CSCI570 Fall2023 HW1

Vinit Pitamber Motwani (USC ID: 8187180925)

September 2023

1. Arrange these functions under the Big-O notation in increasing order of growth rate with g(n) following f(n) in your list if and only if f(n) = O(g(n)) (here, log(x) is the natural logarithm1 of x, with the base being the Euler's number e) :

   $2^{log(n)}$, 2^{3n}, 3^{2n}, $n^{nlog(n)}$, $log(n)$, $nlog(n^2)$, n^{n^2} , $log(n!)$, $log(log(n^n))$

   Solution

   The increasing order is:

   $$\text{O(log(n))} \leq \mathcal{O}(log(log(n^n))) \leq \mathcal{O}(2^{log(n)}) \leq \mathcal{O}(log(n!)) \leq \mathcal{O}(nlog(n^2)) \leq \mathcal{O}(2^{3n}) \leq \mathcal{O}(3^{2n}) \leq \mathcal{O}(n^{nlog(n)}) \leq \mathcal{O}(n^{n^2})$$

   OR

   Let:

   $2^{log(n)} \to f_1$

   $2^{3n} \to f_2$

   $3^{2n} \to f_3$

   $n^{nlog(n)} \to f_4$

   $log(n) \to f_5$

   $nlog(n^2) \to f_6$

   $n^{n^2} \to f_7$

   $log(n!) \to f_8$

   $log(log(n^n)) \to f_9$

   Then the increasing order is:

   $f_5 < f_9 < f_1 < f_8 < f_6 < f_2 < f_3 < f_4 < f_7$

2. Show by induction that for any positive integer k, (k^3+5k) is divisible by 6.

   Solution

   By mathematical induction :

Step 1:
To prove the result is true for $k = 1$

Substituting $k = 1$
$(k^3 + 5k) = (1)^3 + 5 \times 1 = 6$ (which is divisible by 6)

Step 2:
Assume the result is true for $k = n$
Therefore,

$(n^3 + 5n)$ is divisible by 6 for any positive integer n

That means $(n^3 + 5n) = 6 \times p$ .....(where p is some constant)

Step 3:
To prove the result is true for $k = n + 1$

Substituting $k = n + 1$

$$\begin{aligned}(k^3 + 5k) &= (n+1)^3 + 5(n+1) \\ &= n^3 + 1^3 + 3 \times (n)^2 \times 1 + 3 \times (1)^2 \times n + 5 \times n + 5 \times 1 \\ &= n^3 + 3n^2 + 8n + 6 \\ &= n^3 + 5n + 3n^2 + 3n + 6 \\ &= 6p + 3n^2 + 3n + 6 \text{ .....}(from\ Step 2) \\ &= 6(p+1) + 3n^2 + 3n\end{aligned}$$

Now we break this into 2 cases

1) If n is even that means $n = 2w$ for some integer w

$$\begin{aligned}6(p+1) + 3n^2 + 3n &= 6(p+1) + 12w^2 + 6w \\ &= 6(p + 1 + 2w^2 + w) \text{ .....}(which\ is\ divisible\ by\ 6)\end{aligned}$$

2) If n is odd that means $n = 2w + 1$ for some integer w

$$\begin{aligned}6(p+1) + 3n^2 + 3n &= 6(p+1) + 3(2w+1)^2 + 3(2w+1) \\ &= 6(p+1) + 12w^2 + 12w + 6w + 6 \\ &= 6(p + 2w^2 + 3w + 2) \text{ .....}(which\ is\ divisible\ by\ 6)\end{aligned}$$

So it's proved by mathematical induction that $(k^3 + 5k)$ is divisible by 6 for any positive integer k

3. Show that $1^3 + 2^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$ for every positive integer n using induction.

Solution

By mathematical induction :

Step 1:
To prove the result is true for $n = 1$

Substituting $n = 1$
LHS : $1^3 = 1$

RHS : $\frac{1^2(1+1)^2}{4} = \frac{4}{4} = 1$

Therefore LHS = RHS

Step 2:
Assume the result is true for $n = k$
Therefore,

$1^3 + 2^3 + \cdots + k^3 = \frac{k^2(k+1)^2}{4}$

Step 3:
To prove the result is true for $n = k + 1$

Substituting $n = k + 1$
$1^3 + 2^3 + \cdots + k^3 + (k+1)^3 = \frac{(k+1)^2(k+2)^2}{4}$

Now,

$$LHS : 1^3 + 2^3 + \cdots + k^3 + (k+1)^3$$
$$= \frac{k^2(k+1)^2}{4} + (k+1)^3 \; .....(from\ Step 2)$$
$$= (k+1)^2[\frac{k^2}{4} + (k+1)]$$
$$= (k+1)^2[\frac{k^2+4k+4}{4}]$$
$$= (k+1)^2[\frac{(k+2)^2}{4}]$$
$$= \frac{(k+1)^2(k+2)^2}{4}$$

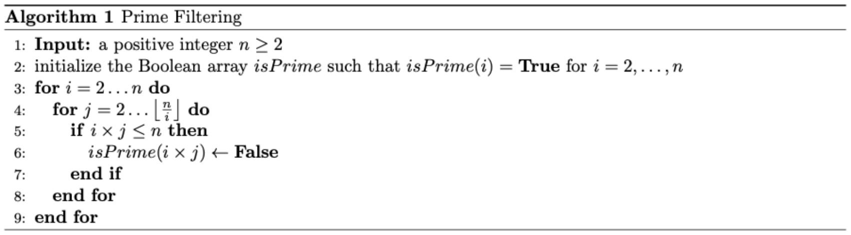
Therefore LHS = RHS

So the result is true for $n = k + 1$

Hence it's proved by mathematical induction that $1^3 + 2^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$ for every positive integer n

4. Consider the following prime filtering algorithm that outputs all the prime numbers in 2, . . . , n (the pseudo code is presented in Algorithm 1).

   - Please prove this algorithm is correct (that is, a positive integer k that $2 \leq k \leq n$ is a prime if and only if $isPrime(k) = \textbf{True}$).
   - Please calculate the time complexity under the Big-$\mathcal{O}$ notation.



Algorithm 1 Prime Filtering

```
Input: a positive integer $n \geq 2$
initialize the Boolean array $isPrime$ such that $isPrime(i) = $ True for $i = 2, \ldots, n$
for $i = 2 \ldots n$ do
  for $j = 2 \ldots \lfloor \frac{n}{i} \rfloor$ do
    if $i \times j \leq n$ then
      $isPrime(i \times j) \leftarrow$ False
    end if
  end for
end for
```

Solution

A prime number is a natural number greater than 1 that is not a product of two smaller natural numbers. Rigorously speaking, a natural number n is a prime, if and only if there does not exist two natural $a, b \in [2, n-1]$numbers such that $n = a \times b$

We can prove the correctness of the algorithm by:

1) If k is a prime number than $isPrime(k) = True$

2) If $isPrime(k) = True$ than it means that k is a prime number

Proof 1:

- Assume k is prime
- The algorithm initially sets all elements in the $isPrime$ array to True, including k
- As the algorithm iterates and marks multiples of other numbers as False, it never marks k as False because k is not a multiple of itself.
- Thus, $isPrime(k)$ remains True

Proof 2:

- Assume $isPrime(k)$ is True
- This means that during the execution of the algorithm, k was not marked as False when checking its multiples
- The only way this can happen is if k is not divisible by any number in the range 2 to $k - 1$.
- If k is not divisible by any number in this range, then it is prime.

We can also prove using Contradiction:

Proof: Assume $isPrime(k) = True$ but k is not a prime number

- If k is not a prime number. This means that k has at least one positive divisor other than 1 and itself.
- Since $isPrime(k)$ is True, it implies that k has not been marked as non-prime during the execution of the algorithm.
- If k has at least one positive divisor other than 1 and itself, then there must exist some prime number p such that p divides k evenly.
- However, if k is divisible by a prime number p, then k would have been marked as non-prime during the execution of the algorithm when i becomes equal to p.
- This contradiction arises because we assumed that $isPrime(k)$ is True, but we have shown that if k is not a prime, then $isPrime(k)$ must be False.
- Therefore, our initial assumption that $isPrime(k)$ is True, but k is not a prime number, must be false.
- Hence, it follows that if $isPrime(k)$ is True, then k is indeed a prime number.

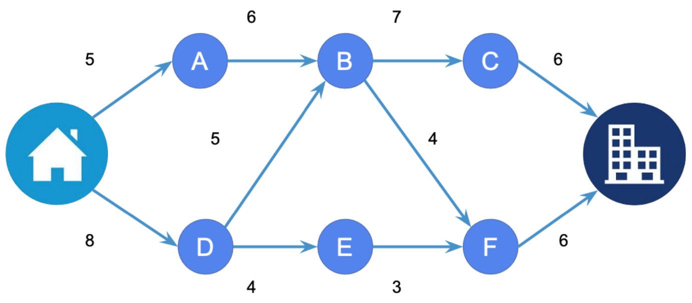
Analyzing the time complexity of the given algorithm using Big-O notation.

$$\begin{aligned}
for\ i = 2\ j\ runs\ \frac{n}{2}\ times \\
for\ i = 3\ j\ runs\ \frac{n}{3}\ times \\
for\ i = 4\ j\ runs\ \frac{n}{4}\ times \\
So\ on\ .... \\
for\ i = n\ j\ runs\ \frac{n}{n} = 1\ times
\end{aligned}$$

$$\begin{aligned}
That\ means\ j\ runs\ in\ total\ for &= \left(\frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \ldots + 1\right) \\
&= n \times \left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}\right) \\
&\leq n \times \left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}\right) \\
&\leq n \times log(n)\ ....(i.e\ \mathcal{O}(nlog(n)))
\end{aligned}$$

Therefore the time complexity of the algorithm is $\mathcal{O}(nlog(n))$

5. Amy usually walks from Amy's house ("H") to SGM ("S") for CSCI 570. On her way, there are six crossings named from A to F. After taking the first course, Amy denotes the six crossings, the house, and SGM as 8 nodes, and write down the roads together with their time costs (in minutes) in Figure 1. Could you find the shortest path from Amy's house to SGM? You need to calculate the shortest length, and write down all the valid paths.



Solution

There exists two shortest path from Amy's house to SGM with the length of 21

$H \to A \to B \to F \to S$

$H \to D \to E \to F \to S$

Below listed are all the valid paths along with their distance

$H \to A \to B \to C \to S$

Distance: $5 + 6 + 7 + 6 = 24$

$H \to A \to B \to F \to S$

Distance: $5 + 6 + 4 + 6 = 21$

$H \to D \to B \to C \to S$

Distance: $8 + 5 + 7 + 6 = 26$

$H \to D \to B \to F \to S$

Distance: $8 + 5 + 4 + 6 = 23$

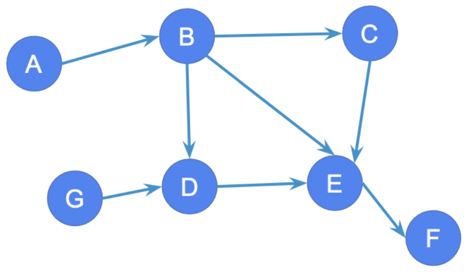
$H \to D \to E \to F \to S$

Distance: $8 + 4 + 3 + 6 = 21$

6. According to the Topological Sort for DAG described in Lecture 1, please find one possible topological order of the graph in Figrue 2. In addition, could you find all the possible topological orders?

Solution

For finding topological order we follow below steps



1. Select a vertex that has zero in-degree.
2. Add the vertex to the output.
3. Delete this vertex and all its outgoing edges.
4. Repeat.

So by following the above steps

- There are two vertex that have in-degree zero i.e A and G so we select A and it to output
- Then delete all the outgoing edges from A
- Then we have two vertex that have in-degree zero i.e B and G we select G and it to output
- Then delete all the outgoing edges from G
- Then we have only one vertex with in-degree zero i.e B we select B and it to output
- Then delete all the outgoing edges from B
- Then we have two vertex that have in-degree zero i.e D and C we select D and it to output
- Then delete all the outgoing edges from D
- Then we have only one vertex with in-degree zero i.e C we select C and it to output
- Then delete all the outgoing edges from C
- Then we have only one vertex with in-degree zero i.e E we select E and it to output
- Then delete all the outgoing edges from E
- Then we have only one vertex with in-degree zero i.e F we select F and it to output
- Then delete all the outgoing edges from F

So one possible topological order is : $A \to G \to B \to D \to C \to E \to F$

All the possible topological orders are:
$A \to B \to C \to G \to D \to E \to F$
$A \to B \to G \to C \to D \to E \to F$
$A \to B \to G \to D \to C \to E \to F$
$A \to G \to B \to C \to D \to E \to F$
$A \to G \to B \to D \to C \to E \to F$
$G \to A \to B \to C \to D \to E \to F$
$G \to A \to B \to D \to C \to E \to F$

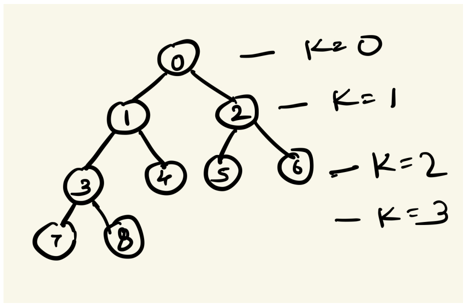
7. A binary tree is a rooted tree in which each node has two children at most. A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible. For a complete binary tree T with k nodes, suppose we number the node from top to down, from left to right with $0, 1, 2, ..., (k-1)$. Please solve the following two questions:

   - For any of the left most node of a layer with label t, suppose it has at least one child, prove that its left child is $2t+1$.
   - For a node with label t and suppose it has at least one child, prove that its left child is $2t+1$.

   Solution

   Proof by Mathematical Induction

   Consider the below Complete Binary Tree T



   Statement 1: For any leftmost node of a layer with label t that has at least one child, its left child is $2t+1$.

   Base Case ($t = 0$): For the first layer ($k = 0$), the leftmost node has label ($t = 0$) root node, and its left child is $2t+1 = 2 \times 0 + 1$. This is true since the left child of the root node is the first node with label 1.

   Induction Hypothesis: Assume that for some layer, the leftmost node with label x has a left child labeled $2x+1$.

Induction Step: We need to prove that for the leftmost node with label $x + 1$ also has a left child labeled $2(x + 1) + 1$.

Now Suppose we consider the leftmost node with label x it will be labelled based on the total number of nodes used before it that will the total sum of nodes present in each layer above the layer in which label x is present we can calculate as shown below

$$2^0 + 2^1 + \cdots + (2)^k = (2)^{k+1} - 1 = x \cdots (1)$$

$$\begin{aligned} So,\ 2x + 1 &= 2(2^{k+1} - 1) + 1 \\ 2x + 1 &= 2^{k+2} - 1 \\ 2x &= 2^{k+2} - 2 \\ x &= (2)^{k+1} - 1 \cdots (2) \end{aligned}$$

So (1) equal to (2)

Now we will prove for $x + 1$:

$$\begin{aligned} x + 1 &= (2)^{k+1} - 1 \\ x &= (2)^{k+1} - 2 \cdots (3) \end{aligned}$$

$$\begin{aligned} So,\ 2(x + 1) + 1 &= 2(2^{k+1} - 2 + 1) + 1 \\ 2x + 3 &= 2(2^{k+1} - 1) + 1 \\ 2x + 3 &= 2^{k+2} - 1 \\ 2x &= 2^{k+2} - 4 \\ x &= (2)^{k+1} - 2 \cdots (4) \end{aligned}$$

So (3) equal to (4)

Hence it is proved that for any leftmost node of a layer with label t that has at least one child, its left child is $2t + 1$.

Statement 2: For a node with label t and suppose it has at least one child, prove that its left child is $2t + 1$.

Induction Hypothesis: Assume that for some layer, the node with label x has a left child labeled $2x + 1$.

Induction Step: We need to prove that for the node with label $x + 1$ also has a left child labeled $2(x + 1) + 1$.

From the statement 1 we saw that leftmost node with $t = (2)^{k+1} - 1$ so as in this statement we need to prove for any node we will do it for $t + 1$ node which will be $t = (2)^{k+1} - 1 + 1$ that means $t = (2)^{k+1}$

$$x = (2)^{k+1} \cdots (1)$$

$$\begin{aligned} So,\ 2x+1 &= 2(2^{k+1})+1 \\ 2x+1 &= 2^{k+2}+1 \\ 2x &= 2^{k+2} \\ x &= (2)^{k+1} \cdots (2) \end{aligned}$$

So (1) equal to (2)

Now we will prove for $x+1$:

$$\begin{aligned} x+1 &= (2)^{k+1} \\ x &= (2)^{k+1} - 1 \cdots (3) \end{aligned}$$

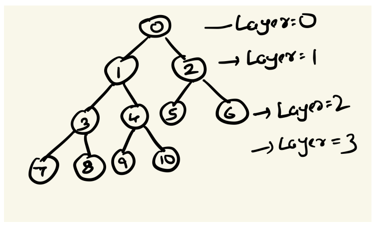
$$\begin{aligned} So,\ 2(x+1)+1 &= 2(2^{k+1}-1+1)+1 \\ 2x+3 &= 2(2^{k+1})+1 \\ 2x+3 &= 2^{k+2}+1 \\ 2x &= 2^{k+2}-2 \\ x &= (2)^{k+1}-1 \cdots (4) \end{aligned}$$

So (3) equal to (4)

Hence it is proved that for a node with label t and suppose it has at least one child, prove that its left child is $2t+1$.

Proof by Example

Consider the below Complete Binary Tree T



Now, let's apply the two statements to this tree:

Statement 1:

- The leftmost node in layer 2 is node $t = 3$.

- Node $t = 3$ has a left child, which is node 7.
- According to the statement $2t + 1 = 2 \times 3 + 1 = 7$
- Node 7 indeed has the label 7, which matches the expected result.

Hence Statement 1 is true

Statement 2:

- Consider node 4 with label $t = 4$.
- Node 4 has a left child, which is node 9.
- According to the statement, $2t + 1 = 2 \times 4 + 1 = 9$.
- Node 9 indeed has the label 9, which matches the expected result.

Hence Statement 2 is true

8. Consider a full binary tree (all nodes have zero or two children) with k nodes. Two operations are defined: 1) $removeLastNodes()$: removes nodes whose distance equals the largest distance among all nodes to the root node; 2) $addTwoNodes()$: adds two children to all leaf nodes. The cost of either adding or removing one node is 1. What is the time complexity of these two operations, respectively? Suppose the time complexity to obtain the list of nodes with the largest distance to the root and the list of leaf nodes is both $\mathcal{O}(1)$.

   Solution

   In a full binary tree with 'k' nodes, the number of nodes at the largest distance from the root (leaf nodes) can be expressed as $\frac{k+1}{2}$

   1) Time Complexity of $removeLastNodes()$:

   Now when the '$removeLastNodes()$' operation is performed we are removing the leaf nodes from the tree. Since it's mentioned that obtaining the list of nodes with the largest distance to the root is $\mathcal{O}(1)$, the time complexity for this operation can be calculated based on how many nodes are to be removed.

   Therefore for removing $\frac{k+1}{2}$ leaf nodes time complexity is $\mathcal{O}(\frac{k+1}{2})$ which simplifies to $\mathcal{O}(k)$

   Thus the time complexity of $removeLastNodes()$ is $\mathcal{O}(k)$

   2) Time Complexity of $addTwoNodes()$:

   Now when the '$addTwoNodes()$' operation is performed we add two children to each leaf nodes of the tree.Since it's mentioned that obtaining the list of leaf nodes is $\mathcal{O}(1)$ we only need to consider the time complexity for adding the nodes.

   Therefore as their are $\frac{k+1}{2}$ leaf nodes and accessing them and adding 2 nodes will result in $\frac{k+1}{2}$ times 2 additions, each with a cost of 1

   Thus the time complexity of $addTwoNodes()$ is $\mathcal{O}(k)$

9. Given a sequence of n operations, suppose the i-th operation cost 2^{j-1} if $i = 2^j$ for some integer j; otherwise, the cost is 1. Prove that the amortized cost per operation is $\mathcal{O}(1)$.

Solution

Proof by Aggregate Analysis

In a sequence of n operations there are $log_2(n) + 1$ exact powers of 2, namely $1, 2, 4..., 2^{log_2(n)}$. The total cost of these is a geometric sum

$$\sum_{j=0}^{log_2(n)} 2^{j-1} = 2^{log_2(n)+1} - 1 \leq 2^{log_2(n)+1} = 2n$$

The rest of the operations are cheap, each having a cost of 1, and there are $n - log(n) \leq n$ such operations

Therefore,

$TotalCost \leq 2n + n = 3n$

So, $Amortizedcost = \frac{TotalCost}{Number\ of\ Operations} = \frac{3n}{n} = 3 = \mathcal{O}(1)$

So the amortized cost per operation is $\mathcal{O}(1)$

Hence Proved

Proof by Accounting Method

The actual cost of the i-th operation is

$$c_i = \begin{cases} 2^{j-1}, & \text{if } i = 2^j \\ 1 & \text{otherwise} \end{cases}$$

We assign the amortized costs as follows:

$$\hat{c_i} = \begin{cases} 2, & \text{if } i = 2^j \\ 3 & \text{otherwise} \end{cases}$$

Now an operation, which is an exact power of 2 uses all previously accumulated credit plus one unit of its own amortized cost to pay its true cost. It then assigns the remaining unit as credit. On the other hand, if i is not an exact power of 2, then the operation uses one unit to pay its actual cost and assigns the remaining two units as credit. This covers the actual cost of the operation. We still have to show that there will always be enough credit to pay the cost of any power-of-2 operation. Clearly this is the case for the first operation ($j = 1$), which happens to be an exact power of two.

So now we prove that

Suppose After 2^{j-2} :th operation there is one unit of credit. Between operations 2^{j-2} and 2^{j-1} there are $2^{j-1}-1$ operations none of which is an exact power of 2. Each assigns two units as credit resulting to total of $1+2(2^{j-1}-1)=2^j-1$ accumulated credit before 2^j th (i-th) operation. This, together with one unit by its own, is just enough to cover its true cost. Therefore the amount of credit stays non negative all the time, and the total amortized cost is an upper bound for the total actual cost.

So amortized cost per operation $\hat{c_i} \leq 3$

actual total cost $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c_i} \leq \sum_{i=1}^{n} 3 = 3n$ ...(i.e $\mathcal{O}(n)$)

So the amortized cost per operation is $= \frac{\mathcal{O}(n)}{n} = \mathcal{O}(1)$

Hence Proved

10. Consider a singly linked list as a dictionary that we always insert at the beginning of the list. Now assume that you may perform n insert operations but will only perform one last lookup operation (of a random item in the list after n insert operations). What is the amortized cost per operation?

    Solution

    Each insert operation involves adding a new item at the beginning of the list, which takes $\mathcal{O}(1)$ time.

    Now, let's consider the lookup operation. Since it's performed after n insert operations, the list could have n items. The worst-case time complexity for searching for an item in an unsorted singly linked list of n items is $\mathcal{O}(n)$. However, since it's mentioned a "random item" we can assume an average-case analysis, which would mean, on average, we might need to look at half of the items before finding the desired one.

    So, the average cost of the lookup operation is $\mathcal{O}(\frac{n}{2})$, which simplifies to $\mathcal{O}(n)$.

    Total cost of operations $=$ n insert operations $+$ 1 lookup operation

    Total cost of operations $=\mathcal{O}(n \times 1) + \mathcal{O}(1 \times n) = \mathcal{O}(n)$

    Total number of operations $= \mathcal{O}(n+1)$

    Amortized cost per operation $= \frac{Total\ cost\ of\ operations}{Total\ number\ of\ operations}$

    Amortized cost per operation $= \frac{\mathcal{O}(n)}{\mathcal{O}(n+1)} \leq \frac{\mathcal{O}(n+1)}{\mathcal{O}(n+1)} = \mathcal{O}(1)$

    So, amortized cost per operation is $\mathcal{O}(1)$