# Homework #4
# Introduction to Algorithms/Algorithms 1
# 600.433/633
# Spring 2018

**Due on:** Thursday, March 8, 5.00pm
**Late submissions:** will NOT be accepted
**Format:** Please start each problem on a new page.
**Where to submit:** Gradescope.
Please type your answers; handwritten assignments will not be accepted.
To get full credit, your answers must be explained clearly,
with enough details and rigorous proofs.

March 13, 2018

## Problem 1 (10 points)

You are given two numbers, $n$ and $k$, such that $n \in \mathbb{N}$ and $k \in \{1, ..9\}$. 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.

## Solution

Let us first reduce this problem to a smaller problem. Note that we only 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 $\forall j \in [0, k], \sum_{i=0}^{c'} \infty[i + j == c'](table[r1][i])$. 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.

**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) bethis 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**

**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 $O(nnk) = O(n^2k)$. 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: $O(nnknkk) = O(n^3k^3)$. 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.

## Problem 2 (10 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 $\sum_{i \in P_A} p_i = \sum_{i \in P_B} p_i$?

## Solution

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.

## Problem 3 (10 points)

Let $X$ be a set of $n$ intervals on the real line. We say that a set $P$ of points stabs $X$ if every interval in $X$ contains at least one point in $P$. Describe and analyze an efficient algorithm to compute the smallest set of points that stabs $X$. Assume that your input consists of two arrays $XL[1 \ldots n]$ and $XR[1 \ldots n]$, representing the left and right endpoints of the intervals in $X$. Prove correctness and running time.

## Solution

We propose the following greedy algorithm:

1. Let $P = \emptyset$
2. Sort arrays $XL, XR$ using any $O(n \log n)$ sorting algorithm such as merge sort. Keep equivalence tables $Y_L, Y_R$ (see note) of the same size of the arrays to remember which ending in $XR$ applies to each start in $XL$. Mark all intervals as *not stabbed*.
3. Add to $P$ the first ending $e$ in $XR$ of an interval that is not stabbed.
4. Mark all intervals that start at $s' < e$ and end at $e' \geq e$ as stabbed.
5. Repeat items 3 and 4 until all intervals are stabbed.

*Note: An equivalence table is just a common array in which the indexes represent the position in the array being sorted and the values the position in the other array. That is, if an interval starts at $XL[2]$ and ends at $XR[0]$, the equivalence tables will have $Y_L[2] = 0$ and $Y_R[0] = 2$. Whenever a swap happens in $XL$, we swap the values stored on the equivalent indexes in $Y_L$ (Adds O(1) per swap). Before sorting the other array, the table can be easily inverted into $Y_R$ (indexes and values flipped) in total O(n) time. Then the same procedure applies to $XR$ and finally $Y_R$ is inverted to update $Y_L$, at which point both tables will be correct. This adds overall $O(n)$ time to a step that is already $O(n \log n)$*

**Proof of correctness**

Now we must prove the correctness of our algorithm. Assume it is not correct.

First, we will show that all of the intervals in $X$ must be stabbed by $P$. Suppose the opposite, that there exists an interval $I$ which is not stabbed by $P$. This would imply that the right end point of $I$ is not in $P$ and thus it was skipped by the algorithm on the step 3. Step 3 of the algorithm skips only those end points which belong to the intervals marked as already stabbed. The interval can be marked as already stabbed only on the Step 4 of the algorihm, and only for those intervals which start at $s' < e$ and end at $e' \geq e$ for some $e \in P$. Thus $I$ is stabbed by some $e \in P$ by definition which leads us to contradiction.

Second, we will show that $P$ is the smallest set that stabs $X$. Assume it is not, i.e. there exist a set $P'$, such that it stabs $X$ and $|P'| < |P|$. Let's use the following notation for the members of $P$ and $P'$:

$$P = \{p_1, \ldots, p_k\} \text{ and } P' = \{p'_1, \ldots, p'_{k'}\} \text{ where } k' < k$$

**Statement:**

$$\forall i \in \{1, \ldots, k'\} : \ p'_i \leq p_i$$

**Proof by induction:**
**BC:**
$p'_1 \leq p_1$, by contraiction: if it is not the case then the interval with the smallest right end point (which is by construction of our algorithm equal to $p_1$) is not stabbed by $P'$, as the smallest point $p'_1 \in P'$ is larger than interval's right end point.
**IH:**
Assume that $p'_{i-1} \leq p_{i-1}$
**IS:**
Suppose the opposite that $p'_i > p_i$, then from the step 3 of the algorithm we can conlude that the interval with $e = p_i$ has $s > p_{i-1}$, otherwise it would already be stabbed. Therefore using the IH we can conclude that $p'_i > p_i = e > s > p_{i-1} \geq p'_{i-1}$, thus interval $(s, e)$ is not stabbed by $P'$. thus we have a contradiction.

**Statement:**
If $k' < k$ then $P'$ is not stabbing all the intervals.
**Proof:**
From previous statement we can conclude that $p_k > p_{k-1} \geq p'_{k'}$, then from the step 3 of the algorithm we can conlude that the interval with $e = p_k$ has $s > p_{k-1}$, otherwise it would already be stabbed. Therefore we can conclude that $p_k = e > s > p_{k-1} \geq p'_{k'}$, thus interval $(s, e)$ is not stubbed by $P'$ .

Thus no such $P'$ exists and our output is optimal. Finally, we will prove the running time of this algorithm. Step 1 takes $O(1)$ time and step 2 takes $O(n \log n)$ since keeping the equivalence table adds only a constant number of extra operations to each swap in the sorting procedure. For steps 3-5 note that we only consider each element in $XR$ and $XL$ once, do a constant number of operations (compare to the ending added, mark as stabbed) and move to the next element. Therefore, we do at most $O(n)$ operations. Hence, the overall complexity of the algorithm is $O(n \log n)$.

## Problem 4 (10 points)

Let $T$ be a red-black tree. Let $x$ and $y$ be two nodes in $T$ such that $bh(x) \leq bh(y) \leq 5bh(x)$. Let $P_1$ be a longest simple path from $x$ to a descendant leaf and let $P_2$ be a shortest simple path from $y$ to a descendant leaf. What is the relation between the lengths of $P_1$ and $P_2$? Prove your answer.

## Solution

In a red-black tree, the shortest possible path from a node to a leaf will include only black nodes, and the longest possible path from a node to a leaf will alternate between red and black nodes. Therefore, a path from a node $x$ to a leaf will have a length between $bh(x)$ and $2bh(x)$. Therefore, from the problem definition and the definition of red-black trees, we have the following three equations:

$$bh(x) \leq bh(y) \leq 5bh(x)$$

$$bh(x) \leq length(P_1) \leq 2bh(x)$$

$$bh(y) \leq length(P_2) \leq 2bh(y)$$

From the first inequality we know that $2bh(y) \leq 10bh(x)$ and $bh(x) \leq bh(y)$, and we can combine this with the third inquality to get:

$$bh(x) \leq length(P_2) \leq 10bh(x)$$

It follows from the second inequality that:

$$bh(x)/2 \leq length(P_1)/2 \leq bh(x)$$

and

$$10bh(x) \leq 10length(P_1) \leq 20bh(x)$$

Thus combining last three inequalities we can get

$$length(P_1)/2 \leq length(P_2) \leq 10length(P_1)$$

## Problem 5 (10 points)

Let $D$ be a data structure with two operations $\alpha(i)$ and $\beta(i)$. Every $\alpha(i)$ operation costs $O(i)$. Any $\beta(i)$ operation takes time $O(1)$. Assume that every $\alpha(i)$ is followed by at least $i$ operations of type $\beta(i)$. That is, if $S$ is a sequence of $m$ operations then every operation $\alpha(i)$ from $S$ is followed by at least $i$ operations $\beta(i)$. Use direct computations to prove that the amortized time of both operations is $O(1)$.

## Solution

Assume that the $m$ operations in question use the $\alpha(i)$ operation $K$ times, to process the $K$ integers, $i_1, i_2, \ldots, i_{K-1}, i_K$ . In this case, our $m$ operations must include exactly $K$ calls to $\alpha(i)$ and at least $\sum_{k=1}^{K} i_k$ calls to $\beta(i)$, as required by the problem definition. We may also have $x$ additional calls to $\beta(i)$, beyond what is required by the problem definition, where $x$ may or may not be zero. This means that

$$m = K + \sum_{k=1}^{K} i_k + x$$

or,

$$m - K - x = \sum_{k=1}^{K} i_k$$

Now, assume without loss of generality that the running time of the $\alpha$ and $\beta$ operations is:

$$T(\alpha(i)) \leq c_\alpha i$$

$$T(\beta(i)) \leq c_\beta$$

for some constants $c_\alpha$ and $c_\beta$ . We will refer to the total running time of the $m$ operations as $T(m)$. We can calculate $T(m)$ as the sum of the costs of $K$ calls to $\alpha$ and $\sum_{k=1}^{K} i_k$ calls to $\beta$, plus an additional $x$ calls to $\beta$:

$$T(m) = \sum_{k=1}^{K} T(\alpha(i_k)) + \sum_{k=1}^{K} i_k T(\beta(i_k)) + \sum_{k=1}^{x} T(\beta(i))$$

$$T(m) = \sum_{k=1}^{K} T(\alpha(i_k)) + i_k T(\beta(i_k)) + \sum_{k=1}^{x} T(\beta(i))$$

$$T(m) = \sum_{k=1}^{K} c_\alpha i_k + c_\beta i_k + \sum_{k=1}^{x} c_\beta$$

$$T(m) = (c_\alpha + c_\beta) \sum_{k=1}^{K} i_k + c_\beta x$$

This simplifies to

$$T(m) \leq (c_\alpha + c_\beta)(m - K - x) + c_\beta x$$

$$T(m) \leq m(c_\alpha + c_\beta)$$

Let $T_0$ be the amortized cost of one of our $m$ operations. Then

$$T_0 = \frac{T(m)}{m} = \frac{(c_\alpha + c_\beta)m}{m} = c_\alpha + c_\beta$$

Thus, the amortized cost of a single operation is no more than $c_\alpha + c_\beta$ , a constant as desired.