
- We know that the counter i of the loop decrements and hence the loop terminates when i=-1. Thus, the value of polynomial in z at termination of loop is $\Sigma a_j * x^j$ such that $n >= j >= 0$. Evaluating this would give the required polynomial. Hence, at the end of the loop the value of z will be the value of polynomial at x as $\Sigma a_j * x^j$ such that $n >= j >= 0$ is $a_nx^n + a_{n-1}x^{n-1} + ... + a_2x^2 + a_1x + a_0$.

For the number of additions and multiplications, the value is equal to number of iterations as in each iteration one multiplication and one addition is performed. Thus, here as number of loops is n-1+1 = n, n multiplications and n additions are performed.

(b) The Horner's method of polynomial evaluation works efficiently when we are to solve for value of polynomials. But for some of the polynomials calculating each and every term may not be useful. Suppose for a polynomial p all of its $a_i = 0$ except for a_n. In such a case we can repeatedly square the polynomial term to obtain the value of polynomial.

Algorithm 2

```
Algorithm1(Ans,Degree)
```

1: $z = a_n$
2: **if** Degree=1 **then**
3: **return** Ans
4: **else if** Degree is even **then**
5: **return** Algorithm1(Ans*Ans, Degree / 2)
6: **else**
7: **return** Ans* Algorithm(Ans*Ans, Degree/2)
8: **end if**

The above algorithm evaluates polynomial of the form $a_n * x^n$ in O(logn) time while solving by Horner's Rule will take O(n) time.

2. **Solution to problem 2**

Algorithm 3

```
Recursive_Sorting (List A, Integer low, Integer high)
```

1: **if** $size(A) = 0$ *or* $size(A) = 1$ **then**
2: return A
3: **else if** $size(A) = 2$ **then**
4: **if** $A[1] <= A[2]$ **then**
5: return A
6: **else**
7: Swap(A[1],A[2])
8: **end if**
9: **end if**
10: FirstPhaseLim = $low + \lceil(2/3 * (high - low))\rceil$
11: SecondPhaseLim =$high - \lceil 2/3 * (high - low)\rceil$
12: $Recursive\_Sorting(A, low, FirstPhaseLim)$
13: $Recursive\_Sorting(A, SecondPhaseLim, high)$
14: $Recursive\_Sorting(A, low, FirstPhaseLim)$

- Correctness of this algorithm, we will use induction method:

  Base Case: Consider the case when array is of Size 1. Trivially the array is sorted and algorithm returns the array as it is. For the case when array is of size 2, the algorithm compares both elements and returns a sorted array.

  Hypothesis: The algorithm correctly sorts the array of size 3i; 1¡=i¡=n

  Step: The algorithm can be proved correct if we can prove here that it works for size 3(i+1)

For array of size 3(i+1): The algorithm first splits it into $2/3^{rd}$ and calls the algorithm recursively for sorting the first $2/3^{rd}$ elements. As, size of this small array can be represented as 3j where 1¡=j¡=n; by hypothesis the first $2/3^{rd}$ elements would be sorted after this call.

Also, we can say that as first $2/3^{rd}$ elements are sorted; the second $1/2^{th}$ elements of these $2/3^{rd}$ elements are the greater ones.

After the recurrence call for first $2/3^{rd}$ elements is made; recurrence call to array of last $2/3^{rd}$ elements is made in the second phase. By hypothesis again; this array of $2/3^{rd}$ elements would be sorted.

Now, as after first phase the second half of first $2/3^{rd}$ elements was having greater elements of the first $2/3^{rd}$ parts, by a similar logic after second phase second half of last $2/3^{rd}$ elements would have all the greater elements of the array.

Hence, now if we symbolize the array after dividing it into 3 equal parts; A,B,C: C would be having all the elements that are greater than $2/3^{rd}$ elements of array in thier respective places. (Because after first phase B has elements that are greater than elements in A; and after secondphase C has to have the greater elements)

Now, in third phase, the first $2/3^{rd}$ elements are again sorted using the recursive call and by hypothesis, they are sorted correctly. Hence, now B will have all elements which are greater than A in their respective positions.

Hence, the final array (A:B:C) would be sorted.

- Recurrence Relation for running time of this algorithm is as given below:
  $T(n) = 3T(2n/3) + c$
  For finding the asymptotic bounds of this algorithm using this recurrence relation, we can use Master Theorem.
  Here, k = 0, a = 3 , b = 3/2 = 1.5
  Using the master theorem as $a > b^k$ T(n) = $O(n^{log_{1.5}3}) = O(n^{log 2.709)}$
- No, I would not use this algorithm for next applications as randomized quick sort has an expected running time of O(nlogn) which is far better than the running time of this algorithm.

3. **Solution to problem 3**

f	g	O	o	Ω	ω	Θ
$n \log^2 n$	$6n^2 \log n$	Yes	Yes	No	No	No
$\sqrt{\log n}$	$(\log \log n)^3$	No	No	Yes	Yes	No
$4 \log n$	$n \log 4n$	Yes	Yes	No	No	No
$n^{3/5}$	$\sqrt{n} \log n$	No	No	Yes	Yes	No
$5\sqrt{n} + \log n$	$2\sqrt{n}$	Yes	No	Yes	No	Yes
$\frac{5^n}{n^8}$	$n^5 4^n$	No	No	Yes	Yes	No
$\sqrt{n}2^n$	$2^{n/2+\log n}$	No	No	Yes	Yes	No
$n \log 2n$	$\frac{n^2}{\log n}$	Yes	Yes	No	No	No
$n!$	2^n	No	No	Yes	Yes	No
$\log n!$	$\log n^n$	Yes	No	Yes	No	Yes

4. **Solution to problem 4**

(a) The algorithm would return different outputs for same given input depending upon the randomly generated item. Thus, it is a kind of Monte-Carlo Algorithms.

The first 2 lines takes constant execution time. The statements inside for loop are executed for n times. The remaining statements take constant execution time.

Hence, T(n) = O(n)

Thus, the randomized Approximate Median is a Monte-Carlo algorithm and has a running time of O(n).

(b) For the success of algorithm, the number to be chosen randomly should be greater than at least n/4 elements and smaller than at least n/4 elements. If we consider a sorted array than the number should be between the middle two quarters.

The probability of selecting a number from those 2 quarters is 1/2. Thus, the success probability of the algorithm is 0.5.

(c) The failure probability of the algorithm from the previous answer would be 1-0.5=0.5. Hence, if we want to increase the success probability, we can repeat the same algorithm for a fixed number of iterations or else continue to call the algorithm till we get the required output.

Considering the case we run it for fixed number of times:

let m be the number of times the algorithm is run, hence the complexity of the new algorithm would be O(m*n) as the algorithm has been called m times.

The probability of getting at least one correct output from these m runs is the probability of success for whole algorithm.

The probability of getting at least one success out of m trials is 1 - (probability of getting failure in all cases).

Algorithm 4

```
Improved Randomized Approximate Median (S)
```

1: $m = 7$ ▷ *M can be any value greater than* 6 *for getting success probability over* 99%
2: Answer=0
3: **for** $i = 1$ to m **do**
4: ans = Randomized Approximate Median(S)
5: **if** ans != error **then**
6: Answer= ans
7: **end if**
8: **end for**
9: **return** Answer

All the runs can be considered independent events and hence probability of intersection of all those events is product of individual probabilities.

Thus, probability of success for whole algorithm is $1 - (0.5)^m$

For success to be greater than 99%,

$1 - (0.5)^m > 0.99$

$(0.5)^m < 0.01$

$2^m > 100$

$m >= 7$

Thus, by selecting any m¿6, we can make the success probability of the new algorithm be over 99%.

5. **Solution to problem 5**

(a) The k-th order statistic returns k-th smallest element from the set S of distinct numbers. For finding the median we can set k to half the length of list so that it returns middle element of the set which is the required median.

(b) The k-th order statistic is an algorithm of Las Vegas type. It returns same output for same input but the running time would be different according to the randomly chosen variable. Thus, expected running time needs to be evaluated.

The expected running time of k-th order statistic can be analyzed as below:

We define an indicator random variable X_i that is 1 when i^{th} item is selected at random. We can show the running time of the algorithm as T(n) and it depends upon the item selected. if we select the item correctly the algorithm performs the for loop (Statements 2-4) and then returns the element. In the other cases when we select the wrong element at first, our search space gets restricted to lower subarray S^- if the size of S^- is greater than k and gets restricted to upper subarray S^+ if size of subarray S^- is less than k. In the later case we need to find $k - 1 - \mid S^- \mid$element from the upper subarray as we already know the number of elements in the lower subarray which are lower than the required element.

Thus, T(n) = work done while in the loop + work done in the recursive call (i.e) for solving a sub-problem of less size (k in case it selects lower subarray and n-k in case it selects upper subarray) .

To upper bound the whole expected running time T(n), we need to find the expected running time of the sub-problems.

$$T(n) = O(n) + X_k T(k); if\ lower\ subarray\ is\ chosen$$

$$T(n) = O(n) + X_k T(n-k); if\ upper\ subarray\ is\ chosen$$

where, X_i is indicator random variable that is 1 when i^{th} item is selected at random.
When we are upper bounding the overall running time, we can select maximum of above 2 and hence;
The running time $T(n) <= O(n) + X_k max(T(Sub - problem))$
We need to multiply the expectation value of the sub-problem to take into account the probability of sub-problem getting chose and get the expected running time.

$$E[T(n)] <= O(n) + \Sigma(E[X_k * max(T(k, n-k))]); 1 <= k <= n$$

$$= O(n) + \Sigma(E[X-k] * E[max(T(k, n-k))]); 1 <= k <= n$$

as X_k is an indicator random variable, it's value is equal to probability of choosing i^{th} element in arrays of size n which is equal to $1/n$.

$$E[T(n)] <= O(n) + \Sigma(1/n * E[max(T(k, n-k))]); 1 <= k <= n$$

As, k here is a number that divides the array into two parts and if we are considering maximum of it only, the lower subarray sizes and upper subarray sizes would be symmetrical pairs when stated in order. (i.e)

$$(1, n-1); (2-n-2), ..., (n-2, 2), (n-1, 1)$$

After selecting maximum out of these pairs: $n-1, n-2, ..., n-(n/2), n-(n/2), ..., n-1, n-2$. Hence, elements $n/2\ to\ n-1$ repeat twice.

$$E[T(n)] <= O(n) + \Sigma(2/n * E[T(k)]); n/2 <= k <= n$$

$2/n$ is invariable of k and hence, can be taken out of the summation.

$$E[T(n)] <= O(n) + 2/n * \Sigma(E[T(k)]); n/2 <= k <= n$$

$$E[T(n-1)] <= O(n) + 2/n * \Sigma(E[T(k)]); (n-1)/2 <= k <= n-1$$

$$E[T(n)] <= O(n) + 2/n * \Sigma(E[T(k)]) + 2/nc(n-1/2)];\ (n)/2 <= k <= n$$

$$E[T(n)] <= E[T(n-1)] + 2/nc[T(n-1/2)]; (n)/2 <= k <= n$$

$$E[T(n-2)] <= O(n) + 2/n * \Sigma(E[T(k)]); (n-2)/2 <= k <= n-2$$

$$E[T(n-1)] <= O(n) + 2/n * \Sigma(E[T(k)]) + 2/nc(n-2/2)];\ (n-1)/2 <= k <= n-1$$

$$E[T(n-1)] <= E[T(n-2)] + 2/nc(n-1/2)]; (n)/2 <= k <= n$$

$$E[T(n)] <= E[T(n-1)] + 2/nc(n-1/2)]; (n)/2 <= k <= n$$

$$E[T(n)] <= E[T(n-2)] + 2/nc(n-1/2)] + 2/nc(n-1/2)] + 2/nc(n-2/2)]$$

Assuming E[T(1)] = $c_1 n$;

$$E[T(n)] <= c_1 n + 2c_2/n\Sigma(i/2); (n)/2 <= i <= n$$

$$E[T(n)] <= c_1 n + 2c_2/n\Sigma(i/2) - 2c_2/n\Sigma(j/2);\ 1 <= i <= n\ and\ 1 <= j < n/2$$

$$E[T(n)] <= c_1 n + 2c_2 * (n * (n+1))/(4 * n) - 2c_2 * (n(n+1))/(16 * n)$$

$$E[T(n)] <= C_1 n + c_2 * (n+1)/4 - c_2 * (n+1)/8$$

$$E[T(n)] <= c * n$$

$$E[T(n)] = O(n)$$