ShanghaiTech University

CS240 Algorithm Design and Analysis

Fall 2024

Problem Set 1

Due: 23:59, Oct. 13, 2024

1. Submit your solutions to the course Gradescope.
2. If you want to submit a handwritten version, scan it clearly.
3. Late homeworks submitted within 24 hours of the due date will be marked down 25%. Homeworks submitted more than 24 hours after the due date will not be accepted unless there is a valid reason, such as a medical or family emergency.
4. You are required to follow ShanghaiTech's academic honesty policies. You are allowed to discuss problems with other students, but you must write up your solutions by yourselves. You are not allowed to copy materials from other students or from online or published resources. Violating academic honesty can result in serious penalties.

Problem 1:

Sort the following functions in ascending order of growth.

$$f_1(n) = 2024^{2025^n} \tag{1}$$
$$f_2(n) = \log_2 4^n \tag{2}$$
$$f_3(n) = 2^{\frac{1}{2}\log_2 n} \tag{3}$$
$$f_4(n) = n^{\log_2 n} \tag{4}$$
$$f_5(n) = 2025^{2024^n} \tag{5}$$
$$f_6(n) = 2024^{2025} \tag{6}$$
$$f_7(n) = n^{2024} \tag{7}$$
$$f_8(n) = n^{\sqrt{n}} \tag{8}$$
$$f_9(n) = n\log_2(\log_2 n) \tag{9}$$

Solution:

(6) < (3) < (2) < (9) < (7) < (4) < (8) < (5) < (1)

Problem 2:

Analyze the time complexities of the following algorithms and explain your reasoning.

Algorithm 1

```
for $i \leftarrow 1$ to $n$ by $i \leftarrow i + 1$ do
    for $j \leftarrow i^2$ downto $0$ by $j \leftarrow j - 1$ do
        for $k \leftarrow 1$ to $j$ by $k \leftarrow k + 1$ do
            $res_1 \leftarrow res_1 + ij + jk$
        end for
        for $k \leftarrow 1$ to $j$ by $k \leftarrow 2k$ do
            $res_2 \leftarrow res_2 + ik + ij$
        end for
    end for
end for
```

Solution:

For get res_1, the outer loop is executed n times, the middle loop is executed n^2 times, and for each value $j = 1, 2, \ldots, n^2 - 1, n^2$, the inner loop has different execution times:

j	Inner iterations
1^2	1
2^2	$\sum_{k=1}^{2^2} k$
3^2	$\sum_{k=1}^{3^2} k$
$\ldots$	$\ldots$
$(n-1)^2$	$\sum_{k=1}^{(n-1)^2} k$
n^2	$\sum_{k=1}^{n^2} k$

which $\sum_{k=1}^{i^2} k = \frac{i^2(i^2+1)}{2} = \frac{1}{2}(i^4 + i^2)$. The total number steps is

$$\begin{aligned}\sum_{i=1}^{n}\frac{1}{2}(i^4+i^2) &= \frac{1}{2}(\sum_{i=1}^{n} i^4 + \sum_{i=1}^{n} i^2)\\ &= \frac{1}{2}(\frac{n(n+1)(2n+1)(3n^2+3n1)}{30} + \frac{n(n+1)(2n+1)}{6})\\ &= O(n^5)\end{aligned}$$

For res_2, for each value $j = 1, 2, \ldots, n^2-1, n^2$, the running time of inner loop is:$1 + 2 + 2^2 + ... + j$, which is no more than $1+2+3+4+...+j$, so is smaller than getting res_1.

Thus, the total complexity is $O(n^5)$.

Problem 3:

ShanghaiTech University is planning to establish new research centers throughout its campus with the goal of maximizing the total research output. The campus is modeled as an undirected graph $G = (V, E)$, where each vertex u corresponds to a potential location for a research center. Every vertex u has a nonnegative integer value p_u, representing the potential research output for that location. Due to constraints on resources, no two research centers can be placed at neighboring vertices to prevent competition for resources. Your task is to develop an algorithm that selects a subset $U \subseteq V$ of locations that maximizes the total research output $\sum_{u \in U} p_u$. Assume that the campus network G is a tree (i.e., it is acyclic).

(a) Consider the following "greedy" algorithm for placing research centers: First, choose the vertex u_0 with the highest research output from the tree and add it to the subset U. Then, remove u_0 and all its neighboring vertices from the graph, making them ineligible for further selection. Continue this process until no vertices remain. Provide a counterexample to demonstrate that this algorithm does not always produce a placement that maximizes total research output.

(b) Now, suppose that ShanghaiTech University does not have specific research output values for different locations, and instead assumes that all potential research center sites are of equal importance. The new objective is to design an algorithm that selects the largest possible number of research center sites. Describe a simple greedy algorithm for this situation and provide a proof of its correctness.

Solution:

(a) A counterexample could be a simple tree structure like a line with vertex outputs (9, 10, 9). The "greedy" algorithm might first choose the middle node with a profit of 10 and then remove its neighbors, resulting in a total output of 10. However, the optimal solution would be to place research centers on the two end nodes, yielding a higher output of 18 (9 + 9).

(b) **Greedy Algorithm:**

1. **DFS Traversal**: Pick any node u_0 as the root of the tree and perform a depth-first search (DFS), storing the nodes in a list N based on the DFS completion order.

2. **Reverse Processing**: Process the nodes in reverse order from N. For each node $v \in N$, if v and its parent have not yet been selected, add v to

the solution set U, and mark its parent as excluded.

3. **Termination**: Repeat this process until all nodes have been processed. The set U now contains the maximum number of non-adjacent sites.

Proof:

Lemma 1: For any tree with equal node weights, there is an optimal placement that includes all leaves.

1. **Base Case**: Assume that there exists an optimal solution O that excludes some leaf node v. By Lemma 1, we know there exists another solution where all leaves can be included. Thus, adding v to O and potentially removing its parent does not reduce the optimality.

2. **DFS Ordering**: As nodes are processed in reverse DFS order, the first valid node is always a leaf. By Lemma 1, including the leaf and removing its parent from consideration guarantees an optimal solution for the subtree rooted at this node.

3. **Induction on Subtrees**: By continuing to process nodes in reverse DFS order, each selected node will be valid (non-adjacent to any selected parent). As the tree is processed bottom-up, we are ensuring that every subtree receives an optimal number of research centers, leading to an optimal solution for the entire tree.

Sorting nodes using DFS takes $O(n)$ time. The greedy algorithm also takes $O(n)$ time since it processes each node once. Overall the algorithm has $O(n)$ complexity.

Problem 4:

You are driving on a highway represented by an array of fuel stations. You start at the first station. Each station in the array has a number representing how far you can drive from that station before needing to stop at another one for fuel. Assume you always have enough fuel to reach the last station. Your goal is to reach the last station using the smallest possible amount of refuel. Please provide a greedy solution for this scenario.

Example:

- **Input:** `stations = [2, 3, 0, 1, 4]`
- **Output:** 2
- **Explanation:** The minimum number of refuels to reach the last station is 2. Refuel at station 0 to drive to station 1, then refuel at station 1 to drive directly to the last station.

Solution:

Since the last station is always reachable, our goal is to minimize the number of refuels. To achieve this, we should maximize our progress at each step by always refueling at the station that allows us to reach the farthest possible point.

Greedy Algorithm:

- **Step 1:** Define `current_end` to mark the farthest point you can reach with the current number of refuels.
- **Step 2:** Traverse the array and, for each station, calculate how far you can reach. If you reach the `current_end`, it's time to refuel.
- **Step 3:** At each step, choose the station that allows you to reach the farthest possible point, maximizing your gain.
- **Step 4:** Continue this process until you reach or exceed the last station.

Proof:

Assume there exists an optimal solution (OPT) that uses fewer refuels than the greedy algorithm (Greedy). We will now use proof by contradiction to show that this assumption is false and that Greedy is indeed optimal.

We assume that both the greedy algorithm and the optimal solution make the same choices for the first i refuels, i.e., they follow the same path from the start to position p_i. We now analyze the behavior of Greedy and OPT in the $i + 1$ step.

- **OPT**: Let OPT choose to refuel at position p_{opt}, which is the optimal position to refuel from p_i.
- **Greedy**: Greedy will refuel at the farthest possible position from p_i, denoted as p_{greedy}, where $p_{\text{greedy}} = p_i + \max(\texttt{nums}[p_i])$.

Since the greedy algorithm always chooses the farthest position, we have:

$$p_{\text{greedy}} \geq p_{\text{opt}}$$

This means that the greedy algorithm will either reach or exceed the position chosen by the optimal solution, potentially requiring fewer or at least the same total refuels than OPT. Therefore, Greedy is optimal.

Problem 5:

The High-Speed Railway Network Company has decided to improve its railway connections across a vast country with n cities spread from West to East. As part of the infrastructure planning team, your job is to design an efficient network of train routes. However, the company has placed specific constraints: all trains must only travel eastward, and every passenger must be able to travel from any city to any other city to the east with at most one stop. Currently, setting up direct routes for every possible journey would require a prohibitively large number of routes, approximately $\Omega(n^2)$. Your task is to develop a more efficient plan, designing no more than $O(nlogn)$ routes while ensuring that any passenger traveling between any two cities to the east can do so with at most a single stop. Prove that your set of routes satisfies these requirements.

Divide Step. Describe the divide step of your algorithm here, making sure to mention what subproblems are produced. Include the total number of routes created during the divide step.

Combine Step. Describe the combined step of your algorithm here. Include the total number of routes created during this combined step.

Algorithm. Describe your whole algorithm here. You do not need to repeat the procedures for divide and combine, you're welcome to simply reference them here.

Correctness. Prove the correctness of your algorithm. Namely, shows that the selected routes have the property that one can travel from any city to an eastward city with at most a single connection.

Number of Routes. Express the number of routes your algorithm selects as a recurrence relation. Describe, using the recursion tree method or induction proof, how you know that this recurrence is $O(nlogn)$.

Solution:

Divide Step: Divide the set of n cities into two roughly equal halves: a "Western" half and an "Eastern" half. Recursively solve the problem for both the Western and Eastern halves. During this division, you will establish routes within each half separately.

Combine Step: After solving the subproblems for both the Western and Eastern halves, establish connections between the two halves to allow for cross-region travel. Specifically, for each city in the Western half, add a route to a single city in the Eastern half. This will ensure that any passenger from the Western half can reach the Eastern half with at most one connection.

Algorithm Outline: Recursively divide the set of cities until each subset is small enough to handle directly (typically a single city or a very small group). For each level of recursion, solve the subproblems in the Western and Eastern halves. Combine the solutions by adding direct connections between the cities in the two halves as described in the Combine Step. Return the complete set of routes.

Correctness: Every city in the Western half can reach any city in the Eastern half with at most one stop: either by taking a direct route to the Eastern half or by making one connection within the Eastern half. Since each recursive division maintains this property, the entire network will have the desired property that a passenger can travel from any city to any city to the east with at most one stop.

Number of Routes (Recurrence Relation): Let T(n) denote the number of routes required for n cities. Each divide step results in two recursive subproblems of size approximately $n/2$. Each subproblem requires $T(n/2)$ routes. The combined step creates an additional $O(n)$ route to connect the two halves. The recurrence relation is: $T(n) = 2T(n/2) + O(n)$.

Applying the induction proof. For $n = 1$, there is only one city, and no routes are needed, which satisfies the requirement with $0 = O(1)$ routes.

Assume that for any set of k cities ($k < n$), we can construct a set of routes that ensures any passenger can travel between any two cities to the east with at most one stop, using at most $O(klogk)$ routes.

The total number of routes $R(n)$ is the sum of the routes within each half and the additional routes connecting the halves:

$$R(n) = R(\lceil n/2 \rceil) + R(\lfloor n/2 \rfloor) + O(n).$$

By the inductive hypothesis, we have:

$$R(n) = O((n/2)\log(n/2)) + O((n/2)\log(n/2)) + O(n).$$

Simplifying, we obtain:

$$R(n) = O(n\log(n/2)) + O(n) = O(n\log n).$$

Applying the recursion tree method.

To analyze this recurrence, we can construct a **recursion tree**:

Root Level (Level 0): At the root, we have n cities to process, and the cost is $O(n)$.

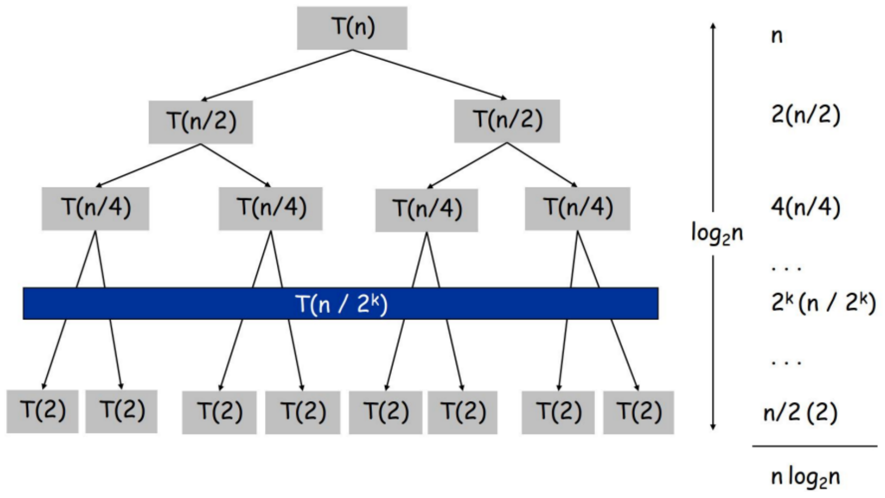

Figure 1: Recursion tree

Level 1: The problem is divided into two subproblems: L and R, each of size approximately $n/2$. The cost at this level is $2 \cdot O(n/2) = O(n)$.

Level 2: Each subproblem from Level 1 is further divided into two subproblems of size $n/4$. The cost at this level is $4 \cdot O(n/4) = O(n)$.

This pattern continues, and at each level of the tree, the total cost is $O(n)$.

Height of the Tree: The height of the recursion tree is $\log n$, as the problem size is halved at each recursive step.

The total cost of the recursion tree is the sum of the costs across all levels:

$$T(n) = O(n) + O(n) + \cdots + O(n) \quad (\text{up to } \log n \text{ levels})$$

Thus, the overall time complexity is:

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

Problem 6:

In a city rich with history, there is an old town that was once the heart of the city. However, as time passed, the streets and buildings of the old town fell into disrepair, and the layout became chaotic. The streets in this old town are connected like a linked list, one after another, and each street has a unique number representing its construction period and architectural style. Over the years, these numbers have fallen out of order, leading to inconvenient traffic, dilapidated buildings, and a decline in the quality of life for the residents. To revitalize the old town, the city government has launched the "Old Town Street Reconstruction Plan". The planners aim to restore the streets to their former glory by arranging them in an orderly sequence according to their numbers, making the layout more logical and facilitating better traffic flow and daily life. Your task is to act as the engineer for this reconstruction project, responsible for reordering these streets.

Given a linked list representing the structure of the streets, where the head of the list is *head*, you are required to use an efficient divide-and-conquer algorithm to sort all the street numbers in ascending order. The reconstructed streets must be sequentially connected, restoring the old town's orderly beauty. Please return the sorted linked list of streets.

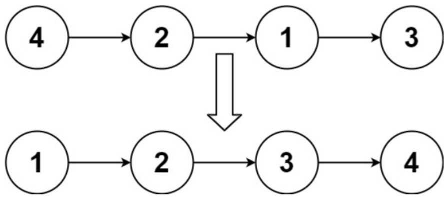

Figure 2: For example

For example, the street numbers of the old town are currently arranged as $[4, 2, 1, 3]$, with the head node $head = 4$. You need to sort them in ascending order as $[1, 2, 3, 4]$.

Solution:

To sort a linked list in ascending order, the most efficient approach is to use Merge Sort. Merge Sort is a divide-and-conquer algorithm with an average time complexity of $O(nlogn)$, which meets the problem's requirements. Merge Sort is particularly well-suited for linked lists because it allows for splitting and merging in constant time without requiring additional space like arrays.

Solution Steps:

1. Divide and Conquer Strategy: Continuously split the linked list into two halves until each sublist has only one node or is empty. Then, merge these sublists back together to create a sorted list.

2. Finding the Middle of the Linked List: To split the list into two halves, use the fast and slow pointer technique (the fast pointer moves two steps at a time, and slow pointer moves one step). When the fast pointer reaches the end, the slow pointer will be at the middle of the list. Disconnect at the middle to form two separate sublists.

3. Recursively Split the Linked List: Recursively apply the above step to each sublist until the list is reduced to its smallest units (individual nodes).

4. Merge Two Sorted Lists: Use a two-pointer approach to merge two sorted sublists. Create a dummy node as the head of the new merged list, then compare nodes from both sublists and append the smaller one to the merged list. Continue until all nodes are merged.

5. Return the Sorted List: After recursively splitting and merging, return the final sorted linked list.

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def sortList(head: ListNode) -> ListNode:
    if not head or not head.next:
        return head
    slow, fast = head, head.next
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    mid = slow.next
    slow.next = None
    left = sortList(head)
    right = sortList(mid)
    return merge(left, right)

def merge(left: ListNode, right: ListNode) -> ListNode:
    dummy = ListNode(0)
```

```
tail = dummy
while left and right:
    if left.val < right.val:
        tail.next = left
        left = left.next
    else:
        tail.next = right
        right = right.next
    tail = tail.next
tail.next = left or right
return dummy.next
```