Weekly Assignment 8

Munia Humaira
Student ID: 21116435
Algorithm Design and Analysis

November 7, 2024

Solution 1:

The algorithm will still work in linear time if the elements are divided into groups of 6.

With this grouping, the median of medians is greater than at least 3 elements from half of the $[n/6]$ groups, which implies that it is larger than approximately $3n/12$ elements overall. Similarly, the median of medians is less than approximately $3n/12$ elements.

Therefore, we are never calling the recursive function on more than $9n/12$ elements, ensuring that the time complexity remains linear. The recurrence relation for the algorithm would thus be:

$T(n) \leq T(n/6) + T(9n/12) + O(n)$

By substitution, we can show that this relation remains linear. Suppose we guess $T(n) < cn$ for $n < k$. Then, for $m \geq k$,

$T(m) \leq T(m/6) + T(9m/12) + \mathcal{O}(m) \leq cm(1/6 + 9/12) + \mathcal{O}(m)$

This means that as long as the constant hidden in the $\mathcal{O}(m)$ term is less than $c/6$, we achieve the desired linear result.

Now, suppose we use groups of size 4. In this case, the recurrence relation becomes less favorable. Specifically, we have fewer elements reliably positioned on each side of the median of medians. For similar reasons, we would find that:

$T(n) = T([n/4]) + T(3n/4) + \mathcal{O}(n) \geq T(n/4) + T(3n/4) + \mathcal{O}(n)$

As a result, this recurrence grows faster than linear. In fact, we would find that it leads to a growth rate of $cnlogn$ rather than $\mathcal{O}(n)$. Therefore, using groups of size 4 results in a complexity that exceeds linear time, while using groups of size 6 allows the algorithm to retain linear time complexity.

Solution 2:

(a) The goal is to prove that choosing the highest possible denomination at each step minimizes the total number of coins required to make an amount a.

Greedy Strategy: Start with the largest coin denomination that does not exceed a, then proceed to the next largest denomination, and so on until the amount is reached.

Base Case: For small values like $a = 1, 2, 3, 4$, using only the $1-$ cent coin is optimal. Inductive Step: Suppose the greedy algorithm yields the minimum number of coins for any amount less than a. For an amount a, the largest coin denomination that can be used is $c_k \leq a$, where $c_k \in 1, 5, 10, 25$. Let $a = d \cdot c_k + r$, where d is the maximum number of c_k coins that fit in a (i.e., $d = a//c_k$), and r is the remainder.

The greedy choice provides d coins of c_k and leaves us with a remainder r, which is less than c_k. By induction, the remainder r can be optimally filled by the smaller denominations using the greedy strategy.

Since this strategy always picks the largest possible denomination first, it minimizes the total number of coins used. Hence, the greedy approach for the given denominations always yields the optimal solution.

(b) The oracle given inputs $<a, C>$, outputs the minimum total number of coins m required for amount a with coin denominations C.

Algorithm Outline:

- Initialize Variables:
  - Set $n = [0] \times k$ (an array of zeros for each coin count n_i).
  - Set remaining amount $R = a$.
- The pseudocode for Binary Search with Oracle:

```
function MinCoinTuple(a, C):
Input: a (amount), C = [c0, c1, ..., ck-1] (sorted coin denominations)
Output: (n0, n1, ..., nk-1) where sum(ni * ci) = a and total coins = minimum

Initialize n = [0] * len(C)        # List to store count of each coin
R = a                              # Remaining amount to be changed
m = Oracle(R, C)                   # Get minimum coins needed for R from oracle

for i from len(C) - 1 down to 0:
    low = 0
    high = R // C[i]
    while low <= high:
        mid = (low + high) // 2
        if Oracle(R - mid * C[i], C) == m - mid:
            n[i] = mid
            R -= mid * C[i]
            m -= mid
            break
        elif Oracle(R - mid * C[i], C) < m - mid:
            high = mid - 1
        else:
            low = mid + 1

return n
```

Explanation:

1. The algorithm iteratively calls the oracle with adjusted amounts to find the exact number of each coin denomination.

2. The binary search ensures efficient calculation, as it narrows down the exact number of each denomination without checking every possible value.

3. The oracle confirms if a particular choice of coins achieves the minimum count, which allows us to adjust m and reduce R accordingly.

Solution 3:

Let $G = g_1, g_2, ..., g_k$ be the set of intervals chosen by our greedy strategy, where each g_i is selected by picking the interval with the latest start time that does not conflict with any intervals already in G. Let $T = t_1, t_2, ..., t_m$ represent an optimal solution set of non-overlapping intervals, with $m \geq k$.

To prove that our greedy choice produces an optimal solution, we will use two claims.

Claim 1: For every $i = 1, ..., k, s(g_i \geq s(t_i)$

- Base Case: For $i = 1$, the greedy choice selects the interval with the latest start time that does not overlap with any other interval. Since T is also a valid set of non-overlapping intervals, we have $s(g_1 \geq s(t_1)$.
- Inductive Step: Assume $s(g_j \geq s(t_j)$ holds for $j = 1, ..., p - 1$. For $i = p$, since $s(g_{p-1}) \geq s(t_{p-1})$, and t_p does not conflict with t_{p-1}, t_p is available to be chosen after g_{p-1}. Therefore, the algorithm's choice of g_p ensures $s(g_p \geq s(t_p)$, as it picks the latest possible start time among all non-overlapping intervals.

Thus, for all $i, s(g_i \geq s(t_i)$.

Claim 2: $k = m$ Assume, for contradiction, that $m > k$. Then there would be a meeting t_{k+1} in T that could be added to G. However, by Claim 1, since $s(g_k \geq s(t_k)$, there are no other intervals available after g_k that don't overlap, contradicting our assumption. Thus, $k = m$, and the greedy choice gives an optimal solution.

Solution 4:

A binary heap is a complete binary tree, meaning all levels are fully filled except possibly the last level, which is filled from left to right.

For a complete binary tree, each level i has 2^i nodes starting with $2^0 = 1$ node at the root. The total number of nodes up to height h is given by the sum:

$$n = 1 + 2 + 4 + ... + 2^h = 2^{(h+1)} - 1$$

Rearranging the above formula,

$$2^{(h+1)} - 1 \leq n$$

$$2^{(h+1)} \leq n + 1$$

Taking the base-2 logarithm on each side,

$$h + 1 \leq \log_2(n + 1)$$

$$h \leq \log_2(n + 1) - 1$$

Therefore, a binary heap with n nodes has a height of $h = [\log_2 n]$