CSE 551
Assignment 1

30^{th} August, 2022

Submission Instructions: Deadline is **11:59pm on 09/07/2022**. Late submissions will be penalized, therefore please ensure that you submit (file upload is completed) before the deadline. Additionally, you can download the submitted file to verify if the file was uploaded correctly. **Please TYPE UP YOUR SOLUTIONS and submit a PDF** electronically, via *Canvas*. Furthermore, please note that the graders will grade 2 out of the 4 questions randomly. Therefore, if the grader decides to check questions 1 and 4,and you haven't answered question 4, you'll lose points for question 4. Hence, please answer all the questions.

1. Prove or disprove the following assertions: **(8+8+9)**

   (i) If $f(n) = O(g(n))$ then $log_2 f(n) = O(log_2 g(n))$
   True

   Given $f(n) \leq c.g(n)$ for all $n \geq n0$ and c is a constant.
   Applying log_2 on both sides.
   $log_2 f(n) \leq log_2 c + log_2 g(n)$ for all $n \geq n0$
   Choose n0 such that $log_2 g(n) \geq log_2 c$ for all $n \geq n0$
   From above 2 we can say $log_2 f(n) \leq log_2 g(n) + log_2 g(n) = 2.log_2 g(n)$
   Hence we can say $log_2 f(n) = O(log_2 g(n))$.

   (ii) If $f(n) = O(g(n))$ then $3^{f(n)} = O(3^{g(n)})$
   False

   Here is a counter example. Consider $f(n) = 2log_3 n$ and $g(n) = log_3 n$. This satisfies $f(n) = O(g(n))$
   $3^{f(n)} = n^2$ and $3^{g(n)} = n$ but n^2 is not $O(n)$.

   (iii) If $f(n) = O(g(n))$ then $f(n)^3 = O(g(n)^3)$
   True

   Proof simiar to (i) where we instead cube on both sides.

2. Algorithm A_1 takes $10^{-4} \times 2^n$ seconds to solve a problem instance of size n and Algorithm A_2 takes $10^{-2} \times n^3$ seconds to do the same on a particular machine. **(8+8+9)**

(i) What is the size of the largest problem instance A_2 will be able solve in one year ?

Ans: 1 year = 31536000 seconds. Assume we can solve a problem size x. Then $10^{-2} * x^3 = 31536000$. Solving it we get x = 1466.

(ii) What is the size of the largest problem instance A_2 will be able solve in one year on a machine one hundred times as fast ?

Ans: From given info we can say A2 is O(n^3) algorithm. From the size of largest problem instance solvable in 1 Hour table we filled in class A2 can solve 4.64N size problem in 1hour on a machine 100 times faster. 4.64 * 1466.64 is approximately 6805.

(iii) Which algorithm will produce results faster, in case we are trying to solve problem instances of size less than 20 ?

Ans: For $n = 20$ A1 takes $10^{-4} * 2^{20} = 104.856$ seconds and A2 takes $10^{-2} * n^3 = 80$ seconds. For $n = 19$ A1 takes 52.43 seconds and A2 takes 68.59 seconds. So both the plots intersect somewhere between 19 and 20. So for problem sizes less than 20 A1 solves the problem faster.

3. Prove or disprove the following with valid arguments: **(8+8+9)**

(i) $3n^3 + 1000 = O(n^2)$.

Ans: False

R.T.P. $3n^3 + 1000 = O(n^2)$

or, $3n^3 + 1000 \leq c.n^2$ where $n \geq n0$

or, $3n + \frac{1000}{n} \leq c$

or, $\frac{3n^2+1000}{n} \leq c$

The L.H.S. is a function of n and its value increases with n. There are no c value for which the above condition is true.

(ii) $2n^2 log(n) = \Theta(n^2)$.

Ans: False

R.T.P. $n^2.log(n) = \Theta(n^2)$

Alternatively, we need to prove :

Case 1 : $n^2.log(n) = O(n^2)$ and

Case 2 : $n^2.log(n) = \Omega(n^2)$

For Case 1 :

R.T.P. : $n^2.log(n) \leq c_1.n^2$

Dividing both L.H.S and R.H.S by n^2 we get :

R.T.P. : $log(n) \leq c_1$

The L.H.S. is a function of n and its value increases with n. There are no c_1 value for which the above condition is true for sufficiently large value of n.

(iii) $3^n n^4 + 8 * 4^n n^3 = O(3^n n^4)$.

ANS: False

R.T.P. $3^n n^4 + 8 * 4^n n^3 = O(3^n n^4)$.

or, $3^n n^4 + 8 * 4^n n^3 \leq c.3^n n^4 for all n \geq n0$

Dividing both sides by $3^n n^4$

$1 + \frac{8*4^n n^3}{3^n n^4} \leq c$

$1 + \frac{8*(1.3)^n}{n} \leq c$

Here, we have reached a contradiction as the L.H.S. becomes infinity as n approaches ∞. So no such constant c exists.

4. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function g(n) immediately follows f(n) in your list, then it should be the case that f(n) is O(g(n)). **(25)**

(i) $f_1(n) = n^{4.6}$.
(ii) $f_2(n) = (2n)^{1.6}$.
(iii) $f_3(n) = n^{4.5} + 87$.
(iv) $f_4(n) = 40^n$.
(v) $f_5(n) = 120^n$

ANS: $f_2 < f_3 < f_1 < f_4 < f_5$ Polynomial functions have less order of growth compared to exponential functions. Polynomial functions have order of growth in the order of their highest exponents. In exponential functions 40^n has lower order of growth than 120^n.

CSE 551 Assignment 2

14^{th} September, 2022

Submission Instructions: Deadline is **11:59pm on 09/20/2022**. Late submissions will be penalized, therefore please ensure that you submit (file upload is completed) before the deadline. Additionally, you can download the submitted file to verify if the file was uploaded correctly. **Please TYPE UP YOUR SOLUTIONS and submit a PDF** electronically, via *Canvas*. Furthermore, please note that the graders will grade 2 out of the 4 questions randomly. Therefore, if the grader decides to check questions 1 and 4,and you haven't answered question 4, you'll lose points for question 4. Hence, please answer all the questions.

1. The Fibonacci series can be computed as follows,

$$F(n) = F(n-1) + F(n-2) \tag{1}$$

In class, we showed how this can be done in $\mathcal{O}(log\ n)$ computation time. Now suppose that the definition is changed in the following way,

$$F'(n) = F'(n-1) + F'(n-2) + F'(n-3) + F'(n-4) \tag{2}$$

Can $F'(n)$ be computed in $\mathcal{O}(log\ n)$? If yes, please show how it can be done. If no, show a counterexample where this fails. Please provide your rationale for both. Assume that $F'(0) = 0, F'(1) = 1, F'(2) = 1, F'(3) = 1$. **[25 Points]**.

ANS:

Given,

$F'(n) = F'(n-1) + F'(n-2) + F'(n-3) + F'(n-4)$

Therefore,

$[F'(n-3)\ \ F'(n-2)\ \ F'(n-1)\ \ F'(n)]$

$= [F'(n-3)\ \ F'(n-2)\ \ F'(n-1)\ \ F'(n-1)+F'(n-2)+F'(n-3)+F'(n-4)]$

$= [F'(n-4)\ \ F'(n-3)\ \ F'(n-2)\ \ F'(n-1)]$ x A

where

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$= [F'(n-5)\ \ F'(n-4)\ \ F'(n-3)\ \ F'(n-2)]$ x A^2

. . . . . .

$= [F'(0)\ \ F'(1)\ \ F'(2)\ \ F'(3)]$ x A^{n-3}

$= [0\ \ 1\ \ 1\ \ 1]$ x A^{n-3}

Since the dimensions of $A = (4 \times 4)$,i.e., a constant value, the matrix multiplication A^{n-3} can be done in O(log n) time (**Refer to class slides**).

2. In class we have discussed the Quick Sort algorithm. Answer the following questions regarding quick sort.

   - When does the worst case for Quick Sort occur?
     Ans: The worst case for quick sort occurs when every time the pivot element picked would be the largest or smallest element in the array. This would make the array into only one part of size $n-1$.

   - What would be the worst case recurrence for Quick Sort? Solve the recurrence and get its time complexity in worst case.
     Ans: The worst case recurrence for quick sort is $T(n) = T(n-1) + T(1) + O(n)$. Here is the solution:
     $T(n) = T(n-1) + T(1) + O(n) and T(1) = O(1)$
     $T(n) = T(n-1) + O(n)$
     $T(n) = [T(n-2) + O(n-1)] + O(n)$
     $= T(n-3) + (n-2) + (n-1) + O(n)$
     .
     .
     .
     $= T(1) + (n-(n-2)) + ... + (n)$
     $= 1 + 2 + 3 + ... + n = n(n-1)/2$
     $T(n) = O(n^2)$

   - What would be the best case recurrence for Quick Sort? Solve the recurrence and get its time complexity in best case.
     Ans: In best case of quick sort every time the array would be split into two equal halves. So the recurrence would be $T(n) = 2T(n/2) + O(n)$. We can solve the recurrence using Master theorem and we get the time complexity of $O(nlogn)$

- Suppose you have an algorithm which can give the median of a set on numbers in O(n) time. How can this be used to improve the worst case time complexity of Quick sort?
  Ans: When we have a algorithm to find median in O(n) time, in every iteration of the quick sort we can use this to get the median of the numbers ans use it as the pivot element. This would make the recurrence $T(n) = 2T(n/2) + O(n)$ solving which we get $T(n) = O(nlogn)$

- In the execution of Quick sort suppose every time the $n/5^{th}$ smallest element is picked as the pivot element among n elements, what would be the recurrence for quick sort? What would be its time complexity?
  Ans: If the n/5th element is picked as the pivot element after one split one bucket has $n/5$ elements and other has $4n/5$ elements.
  The recurrence becomes $T(n) = T(n/5) + T(4n/5) + O(n)$. We can use recurrence tree method to solve this. The sum of nodes in each level would be n and the height of the tree would be at most $log_{5/4}n$. So the time complexity would be $\leq nlog_{5/4}n = O(nlogn)$.

3. If k is a non-negative constant, then prove that the recurrence: **[25 points]**

$$T(n) = k \text{ , for } n = 1 \text{ and}$$
$$T(n) = 3T(\tfrac{n}{2}) + kn, \text{ for } n > 1$$

has the following solution (for n a power of 2):

$$T(n) = 3kn^{log_2 3} - 2kn$$

Ans: Here is the solution to the recurrence.

$T(n) = 3T(n/2) + kn$
$T(n) = 3[3T(n/4) + kn/2] + kn$
$= 9T(n/4) + 3kn/2 + kn$
$= 9[3T(n/8) + kn/4] + 3kn/2 + kn$
$= 27T(n/8) + 9kn/4 + 3kn/2 + kn$

After i levels we get $T(n) = 3^i T(n/2^i) + kn + \Sigma_{j=0}^{i-1}(3/2)^j$

Since there are $log_2 n$ levels substituting i we get

$T(n) = 3^{logn} * k + 2kn * ((3/2)^{logn} - 1)$ which on simplifying gives $T(n) = 3kn^{log_2 3} - 2kn$

4. Design an algorithm to compute the 2nd smallest number in an unordered (unsorted) sequence of numbers $\{a_1, a_2, ..., a_n\}$ in $n + \lceil log_2(n) \rceil - 2$ comparisons in the worst case. If you think such an algorithm can be designed,

then show how it can be done. If your answer is no, then explain why it cannot be done. **[25 points]**

ANS:

Let us consider a tournament. Assume that you have a list of n players (denoted by unique integers). One tournament rule could be the following: the smallest number of two numbers, wins. Now, we can use the Find-MinMax algorithm taught in class to find out the smallest number in n - 1 comparisons. Realize that, in such a tournament the smallest number (best player) is guaranteed to play the second smallest number (second best player), if we follow our rule. Therefore, once we have discovered the best player, we need to examine *the subtree from where the best player originated*. This is because, the best player could have played the second best player at any level in the tournament (and not just the final round). This analysis takes $\lceil log_2(n) \rceil - 1$ comparisons. Therefore, in total, we have at most $n + \lceil log_2(n) \rceil - 2$ comparisons to find the 2nd second best player in the tournament.

Another similar way to solve this to construct a min heap. Construction of the min heap would take $O(n)$. This would entail that the minimum value in the sequence is present in the root. Remove the minimum and replace it with the rightmost element in the tree, following level order traversal. Heapify the new tree and you will have the second smallest element as the root now. Heapify takes $O(logn)$ time. Hence the total time to find the second smallest element is less than $n + \lceil log_2(n) \rceil - 2$.

CSE 551: Foundations of Algorithms
Midterm Solutions

Set 1

Problem 1: Suppose that the dimensions of the matrices A,B,C and D are 20*22, 22*25, 25*50 and 50*5. Find the fewest number of multiplications that will be necessary to compute the final matrix and the order in which the matrix chain be multiplied so that it will require the fewest number of multiplications. Show all your work.

Solution: 11200. A*(B*(C*D))

Problem 1*: Suppose that the dimensions of the matrices A,B,C and D are 20*12, 12*5, 5*60 and 60*4. Find the fewest number of multiplications that will be necessary to compute the final matrix and the order in which the matrix chain be multiplied so that it will require the fewest number of multiplications. Show all your work.

Solution: 2400. A*(B*(C*D))

Problem 1:** Suppose that the dimensions of the matrices A,B,C and D are 20*2, 2*15, 15*40 and 40*4. Find the fewest number of multiplications that will be necessary to compute the final matrix and the order in which the matrix chain be multiplied so that it will require the fewest number of multiplications. Show all your work.

Solution: 1680. (A*(B*C)*D))

Problem 2: Let $A_1,....A_n$ be matrices where the dimensions of the matrices A_i, $1<= i <= n$ are $d_{i-1} * d_i$. The following algorithm is proposed to determine the best order in which to perform the matrix multiplications to compute $A_1*A_2*....*A_n$.

At each step, choose the largest remaining dimensions, and multiply two adjacent matrices that share that dimension. What is the complexity of this algorithm? Is this algorithm always going to find the order in which to multiply the matrix chain that will require the fewest number of multiplications? If your answer is **yes**, then provide arguments to support the assertion that this algorithm will always find the best order. If your answer is **no**, then provide an example where the algorithm fails to find the best order.

Solution: No, this algorithm will not always find the fewest number of multiplications.
Let us consider the following example, A * B * C, with dimensions $d_0 * d_1$, $d_1 * d_2$, $d_2 * d_3$.
Let us assume that $d_1 > d_2$. (You could also consider $d_2 > d_1$).

Following our assumption, we will multiply A with B first and the resultant matrix with C. The number of multiplications required is,

$d_0d_1d_2 + d_0d_2d_3$

Now, this should be lesser than the number of multiplications required if we multiply B and C first and then the resultant with A. Therefore,

$d_0d_1d_2 + d_0d_2d_3 < d_1d_2d_3 + d_0d_1d_3$, or,

$d_0d_1d_2 + d_0d_3d_2 < d,d_1d_2 + d_0d_3d_1$, rearranging the terms.

Now, $d_0d_3d_2 < d_0d_3d_1$, as $d_1 > d_2$.
But, $d_0d_1d_2$ need not be less than $d_3d_1d_2$, as d_0 and d_3 are independent numbers.

Let us look at the following example,

100*2, 2*1, 1*1. Going by our approach, we multiply 100*2 and 2*1 first, resulting in 200 multiplications. We then multiply 100*1 with 1*1, resulting in 100 multiplications. Therefore the total number of multiplications required is 200 + 100 = 300.
But, we can do this is in fewer number of multiplications,
Multiply 2*1 with 1*1, resulting in a 2*1 matrix, with 2 multiplications. Then multiply this with 100*2, resulting in a 100*1 matrix, with 200 multiplications. Thus, the total number of multiplications required is 2 + 200 = 202.

Problem 3: Let S be an ordered sequence of n distinct integers. Develop an algorithm to find a longest increasing subsequence of the entries of S. The subsequence is not required to be contiguous in the original sequence. For example, if S = {11, 17, 5, 8, 6, 4, 7, 12, 3}, a longest increasing subsequence of S is {5, 6, 7, 12}. Analyze your algorithm to establish its correctness and also its worst-case running time and space requirement.

Solution:
Say the given list is S. Create another list T which is the sorted version of S.
Now compute LCS(S,T) in $O(n^2)$ time.

Problem 4: Suppose that you are choosing between the following three algorithms:

1. A solves the problem by dividing it into 4 subproblems of half the size, recursively solving each subproblem and then combining the solutions in linear time.
2. B solves the problem of size n by recursively solving 4 subproblems of size n - 1 and then combining the solutions in constant time.
3. C solves the problem of size n by dividing them into 8 subproblems of size n/3, recursively solving each subproblem, and then combining the solution in $O(n^2)$ time.

What are the running time of each of the algorithms and which would you choose?

Solution:
For 1, $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. Solving $4T(n/2)$ would result in $\Theta(n^2)$, which is greater than $O(n)$, therefore the complexity is $\Theta(n^2)$.

For 2, $T(n) = 4T(n - 1) + k$, $T(1) = 1$. Solving $4T(n - 1)$, would result in an exponential function (2^n), hence the complexity is $O(2^n)$.

For 3, $T(n) = 8T(n/3) + O(n^2)$. Solving $8T(n/3)$, would result in $\Theta(n^{1.89})$, but $O(n^2)$ would dominate and hence, the complexity is $O(n^2)$.

You can choose either 1 or 3 to receive full points.

Problem 4*: Suppose that you are choosing between the following three algorithms:

4. A solves the problem by dividing it into 5 subproblems of half the size, recursively solving each subproblem and then combining the solutions in linear time.
5. B solves the problem of size n by recursively solving 2 subproblems of size n - 1 and then combining the solutions in constant time.
6. C solves the problem of size n by dividing them into 9 subproblems of size n/3, recursively solving each subproblem, and then combining the solution in $O(n^2)$ time.

What are the running time of each of the algorithms and which would you choose?

Solution:
For 1, $T(n) = 5T(n/2) + O(n)$, $T(1) = 1$. Solving $5T(n/2)$ would result in $\Theta(n^{2.32})$, which is greater than $O(n)$, therefore the complexity is $\Theta(n^{2.32})$.

For 2, $T(n) = 2T(n - 1) + k$, $T(1) = 1$. Solving $2T(n - 1)$, would result in an exponential function (2^n), hence the complexity is $O(2^n)$, where k is a constant.

For 3, $T(n) = 9T(n/3) + O(n^2)$. Solving $9T(n/3)$, would result in $\Theta(n^2)$, which is same as the recombination, hence the complexity is $\Theta(n^2)$.

Select 3 to receive full points.

Problem 4:** Suppose that you are choosing between the following three algorithms:

7. A solves the problem by dividing it into 6 subproblems of half the size, recursively solving each subproblem and then combining the solutions in linear time.
8. B solves the problem of size n by recursively solving 3 subproblems of size n - 1 and then combining the solutions in constant time.
9. C solves the problem of size n by dividing them into 10 subproblems of size n/3, recursively solving each subproblem, and then combining the solution in $O(n^2)$ time.

What are the running time of each of the algorithms and which would you choose?

Solution:
For 1, $T(n) = 6T(n/2) + O(n)$, $T(1) = 1$. Solving $6T(n/2)$ would result in $\Theta(n^{2.58})$, which is greater than $O(n)$, therefore the complexity is $\Theta(n^{2.58})$.

For 2, $T(n) = 3T(n - 1) + k$, $T(1) = 1$. Solving $3T(n - 1)$, would result in an exponential function (2^n), hence the complexity is $O(2^n)$, where k is a constant.

For 3, $T(n) = 10T(n/3) + O(n^2)$. Solving $10T(n/3)$, would result in $\Theta(n^{2.09})$, which is greater than the recombination, hence the complexity is $\Theta(n^{2.09})$.

Select 3 to receive full points.

Set 2

Problem: Let u and v be two n bit numbers where n is a power of 2. the traditional multiplication algorithm requires O(n2) operations. A Divide-and-Conquer based algorithm splits the numbers into two equal parts, computing the product as

$$uv = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^{n/2} + (ad + bc)2^{n/2} + bd$$

The multiplication of ac, ad, bc and bd are done using this algorithm recursively.
(i) Determine the computing time of the above algorithm.
(ii) What is the computing time if ad + bc is computed as $(a + b)(c + d) - ac - bd$?

Solution:

(i) The expression makes recursive calls to four subproblems of half the size each, and then evaluates the preceding expression in O(n) time.

$$T(n) = 4T(n/2) + O(n).$$

Using Master's Theorem, we solve the recurrence to obtain $O(n^2)$.

(ii) Note that this method gives the recurrence

$$T(n) = 3T(n/2) + O(n)$$

Since the transition from 4 to 3 occurs at every level of the recursion, we get lower time bound of $O(n^{1.59})$ (again, use Master's Theorem to prove this).

Problem: The n-th Fibonacci number is defined by the recurrence relation $F(n) = F(n-1) + F(n-2)$, with initial conditions $F(0) = 0$ and $F(1) = 1$. $F(n)$ can easily be computed with an algorithm with complexity $\Theta(n)$. Develop an algorithm to compute $F(n)$ with complexity $\Theta(\log n)$. Explain the idea of your algorithm and analyze the algorithm to show that the complexity is indeed $\Theta(\log n)$. Show all your work.

Solution:

Discussed in class.
Check the class notes for the exact solution.

Problem: $T(n) = k, n = 1$
$T(n) = 3T(n/2) + kn, n > 1.$

Solution:

$T(n) = 3T(n/2) + kn$
$T(n/2) = 3T(n/4) + kn/2$
.
.
.
$T(n/2^{m-1}) = 3T(1) + kn/2^{m-1}$
Substituting,

$T(n) = 3^m k + (3/2)^{m-1}kn + (3/2)^{m-2}kn + \ldots\ldots + (3/2)^0 kn$
Where $n = 2^m$

$= 3^{\log_2(n)}k + kn\,[\,(3/2)^{m-1} + \ldots\ldots + (3/2)^0]$
$= 3^{\log_2(n)}k + 2kn[(3/2)^m - 1]$
$= 3^{\log_2(n)}k + 2kn[(3^{\log_2(n)}/ 2^{\log_2(n)}) - 1]$
$= 3^{\log_2(n)}k + 2kn[3^{\log_2(n)} / n - 1]$
$= 3^{\log_2(n)}k + 2k[3^{\log_2(n)} - n]$
$= 3^{\log_2(n)}k + 2k3^{\log_2(n)} - 2kn$
$= 3k3^{\log_2(n)} - 2kn$
$= 3kn^{\log_2(3)} - 2kn \qquad [3^{\log_2(n)} = n^{\log_2(3)}]$

Problem: While discussing the LCS problem, we noted that there may be more than one LCS between two given LCSs S and Y. The algorithm discussed in class produces only one LCS. Develop an algorithm to find all the possible LCSs between two given sequences X and Y.

Solution:

In the LCS-length matrix, in the final row, if multiple instances of the same maxima, say *m*, exists, then instead of just following one route(sequence of arrows, i.e. "↑" and "←") from one of the instances of *m*, follow the arrows from all the instances of it from the last row. This will give you all the possible LCSs for a pair of strings. In your solution, you have to demonstrate the full algorithm and not just the changes from the single LCS algorithm.

CSE 551 Assignment 2

February 24, 2021

Submission Instructions: Deadline is **11:59pm on 02/21**. Late sub-missions will be penalized, therefore please ensure that you submit (file upload is completed) before the deadline. Additionally, you can download the submitted file to verify if the file was uploaded correctly. Submit your answers electronically, in a single PDF, via *Canvas*. **Please type up the answers and keep in mind that we'll be checking for plagiarism**.
Furthermore, please note that the graders will grade 3 out of the 6 questions randomly. Therefore, if the grader decides to check questions 1, 2 and 4, and you haven't answered question 4, you'll lose points for question 4. Hence, please answer all the questions.

1. The Fibonacci series can be computed as follows,

$$F(n) = F(n-1) + F(n-2) \tag{1}$$

In class, we showed how this can be done in $\mathcal{O}(log\ n)$ computation time. Now suppose that the definition is changed in the following way,

$$F'(n) = F'(n-1) + F'(n-2) + F'(n-3) + F'(n-4) \tag{2}$$

Can $F'(n)$ be computed in $\mathcal{O}(log\ n)$? If yes, please show how it can be done. If no, show a counterexample where this fails. Please provide your rationale for both. Assume that $F'(0) = 0, F'(1) = 1, F'(2) = 1, F'(3) = 1$. **[25 Points]**.

ANS:

Given,

$F'(n) = F'(n-1) + F'(n-2) + F'(n-3) + F'(n-4)$

Therefore,

$[F'(n-3)\ \ F'(n-2)\ \ F'(n-1)\ \ F'(n)]$

$= [F'(n-3)\ \ F'(n-2)\ \ F'(n-1)\ \ F'(n-1) + F'(n-2) + F'(n-3) + F'(n-4)]$

$= [F'(n-4)\ \ F'(n-3)\ \ F'(n-2)\ \ F'(n-1)]$ x A

where

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$= [F'(n-5)\ \ F'(n-4)\ \ F'(n-3)\ \ F'(n-2)]$ x A^2

. . . . . .

$= [F'(0)\ \ F'(1)\ \ F'(2)\ \ F'(3)]$ x A^{n-3}

$= [0\ \ 1\ \ 1\ \ 1]$ x A^{n-3}

Since the dimensions of $A = (4 \times 4)$,i.e., a constant value, the matrix multiplication A^{n-3} can be done in O(log n) time (**Refer to class slides**).

2. Consider the following problem. You're given an array A consisting of n integers $A[1], A[2], ..., A[n]$. You'd like to output a two-dimensional $n \times n$ array B in which $B[i, j](for\ i < j)$ contains the sum of array entries $A[i]$ through $A[j]$, i.e., the sum $A[i] + A[i+1] + ... + A[j]$. (The value of array entry $B[i, j]$ is left unspecified whenever $i \geq j$, so it doesn't matter what is output for these values.)

   A simple algorithm to solve this problem illustrated in Algorithm 1:

```
Data: Input A and n
Result: Return B
for i = 1, 2, ..., n do
    for j = i + 1, i + 2, ..., n - 1 do
        Add up array entries A[i] through A[j]
        Store the result in B[i, j]
    end
end
Return B;
```

Algorithm 1: Algorithm for Q2

```
B[i][j] ← 0, ∀i, j ≤ n
for i = 1, 2, ..., n do
    sum ← A[i]
    for j = i + 1, i + 2, ..., n - 1 do
        sum ← sum + A[j]
        B[i][j] ← sum
    end
end
Return B;
```

Algorithm 2: Improved Algorithm for Q3

- For some function f that you should choose, give a bound of the form $\mathcal{O}(f(n))$ on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm). **[8 Points]**.

  ANS: The first two for loops each iterate n and $n-1$ times respectively. Therefore, we have n^2 computation for the for loops. The computation of the sum from $A[i]$ to $A[j]$, can take atmost n iterations. Storing the result takes constant time. Therefore, we can select a $f(n) = n^3$ and the upper bound is $\mathcal{O}(n^3)$.

- For this same function f, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.) **[8 Points]**.

  ANS: Let us count the number of additions, suppose $i = 1$,
  when $j = 2$, we have 1 addition
  when $j = 3$, we have 2 additions
  when $j = 4$, we have 3 additions
  when $j = n$, we have $n - 1$ additions

  Therefore, the number of additions taking place $= 1 + 2 + (n-1) = n \times (n-1)/2$

  Now, i iterates n times, therefore select $f(n) = n \times n \times (n-1)/2 = (n^3 - n^2)/2$. You can now easily show $\Theta(n^3)$.

- Although the algorithm you analyzed in parts (a) and (b) is most natural way to solve problem-after all, it just iterates through the relevant entries of the array B, filling in a value for each - it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time $\mathcal{O}(g(n))$, where $lim_{n\to\infty} g(n)/f(n) = 0$. **[9 Points]**.

  ANS: One modification would be to pre-compute the values in the outer loop, to avoid re-computation. Consider the algorithm 2:
  This improves the time complexity to $O(n^2)$.

3. Design an algorithm to compute the 2nd smallest number in an unordered (unsorted) sequence of numbers $\{a_1, a_2, ..., a_n\}$ in

$n + \lceil log_2(n) \rceil - 2$ comparisons in the worst case. If you think such an algorithm can be designed, then show how it can be done. If your answer is no, then explain why it cannot be done. **[25 points]**

ANS:

Let us consider a tournament. Assume that you have a list of n players (denoted by unique integers). One tournament rule could be the following: the smallest number of two numbers, wins. Now, we can use the FindMinMax algorithm taught in class to find out the smallest number in n - 1 comparisons. Realize that, in such a tournament the smallest number (best player) is guaranteed to play the second smallest number (second best player), if we follow our rule. Therefore, once we have discovered the best player, we need to examine *the subtree from where the best player originated.* This is because, the best player could have played the second best player at any level in the tournament (and not just the final round). This analysis takes $\lceil log_2(n) \rceil - 1$ comparisons. Therefore, in total, we have at most $n + \lceil log_2(n) \rceil - 2$ comparisons to find the 2nd second best player in the tournament.

Another similar way to solve this to construct a min heap. Construction of the min heap would take $O(n)$. This would entail that the minimum value in the sequence is present in the root. Remove the minimum and replace it with the rightmost element in the tree, following level order traversal. Heapify the new tree and you will have the second smallest element as the root now. Heapify takes $O(logn)$ time. Hence the total time to find the second smallest element is less than $n+\lceil log_2(n) \rceil - 2$.

4. Let $n = 2p$, $V = (v_1, ..., v_n), W = (w_1, ..., w_n)$. Then we can compute the vector product VW by the formula:
$\sum_{1 \le i \le p} (v_{2i-1} + w_{2i}) \times (v_{2i} + w_{2i-1}) - \sum_{1 \le i \le p} v_{2i-1} \times v_{2i} - \sum_{1 \le i \le p} w_{2i-1} \times w_{2i}$

which requires $3n/2$ multiplications. Show how to use this formula for the multiplication of two $n \times n$ matrices giving a method which requires $n^3/2 + n^2$ multiplications rather than the usual n^3 multiplications. **[25 points]**

ANS:

Multiplications of two $n \times n$ matrices to compute the resultant $n \times n$ matrix C involves product of n^2 vector multiplications. The matrix A may be viewed as made up of n row vectors $V_1, V_2, ..., V_n$ and similarly B may be viewed as made up of n column vectors $W_1, W_2, ..., W_n$. The element C_{11} of C is $V_1 W_1$, C_{12} is $V_1 W_2$ and so on.

In order to compute C_{11} we will require 3p multiplications. In order to compute $C_{12}, ..., C_{1n}$ will require only 2p multiplications, as the second term in the formula involving V_1 is already computed while computing C_{11}.

Similarly, in order to compute $C_{21}, ..., C_{n1}$ we will require only 2p multiplications, as the third term in the formula involving W_1 is already computed while computing C_{11}.

In order to compute C_{22} to C_{2n} and C_{32} to C_{3n}, ..., C_{n2} to C_{nn}, $((n-1)^2)$ entries, only p multiplications will be needed as the second and the third terms of the formula need not be computed again. Thus the total number of multiplications that will be needed is:

$= 3p + (n-1) \times 2p + (n-1) \times 2p + (n-1)^2 \times p$

$= 3n/2 + (n-1) \times n + (n-1) \times n + (n-1)^2 \times n/2$

$= n^3/2 + n^2$

5. Find the solution to the following recurrence relation:

$T(n) = 16T(n/4) + n^2$,

for n a power of 4 (or, $n = 4^k$) and $n > 1$. **Show all your work. [25 points]**

$T(n) = 16T(n/4) + n^2$.

$$\begin{aligned}
T(n) &= n^2 + 16T(\frac{n}{4}) \\
&= n^2 + 16((\frac{n}{4})^2 + 16T(\frac{n}{4^2})) \\
&= n^2 + 16(\frac{n}{4})^2 + 16^2T(\frac{n}{4^2}) \\
&= n^2 + 16(\frac{n}{4})^2 + 16^2((\frac{n}{4^2})^2 + 16T(\frac{n}{4^3})) \\
&= n^2 + 16(\frac{n}{4})^2 + 16^2(\frac{n}{4^2})^2 + 16^3T(\frac{n}{4^3}) \\
&\quad \cdots \\
&= n^2 + 16(\frac{n}{4})^2 + 16^2(\frac{n}{4^2})^2 + 16^3(\frac{n}{4^3})^2 + 16^4(\frac{n}{4^4})^2 + ... + 16^{\log_4 n}(\frac{n}{4^{\log_4 n}})^2 \\
&= \underbrace{n^2 + n^2 + n^2 + n^2 + n^2 + ... + n^2}_{\log_4 n + 1} \\
&= n^2(\log_4 n + 1) \\
&= \Theta(n^2 \lg n)
\end{aligned}$$

6. Find the solution to the following recurrence relation:

$T(1) = 1$
$T(n) = 3^n T(n/2)$,

for n a power of 2 (or, $n = 2^k$) and $n > 1$. **Show all your work. [25 points]**

ANS:

$T(n) = 3^n T(n/2)$
$= 3^n * 3^{\frac{n}{2}} * T(n/4)$
$= 3^n * 3^{\frac{n}{2}} * 3^{\frac{n}{4}} * T(n/8)$
.....
.....
$= 3^n * 3^{\frac{n}{2}} * 3^{\frac{n}{4}} * ... * 3^{\frac{n}{n/2}} * T(n/n)$
$= 3^n * 3^{\frac{n}{2}} * 3^{\frac{n}{4}} * ... * 3^2 * T(1)$
$= 3^{n+\frac{n}{2}+\frac{n}{4}+...+2} * 1$
For n a power of 2, we know: $n + \frac{n}{2} + \frac{n}{4} + ... + 2 + 1 = 2n - 1$
$=> n + \frac{n}{2} + \frac{n}{4} + ... + 2 = 2n - 1 - 1 = 2n - 2$
Therefore,
$T(n) = 3^{2n-2} = \frac{3^{2n}}{9}$
$T(n) \in O(3^{2n})$

CSE 551 Assignment 3
Solutions

March 17, 2021

Submission Instructions: Deadline is **11:59pm on 03/16**. Late sub-missions will be penalized, therefore please ensure that you submit (file upload is completed) before the deadline. Additionally, you can download the submitted file to verify if the file was uploaded correctly. Submit your answers electronically, in a single PDF, via *Canvas*. **Please type up the answers and keep in mind that we'll be checking for plagiarism**.
Furthermore, please note that the graders will grade 2 out of the 4 questions randomly. Therefore, if the grader decides to check questions 1 and 4, and you haven't answered question 4, you'll lose points for question 4. Hence, please answer all the questions.

1. Find the optimal order of multiplying four matrices whose sequence of dimensions is (13, 5, 89, 3, 34), i.e., the First matrix is 13 x 5, the second matrix is 5 x 89 and so on. Furthermore, compute the fewest number of multiplications needed to find the resulting matrix. Show all the steps of your computation.

   Solution: The optimal number of multiplications required is 2856. Let A be 13×5, B be 5×89, C be 89×3 and D be 3×34. The optimal order is $(A \times (B \times C)) \times D$.

2. Let $A_1, A_2, ..., A_n$ be matrices where the dimensions of A_i are $d_{i-1} * d_i$ for $i = 1, ..., n$. Here is a strategy to compute the best order (i.e., the order that requires the fewest number of multiplications) in which to perform the matrix multiplications to compute $A_1, A_2, ..., A_n$. At each step, choose the largest remaining dimension (from among $d_1, ..., d_{n-1}$) and multiply two adjacent matrices that share the same dimension. State with reasoning whether the above strategy will always minimize the number of multiplications necessary to multiply the matrix chain $A_1, ..., A_n$ (i.e., if you think that this will result in the optimal number of multiplications, then please provide arguments as to why you think so, else, provide a counter example).

   Solution: The above strategy will not result in the optimal number of multiplications. Here is the counter example. Consider the following matrices and their dimensions: M_1, M_2, M_3 with dimensions $(1 \times 1, 1 \times 2, 2 \times 3)$. The above method will multiply $(M_1 \times (M_2 \times M_3))$ resulting in 9 multiplications $(1 \times 2 \times 3 + 1 \times 1 \times 3)$. However, if we multiply $((M_1 \times M_2) \times M_3)$, we will end up with 8 multiplications $(1 \times 1 \times 2 + 1 \times 2 \times 3)$.

3. Using Dynamic Programming technique, find the optimal solution to the Traveling Salesman Problem for the following data set (Distance Matrix).

$$M = \begin{bmatrix} 0 & 2 & 5 & 9 \\ 10 & 0 & 2 & 8 \\ 3 & 8 & 0 & 6 \\ 9 & 2 & 6 & 0 \end{bmatrix}$$

   Show all your work. Also show the node ordering in the optimal tour.

   Solution: $g(2, \Phi) = 10$, $g(3, \Phi) = 3$, $g(4, \Phi) = 9$.
   $g(2, 3) = L_{23} + g(3, \Phi) = 5$
   $g(2, 4) = L_{24} + g(4, \Phi) = 17$
   $g(3, 2) = L_{32} + g(2, \Phi) = 18$
   $g(3, 4) = L_{34} + g(4, \Phi) = 15$
   $g(4, 2) = L_{42} + g(2, \Phi) = 12$
   $g(4, 3) = L_{43} + g(3, \Phi) = 9$

   $g(2, \{3, 4\}) = min(L_{23} + g(3, 4), L_{24} + g(4, 3)) = min(2 + 15, 8 + 9) = 17$
   $g(3, \{2, 4\}) = min(L_{32} + g(2, 4), L_{34} + g(4, 2)) = min(8 + 17, 6 + 12) = 18$
   $g(4, \{2, 3\}) = min(L_{42} + g(2, 3), L_{43} + g(3, 2)) = min(2 + 5, 6 + 18) = 7$

$g(1, \{2,3,4\}) = min(L_{12} + g(2,(3,4)), L_{13} + g(3,(2,4)), L_{14} + g(4,(2,3))) = min(2+17, 5+18, 9+7) = 16$

Optimal tour: $1 \to 4 \to 2 \to 3 \to 1$.

4. A subset of nodes in $S \subset V$ is a "Special Node Set (SNS)" of a graph $G = (V, E)$, if there are no edges between the nodes in S. The SNS of the largest cardinality is referred to as "Largest Special Node Set (LSNS)". For instance, in Fig. 1, the set $\{1, 5\}$ form an SNS but the set $\{1, 4, 5\}$ do not because there is an edge between nodes 4 and 5. The largest SNS set here is $\{2, 3, 6\}$.

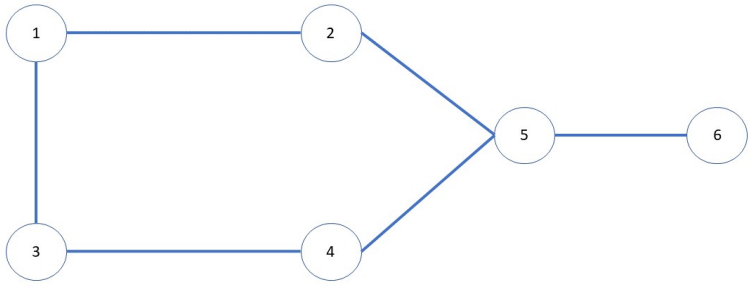


Figure 1: Figure for Q4

Design a Dynamic Programming based algorithm to compute the LSNS of a tree. Write down the recurrence relation on which your algorithm is based and show how the recurrence relation is being used by your algorithm to compute the LSNS of the tree. Analyze your algorithm to find it's complexity. **Show all our work**. Can your algorithm be used to find the LSNS of a graph which isn't a tree? If your answer is "yes", explain how it can be used, otherwise, explain why it can't be used to compute the LSNS of a graph which isn't a tree.

Solution: If the given graph is a tree, by using Dynamic Programming, we can show that the LSNS problem can be solved in linear time. Consider the following:

Root the tree at an arbitrary vertex. Now each vertex defines a subtree. Suppose that we proceed in a bottom up manner and we know the size of the LSNS of all subtrees below a node j. There can be two cases regarding the LSNS in the subtree rooted at j: either j is in the LSNS or it is not. If j is not in the solution, then the LSNS is simply the union of the LSNS of the subtrees of the children of j. However, if j is in the solution, then the LSNS includes j and the union of the LSNS of the grandchildren of j. The recursive equation can be written as follows:

$$I(j) := max\{\sum_{k\ child\ of\ j} I(k),\ 1 + \sum_{k\ grandchild\ of\ j} I(k)\} \tag{1}$$

For each vertex, the algorithm only looks at its children and its grandchildren; hence, each vertex j is looked at only three times: when the algorithm is processing vertex j, when it is processing $j's$ parent, and when it is processing $j's$ grandparent. Since each vertex is looked at only a constant number of times, the total number of steps is $O(n)$.

In the case of trees, we know for sure that there are no cycles present. This is not the case for graphs. As a result, it will be difficult to isolate the children/grandchildren in the above recurrence relation, due to the presence of cycles. Therefore, the above algorithm cannot be used to compute the LSNS of a graph which isn't a tree.

CSE 551 Assignment 1 Solutions

January 24th, 2021

Submission Instructions: Deadline is **11:59pm on 01/31**. Late submissions will be penalized, therefore please ensure that you submit (file upload is completed) before the deadline. Additionally, you can download the submitted file to verify if the file was uploaded correctly. Submit your answers electronically, in a single PDF, via *Canvas*. **Please type up the answers and keep in mind that we'll be checking for plagiarism**.
Furthermore, please note that the graders will grade 2 out of the 4 questions randomly. Therefore, if the grader decides to check questions 1 and 4, and you haven't answered question 4, you'll lose points for question 4. Hence, please answer all the questions.

1. Prove of disprove the following with valid arguments: **(5+5+5+5+5)**

   Solution:
   (i) $n! \in O(n^n)$.

   $n! \in O(n^n)$ implies,
   $n! \leq c \times n^n$, (using the definition of Big-O)
   $n \times (n-1) \times ... \times 1 \leq c \times n^n$,
   Dividing both sides by n^n, we get,
   $\frac{n}{n^n} \times \frac{(n-1)}{n^n} \times ... \times 1 \leq c$

   As $n \to \infty$, all the terms on the LHS $\to 0$. Therefore, there exists a constant $c > 0$, which will satisfy $n! \leq c \times n^n$. Therefore, **TRUE** that $n! \in O(n^n)$.

   (ii) $2n^2 2^n + nlog(n) \in \Theta(n^2 2^n)$.

   We have to show that $2n^2 2^n + nlog(n) \in \Omega(n^2 2^n)$ and $2n^2 2^n + nlog(n) \in O(n^2 2^n)$, in order to show that $2n^2 2^n + nlog(n) \in \Theta(n^2 2^n)$.

   Case 1: $2n^2 2^n + nlog(n) \in \Omega(n^2 2^n)$
   $2n^2 2^n + nlog(n) \geq c_1 \times n^2 2^n$, using the definition for Big-Omega,

Dividing both sides by $n^2 2^n$, we get,
$2 + \frac{nlogn}{n^2 2^n} \geq c_1$,
$2 + \frac{logn}{n2^n} \geq c_1$,

As $n \to \infty$, $\frac{logn}{n2^n} \to 0$. Therefore by selecting $c_1 = 1$, we can satisfy the Big-Omega inequality. Therefore, $2n^2 2^n + nlog(n) \in \Omega(n^2 2^n)$.

Case 2: $2n^2 2^n + nlog(n) \in O(n^2 2^n)$
$2n^2 2^n + nlog(n) \leq c_2 \times n^2 2^n$, using the definition for Big-O,
Dividing both sides by $n^2 2^n$, we get,
$2 + \frac{nlogn}{n^2 2^n} \leq c_2$,
$2 + \frac{logn}{n2^n} \leq c_2$,

As $n \to \infty$, $\frac{logn}{n2^n} \to 0$. Therefore by selecting $c_2 \geq 2$, we can satisfy the Big-O inequality. Therefore, $2n^2 2^n + nlog(n) \in O(n^2 2^n)$. Hence, **TRUE** that $2n^2 2^n + nlog(n) \in \Theta(n^2 2^n)$.

(iii) $10n^2 + 9 = O(n)$.
$10n^2 + 9 \leq c \times n$,
Dividing both sides by n, we get,
$10n + \frac{9}{n} \leq c$,

Since the LHS is a a function of n, we cannot find a constant which will satisfy the above inequality. Therefore, **FALSE** that $10n^2 + 9 = O(n)$.

(iv) $n^2 log(n) = \Theta(n^2)$.
Case 1: $n^2 log(n) = \Omega(n^2)$
$n^2 log(n) \geq c_1 \times n^2$
$log(n) \geq c_1$, after dividing both sides by n^2,
Selecting $c_1 = 1$ for $n > 1$ will satisfy the above equation. Therefore $n^2 log(n) = \Omega(n^2)$.

Case 2: $n^2 log(n) = O(n^2)$
$n^2 log(n) \leq c_2 \times n^2$
$log(n) \leq c_2$, after dividing both sides by n^2,

Since the LHS is a a function of n, we cannot find a constant which

will satisfy the above inequality. Therefore, $n^2log(n) \neq O(n^2)$ and by extension, **FALSE** that $n^2log(n) = O(n^2)$.

(v) $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$.
$n^3 2^n + 6n^2 3^n \leq c \times n^3 2^n$,
$1 + 6 \times \frac{n^2 3^n}{n^3 2^n} \leq c$,
$1 + 6 \times \frac{3^n}{n 2^n} \leq c$,
$1 + 6 \times \frac{1}{n} \times (\frac{3}{2})^n \leq c$,

The RHS is a function of n, therefore we cannot find a constant. Therefore, **FALSE** that $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$.

2. Suppose that you have algorithms with the size running times listed below.Assume that these are the exact number of operations performed as a function of the input size n. Suppose you have a computer that can perform 10^{10} operations per second, and you need to compare a result in at most an hour of computation. For each of the algorithms, what is the largest input size n for which you would be able to get the result within an hour? **(6+6+6+7)**

Solution: Number of operations done in 1 hour = 60 * 60 * 10^{10}.

(i) n^2.
Solving for $n^2 = 60 * 60 * 10^{10}$, we get,
$n = 60 \times 10^5$.

(ii) n^3.
Solving for $n^3 = 60 * 60 * 10^{10}$, we get,
$n = 33019$.

(iii) $50n^2$.
Solving for $50n^2 = 60 * 60 * 10^{10}$, we get,
$n = 848528$.

(iv) 3^n.
Solving for $3^n = 60 * 60 * 10^{10}$, we get,
$n = 28$.

3. Algorithm A_1 takes $10^{-3} \times 2^n$ seconds to solve a problem instance of size n and Algorithm A_2 takes $10^{-2} \times n^4$ seconds to do the same on a particular machine. **(8+8+9)**

Solution:

(i) What is the size of the largest problem instance A_2 will be able solve in one year ?

Ans: Amount of seconds in a year = 365 * 24 * 60 * 60 = 31536000.

Now, we solve for $10^{-2} \times n^4 = 31536000$

$n = 236.97$

Now since the size of a problem instance cannot be fractional, the largest problem instance the algorithm will *finish* solving is 236.

(ii) What is the size of the largest problem instance A_2 will be able solve in one year on a machine one hundred times as fast ?

Ans: On a machine 100x fast, we have
$10^{-2} \times n^4 \times 10^{-2} = 31536000$

$n = 749.37$
Or, $n = 749$

(iii) Which algorithm will produce results faster, in case we are trying to solve problem instances of size less than 20?

Ans: Problem instance of size less than 20 implies $n = 19$. Plugging in the value of n in the amount of time taken for algorithm A_1 we get, $10^{-3} \times 2^n = 10^{-3} \times 2^{19} = 10^{-3} \times 524288 = 524.28s$. Algorithm A_2 will take $10^{-2} \times n^4 = 10^{-2} \times 19^4 = 10^{-2} \times 130321 = 1303.21s$. The process yields similar results for values $1 \leq n \leq 18$. Therefore, algorithm A_1 is faster for instances less than 20.

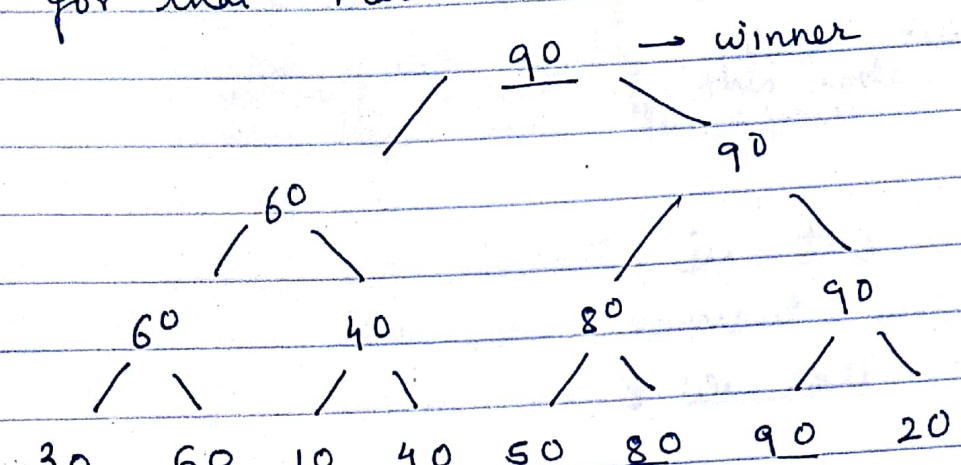
4. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function g(n) immediately follows f(n) in your list, then it should be the case that f(n) is O(g(n)). **(25)**

(i) $f_1(n) = n^{4.5}$.
(ii) $f_2(n) = (3n)^{0.6}$.
(iii) $f_3(n) = n^4 + 20$.
(iv) $f_4(n) = 25^n$.
(v) $f_5(n) = 260^n$

Solution: $f_2 < f_3 < f_1 < f_4 < f_5$

CSE 551: Homework 2

(1) Example of tournament tree:
[30, 60, 10, 40, 50, 80, 90, 20]
The node with max value will be winner for that match.



In order to get this tree, we need (n-1) comparisons since there are (n-1) matches between n players.
(n-1) represents all the internal nodes in the above tree. .. (1)

Algo to construct tree:
The logic here is to maintain a list of nodes, remove 2 nodes from the front of list & push their minimum to the end of list.

Push all the nodes in the array to a queue.
while (queue.size != 1)
{
    pop 2 nodes from queue (x & y)
    create a node with value = max(x & y)
    add x & y as child of this node.
    push this node to queue.
}

The last element in the queue will be the root element with greatest value. or I we call it best player here -

Algo to find second-best player or second max value in the tree constructed above.

Logic used:

At sometime, second best player must have played match with the best player and lost. We need to find ~~that~~ the max value of ~~the no~~ all the players that played match with best player. Other matches can be ignored. Thus, we need ~~O(1)~~ 1 comparison at every level in tree except the root node. Therefore, the

total no. of comparisons = $\log N - 1$ ..(2)

∵ We use recursion here.

```
recur( node* temp)
{
    if ( temp -> right == NULL or
         temp -> left == NULL)
        Return;
    if ( temp -> left -> value == root -> value)
    {
        mx = max( temp -> right -> value ,
                  mn)
        recur( temp -> left);
    }
    if (temp -> right -> value == root -> value)
    {
        mx = max( temp -> left -> value , mn)
        recur ( temp -> right)
    }
}
```

From the above algo, we get second best player in mx variable.

From (1) & (2), we can see that the complexity is $N - 1 + \log N - 1 = (N + \log N - 2)$