CS456: Algorithm Design and Analysis

Elikem Asudo Tsatsu Gale-Zoyiku

February 25, 2024

Assignment 3

Question 1

Recursive Calls of Partition Method

List:

$$Q, U, E, B, R, A, C, H, O$$

When the pivot is List[0] == Q:

$$[A, E, B, C, H, O], [Q], [U, R]$$

For sublist $[A, E, B, C, H, O]$ with pivot Left[0] == A:

$$[A], [E, B, C, H, O]$$

For sublist $[E, B, C, H, O]$ with pivot Right[0] == E:

$$[B, C], [E], [H, O]$$

For sublist $[B, C]$ with pivot Left[0] == B:

$$[B], [C]$$

For sublist $[H, O]$ with pivot Right[0] == H:

$$[H], [O]$$

For sublist $[U, R]$ with pivot Right[0] == U:

$$[R], [U]$$

List After Each Partition Call

After Each Call of Partition Method:

$$[Q, U, E, B, R, A, C, H, O]$$

$$[A, E, B, C, H, O], [Q], [U, R]$$

$$[A], [E, B, C, H, O], [Q], [U, R]$$

$$[A], [B, C], [E], [H, O], [Q], [U, R]$$

$$[A], [B], [C], [E], [H,], [O], [Q], [U], [R] \textit{(singletons so recursive calls end)}$$

Sorted list:

$$[A, B, C, E, H, O, Q, R, U]$$

Question 2

Analyzing the running times of each algorithm:

Algorithm A

- Divides the problem of size n into 5 subproblems of size $n/2$.
- Recursively solves each subproblem.
- Combines the solutions in linear time.

The recurrence relation for Algorithm A can be expressed as:

$$T(n) = 5T(n/2) + O(n)$$

$$A = 5, B = 2, k = 1, p = 0, and A > b^k$$

By using the Master Theorem, the time complexity of Algorithm A is $O(n^{\log_b a}) == O(n^{\log_2 5})$ which is equivalent to $O(n^{2.322})$

Algorithm B

- Divides the problem of size n by recursively solving two subproblems of size $n-1$
- Combines the solutions in constant time.

The recurrence relation for Algorithm B can be expressed as:

$$T(n) = 2T(n-1) + O(1)$$

Solving the recurrence relation using substitution:

$$\begin{aligned} T(n) &= 2T(n-1) + O(1) \\ &= 2[2T(n-2) + O(1)] + O(1) \\ &= 2^2T(n-2) + 2O(1) + O(1) \\ &= 2^3T(n-3) + 2^2O(1) + 2O(1) + O(1) \\ &= \cdots \\ &= 2^kT(n-k) + 2^{k-1}O(1) + \cdots + 2^2O(1) + 2O(1) + O(1) \end{aligned}$$

At each step, the problem size n reduces by 1, so $n - k = 1$ when $k = n - 1$ (base case).

When $k = n - 1$:

$$T(n) = 2^{n-1}T(1) + 2^{n-2}O(1) + \cdots + 2^2O(1) + 2O(1) + O(1)$$

$T(1)$ is a constant because only the constant time portion of the relation will remain when $n = 1$.

Thus, T(1) is denoted as c.

$$T(n) = 2^{n-1}c + 2^{n-2}O(1) + \cdots + 2^2O(1) + 2O(1) + O(1)$$

In this expression, the dominant term is $2^{n-1}c$. Dropping the lower-order terms and constants,

the time complexity of Algorithm B:

$$T(n) = O(2^{n-1}) = O(2^n)$$

Therefore, the time complexity of Algorithm B is $O(2^n)$.

Algorithm C

- Divides the problem of size n into nine subproblems of size $n/3$.
- Recursively solves each subproblem.
- Combines the solutions in $O(n^2)$ time.

The recurrence relation for Algorithm C can be expressed as:

$$T(n) = 9T(n/3) + O(n^2)$$

$$A = 9, B = 3, k = 2, p = 0, and A == b^k$$

By using the Master Theorem, the time complexity of Algorithm C is $O(n^k) == O(n^2)$.

Comparison of Time Complexities

- Algorithm A has a time complexity of $O(n^{\log_2 2.322})$.
- Algorithm B has a time complexity of $O(2^n)$.
- Algorithm C has a time complexity of $O(n^2)$.

Algorithm C has the lowest time complexity of $O(n^2)$.

Therefore, Algorithm C would be the preferred choice for solving the problem of size n.

Question 3

a

Algorithm for Identifying the Heavier Ball in a Group of 9 Balls

- First Iteration: Divide the 9 balls into three groups of three balls each.
  - Step 1: Place three balls on each side of the weighing scale.
  - Step 2: If the scale tips to one side, it indicates that the heavier ball is in that group of three. If the scale remains balanced, the heavier ball is among the remaining three balls.
  - Step 3: Take the group of three balls that contains the heavier ball (if any) and set the other six balls aside.

  Since there are three balls on each side of the scale, there are only two possible outcomes: one side is heavier, indicating that the heavier ball is among those three balls, or the scale remains balanced, indicating that the heavier ball is among the remaining three balls.

- Second Iteration: Divide the three remaining balls into three groups of one ball each.
  - Step 1: Place one ball on each side of the weighing scale, leaving one ball aside.
  - Step 2: If one side of the scale is heavier, it indicates that the heavier ball is among those two balls. If the scale remains balanced, the heavier ball is the one left aside.

  With only three balls remaining, there are only two possible outcomes: one side is heavier, indicating that the heavier ball is among those two balls, or the scale remains balanced, indicating that the heavier ball is the one left aside.

b

Algorithm for Identifying the Heavier Ball in a Group of N Balls where N is a perfect power of 3

To solve this problem where N-1 balls are of exactly the same weight and one ball is heavier, an extended form of the previous algorithm can be used.

This technique would divide the balls into three equal groups in each iteration and compare the weights of the groups on the balance:

- Initialization:
  - Divide the N balls into three equal groups of size N/3 each.
  - Place one group on each side of the balance, leaving one group aside.
- Weighing:
  - If the balance tips to one side, it indicates that the heavier ball is among the balls on that side.
  - If the balance remains balanced, it means the heavier ball is among the balls that were left aside.
- Recursion:
  - Recursively apply the same process to the group of balls identified as potentially containing the heavier ball.
- Base case:
  - When the number of balls in a group becomes 1, that ball is the heavier one.

- Termination:
  - The process terminates when, in each iteration, N is a perfect power of 3.
  - The number of iterations needed to find the heavier ball is logarithmic with base 3.
  - Therefore, the number of times the balance will be weighed is approximately log3(N).

Input : N balls, where N is a perfect power of 3
Output: The index of the heavier ball
Function `IdentifyHeavierBall(`*balls*`)`**:**
  if *length of balls is 1* **then**
    return the index of that single ball as it will be the heavier ball;
  end
    - Divide the balls into three equal groups of size N/3 each and assign them to groups A, B, and C;
    - Place group A on the left side of the balance and group B on the right side, leaving group C aside;

  if *the balance tips to the side of A* **then**
    `IdentifyHeavierBall(`*group A*`)` The heavier ball is in group A;
  end
  else if *the balance tips to the other side* **then**
    `IdentifyHeavierBall(`*group B*`)` The heavier ball is in group B;
  end
  else
    `IdentifyHeavierBall(`*group C*`)` The heavier ball is in group C because A and B weigh the same;
  end
`IdentifyHeavierBall(`*N balls*`)`;

c

Let T(N) be the number of weighings needed to find the heavier ball among N balls. Since the algorithm divides the balls into three equal groups in each iteration, and N is a perfect power of 3, the recurrence relation can be expressed as follows:

$$T(N) = T\left(\frac{N}{3}\right) + O(1)$$

This recurrence relation represents the number of weighings needed to find the heavier ball among N balls by recursively applying the same process to the group of balls identified as potentially containing the heavier ball. Each recursion involves dividing the number of balls by 3 and adding some constant time to account for the weighing operation in each iteration.
The base case of the recurrence relation occurs when there is only one ball left to be weighed, in which case T(1) = 1, as no additional weighings are needed, and the index of the ball is returned.
Therefore, the recurrence relation for the algorithm is:

$$T(N) = \begin{cases} 1 & \text{if } N = 1 \\ T\left(\frac{N}{3}\right) + 1 & \text{if } N > 1 \end{cases}$$

d

$$T(N) = T\left(\frac{N}{3}\right) + 1$$

$$T\left(\frac{N}{3}\right) = T\left(\frac{N}{9}\right) + 1$$

$$T\left(\frac{N}{9}\right) = T\left(\frac{N}{27}\right) + 1$$

After k divisions:

$$T\left(\frac{N}{3^k}\right) = T(1) + k$$

Since $N = 3^k$, k:

$$3^k = N$$

$$k = \log_3(N)$$

Substituting k back into the expression:

$$T\left(\frac{N}{3^{\log_3(N)}}\right) = T(1) + \log_3(N)$$

$$T(1) = 1$$

Therefore, the closed-form solution for the recurrence relation is:

$$T(N) = \log_3(N) + 1$$

equivalent to

$$O(\log_3(N))$$

Using induction to prove:

Base Case:

When $N = 1$, $T(1) = \log_3(1) + 1 = 0 + 1 = 1$

This satisfies the base case.

Inductive Step:
Assume that for all $N' < N$, $T(N') = \log_3(N') + 1$.

Consider $T(N)$:

$$T(N) = T\left(\frac{N}{3}\right) + 1$$

By the inductive hypothesis, we have:

$$T\left(\frac{N}{3}\right) = \log_3\left(\frac{N}{3}\right) + 1$$

Hence,

$$T(N) = \log_3\left(\frac{N}{3}\right) + 1 + 1$$

$$T(N) = \log_3(N) - 1 + 1 + 1$$

$$T(N) = \log_3(N) + 1$$
$$T(N) = O(log_3(N))$$

This completes the proof by induction.

Question 4

a

Function `topologicalSort(`*graph*`):`

- Initialize an empty stack S to store vertices in the correct order;
- Mark all vertices as unvisited;

for *each vertex in graph* **do**
  if *vertex is unvisited* **then**
    performDFS(vertex, S);
  end
end
while S *is not empty* **do**
  vertex = pop from stack S;
  print vertex;
end

initialize an empty stack S to store vertices in the correct order;
mark all vertices as unvisited;
for *each vertex in graph* **do**
  if *vertex is unvisited* **then**
    performDFS(vertex, S);
  end
end
while S *is not empty* **do**
  vertex = pop from stack S;
  print vertex;
end

Function `performDFS(`*vertex, S*`):`

mark vertex as visited;
for *each neighbor of vertex* **do**
  if *neighbor is unvisited* **then**
    performDFS(neighbor, S);
  end
end
push vertex onto stack S;

b

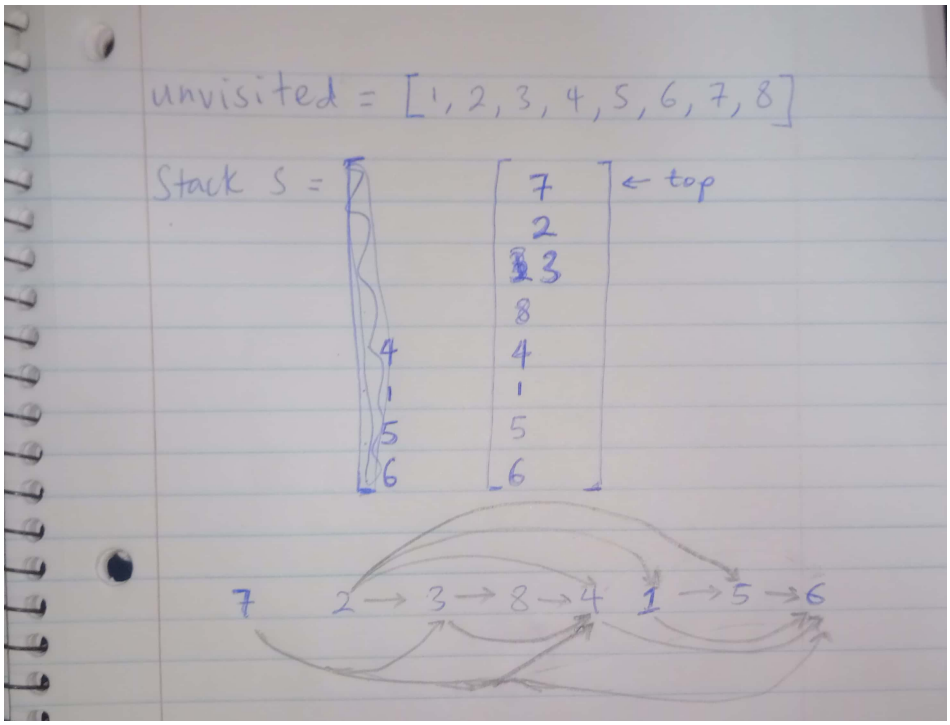

Figure 1: Topological Sort Using DFS

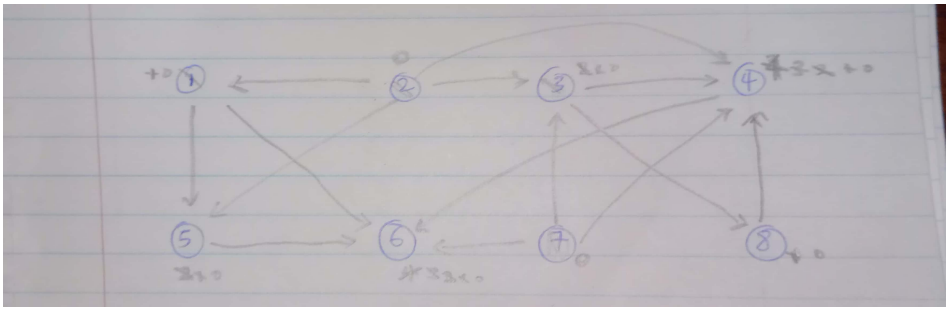

Figure 2: Source Removal

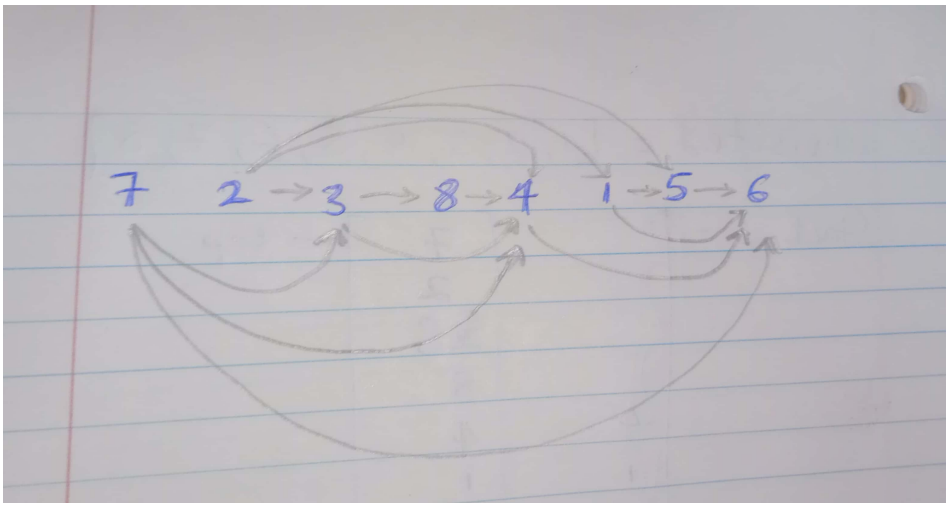

Figure 3: Topological Sort with Source Removal

c

d

1. Initialization:
   - Start with an empty list to store the topological order of vertices.
   - Identify all source vertices in the DAG. These are vertices with no incoming edges.
2. Source Removal:
   - While there are source vertices remaining in the DAG:
     - Select a source vertex from the remaining source vertices.
     - Remove the selected source vertex from the graph.
     - Update the remaining source vertices (i.e., vertices that become sources after the removal of the selected vertex).
     - Add the selected source vertex to the topological order list.
3. Completion:
   - After removing all source vertices, the topological order list contains the vertices in a valid topological order.
4. Output:
   - Return the obtained topological order list.

e

25 TopoSorts can be obtained from the given graph. The following are 25 possible topological orderings of the vertices in the graph:

TopoSort1 [2, 1, 5, 7, 3, 8, 4, 6]

TopoSort2 [2, 1, 7, 3, 8, 4, 5, 6]

TopoSort3 [2, 1, 7, 3, 8, 5, 4, 6]

TopoSort4 [2, 1, 7, 3, 5, 8, 4, 6]

TopoSort5 [2, 1, 7, 5, 3, 8, 4, 6]

TopoSort6 [2, 7, 1, 3, 8, 4, 5, 6]

TopoSort7 [2, 7, 1, 3, 8, 5, 4, 6]

TopoSort8 [2, 7, 1, 3, 5, 8, 4, 6]

TopoSort9 [2, 7, 1, 5, 3, 8, 4, 6]

TopoSort10 [2, 7, 3, 1, 8, 4, 5, 6]

TopoSort11 [2, 7, 3, 1, 8, 5, 4, 6]

TopoSort12 [2, 7, 3, 1, 5, 8, 4, 6]

TopoSort13 [2, 7, 3, 8, 1, 4, 5, 6]

TopoSort14 [2, 7, 3, 8, 1, 5, 4, 6]

TopoSort15 [2, 7, 3, 8, 4, 1, 5, 6]

TopoSort16 [7, 2, 1, 3, 8, 4, 5, 6]

TopoSort17 [7, 2, 1, 3, 8, 5, 4, 6]

TopoSort18 [7, 2, 1, 3, 5, 8, 4, 6]

TopoSort19 [7, 2, 1, 5, 3, 8, 4, 6]

TopoSort20 [7, 2, 3, 1, 8, 4, 5, 6]

TopoSort21 [7, 2, 3, 1, 8, 5, 4, 6]

TopoSort22 [7, 2, 3, 1, 5, 8, 4, 6]

TopoSort23 [7, 2, 3, 8, 1, 4, 5, 6]

TopoSort24 [7, 2, 3, 8, 1, 5, 4, 6]

TopoSort25 [7, 2, 3, 8, 4, 1, 5, 6]

Question 5

1. Base Cases:
    - If the tree is empty, return 0 (no levels).
    - If the tree has only one node, return 1 (one level).
2. Divide:
    - Divide the binary tree into its left and right subtrees.
3. Conquer:
    - Recursively compute the number of levels in the left and right subtrees.
4. Combine:
    - The number of levels in the entire tree is the maximum of the number of levels in its left and right subtrees plus one (to account for the current level).

```
function computeLevels(tree):
    if tree is empty:
        return 0
    else if tree has only one node:
        return 1
    else:
        leftLevels = computeLevels(left subtree of tree)
        rightLevels = computeLevels(right subtree of tree)
        return max(leftLevels, rightLevels) + 1
```

Time Complexity:

The time complexity of this divide-and-conquer algorithm for computing the number of levels in a binary tree can be analyzed as follows:

- At each level of recursion, the algorithm divides the problem into two subproblems (left and right subtrees) and conquers each subproblem.
- The recurrence relation for the time complexity is $T(n) = 2T(n/2)+O(1)$, where n is the number of nodes in the binary tree.
- $$A = 2, B = 2, k = 0, p = 0, and A > b^k$$
- Using Master Theorem, the efficiency class of the algorithm is $\Theta(n^{log_2(2)} = \Theta(n))$.

Therefore, the time efficiency class of this algorithm is $O(n)$,
where n is the number of nodes in the binary tree.

Question 6

(a) For the recurrence relation $T(n) = 100T(n^{1/10}) + 100(\log n)^2$, a change of variables is needed to solve:

- Let $n = 2^m$. Then, the recurrence becomes:

$$T(2^m) = 100T(2^{m/10}) + 100m^2$$

- Let $S(m) = T(2^m)$. Thus, the recurrence becomes:

$$S(m) = 100S(m/10) + 100m^2$$

- This is of the form $S(m) = aS(m/b) + f(m)$. The solution is:

$$S(m) = O(m^2)$$

- Substituting back the original variables, the solution is:

$$T(n) = T(2^m) = S(m) = O((\log n)^2)$$

(b) Considering the recurrence relation $T(n) = 7T(n/8) + n \ln n$, using Master Theorem to solve:

- Given $a = 7$, $b = 8$, $k = 1$, $p = 1$, and $A < B^k$ and $f(n) = n \ln n$, as $f(n) = O(n^k \log^p n)$ for $k = 1$, Master Theorem applies. Hence, the solution is:

$$T(n) = O(n \log n)$$

(c) In the recurrence relation $T(n) = 9T(n/3) + n^2 \log n$, using Master Theorem to solve:

- Given $a = 9$, $b = 3$, $k = 2$, $p = 1$, and $f(n) = n^2 \log n$, and $f(n) = O(n^{\log_b a} \log^{p+1} n)$ for $k = 1$, The solution is:

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

(d) For the recurrence relation $T(n) = T(n^{1/10}) + T(n^{1/5} + 2) + T(n^{1/4}) + \log n$.

Question 7

Question 8

Explanation of how the function works:

1. Base Case:
   - If the length of the input array is less than 2, indicating either an empty array or an array with only one element, the function returns 0 since there are no elements to compare.
2. Divide:
   - The function divides the input array into two equal halves.
3. Conquer:
   - It recursively calls itself on each half of the input array to compute the maximum difference within each half.
4. Combine:
   - The function then combines the results from the left and right halves to determine the maximum difference across the entire array.
   - It calculates the maximum difference as the maximum of three values:
     - The maximum difference computed from the left half.
     - The maximum difference computed from the right half.
     - The difference between the maximum and minimum values in the entire array.
5. Return Result:
   - Finally, the function returns the maximum difference computed.

The time complexity of the `max_difference` function can be analyzed using a recurrence relation. The recurrence relation is $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(1)$, where n is the size of the input array.

- At each recursive call, the function divides the input array into two equal halves, resulting in $O(1)$ time complexity for each division.
- The function then recursively calls itself on each half, resulting in $2 \cdot T\left(\frac{n}{2}\right)$ time complexity for combining the results from the left and right halves.
- Additionally, finding the minimum and maximum values in the input array using the `min` and `max` functions also takes constant time, denoted as $O(1)$.

Using the Master Theorem, the time complexity of the `max_difference` function is $O(n)$, where n is the size of the input array. This is because the time complexity of the function falls into case $A > B^k$ of the Master Theorem, where $a = 2$, $b = 2$, $k = 0$ and $p = 0$, resulting in a time complexity of $O(n^{\log_b a}) = O(n^1) = O(n)$.

Therefore, the `max_difference` function computes the maximum difference between any two elements in an input array with a time complexity of $O(n)$.