# Homework #4
# Algorithms I
# 600.463
# Spring 2017

**Due on:** Tuesday, March 7th, 11:59pm
**Late submissions:** will NOT be accepted
**Format:** Please start each problem on a new page.
**Where to submit:** On Gradescope, under HW4
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 9, 2017

## 1 Problem 1 (20 points)

When you are checking out at BalMart, you want to make change for $A$ cents. Assuming that the cashier has infinite supply of each of $C = \{C_1, C_2, \ldots, C_t\}$ valued coins, can you count how many ways to make change for $A$ cents? Give an efficient dynamic programming algorithm and analyze the running time. Here we don't consider the order of the coins.

For example, when $A = 3$ and $C = \{1, 2, 3\}$, there are three solutions: $\{1, 1, 1\}$, $\{1, 2\}$, $\{3\}$.

**Brief Answer:**
For a dynamic programming solution, let's first define the optimal substructure. To count the number of different ways, we can divide the solutions into two parts:

- Solutions that have one or more $C_t$ coins.
- Solutions that do not have $C_t$ coins.

Thus we can define function Coin-Count$(C, t, A)$ = Coin-Count$(C, t - 1, A)$ + Coin-Count$(C, t, A - C_t)$.

In this recursion formula, the same subproblems are repeatedly called. We can find there are many of these subproblems are overlapped. We can use a table from bottom up to store the results of these overlapped subproblems to avoid the repeated same computations.

Coin-Count($C[], t, A$):

```
1 initialization table[0...A] to all '0'
2 table[0] = 1
3 for i = 1 to t do
4 |   for j = C[i] to A do
5 |   |   table[j] = table[j] + table[j - C[i]]
6 |   end
7 end
8 return table[A]
```

Here, line 2 gives the base case when $A$ is $0$ and the initialization covers all other base cases with $0$ solutions. Lines 3-6 check the coins one by one and update the table (when the index $j$ is greater than or equal to $C[i]$). A formal correctness proof is omitted.

The two for loops takes $O(t \cdot A)$ iterations and each table update takes $O(1)$. So in total the algorithm runs in $O(tA)$.

## 2 Problem 2 (20 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

$$< x_1, x_2, x_3, x_4 > = < 6, 7, 12, 14 >,$$

and

$$< p_1, p_2, p_3, p_4 >=< 5, 6, 5, 1 > .$$

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 efficient 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 from a valid subset of sites. Analyze the running time of your algorithm. Your solution must clearly define a recursive formula.

**Brief Answer:**

**Solution 1:** 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 undefined. 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 $P(j) = max\{P(j-1), P(prev(j)) + p_j\}$, where $P(j-1)$ means "don't place a billboard at $x_j$" and $P(prev(j)) + p_j$ means "place a billboard at $x_j$".

The dynamic program would then look as the following:

```
1 initialization P[0...n]
2 P[0] = 0
3 for j = 1 to n do
4 |  do calculate prev(j)
5 |     P[j] = max{P[j-1], P[prev(j)] + p_j}
6 end
7 return P[n]
```

Ignoring Line 4, the cost of filling in the table is $\Theta(n)$, since its just doing two table lookups and a few calculations in each iteration. The cost of Line 4 thus dominates. Naively, you could compute $prev(j)$ by looping from $j - 1$ down to

0, stopping when you find a position that is far enough away. If all $x_1 > x_n - 5$ meaning that all billboards are within 5 miles, billboards are within 5 miles, however, the $j$-th iteration would then take $\Theta(j)$ time, giving a total of $\Theta(n^2)$.

**Solution 2:** A better solution for computing $prev(j)$ is to use binary search over the $\{x_i\}$ for the value $x_j - 5$ (after all, they are already sorted, thus binary search is valid), giving a running time of $O(\log n)$ per iteration. In total it is $O(n \log n)$.

**Solution 3:** Now, can we do even better? The idea is to precompute all values of $prev(j)$ up front, before running the dynamic program. To do this, To do this, keep two pointers, $i$ and $j$, such that $x_i < x_j - 5$. Each time you increment $j$, advance $i$ as far as you can to make this condition hold. You can do this with the following pseudo code:

build a table $prev[1 \dots n]$
$i = 0$
**for** $j = 1$ *to* $n$ **do**
    **while** $x_{i+1} < x_j - 5$ **do**
        $i = i + 1$
    **end**
    $prev[j] = i$
**end**

While any iteration of the outer loop may be very expensive, in total the pointer $i$ can only be advanced $n$ times. So the total cost is $\Theta(n)$ for computing all values of $prev[1 \dots n]$. When incorporated into the dynamic program, the total running time comes out to $\Theta(n)$.