ShanghaiTech University

CS240 Algorithm Design and Analysis

Fall 2023

Problem Set 1

Due: 23:59, Oct. 29, 2023

1. Submit your solutions to Gradescope (www.gradescope.com).
2. In "Account Settings" of Gradescope, set your FULL NAME to your Chinese name and enter your STUDENT ID correctly.
3. If you want to submit a handwritten version, scan it clearly. Camscanner is recommended.
4. When submitting your homework, match each of your solution to the corresponding problem number.

Problem 1:

Given a sequence $S = a_1, a_2, \cdots, a_n$. For some t between 1 and n, the sequence satisfies $a_1 < a_2 < \cdots < a_t$ and $a_t > a_{t+1} > \cdots > a_n$. You would like to find the maximum value of the S by reading as few elements of S as possible. Show your algorithm and analyze the time complexity of it. For example, suppose the sequence S is $1, 7, 8, 9, 4, 3, 2$, then the max value is 9.

Solution:

```
fn find_max_value(nums: Vec<i32>) -> i32 {
    let n = nums.len();
    let mut max_val = 0;
    let (mut left,mut right,mut mid) = (0,n,n/2);
    while left < right {
        if nums[mid]>nums[mid-1] && nums[mid]>nums[mid+1]
        {
        max_val = nums[mid];W
        break;
        }
        if nums[mid]>nums[mid+1]
        {
            left = mid +1;
            mid = (left+right)/2;
        }
        else
        {
            right = mid-1;
            mid = (left+right)/2
}
    }

    max_val
}
```

When traversing an array using binary search, if the current element is greater than both its adjacent elements, then stop the traversal. This node is the maximum element. Otherwise, if the sequence is monotonically increasing, update left to mid + 1. If the sequence is monotonically decreasing, update right to mid - 1.

The time complexity:

$$O(logN)$$

The space complexity

$$O(1)$$

Problem 2:

Suppose there are n trees in the campus of ShanghaiTech and their heights are denoted as $h_1, h_2, \cdots, h_n$. Now we want to find the m ($m \leq n$) trees that are closest to this height for a given arbitrary height h. Please give an efficient algorithm to achieve this. For example, suppose the input of height of trees, a target height and a target number of trees are $h_1 = 8, h_2 = 5, h_3 = 3, h_4 = 1$, $h = 4$ and $m = 2$ respectively, then the output should be h_2, h_3. Note that the height of trees in the output must be in the original order.

Solution:

1. Create an empty result list 'result' and an empty distance list 'distances'.
2. For each tree's height h_i, calculate the absolute difference between it and the target height h, and append the result to the 'distances' list.
3. Sort the 'distances' list along with the corresponding tree height list 'indices', maintaining their correspondence.
4. Select the heights of the first m trees from the sorted list and append them to the 'result' list.

The code is as follows:

```
def find_closest_trees(heights, h, m):
    distances = [abs(tree_height - h) for tree_height in heights]
    sorted_distances = quick_sort(enumerate(distances), key=lambda x:
        x[1])
    result = [f'h{i+1}' for i, _ in sorted_distances[:m]]
    return result
```

The time complexity:

$$O(n * \log n)$$

The space complexity:

$$O(n)$$

Problem 3:

Asymptotic Order of Growth. Sort all the functions below in increasing order of asymptotic (Big-Oh) growth.

1. $8n$
2. lgn
3. $10lg(lgn)$
4. $n^{3.14}$
5. n^{n^2}
6. lgn^{10lgn}
7. n^{lgn}

Solution:

The sorted outcome

$$O(10lgn(lgn) < O(lgn) < O(8n) < O(lgn^{10lgn}) < O(n^{3.14}) \leq O(n^{lgn}) < O(n^{n^2})$$

Problem 4:

Asymptotic Order of Growth. Analyze the running time of following algorithm, and express it using "Big-Oh" notation.

Algorithm 1

Input: integers $n > 0$

```
i = 0, j = 0
while i < n do
  while j < i^2 do
    j += 1
  end while
  i += 1
  j = 0
end while
return j
```

Solution:

The loop in j can be inferred to follow a function expression of n^2, so the total number of computations for this algorithm is:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Therefore, the time complexity of this algorithm is $O(n^3)$.

Problem 5:

Greedy Algorithm. You are given n sorted arrays. Your task is to merge them into a single sorted array. You can only merge two arrays at a time, and the cost of merging is the sum of the lengths of the two arrays. Your goal is to find the merge strategy with the minimum total cost.

Detailed Problem Description:

Suppose we have 3 sorted arrays:

$$[1, 3, 9], [2, 4, 6], [0, 5, 7, 8]$$

We first merge the first two arrays, the cost of merging is 6 (the sum of the lengths of the two arrays). Then, we merge the resulting array with the third array, the cost of merging is 10 (the sum of the lengths of the two arrays). So, the total cost of merging is 16.

Solution:

During each merge operation, we start by merging the two smallest arrays. This means that we need to find the two smallest arrays to merge each time, until only one array remains. We represent each array as $Array_i$. We create a min-heap based on the array lengths, with the array as the element and the array length as the key. Each time, we extract $Array_0$ and $Array_1$, which are the two smallest arrays. We then merge them, and insert the resulting merged array $Array_{new}$ back into the min-heap. This process continues until there is only one array left in the min-heap, at which point the algorithm concludes.

```
def merge_array(Input)W
    Input: n arrays

    minHeap = CreateMinHeap(Input)
    let i = 1 - n insert minHeap

    totalComparisons = 0

    while minHeap.len() >1:
        array1 = minHeap.pop()
        array2 = minHeap.pop()
        totalComparisons += length(array1) + length(array2) - 1
        mergedArray = merge(array1, array2)
        InsertHeapArray(mergeArray)
```

```
Output: totalComparisons
return Output
```

The time complexity for both inserting and adjusting the min-heap is $O(\log K)$, where K is the number of arrays. Therefore, the total time for heap adjustment is $O(K \log K)$. The total time for merging is $O(n)$, where n is the total number of elements in all the arrays. The time complexity:

$$O(nlogn)$$

The space complexity:

$$O(n)$$

proof: The greedy algorithm utilizes a Huffman tree to solve this problem, and it suffices to demonstrate the optimality of the Huffman tree. Let's assume that the Huffman tree is not the optimal binary tree. In that case, we can find a more profitable feasible tree. We define a swap operation in tree T, exchanging a pair of leaf nodes i and j. There are three possible cases for feasible trees:

(W P L = P valuei * leveli, i = 1,2,..., n, where valuei is the value of node i, and leveli is the level of node i)

Case 1: Nodes i and j are on the same level in the binary tree, meaning $Level_i = Level_j$. Thus, $value_i * Level_j + value_j * Level_i = value_i * Level_i + value_j * Level_j$. In this case, the WPL of the feasible tree is equal to the WPL of the Huffman tree.

Case 2: Node i is below node j, so $value_i <$value$_j, and$level$_i > level_j$. Because $value_i * level_j + value_j * level_i > value_i * level_i + value_j * level_j$, the WPL of the feasible tree is greater than the WPL of the Huffman tree. Therefore, in this case, the Huffman tree is superior to the feasible tree.

Case 3: It is not possible for node i to be above node j, as it would violate the definition of a Huffman tree.

In conclusion, the Huffman tree is the optimal binary tree. Hence, the greedy algorithm provides the optimal solution.

Problem 6:

Greedy Algorithm. Suppose you're a manager of a candy store. The store has different candies for sale every day. Each day, you can choose to buy a type of candy and then sell it on a future day. Your goal is to maximize your profit by buying low and selling high. However, you cannot buy candies on different days and sell them all on another day.
Given an array where the i-th element is the price of a candy on day i, design an algorithm to find the maximum profit.
Detailed Description:
For instance, consider the following input:
Candy prices: $[7, 1, 5, 3, 6, 4]$
You should buy candy on day 2 (price $= 1$) and sell it on day 3 (price $= 5$), buy again on day 4 (price $= 3$) and sell it on day 5 (price $= 6$). The total profit is $5 - 1 + 6 - 3 = 7$.

Solution:

For today's candy price, if it is greater than yesterday's candy price, we sell it and accumulate the profit.

```
pub fn max_profit(prices: Vec<i32>) -> i32 {
    prices.windows(2).map(|p| (p[1] - p[0]).max(0)).sum()
}
```

Time Complexity: $O(N)$ Space Complexity: $O(1)$

Proof: Assuming that the solution obtained by the 'greedy algorithm' is not the optimal solution, meaning we can find a feasible solution that yields more profit than the one obtained by the 'greedy algorithm'. In the difference array, apart from the entries with positive differences, there are also entries with zero differences and negative differences. The result obtained by the 'greedy algorithm' is the sum of all entries with positive differences. There are three possible scenarios:

If the feasible solution, when based on the 'greedy algorithm', includes entries with zero differences, the result will be the same as the one obtained by the 'greedy algorithm'. Therefore, adding entries with zero differences will not yield a better result than what the 'greedy algorithm' produces.

If the feasible solution, when based on the 'greedy algorithm', includes entries with negative differences, adding a negative number will definitely result in a smaller total than what the 'greedy algorithm' produces. Hence, adding entries

with negative differences will always lead to a worse result than the 'greedy algorithm'.

If the feasible solution, when based on the 'greedy algorithm', removes any entry with a positive difference, as above, the result will definitely be smaller than what the 'greedy algorithm' produces. Thus, none of the components of the 'greedy algorithm' can be removed.

Thus,greedy algoirthm is the best solution.