CS 341, Spring 2020
Semih Salihoglu

Assignment 4 (due Sunday, July 12th, midnight EST)

Instructions:

- Hand in your assignment using Crowdmark. Detailed instructions are on the course website.
- Give complete legible solutions to all questions.
- Your answers will be marked for clarity as well as correctness.
- For any algorithm you present, you should justify its correctness (if it is not obvious) and analyze the complexity.

1. [*12 marks*] *Asymptotic Notation* For parts (b) and (c) below, for each pair of functions $f(n)$ and $g(n)$, fill in the correct asymptotic notation among Θ, o, and ω in the statement $f(n) \in \sqcup(g(n))$. You must include brief justifications of all your answers.

   (a) [*6 marks*] Arrange the following functions in **increasing** order of growth rate. All logarithms are in **base 10**. You do not need to formally justify your answer.

   $$1.01^n, \quad n\log_{10}(n), \quad n^{\log_{10}(n)}, \quad \log_{10}(\pi^n), \quad 3^{\log_{10}(n)}, \quad n^{2017}$$

   The order is $\log_{10}(\pi^n), \quad 3^{\log_{10}(n)}, \quad n\log_{10}(n), \quad n^{2017}, \quad n^{\log_{10}(n)}, \quad 1.01^n$

   (b) [*2 marks*] $f(n) = \sum_{i=0}^{\lfloor \log_3 n \rfloor} 3^i$ vs. $g(n) = 0.5n$

   $$\begin{aligned} f(n) &= \sum_{i=0}^{\lfloor \log_3 n \rfloor} 3^i \\ &= \frac{3^{\lfloor \log_3 n \rfloor + 1} - 1}{3-1} + 1 \\ &\leq \frac{3n-1}{2} + 1 \\ &= \Theta(n) \end{aligned}$$

   Since $g(n) = \Theta n$, $f(n) \in \Theta(g(n))$

   (c) [*2 marks*] $f(n) = 4^{n^2}$ vs. $g(n) = 100^n$
   $\lim_{n\to\infty} \frac{4^{n^2}}{100^n} = \lim_{n\to\infty} (\frac{4}{100})^n \cdot 4^{n^2-n}$
   Since, $\lim_{n\to\infty} (\frac{4}{100})^n = \infty$ and $\lim_{n\to\infty} 4^{n^2-n} = \infty$, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. Therefore, $f(n) \in \omega(g(n))$

   (d) [*2 marks*] Answer with a YES or NO with a brief justification: If $f(n) = \Theta(n\log n + n)$ and $g(n) = \Theta(n\log n)$, does it always follow that $f(n) - g(n) = \Theta(n)$?
   No. Let $f(n) = 2n\log n + n$ and $g(n) = n\log n$.
   Clearly, $f(n) = \Theta(n\log n + n)$ and $g(n) = \Theta(n\log n)$
   But $f(n) - g(n) = n\log n + n \neq \Theta(n)$

2. [*12 marks*] *Divide and Conquer*

Suppose two people rank a list of n items (say movies), denoted $M_1, \ldots, M_n$. A *conflict* is a pair of movies $\{M_i, M_j\}$ such that $M_i > M_j$ in one ranking and $M_j > M_i$ in the other ranking. The number of conflicts between two rankings is a measure of how different they are.

For example, consider the following two rankings:

$$M_1 > M_2 > M_3 > M_4 \quad \text{and} \quad M_2 > M_4 > M_1 > M_3.$$

The number of conflicts is three; $\{M_1, M_2\}$, $\{M_1, M_4\}$, and $\{M_3, M_4\}$ are the conflicting pairs.

The purpose of this question is to find an $O(n \log n)$ divide-and-conquer algorithm to compute the number of conflicts between two rankings of n items. For simplicity, you can assume n is a power of two. Also, without loss of generality you can simplify your notation and assume that the *first ranking* is $M_1 > M_2 > \cdots > M_n$. Give a pseudocode description of your algorithm, briefly justify its correctness and analyze the complexity using a recurrence relation.

We use merge sort on the second array and count the number of elements in L less than $R[i]$ when merge. Let n_i be he number of elements in L less than $R[i]$ when merge.

Algorithm 1: findConflict($M[M_1, \ldots, Mn], M'$)

```
1  mergeSort(M):
2      Output: a pair (M, val) where M is sorted list and val = sum_{i=0}^{n-1} n_i.
3      if |M| ≤ 1 then
4          return (M, 0)
5      (L, n_l) = mergeSort(M[0, ..., n/2 − 1])
6      (R, n_r) = mergeSort(M[n/2, ..., n])
7      i, j, k, n = 0
8      while i < n/2 or j < n/2 do
9          suppose L[i] = M_u and R[j] = M_v
10         if j ≥ n/2 or u < v then
11             A[k] = L[i]
12             n = n + j
13             i = i + 1
14         else
15             A[k] = L[j]
16             j = j + 1
17         k = k + 1
18     return (A, n + n_l + n_r)
19 findConflict(M[M_1, ..., Mn], M'):
20     suppose M_1 < M_2 < ··· < M_n
21     return mergeSort(M').first
```

Proof of Correctness: We first proof that n is the number of conflicts between L and R. Assume n_i is the number of elements in L that is less than $R[i]$ at some iteration. Then i has

the smallest index in the orgional array than every elements in L. Since there are n_i elements is less than $R[i]$. The number of conflict is exactly n_i. $n = \sum_{i=0}^{n-1} n_i$ is the number of conflicts between L and R.
Since total number of conflicts is the sum of the number of conflicts in L, the number of conflicts in R and the number of conflicts between L and R. The total number of conflicts is the sum of n_i.

Runtime analysis:

$$T_{ms}(n) = \begin{cases} 2T_{ms}(\frac{n}{2}) + O(n) & x > 1 \\ O(1) & x = 1 \end{cases}$$

Since $a = 2, b = 2, d = 1$, by master theorem, $T_{ms}(n) = \Theta(n \log n)$
Thus, $T(n) = T_{ms}(n) = O(n \log n)$

3. [*12 marks*] *Greedy* Given n intervals $[a_1, b_1], ..., [a_n, b_n]$, decide whether there are n points $p_1, ..., p_n$ such that: (i) each $p_i \in [a_i, b_i]$; and (ii) consecutive points are at least distance 2 apart, i.e., $p_{i+1} - p_i \geq 2$. Design an $O(n)$ time greedy algorithm to solve this problem.

   For example, if the input is $[0.9, 4]$ $[2.2, 3.3]$ $[4.5, 5]$ $[6.5, 7.5]$, then the answer is YES, since, e.g., 1, 3, 5, 7, satisfies the conditions mentioned above (i.e., cut each line and are at least 2 apart). Your algorithm needs to only return YES/NO but can also return the actual set of points if the answer is YES. As usual, show the runtime of your algorithm and prove its correctness.

   [Hint: You may not need an exchange argument or greedy-stays-ahead argument here but any correct argument will get full credits.]

   we try to pick the smallest point in each interval that satisfies the constraint $p_{i+1} - p_i \geq 2$, that is $min(a_i, a_{i-1} + 2)$.

Algorithm 2: cut($[a_1, b_1], \ldots, [a_n, b_n]$)

```
P[1] = a_1
for i = 2, ··· , n − 1 do
    if b_i < a_{i−1} + 2 then
        return NO
    else
        P[i] = max(a_i, a_{i−1} + 2)
return (YES, P)
```

Proof of Correctness: If the algorithm returns YES, clearly, $p_{i+1} - p_i > 2$ and point p_i is in $[a_i, b_i]$. If the algorithm returns NO, let $[a_i, b_i]$ be the interval when the algorithm terminates. So $b_i < a_{i-1} + 2$. Assume for contradiction, that there is a solution S. Then $p_{i-1} \in S$ and $p_{i-1} < P[i-1]$. However, $P[i-1] = max(a_{i-1}, a_{i-2} + 2)$. $p_{i-2} < P[i-2]$ which is the most left point that we can pick. We keep decrease the index i to $i = 1$ and get $p_1 < P[1]$. But $P[1] = a_1$, $p_1 \notin [a_1, b_1]$, a contradiction.

Runtime Analysis: $T(n) = nO(1) = O(n)$

4. [*15 marks*] *Dynamic Programming* You are given n positive integers $a_1, ..., a_n$, a number k, and a target amount W, where both k and W are also positive integers. Find a subset $S \subseteq \{a_1, ..., a_n\}$ of size at most k such that sum of the elements in S is as close to W as possible, i.e., minimize the quantity $|\Sigma_{a \in S} a - W|$. $S = \emptyset$ has sum 0 and is W away from the target. Design an $O(nW)$ time dynamic programming algorithm to solve this question. An $O(nkW)$ algorithm will get at most 10 marks (so most but not all the marks).

   Let $A_{(n+1)\times(2W+1)}$, $A[i][j] = (true \backslash false, l)$ where $true \backslash false$ represents if there exists a $S \subseteq \{a_1, ..., a_i\}$ such that $\sum_{a \in S} a = j$ and $|S| = l$.

Algorithm 3: TraceBack($[a_1, \ldots, a_n], A, i, j$)

```
while i > 0 do
    if j − a_i > 0 and A[i − 1][j].first = true and A[i − 1][j − a_i].first = true and
      A[i − 1][j].second > A[i − 1][j − a_i].second then
        S = S ∪ {a_i}
        j = j − a_i
    else if j − a_i > 0 and A[i − 1][j − a_i].first = true then
        S = S ∪ {a_i}
        j = j − a_i
    i = i − 1
return S
```

Algorithm 4: findClosest($[a_1, \ldots, a_n], k, W$)

```
1  A_{(n+1)×(2W+1)} where A[i][j] = (false, 0)   ∀i, j
2  A[i][0] = (true, 0)
3  for i = 1, ..., n do
4      for j = 1, ..., 2W do
5          if j − a_i > 0 and A[i − 1][j].first = true and A[i − 1][j − a_i].first = true then
6              if A[i − 1][j].second <= A[i − 1][j − a_i].second then
7                  A[i][j] = A[i − 1][j]
8              else
9                  A[i][j] = (true, A[i − 1][j − a_i].second + 1)
10         else if j − a_i < 0 or A[i − 1][j].first = true then
11             A[i][j] = A[i − 1][j]
12         else if A[i − 1][j − a_i].first = true then
13             A[i][j].first = (true, A[i − 1][j − a_i].second + 1)
14         else
15             A[i][j].first = (false, 0)
16 j = W
17 B = [W, W − 1, W + 1, W − 2, W + 2, ..., 0, 2W]
18 for j ∈ B do
19     for i = 0, ..., n do
20         if A[i][j].first = true and A[i][j].second <= k then
21             return TraceBack([a_1, ..., a_n], A, i, j)
```

Proof of Correctness: Consider the subproblem with $i-1$ positive integers $\{a_1, \ldots, a_{i-1}\}$ and a target amount j such that there exists a subset S, $\sum_{a \in S} a = j$. We claim that a_i is either in S or not in S. If $a_i \in S$, then there exists a subset $S' \subseteq \{a_1, ..., a_{i-1}\}$ such that $\sum_{a \in S'} a = j - a_i$. If $a_i \notin S$, then either S does not exists, or we can find a $S' \subseteq \{a_1, ..., a_{i-1}\}$ such that $\sum_{a \in S'} a = j$. Therefore, $A[i][j]$ always contains the correct output.
In TraceBack($[a_1, \ldots, a_n], A, i, j$), i decreases 1 in each iteration, therefore it always terminates.
Let $S \subseteq \{a_1, ..., a_n\}$ be the solution that minimize $|\sum_{a \in S} a - W|$. $\sum_{a \in S} \in [0, 2W]$, otherwise, the solution would be $\emptyset$ if $\sum_{a \in S} > 2W$. Therefore, We can find S by checking each column in A in the order of $W, W-1, W+1, W-2, W+2, \ldots, 0, 2W$ and back track the elements.

Runtime Analysis: Since the for loop starting from line 18 in findClosest goes throught at most $O(nW)$ elements, it has runtime $O(nW)$. $T(n, W) = 2W \cdot nO(1) + O(nW) = O(nW)$

5. [*8 marks*] *Graph Algorithm* Given a directed graph $G = (V, E)$ with m edges and n vertices, an edge e is circular if there exists a cycle that contains e. Give an $O(m + n)$-time algorithm to identify all circular edges in G. As usual show the runtime of your algorithm and prove its correctness.

We can run Kosaraju's Alogrithm to indetify the strongly connected components in G. Once the algorithm successfully indetified a strongly connected component, we return all the edges inside that component. We keep running the algorithm until we go through all strongly connected components in G.

Proof of Correctness: Since the SCC graph of G is a DAG, the edges in the SCC graph of G cannot be contained in a cycle. So only the edges inside each SCC can be contained in a cycle. We claim that all edges in all SCCs are contained in some cycle.

Proof. Assume for contradiction, that there exists an edge e that are not contained in any cycle in a SCC.
Let u, v be the vertices that are incident to e. WLOG, let e points to v from u. We cannot find a path from v to u since e are not contained in any cycle. But by the definition of SCC, there exists a path form v to u, a contradiction. □

Runtime Analysis: Since we can report the edges in each SCC when doing DFS/BFS in each SCC, the runtime of collection edges is $O(1)$, so it has the same runtime as Kosaraju's Alogrithm which is $O(m + n)$.