Homework #4
Introduction to Algorithms/Algorithms 1
601.433/633
Spring 2020

Due on: Tuesday, March 10th, 12pm

Where to submit: On Gradescope.
Please type your answers; handwritten assignments will not be accepted.

1 Problem 1 (15 points)

Suppose you are managing the construction of billboards on an east-west highway that extends in a straight line. The possible sites for billboards are given by numbers $x_1, x_2, \ldots, x_n$ with $0 \leq x_1 < x_2 < \cdots < x_n$, specifying their distance in miles from the west end of the highway. If you place a billboard at location x_i, you receive payment $p_i > 0$.

Regulations imposed by the Baltimore County's Highway Department require that any pair of billboards be more than 5 miles apart. You'd like to place billboards at a subset of the sites so as to maximize your total revenue, subject to that placement restriction.

For example, suppose n = 4, with

$$\langle x_1, x_2, x_3, x_4 \rangle = \langle 6, 7, 12, 14 \rangle,$$

and

$$\langle p_1, p_2, p_3, p_4 \rangle = \langle 5, 6, 5, 1 \rangle.$$

The optimal solution would be to place billboards at x_1 and x_3, for a total revenue of $p_1 + p_3 = \$10$.

Give an $O(n)$ time dynamic-programming algorithm that takes as input an instance (locations $\{x_i\}$ given in sorted order and their prices $\{p_i\}$) and returns the maximum revenue obtainable. As usual, prove correctness and running time of your algorithm.

Proof. To see the optimal substructure, suppose an optimal solution S places a billboard at position x_j. Let x_i be the latest position such that $x_i < x_j - 5$, and let x_k be the earliest position such that $x_k > x_j + 5$. Then S consists of an optimal solution for billboards $x_1, \ldots, x_i$, the billboard at x_j, and an optimal solution for $x_k, \ldots, x_n$. You could prove this formally with a cut-and-paste (contradiction) argument. For the purposes of the recursive formula, choosing x_j to be the latest billboard so that subproblems correspond to a prefix is the most efficient.

Before we define a recursive formula, lets define some notation. Looking at the sub-structure, we need a "latest position such that $x_i < x_j - 5$", so lets define $prev(j) = max\{i < j | x_i < x_j - 5\}$ to denote this value, with $prev(j) = 0$ if it is otherwise un- defined. Well come back to how to calculate this later.

Now we can define the recursive formula. Let $P(j)$ denote the optimal revenue obtainable from billboards $1, 2, \ldots, j$. That is, $P(j)$ is the solution using positions $x_1, x_2, \ldots, x_j$. For notational convenience, we set $P(0) = 0$. Then we can calculate $P(j)$ for $j > 0$ recursively as $P(j) = max\{P(j-1), P(prev(j)) + p_j\}$. □

2 Problem 2 (20 points)

You are given two numbers, n and k, such that $n \in \mathbb{N}$ and $k \in \{1, \ldots, 9\}$. Use dynamic programming to devise an algorithm which will find the number of $2n$-digit integers for which the sum of the first n digits is equal to the sum of the last n digits and each digit takes a value from 0 to k.

For example, when $k = 2$ and $n = 1$: you have only 3 such numbers 00, 11, 22. For example, when $k = 1$ and $n = 2$: you have only 6 such numbers 0000, 0101, 0110, 1001 , 1010, 1111.

Your algorithm should work in time polynomial of n and k. Prove correctness and provide running time analysis.

Proof. Let us first reduce this problem to a smaller problem. Note that we only

Algorithm 1 Algorithm 1 Problem 1 Dynamic Programming Pseudocode

input: n and k
table = int$[n+1][nk+1]$ ▷ initialize the table with all zeros
for $col = 0 \ldots k$ **do** ▷ add 1s to row r = 1
    table[1][col] = 1
end for
▷ populate the table
▷ traverse through the table row then column
for $row = 2 \ldots n$ **do**
    for $col = 0 \ldots nk$ **do**
        ▷ go through all columns (sums) before this current column col
        for $i = 0 \ldots col$ **do**
            ▷ go through all possible digit values to add to previous sums
            for $j = 0 \ldots k$ **do**
                ▷ if new sum == current sum col, add old sum table entry
                if $i + j == col$ **then**
                    table[row][col] += table[$row - 1$][i]
                end if
            end for
        end for
    end for
end for
▷ calculate the answer
$ans = 0$
for $col = 0 \ldots nk$ **do**
    $ans = ans + (table[n][col])^2$
end for

need to be concerned with the number of $2n$-digit integers. Also note that we only need to concern ourselves with the number of ways to construct n digit integers and then we can square that number to get the number of ways to construct $2n$ digit integers that satisfy our condition. Consider the example of $n = 2$ and $k = 2$. There are two ways to create a sum of 1 using $n = 2$ digits: 01 and 10. There are four ways to create a sum of 1 on both ends of a 2n-digit number: $0101, 0110, 1001, 1010$. You can see that the number of permutations that satisfy our condition for a $2n$-digit number is the square of the number of permutations for that sum using n digits. We can then sum the squares of all possible sums for n digits and we arrive at our answer. In total, there are $1 + 4 + 9 + 4 + 1 = 19$ numbers that satisfy our given example.

Now we can use dynamic programming to solve this reduced problem. We will memorize the number of ways to construct a sum S_{nk} using n digits. Thus, our table will be of size $n \times nk$. We will be storing integers in each entry. (See algorithm pseudocode below). We will now show that this is correct. To do this, we must show that what we are memoizing is correct and how we are using our memoized entries is correct.

Claim: The table entries at row r are correct.

Proof: Base Case: $r = 0$. All entries in the row of $r = 0$ are trivially 0 because there are 0 ways to ever make a number of 0 digits for any sum. For $r = 1$, it is clear to see that we can only use the digit $d = c$ to construct a sum of c with $r = 1$ digit. The sum is simply the number we choose for the 1 digit, and any other number $x \neq d$ will never be c.

Induction Hypothesis: Let us assume that this is correct for $r = k$.

Induction Step: We want to show that this is correct for $r = k + 1$. WLOG let us consider an arbitrary column c' in the row $r = k + 1$. For any entry in the row $r = k + 1$, its value is the sum $\forall j \in [0, k], \sum_{i=0}^{c'} \infty[i + j == c'](table[r - 1][i])$. Where $\infty[i + j == c']$ is the indicator variable that represents 1 when $i + j == c'$ and 0 otherwise. It can be seen that this equation adds the number of ways to create this $k + 1$-digit number using the k-digit number and appending any single digit in $[0, k]$. This sum encompasses the total number of ways to create the sum c' because, since we have shown by the IH that all values in the previous row $r = k$ are correct, we know that the numbers are correct for k-digit numbers and we know we can only make this $r = k+1$-digit number by appending some number in $[0, k]$, which we exhaustively check for in the algorithm. This extends to not just column c' but all columns in the row $r = k + 1$. Thus, by IH, we show that the entries in the table are correct.

Claim: The algorithm is correct.

Proof: Note that there is a maximum of nk possible sums for this problem. This is because with n-digit numbers and a max digit value of k, we only have nk options. The algorithm, after memorizing everything and storing all number entries in the table, requires we take the sum of the squares of the entries in the last row of the table. Consider a single entry at row $r = n$, column $c = c_f$ with entry $table[r][c_f]$. This is equivalent to looking at the number of possible permutations of an n-digit number with a sum of c_f . By squaring this number, $(table[r][c_f])^2$, we find the number of permutations for a $2n$-digit number respecting the restriction

$\sum_{i=0}^{n} N[i] = \sum_{j=n+1}^{2n} N[j]$, where $N[i]$ is the i-th digit of the number N . We count the number of times we can match the sum c_f for n-digit number with itself to create a $2n$-digit number. When we sum this square from the bottom row, where $r = n$, going through all possible sums (nk columns), we get the total number of ways to get $2n$-digit numbers total for the problem. Now let us prove runtime is in order n and k. Initialization of the table is of time $O(n * nk) = O(n^2k)$. We must go through all entries of the table, all previous column entries in the previous row for every entry, and through all possible k values in lines 9-22: $O(n*nk*nk*k) = O(n^3k^3)$. We must then go through the last row and sum all the squares of the columns: $O(nk)$. The total time is thus $O(n^2k) + O(n^3k^3) + O(nk) = O(n^3k^3)$ which is in the order of n and k. □

3 Problem 3 (15 points)

Alice and Bob found a treasure chest with different golden coins, jewelry and various old and expensive goods. After evaluating the price of each object they created a list $P = \{p_1, \ldots, p_n\}$ for all n objects, where $p_i \in \{1, \ldots, K\}$ is the price of the object i. Help Alice and Bob to check if the treasure can be divided equally, i.e. if it is possible to break the set of all abjects P into two parts P_A and P_B such that $P_A \cup P_B = P$, $P_A \cap P_B = \emptyset$ and $\sum_{i \in P_A} p_i = \sum_{i \in P_B} p_i$?

Your algorithm should run in time polynomial in n and K. As usual, prove correctness and running time of your algorithm.

Proof. Let's denote the total cost of all objects as $T = \sum_{i=1}^{n} p_i$. If such P_A and P_B exist, then

$$\sum_{i \in P_A} p_i = \sum_{i \in P_B} p_i = \frac{T}{2}.$$

Then we can conlude that the problem is equivalent to checking if

$$\exists P_A, \text{ s.t. } \sum_{i \in P_A} p_i = \frac{T}{2}.$$

For all $i \in \{1, \ldots, n\}$ define $P_i \subset P$ as a list of prices of first i objects:

$$P_i = \{p_1, \ldots, p_i\}$$

We will create a table T with dimensions $(n+1) \times (\frac{T}{2}+1)$, i.e. rows with indeces $i \in \{0, \ldots, n\}$ and columns with indices $j \in \{0, \ldots, T/2\}$. $T(i,j) = TRUE$ if and only if $\exists P' \subset P_i : \sum_{k \in P'} p_k = j$, i.e. there is a subset of objects with indices $\leq i$ with the total cost equals j. We will initialize first row of the table as:

$$T(0,0) = TRUE \text{ and } \forall j > 0 : T(0,j) = FALSE,$$

And will compute every next row i based on the row $i-1$ as follows:

$$T(i,j) = (T(i-1, j-p_i)) \textbf{ OR } (T(i-1,j))$$

. When all the values in the table are computed the algorithm need to output the value in the cell $(n, T/2)$.

Correctness

We will prove that all values in the table are correct by induction.

BC:

$$T(0,0) = TRUE \text{ and } \forall j > 0 : T(0,j) = FALSE$$

First row is correct as empty set costs 0.

IH:

Suppose that $(i-1)$-th row is correct.

IS:

For every $j \in \{1, \ldots, T/2\}$ argument is the following.
If $\exists P' \subset P_i : \sum_{k \in P'} p_k = j$ then there are two cases:

1. $p_i \in P'$ then there should exist $\exists P'' \subset P_{i-1} : \sum_{k \in P''} p_k = j - p_i$ and the algorithm will correctly put $T(i,j) = T(i-1, j-p_i)$ (due to IH).
2. $p_i \notin P'$ then there should exist $\exists P'' \subset P_{i-1} : \sum_{k \in P''} p_k = j$ and the algorithm will correctly put $T(i,j) = T(i-1,j)$ (due to IH).

If there is no such P' the argument is in the similar fashion: if there is no such P' then there is no such P'', thus due to IH i-th row is also correct

Running time

To find out the total cost algorithm need to read the entire array which costs $O(n)$, then it takes $O(n)$ steps to fill in the first row of the table. To compute each cell algorithm check two cells on the previous row, thus in total to fill in the table algorithm will spend $O(nT/2) = O(n^2K)$, as total sum is bounded by maximum cost of each item times the number of items. $\square$