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:

We can solve the problem by a variation of binary search as follows. The time complexity of the algorithm is $O(\log n)$.

```
function findMaxValue(arr, left, right)
    if left == right:
        return arr[left]

    mid = (left + right) / 2
    if array[mid] < array[mid + 1]
        return findMaxValue(mid + 1, right)
    else
        return findMaxValue(left, mid)

findMaxValue(arr, 0, arr.length - 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:

We can convert this problem into a new problem of finding the smallest m elements in a given list. This can be achieve by constructing a new list $|h_1 - h|, |h_2 - h|, \cdots, |h_n - h|$ and then finding the smallest m elements in the new list. We can solve the new problem by a variation of quick-select algorithm as follows.

```
function mQuickSelect(arr, m)
    pivot := arr[rand(0, arr.length)]; # We choose a random pivot.
    left := [], mid := [], right := []
    i := 0
    while i < arr.length: # Partion the arr into left, mid and
        right subarray by the pivot
        if arr[i] < pivot:
            left.append(arr[i])
        else if arr[i] == pivot:
            mid.append(arr[i])
        else
            right.append(arr[i])
    if m <= left.length: # In this case, the smallest elements must
        be in the left subarray.
        return mQuickSelect(left, m)
    else if m <= left.length + mid.length: # In this case, the
        smallest elements must be in the left and mid subarray.
        return pivot, left.length, k - left.length
    else
        found := left.length + mid.length # In this case, we have
            found left.length + mid.length smallest elements, then
            find the rest elements in the right subarray.
        r0, r1, r2 := mQuickSelect(arr, k - found)
        return r0, r1 + found, r2
```

```
function find(arr, h, m)
    newArr := map(x => abs(x - h), arr) # Generate the new list.
    maxH, left, mid := mQuickSelect(newArr, m) # maxH is maximum
        value of smallest m elements in newArr, left is the number
        of those elements in newArr smaller than maxH, and mid is
        the number of those elements in newArr equal to maxH.
    i := 0 , res := [] # Initialize with empty array
    while left > 0 or mid > 0:
        if newArr[i] < maxH and left > 0:
            res.append(arr[i])
            left := left - 1
        if newArr[i] == maxH and mid > 0:
            res.append(arr[i])
            mid := mid - 1
        i := i + 1
    return res
```

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:

$3 < 2 < 1 < 4 < 6 < 7 < 5$

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 number of times the inner loop is performed is the sum of squares from 1 to n. $\sum_{i=1}^{n} i^2 = n * (n+1) * (2*n+1)/6$ Thus, the running time 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 9 (the sum of the lengths of the two arrays). So, the total cost of merging is 15.

Solution:

This problem can be solved with a priority queue (or heap). A priority queue is a special type of queue in which each time the element to be dequeued is the one with the highest (or lowest) priority. In this problem, we need a minimum priority queue, where each time the array to be dequeued has the smallest length.

The steps of the algorithm are as follows:

1. Put all arrays into the priority queue.
2. Each time, remove two shortest arrays from the priority queue, merge them into a new array, and then put the new array back into the priority queue. Add the sum of the lengths of these two arrays to the total cost.
3. Repeat the second step until there is only one array left in the priority queue.

This greedy strategy ensures that each time the two shortest arrays are merged, thus minimizing the total cost of merging.

Proof of Correctness:

We will prove the algorithm's correctness via the "Exchange argument" strategy as detailed in the lecture slides. Assume that we have an optimal solution OPT different from our greedy solution $GREEDY$. We will gradually transform OPT into $GREEDY$ without increasing the cost, thus proving $GREEDY$ is indeed an optimal solution.

Let's start by considering the first operation where OPT and $GREEDY$ differ. Without loss of generality, suppose OPT merges arrays A and B with lengths $|A|$ and $|B|$ where $|A| \geq |B|$, while $GREEDY$ merges C and D with lengths $|C|$ and $|D|$ where $|C| \leq |D|$. Note that $|C|$ and $|D|$ are the smallest lengths among all arrays at this step, so $|C| \leq |A|$ and $|C| \leq |B|$.

The cost of the merge operation in OPT is $|A| + |B|$, while in $GREEDY$ it is $|C| + |D|$. Given $|C| \leq |A|$ and $|C| \leq |B|$, we can safely say that the cost of the merge operation in $GREEDY$ is less than or equal to the cost in OPT. Therefore, we can replace the operation in OPT with the operation in $GREEDY$ without increasing the cost.

By repeating this process, we can transform every operation in OPT into the corresponding operation in $GREEDY$ without increasing the cost. Therefore, $GREEDY$ is an 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.

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:

This problem can be solved using a greedy approach. The greedy strategy here is to make as many transactions as possible and try to make each transaction as profitable as possible. This is equivalent to finding all ascending sequences in the array and summing their differences.

We iterate through the array, and whenever we find a pair of elements where the second element is larger than the first, we consider the difference as a profit and add it to the total profit. Thus, we are effectively summing all the profits from the ascending sequences in the array.

Proof of Correctness:

We will prove this by contradiction. Assume there is an optimal solution OPT different from our greedy solution $GREEDY$. The OPT solution might choose not to sell on a day when the price increases and hold onto the candy instead, aiming to sell it on a later day with a higher price.

Let's consider the first operation where OPT and $GREEDY$ differ. Suppose OPT chooses not to sell on a day i when the price of candy increases from day $i-1$, i.e., price$[i] \geq$ price$[i-1]$, aiming to sell it on a later day j where $j > i$ and price$[j] \geq$ price$[i]$.

However, we can always break this operation into two operations without changing the profit: sell on day i and buy back on day i to sell on day j. The profit of these two operations is (price$[i]$-price$[i-1]$) + (price$[j]$-price$[i]$), which equals to (price$[j]$-price$[i-1]$), the same as the profit of the original operation in OPT.

By doing this, we have transformed the operation in OPT into two operations that match $GREEDY$ without decreasing the profit. By repeating this process, we can transform every operation in OPT into the corresponding operation in $GREEDY$ without changing the total profit. Therefore, $GREEDY$ is an optimal solution.