COMPSCI 311: Introduction to Algorithms **Spring 2020**

Homework 2

Released 2/10/2020 Due 2/24/2020 11:59pm in Gradescope

1. **Bipartite Matchings (10 points)**

   It is not correct. For example, for a graph with 3 points in each group, we have links 0-0, 0-1, 0-2, 1-0, 1-2, 2-0

   In student's method, it will have the preference matrixes for A and B

0: 0 1 2	0: 0 1 2
1: 0 2 1	1: 0 1 2
2: 0 1 2	2: 0 1 2

   Apply Gale-Shapley matching on it, we will have the result 0-0, 1-2, 2-1 which is stable. However, in the graph there is no link from 2-1, but we can still find a perfect match of 0-1, 1-2, 2-0.

2. **Greedy Partition (10 points)**

   It does not always produce the most balanced split. Consider the array [8, 7, 6, 5, 4], the process would be $A:[8], B[], S_A = 8, S_B = 0 \Rightarrow A[8], B[7], S_A = 8, S_B = 7 \Rightarrow A[8], B[7,6], S_A = 8, S_B = 13 \Rightarrow A[8,5], B[7,6], S_A = 13, S_B = 13 \Rightarrow A[8,5,4], B = [7,6], S_A = 17, S_B = 13$, whish has $|S_A - S_B| = 4$, however, we can have set $A' = [8,7], B' = [6,5,4], S_{A'} = 15, S_{B'} = 15$ where $|S_{A'} - S_{B'}| = 0$, which is contaidictory to the condition that no $|S_{A'} - S_{B'}| < |S_A - S_B|$.

3. **Checking Tasks (20 points)**

   Algorithm: T as input sorted task list where $T = [[s_1, e_1], ..., [s_n, e_n]], e_1 \leq e_2 \leq ... \leq e_n$

```
CheckingTasks(T)
  sj, ej = T[1]
  result = [ej]
  for i = 2 to T.length:
    si, ei = T[i]
    if si > ej:
      sj, ej = si, ei
      result.add(ei)
    endif
  endfor
  return result
```

   In words, that is for sorted T, for tasks that start after any tasks that ends before it, its ending should be a check time point. It should return the end time of the first k number of tasks in the sorted list, where k is the number of checkings.

   Intuition: The intuition of this algorithm is that because every task needed to be checked, then we always want a check before each task ends, so we'd like to check from the tasks that ends early. Before they end early than any of its overlap tasks, its ending should overlap maximum number of tasks, gives least checkings.

   Proof: The T is sorted by end time increasingly, assume the solution of this algorithm is A, and there is an arbitrary optimal solution O, then there must be at least one check point not in A, and at least one check point in A not in O. Consider the time points in O, A are sorted increasingly. Suppose that all other parts in A are same of O except the first point. The first point of O should cover the first task T_1 in the list, because this task ends first. We can tune that check point to e_1, because no other tasks end before e_1, the number of tasks checked by this point can only increase, create an O' which is no worse than O, where O' is closer to A than O, as the first check point is closer to e_1. Finally, we can transform O into A, means A is optimal.

Complexity: The complexity of this algorithm is $O(n \log n)$, because there is a sorting operation at the beginning. The rest of the algorithm is simply a loop from 2 to n with some comparation, list push and list get, which is $O(n)$

4. **Surfboard Rental (20 points)**

Algorithm: To make it simple, assume the h and l are already sorted, that is, $\forall 0 < i < n : h_i \leq h_{i+1} \wedge l_i \leq l_{i+1}$. Then the pseudocode is

```
H as the heights, L as the lengths
SurfboardRental(H, L)
  a = {}
  for i = 1 to n
    a(i) = i
  endfor
  return a
```

If the arrays are not sorted, sort them and keep their original index then retrive then on the $a(i) = i$ line to make the two i be the original indexes for l and h.

Intuition: The intuition of this algorithm is that to make the shortest tourist has the shortest surfboard, and make the second shortest tourist has the second shortest and so on.

Proof: To make it simple, take the assumption from the pseudocode. When the arrays are not sorted, just mapping the indexes.

Set a as the solution. Suppose there is an optimal solution $o \neq a$. Suppose that i is the smallest index where $o(i)! = i$, which is different compare to a. Let $o(i) = j$, consider o' by switch h_j and h_i to make $o'(i) = i$. As assumption, we have $i < j$, so $h_i \leq h_j, l_i \leq l_j$

The cost difference between o and o' is $D = \frac{1}{n}(|h_i - l_j| + |h_j - l_i| - |h_i - l_i| - |h_j - l_j|), cost(o') = cost(o) - D$, which has 6 cases:

1. $(h_i \leq h_j \leq l_i \leq l_j) \Rightarrow (D \geq 0)$
2. $(h_i \leq l_i \leq h_j \leq l_j) \Rightarrow (D \geq 0)$
3. $(h_i \leq l_i \leq l_j \leq h_j) \Rightarrow (D \geq 0)$
4. $(l_i \leq l_j \leq h_i \leq h_j) \Rightarrow (D \geq 0)$
5. $(l_i \leq h_i \leq l_j \leq h_j) \Rightarrow (D \geq 0)$
6. $(l_i \leq h_i \leq h_j \leq l_j) \Rightarrow (D \geq 0)$

All these proved that $D \geq 0$, means the cost of o' is less or equal to the cost of o, so o' is no worse than o. This can be done repeatedly on the remains of the solution final got a, so a is no worse than o which is an optimal solution, so a is optimal.

The running time should be $O(n)$ if the h and l are already sorted, and $O(n \log n)$ if they are not sorted.

The $O(n)$ is easy to justify because there is only one loop in the pseudocode has n steps, and only assign operation ($O(1)$) is done in each step. The $O(n \log n)$ is because the sort operation on two arrays. Keep index should be a $O(n \log n)$ operation, as we should keep track on each operation in the sort and update the lookup list. In the loop with n steps, there will be two mapping operations and an assign operation, all of them are $O(1)$, so the whole loop is $O(n)$, then the whole complexity is $O(n \log n)$

5. **Board Coloring (20 points)**

Algorithm: Consider the sort rule for list R, C, the elements inside them are ordered decreasingly by the element minus the sum of the black squares in the row/column, and the elements keep the original index.

```
CheckingTasks(R, C)
  R = sort(R)
  C = sort(C)
  let result be an nxn matrix all white
  foreach R_i in R:
    rn = R[i]
    foreach C_j in C:
      let csum be the number of black on column j
      if csum < C[j]:
        result[i, j] = black
        rn -= 1
        if rn == 0:
          break
        endif
      endif
    endfor
    C = sort(C)
  endfor
  let R' be the sum of number of black squares in the result group by rows
  let C' be the sum of number of black squares in the result group by columns
  if any(R'_i != R_i) for i = 1 to n
    or any(C'_i != C_i) for i = 1 to n:
    Exception('No such board')
  endif
  return result
```

The idea of this algorithm is sort the R, C by the remaining black squares to fill, and for each row, fulfill the required black squares, in the order on the remaining black squares to fill on the column in decreasing order, means first fill the column currently require most black squares.

Intuition: It easily come to the intuition that the columns/rows with most black squares to fill should be filled first, because they have less freedom. For example, if $R[i] = n$, then the only choice is to fill all the squares in that row.

Proof: Set the algorithm solution to be A, and consider an arbitrary solution O. We know that there is at least one pair of elements are different in O and O'. Set these elements to be $(O_a, O_b), (O'_a, O'_b)$. When we have $R[O_a] = R[O'_b], R[O_b] = R[O'_a]$ and $C[O_a] = C[O'_a], C[O_b] = C[O'_b]$, swap O_a, O'_a and O_b, O'_b, O' will be no worse than O. It also works when swaping the columns and rows. Because we first fill the rows/cols with most remaining black squares, it can finally swap to A, means A is no worse than O, so optimal.

Complexity: The complexity is $O(n^2 \log n)$, because there is are two nested loops with $O(n)$, in total $O(n^2)$. In the outer loop, there is a sorting operation, gives $O(n \log n)$, in total $O(n^2 \log n)$ with the loop. The sort at the beginning are $O(n \log n)$, the comparation in the ending are $O(n)$, so the complexity is $O(n^2 \log n)$

6. **Transforming Trees (20 points)**

Algorithm: Consider $n \times n$ adjacency matrix for the edge information

```
TransformingTrees(T1, T2):
  result = 0
  for i = 1 to n:
    for j = i + 1 to n:
      if T1[i, j] == 0 and T2[i, j] == 1:
        result += 1
      endif
    endfor
  endfor
  return result
```

Basically, this algorithm counts the number of edges that in T_1 but not in T_2, and this should be the result.

Intuition: We know that a spanning tree is a connected acyclic graph based on G, means that T_1 and T_2 have same vertices, so we can safely add an edge that only in T_2 to T_1. Also, we should remove same amount of edges from T_1, because the number of edges in a spanning tree is constant, and it fulfills the requirement of stepwise.

Proof: For T_1, T_2, they have same vertices and number of edges. Set A be the edges only in T_1, and B be the edges only in T_2. If we add an edge from B to T_1, T_1 would still be connected, but will create a cycle, and in this cycle, there must be an edge in A, because if there are different spanning trees for G, there must be at least 1 cycle or there will be an unique spanning tree instead, and each tree will choose some different edges from this cycle. Remove this edge from T_1, T_1 will still be a spanning tree because it is still an acyclic connected graph which connects all the vertices in G. So, this is a valid step. Repeatedly swap edges, it is obvious that the total number of steps should be the number of edges that in T_2 but not in T_1.

Complexity: The complexity would be $O(|V|^2)$, becuase the adjacency matrix is a matrix with $|V| \times |V|$ dimension, and there are two nested loops in the algorithm that are $O(|V|)$, so the total complexity should be $O(|V| \times |V|) = O(|V|^2)$

7. **(0 points).** How long did it take you to complete this assignment?