# Solution to Sample Test

Basic Algorithms CSCI-UA.0310

Spring 2022

## Problem 1

(a) The statement is not true.
Counterexample:
Let $f = n$ and $g = n^4$ therefore $f = O(g)$ holds.
Let $h = n^2$ therefore $h = O(g)$ holds.
However for $f = \Omega(h)$ to be true, we need to prove $f \geq c \cdot h$ but $n \leq n^2$ and hence the statement doesn't hold true.

(b) We are given $f(n) = \log(n!)$ which can be written as

$$\log(n!) = \log(1) + \log(2) + ... + \log(n-1) + \log(n)$$

We can get the upper bound as

$$\log(1) + \log(2) + ... + \log(n) \leq \log(n) + \log(n) + ... + \log(n) = n\log(n)$$

Similarly, we can obtain the lower bound as

$$\begin{aligned} \log(1) + ... + \log(n/2) + ... + \log(n) &\geq \log(n/2) + ... + \log(n) \\ &= \log(n/2) + \log(n/2+1) + ... + \log(n-1) + \log(n) \\ &\geq \log(n/2) + ... + \log(n/2) \\ &= (n/2)\log(n/2) \end{aligned}$$

Therefore, we can say $f(n) = \Theta(n \cdot \log n)$

We have $g(n) = \log(n^n) = n\log n$

Hence $f = \Theta(g)$

(c) The recursion tree for the relation can be drawn as follows:

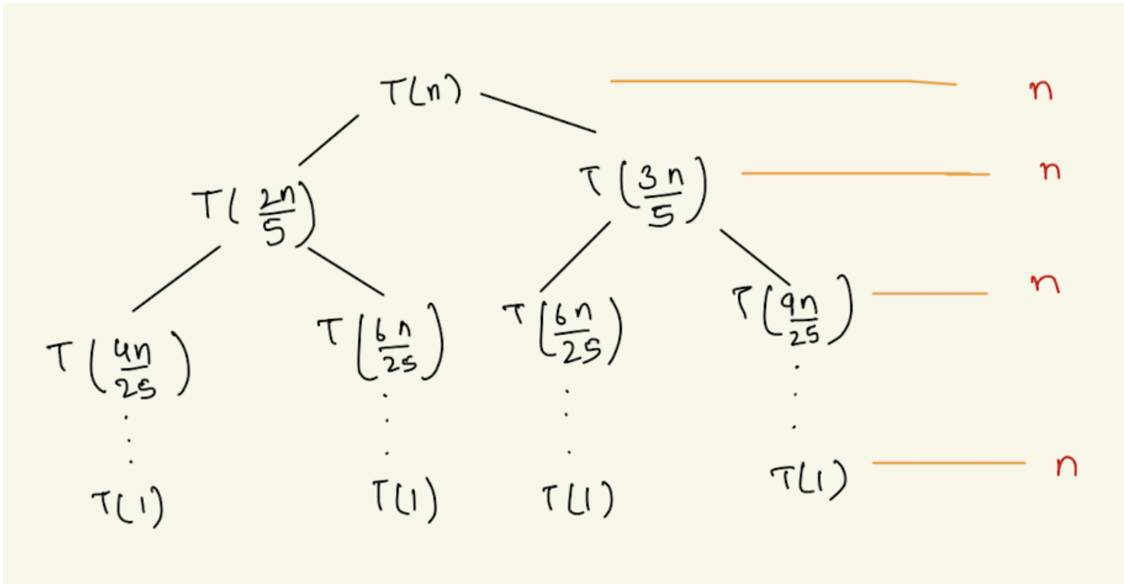


Figure 1: Recursion Tree

The work done at each level is $n$. Every leaf in the recursion tree has depth between $\log_{\frac{5}{3}} n$ and $\log_{\frac{5}{2}} n$.
To derive an upper bound, we overestimate T(n) by ignoring the base cases and extending the tree downward to the level of the deepest leaf, and for the lower bound, likewise to the shallowest leaf.
So we have bounds $n \cdot \log_{\frac{5}{2}} n \leq T(n) \leq n \cdot \log_{\frac{5}{3}} n$.
These bounds differ by a constant factor, so we can conclude $T(n) = \Theta(n \cdot \log n)$

# Problem 2

(a) Given below is the algorithm that outputs the elements of A in such a way that the positive elements follow negative elements while maintaining the relative ordering of the elements in A.

tmp = A.copy()
SOLVE(A, tmp, 1, n)

##Definition of Solve
SOLVE($A, tmp, left, right$)
    if high <= low:    return

```
        mid = (low + high)/2
        Solve(A, tmp, low, mid)
        Solve(A, tmp, mid + 1, high)
        Merge(A, tmp, low, mid, high)


##Definition of Merge
Merge(A, tmp, left, mid, right)
        k = low

        ##copy negative elements from the left sublist
        for i in range(low, mid):
                if A[i] < 0:
                        tmp[k] = A[i]
                        k = k + 1

        ##copy negative elements from the right sublist
        for j in range(mid + 1, high):
                if A[j] < 0:
                        tmp[k] = A[j]
                        k = k + 1

        ##copy positive elements from the left sublist
        for i in range(low, mid):
                if A[i] >= 0:
                        tmp[k] = A[i]
                        k = k + 1

        ##copy positive elements from the right sublist
        for j in range(mid + 1, high):
                if A[j] >= 0:
                        tmp[k] = A[j]
                        k = k + 1

        ## copy back to the original list to reflect the new order
        for i in range(low, high):
                A[i] = tmp[i]
```

(b) The recursive relation for the time complexity of the above algorithm is

$$T(n) = 2 \cdot T(\tfrac{n}{2}) + cn$$

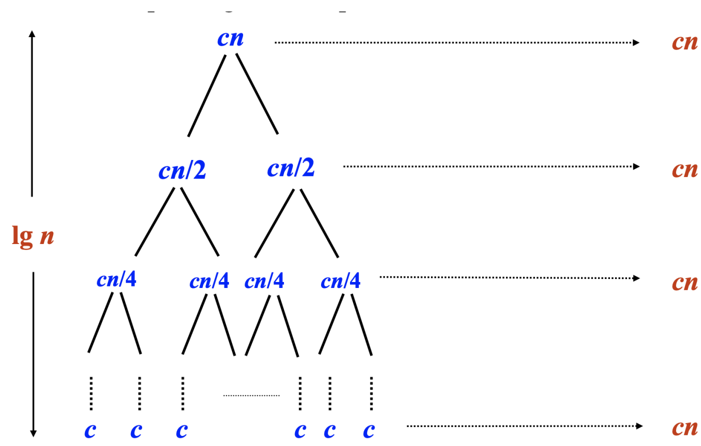


Figure 2: Recursion Tree

Each level has total cost $cn$ and there are $\log n + 1$ levels. Therefore, the total cost is $cn \cdot (\log n + 1) = cn \log n + cn = \Theta(n \log n)$

(c) Yes, the task can be done in $\Theta(n)$ time. We use extra space and save all the negative numbers in first iteration. We iterate over all the elements again and save the positive numbers in this iteration.

```
SOLVE(A)
    ## Initialize the temp array as an empty array
    temp[1...n]=[]
    ## Traverse A and store negative elements in temp array
    j = 1; # index of temp
    for (int i = 1; i ≤ n; i++)
        if (A[i] < 0)
            temp[j++] = A[i]

    ## Store positive elements in temp array
    for (int i = 1; i ≤ n; i++)
        if (A[i] >= 0)
            temp[j++] = A[i]

    ##Copy contents of temp[] to arr[]
    for (int i = 1; i ≤ n; i++)
        A[i] = temp[i];
```

## Problem 3

(a) Given the chocolate bar bar$(i,j)$, we have $i$ options to cut it horizontally, i.e., we can make the horizontal cut at $k = 1,\dots,i$, and we have $j$ options to cut it vertically, i.e., we can make the vertical cut at $l = 1,\dots,j$.
Note that in the rod cutting problem, we could only make vertical cuts.
Thus, among the aforementioned $i+j$ possibilities for the cut, we choose the one maximizing our selling price (there are actually $i+j-1$ possibilities!).
If, for example, we make a horizontal cut at some $1 \le k \le i$, then we get the two pieces bar$(k,j)$ and bar$(i-k,j)$.
The recursive relation to maximize the total selling price is as follows (Justify why the following recursion holds):

$$S(i,j) = \begin{cases} 0 & \text{if i ==0 or j==0} \\ \max\left\{ \ \max_{k=1,\dots,i}(S(k,j)+S(i-k,j)), \quad \max_{l=1,\dots,j}(S(i,l)+S(i,j-l)) \ \right\} & \text{o.w.} \end{cases}$$

(b) Following is the bottom-up dynamic programming algorithm to compute $S(m,n)$:

```
MAXIMIZESELLINGPRICE(P,n,m)
    ## Initialize the memory array with all zeros (includes the base cases)
    memo[0 ... m][0 ... n]=[0]
    ## Check maximum at each possible cut
    for i=1 to m
        for j=1 to n
            for k=1 to i
                memo[i][j] = max( memo[i][j], memo[k][j] + memo[i-k][j] )
            for l=1 to j
                memo[i][j] = max( memo[i][j], memo[i][l] + memo[i][j-l] )

    return memo[m][n]
```

(c) The time complexity of the above algorithm is much slower than the time complexity for LCS. The reason is that the time needed to solve each subproblem $S(i,j)$ for the above algorithm is $\Theta(i+j)$, while for LCS it is only $O(1)$.
**Exercise:** Can you find the time complexity of the above algorithm?