CSE 6140 / CX 4140 Test1
due Sep 18, 2020 at 11:59pm on Canvas

Shasha Liao

1 Problem1

1. Dijkstra's algorithm is a type of Greedy algorithm and it finds the shortest path from a source node to all the other nodes in the graph.
2. $f \in \Omega(g)$, $f \in \Omega(g)$, $f \in O(g)$, $f \in O(g)$
3. $a = 7, b = 2, d = 2$ and $a > b^d$, so we have $T(n) = \Theta(n^{\log_2 7})$
4. bonus

2 Problem2

1. True. $f(n) \in O(g(n))$ gives that there exists $c_1 > 0$ and $n_1 \geq 0$ such that $f(n) \leq c_1 g(n)$ for all $n > n_1$. $g(n) \in O(h(n))$ gives that there exists $c_2 > 0$ and $n_2 \geq 0$ such that $g(n) \leq c_2 h(n)$ for all $n > n_2$. Therefore, for $n > \max(n_1, n_2)$, we have $f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$. So, $f(n) \in O(h(n))$ by definition.
2. False. Let $f(n) = n$ and $g(n) = \frac{n}{2}$. Then obviously $f(n) = O(g(n))$. But

$$\lim_{n\to\infty} \frac{2^{f(n)}}{2^{g(n)}} = \lim_{n\to\infty} 2^{n/2} = \infty.$$

   So there is no way for $2^{f(n)} \in O(2^{g(n)})$.

3 Problem3

We can directly prove that the greedy algorithm gives us the maximum rewards. For any schedule, the jth job will have a completion time $c_j = l_1 + l_2 + ... + l_j$. The reward we get from it will be $r_j = f_j - c_j$. The total rewards will be $R = r_1 + r_2 + ... + r_n$ where we think of negative rewards as penalties. Therefore

$$R = (f_1 - l_1) + (f_2 - l_1 - l_2) + ... + (f_n - l_1 - l_2 - ... - l_n) = \sum_{i=0}^{n} f_i - (nf_1 + (n-1)f_2 + ... + f_n).$$

Since $\sum_{i=0}^{n} f_i$ is a fixed number, to maximum R, we only need to minimize $nl_1 + (n-1)l_2 + ... + l_n$, which will be achieved by scheduling jobs with ascending lengths.

Personally I think this is enough to prove the optimality. But to be safe, I am also going to provide a proof using exchange argument.

Denote greedy solution as G, and an optimal solution O.

- Compare Solutions. Greedy solution has no inversion, namely, job i starts before job j, i.e. $i < j$, if and only if $l_i \leq l_j$. If $G \neq O$, then O must have at least one inversion. Next, we will prove that solution O can be gradually converted into G without hurting the quality of O.

- Exchange Pieces. First, the argument in the proof of Observation 4 in Slides-08-25 tells us that O has at least one adjacent inversion. That is, there exists $i < j$ such that $l_i \geq l_j$. Now we claim that exchanging these two adjacent, inverted jobs i and j reduces the number of inversions by 1 and dose not decrease the rewards we can obtain. Let R be the total rewards before the swap, and let R' be it afterwards.
  Firstly we have $R_k = R'_k$ for all $k \neq i, j$.
  Secondly, $R'_i = f_i - (l_1 + l_2 + ... + l_{i-1} + l_i + l_j)$, $R'_j = f_j - (l_1 + l_2 + ... + l_{i-1} + l_j)$,
  $R_i = f_i - (l_1 + l_2 + ... + l_{i-1} + l_i)$ and $R_j = f_j - (l_1 + l_2 + ... + l_{i-1} + l_i + l_j)$.
  So we have
  $$(R'_i + R'_j) - (R_i + R_j) = l_i - l_j \geq 0.$$
  So we have $R' \geq R$ and thus swapping two adjacent, inverted jobs i and j reduces the number of inversions by 1 and dose not decrease the total rewards.

- Iteration. Since there are at most $n(n-1)/2$ inversions in O, we can swap the adjacent inversions finitely many times and convert O to G without hurting the quality of O.

Therefore, our greedy algorithm is optimal.

4 Problem4

1. Use two for loops to compare each pair of a_i and a_j with $i > j$ and add one to the count if $a_i < a_j$.

2. We can modify the Merge-Sort algorithm a little bit to achieve the goal. During the merge process, we use a smart way to count the number of inversions in two sorted subsequences s_1 and s_2, where the elements of s_1 comes from the first half of the elements in the original sequence and s_2 contains the second part of the original sequence. we only need to count for each element in s_1, how many elements in s_2 are larger than it. And then summing up all these counts gives us the number of inversions of these two sorted subsequence s_1 and s_2. And we can obtain the number of inversions of the original sequence by summing up all the inversions we obtained during each merge step.

   Pseudocode:

```
def merge(s1, count1, s2, count2):
    n = len(s1)
    count = 0
    merged_list = []
    i = 1; j = 1
    while s1 and s2:
        if s1[i] > s2[j]:
            merged_list.append(s2[j])
            j += 1
            count += (n - i) # number of elements in s1 that is greater than s2[j]
        else:
            merged_list.append(s1[i])
            i += 1
    for number in s1[i:]:
        merged_list.append(number)
    for number in s2[j:]:
```

```
        merged_list.append(number)
    return merged_list, count + count1 + count2

def mergeSort(s):

    if len(s) == 1:
        return s, 0
    #Divide s into two halfs, s1 and s2
    s1, count1 = mergeSort(s1)
    s2, count2 = mergeSort(s2)

    L, count = merge(s1,count1,s2,count2)
    return L, count
```

The run time of my algorithm is $O(n \log n)$ because the recurrence is $T(n) = 2T(n/2) + f(n)$ since the mergeSort algorithm divide the sequence evenly into two parts and calls itself two times. And the merge step is of time complexity $O(n)$. So by master theorem, we have $T(n) = O(n \log n$.