CS 341, Spring 2020
Semih Salihoglu

Assignment 3 (due Thursday, July 2nd, 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. [*25 marks*] *Programming Question:* This is a programming question. Details on how you will submit your code will be posted on Piazza.

   Gerrymandering is the practice of carving up electoral districts in very careful ways so as to lead to outcomes that favor a particular political party. This (terrible) practice is effectively the solution to the following computational problem. Suppose we have a set of n precincts (with n even), each containing m registered voters and two political parties: party A and party B. Suppose further that for each precinct, we have information on how many voters are registered to each of two political parties. You can assume each voter is registered to one of the parties. We say that the set of precincts is susceptible to gerrymandering if it is possible to divide the precincts into 2 districts, each consisting of $n/2$ of the precincts in such a way that the same party holds a majority in both districts.

   Example. Suppose we have $n = 4$ precincts, and the following information on registered voters.

Precinct	1	2	3	4
Number registered for party A	55	43	60	47
Number registered for party B	45	57	40	53

   This set of precincts is susceptible since, if we grouped precincts 1 and 4 into one district, and precincts 2 and 3 into the other, then party A would have a majority in both districts. (Presumably, the "we" who are doing the grouping here are members of party A.) This example is a quick illustration of the basic unfairness in gerrymandering: Although party A holds only a slim majority in the overall population (205 to 195), it ends up with a majority in not one but both districts.

   In this problem, you are a software engineer working for party A (of course, do not work for such parties in real life). Your program will be given an n-size array $A[n]$ and a number m as input, where $A[k]$ stores the number of registered voters for party A in precinct k. So the number of registered voters for party B in precinct k is $m - A[k]$. Your goal is to write an

$O(mn^3)$-time dynamic programming algorithm (so quite slow) that solves the gerrymandering problem. That is, let $P = \{1, ..., n\}$ be the set of all precincts. Your algorithm should return true if there exists a subset $S \subset P$ of size $n/2$ such that in both S and $P \setminus S$, party A holds a majority (i.e., if $\sum_{k \in S} a_k > nm/4$ and $\sum_{k \in P \setminus S} a_k > nm/4$).

We will give you the recurrence you need to get you stared. Aside from programming the algorithm, you need to think about what your base cases should be and how to return the final answer. Let a_k denote the number of voters registered for party A in precinct k. (Then $m - a_k$ is the number of voters registered for party B in precinct k.) Define the following subproblems ($i = 0, \ldots, n$, $j = 0, \ldots, n$, $\ell = 0, \ldots, mn$):

$$C[i, j, \ell] = \begin{cases} 1 & \text{if there exists a subset } S \subseteq \{1, \ldots, i\} \text{ such that } |S| = j \text{ and } \sum_{k \in S} a_k = \ell \\ 0 & \text{else} \end{cases}$$

Forgetting about the base cases, $C[i, j, \ell]$ can be expressed through the following recurrence:

$$C[i, j, \ell] = C[i-1, j, \ell] \vee C[i-1, j-1, \ell - a_i].$$

(For $\ell < a_i$, just take $C[i, j, \ell] = C[i-1, j, \ell]$.)

This is because, there is a subset S of size j from precincts $1, ..., i$ such that party A's total votes in S sum to ℓ either: (i) when precinct $i \notin S$ and there is a subset S' from precincts $1, ..., i-1$ that sum to exactly ℓ; or (ii) when precinct $i \in S$ and and there is a subset S' from precincts $1, ..., i-1$ that sum to exactly $\ell - a_i$;

Using the above recurrence, write a complete and correct dynamic programming algorithm that returns true (if party A can obtain a majority in both districts through gerrymandering) or false (otherwise). In addition, if your algorithm returns true, it should also return such a subset S.

You can either use Python2 or C++. Your program should read from standard input and print to standard output. The input consists of two lines. The first contains the two integers n, m separated by a space. The second contains the n integers entries of matrix $A[n]$ separated by a space. Your program should print either `true` or `false`. In the positive case, it should also print in a new line $n/2$ space-separated integers from the set $\{1, \ldots, n\}$ that represent the subset S.
Input sample:

```
4 100
55 43 60 47
```

Output sample:

```
true
2 3
```

2. [*12 marks*] Consider the following coin changing problem. We have a set of coin denominations $d_1 > d_2 > .... > d_n = 1$, such that d_j is divisible by d_{j+1} for all $1 \leq j \leq n-1$, and a target sum T_0. As in real life, we assume that d_i are positive integers. Prove that the greedy "cashier algorithm" that gives back the maximum number of coins using the highest valued denominator next is optimal. You can think of any algorithm solving this problem as giving back a tuple $[x_1, ..., x_n]$, such that $x_1d_1 + x_2d_2 + ... + x_nd_n = T_0$. The greedy cashier algorithm first gives $x_1 = \lfloor \frac{T_0}{d_1} \rfloor$ many d_1 coins. Then let $T_1 = T_0 - x_1d_1$. Then the algorithm gives $x_2 = \lfloor \frac{T_1}{d_2} \rfloor$ many d_2 coins, so on and so forth.

   For the rest of the question, let $S_g = [x_1, ..., x_n]$ be the greedy solution.

   (a) [*2 marks*] Let $f_i = \frac{d_{i-1}}{d_i}$ for $i \geq 2$. Show that in S_g, $0 \leq x_i < f_i$ for all $i \geq 2$.

   Apperantly, $x_i \leq 0$ for all i. Assume there exists an x_i such that $x_i \geq f_i$. So $x_i \geq \frac{d_{i-1}}{d_i}$, $d_ix_i \geq d_{i-1}$ since d_i is positive. Since $x_i \leq \frac{T_i}{d_{i-1}}$, $d_ix_i \leq T_{i-1}$. Hence $d_{i-1} \leq T_{i-1}$. However,

$$\begin{aligned}
T_{i-1} &= T_{i-2} - x_{i-1}d_{i-1} \\
&= T_{i-2} - \lfloor \frac{T_{i-2}}{d_{i-1}} \rfloor d_{i-1} \\
&< T_{i-2} - (\frac{T_{i-2}}{d_{i-1}} - 1)d_{i-1} \\
&= d_{i-1}
\end{aligned}$$

   a contradiction.

   (b) [*2 marks*] Let $S = [y_1, ..., y_n]$ be an arbitrary solution. Show that if $y_i \geq f_i$ for any $i \geq 2$, then S is not optimal. Note that this proves that if S is an optimal solution, similar to the property of S_g we proved, $0 \leq y_i < f_i$, for $i \geq 2$.

   Since d_{i-1} is dividsible by d_i, let $d_{i-1} = kd_i$ for some integer $k > 1$.

$$\begin{aligned}
y_{i-1}d_{i-1} + y_id_i &= y_{i-1}d_{i-1} + d_{i-1} + y_id_i - d_{i-1} \\
&= (y_{i-1} + 1)d_{i-1} + y_id_i - kd_i \\
&= (y_{i-1} + 1)d_{i-1} + (y_i - k)d_i
\end{aligned}$$

   Since $d_iy_i \geq d_{i-1}$, $(y_i - k)d_i$ is positive. Therefore,

$$\begin{aligned}
T_0 &= y_1d_1 + \cdots + y_{i-1}d_{i-1} + y_id_i + \cdots + y_nd_n \\
&= y_1d_1 + \cdots + (y_{i-1} + 1)d_{i-1} + (y_i - k)d_i + \cdots + y_nd_n
\end{aligned}$$

   But $y_{i-1} + y_i > y_{i-1} + 1 - k$ since $k > 1$. Therefore S is not optimal.

   (c) [*8 marks*] Now suppose that $S = [y_1, ..., y_n]$ is a hypothetical optimal solution, so $0 \leq y_i < f_i$, for $i \geq 2$, but for the purpose of contradiction, $S \neq S_g$. Let i^* be the *highest* index (so the smallest denomination) where S_g and S differ. Derive a contradiction that S is optimal. This completes the proof that S_g is the optimal solution.

Let $K = \{i : x_i \neq y_i\}$. Since d_i is divisible by d_{i+1} and d_{i^*} is smallest in K, $d_k = c_k d_i$ for all $k \in K$.
Also $\sum_{i=1}^{n} y_i d_i = \sum_{i=1}^{n} x_i d_i$, $\sum_{k \in K}(x_k - y_k)d_k = 0$

$$(x_{i^*} - y_{i^*})d_{i^*} + \sum_{k \in K \setminus \{i^*\}} (x_k - y_k)d_k = 0$$

$$(x_{i^*} - y_{i^*})d_{i^*} = -d_{i^*-1} \sum_{k \in K \setminus \{i^*\}} c_k(x_k - y_k)$$

$$|x_{i^*} - y_{i^*}| = f_{i^*} \sum_{k \in K \setminus \{i^*\}} |c_k(x_k - y_k)|$$

$$|x_{i^*} - y_{i^*}| > f_{i^*} \quad \text{since } c_k > 1, |x_k - y_k| > 1$$

However, $y_{i^*} < f_{i^*}$ and $x_{i^*} < f_{i^*}$, $|x_{i^*} - y_{i^*}| < f_{i^*}$, a contradiction.

3. [*13 marks*] Consider the input for the activity selection problem that we covered in Lecture 7. The input consists of a set of line segments $r_1, ..., r_n$, with start times $s(i)$ and finishing times $f(i)$. Now consider the version of the problem where instead of trying to pick the maximum number of non-overlapping line segments, we want to pick the minimum number of line segments S such that each line segment r_i overlaps with at least one of the line segments in S. Develop a greedy algorithm for this problem. Your algorithm can be $O(n^2)$ time or faster, e.g., $O(n \log n)$. You will not lose points if your algorithm is $O(n^2)$ (but you will lose points if it's slower than $O(n^2)$.)

   Let *overlap*(r_i) denot the set of intervals than overlaps with r_i(which includes r_i itself).

Algorithm 1: overlapLs($R[r_1, \ldots, r_n]$)

```
1  sort($R$) according to $f(i)$ in ascending order
2  $S = \emptyset$
3  while $|R| > 1$ do
4      $A = overlap(r)$
5      $S = S \cup a$ where $a$ has the latest ending time in $A$
6      $R = R \backslash overlap(r)$
7  if $|R| = 1$ and $R[0]$ does not overlap with some segment in $S$ then
8      return $S \cup \{R[0]\}$
9  else
10     return $S$
```

Proof of Optimality and Correctness: At each iteration, we only remove the line segments that overlap with r. We claim that a overlaps with all line segments in *overlap*(r). If there exists a line segment r' such that $f(r') < s(a)$, then before this iteration, r is already in S not in R, a contradiction. Since a overlaps with all line segments in *overlap*(r), this is correct.
At each iteration, the greedy algorithm selects the line segment that overlaps with most segments containing the one with earilest starting time in R. So by construction, a is the longest line segment that also contains the one with earilest finishing time. Hence we cannot find another segment that overlaps as many as a can and also contains the one with earilest finishing time. Therefore, the greedy algorithm always stays ahead.

Runtime Analysis: Sorting takes $O(n \log n)$ time. The while loop iterates at most n times, and takes $O(n)$ time to find all overlapping intervals of a for each $r \in R$. So the total runtime is $T(n) \leq O(n \log n) + nO(n) = O(n^2)$.

4. [*12 marks*] There is an election that will take place in your country between two parties, party A and party B. The election takes place in n states and you have calculated the probability of party A winning in each state i as a_i (so party B winning in state i is $(1 - a_i)$). You can assume that the outcomes of the elections in each state is independent from each other. Given the n probabilities of party A winning in each state, $a_1, ..., a_n$, design an $O(n^2)$ dynamic programming algorithm that computes the probability of party A winning the general election, i.e., the probability that A wins in strictly more than $n/2$ states.

   Here are two facts from probability theory that will be needed for this problem:

   Independence: When two events e_1 and e_2 are independent, then the probability of both events happening is the multiplication of their probabilities. For example, in our example, the probability of party A winning in states 1 and 2 are $a_1 a_2$.

   Union/OR of Mutually Exclusive Events: The probability of one of two mutually exclusive events e_1 and e_2 is happening the sum of their probabilities. Two events are mutually exclusive if they cannot occur at the same time. For example, let e_1 be the probability of party A winning in state 1 and state 2, which happens (by independence) with probability $a_1 * a_2$. Let e_2 be the event that party A loses in both state 1 and state 2, which happens with probability $(1 - a_1)(1 - a_2)$. The probability that party A either wins or loses in states 1 and 2 is $a_1 a_2$ + $(1 - a_1)(1 - a_2)$.

Algorithm 2: ProbAWin($a_1, \dots, a_n$)

```
1 $P = 0_{(n+1)\times(n+1)}$
2 $P[0, 0] = 1$
3 $P[0, i] = \Pi_{j=1}^{i}(1 - a_j)$ for $i \neq 0$
4 for $i = 1, \dots, n$ do
5     for $j = 1, \dots, i$ do
6         $P[i, j] = P[i - 1, j - 1] \times a_j + P[i - 1, j] \times (1 - a_j)$
7 return $sum(P[\lceil n/2 \rceil, n], P[\lceil n/2 \rceil + 1, n], \dots, p[n, n])$
```

Proof of Correctness: Let (A, i, j) be the event that A wins in j states with i states in total. Since for each state j, the outcome is either win or loose.

$$\begin{aligned} P((A, i, j)) &= P((A, i-1, j-1)) \times P(\text{state j wins}) + P((A, i-1, j)) \times P(\text{state j loses}) \\ &= P((A, i-1, j-1)) \times a_j + P((A, i-1, j)) \times (1 - a_j) \end{aligned}$$

And, P(A wins in strictly more than n / 2 states) $= P((A, \lceil n/2 \rceil, n)) + P((A, \lceil n/2 \rceil + 1, n)) + \cdots + P((A, n, n))$.

Runtime Analysis: $T(n) = O(n^2) + O(n) = O(n^2)$