**CS141, Programming III Report** **Due: 23:59, Tue 10/25, 2022**

**Name: Majd Kawak / NetID: mkawa025 / CodeForces ID: mkawa025 / Student ID: 862273310**

---

# A. Merge the Candies!

Submission#: 177418310

In this program, firstly declared an *int* that would hold how many time Yihan needs to count her candies, an *int* for bag number and lastly a *vector* to hold candies number in each bag. Then I went on and implemented Merge Sort algorithm where I sorted *vector* of candy bags in a descending order(largest first to smallest last) Time complexity is $O(n\ log\ n)$. I get last two elements of my *vector* using *.back*(), which are the smallest 2 candy bags I have. Count their candies twice $(1st_{bag} + 2nd_{bag} + (1st_{bag} + 2nd_{bag}) = 2 * (1st_{bag} + 2nd_{bag}))$ and save it to my total candies count Time complexity is $O(1)$. Pop them using *.pop*() from my *vector* and push their total into a new bag, Time complexity is $O(2 * n) = O(n)$. I repeat this till I reach 1 nag which is my total candies bag. Lastly I display my total candies counted which is number of candies Yihan needs to count. Total time complexity $= O(n\ log\ n) + O(1) + O(n) = O(n\ log\ n)$.

# B. The simplest knapsack problem in the world

Submission#: 177613526

In this program, I start with declaring an *int* to hold my knapsack size and *int* for number of items. Then I go and create a two 2D *vector* to represent my weight/value and solution Matrix with size $[items+1][knapsack+1]$ to offset first row and column 0/0. After that I start computing my solution Matrix solution using this formula:

$$i = \text{item row}$$

$$w = \text{weight of column}$$

$$v_i = \text{value of item}$$

$$V[i][w] = \begin{cases} max(V[i-1][w], V[i-1][w-w_i] + v_i), \text{ if } w_i \leq w \\ V[i-1][w] \end{cases}$$

First solution is to add current row item to column weight and offset of current weight column backward and add item available, in case I cant add First solution I just copy previous solution(Second solution). I repeat this operation till Solution Matrix gets populated, display max value I got at the end(its usually last item in matrix). Time complexity is $O(n * w)$ where $n$ is the number of items and $w$ is weight limit.

# C. A slightly more complicated knapsack problem

Submission#: 177809048

This program is basically identical to my B program with 1 extra condition, items can be repeatedly picked. I start with declaring an *int* to hold my knapsack size, *int* for number of items and *int* to hold allowed repetition number. Then I go and create a two 2D *vector* to represent my weight/value and solution Matrix

with size $[items + 1][knapsack + 1]$ to offset first row and column 0/0. After that I start computing my solution Matrix solution using this formula:

$$i = \text{item row}$$

$$w = \text{weight of column}$$

$$v = \text{value of item}$$

$$k = \text{allowed item repetition}$$

$$V[i][w] = \begin{cases} max(V[i-1][w], V[i-1][w-w_i] + k * (v_i)), \text{ if } k * (w_i) \leq w \\ V[i-1][w] \end{cases}$$

First solution is to add current row ($k$ items) to column weight and offset of current weight column backward and add item available, in case I cant add First solution I just copy previous solution(Second solution). I repeat this operation till Solution Matrix gets populated, display max value I got at the end(its usually last item in matrix). Time complexity is $O(n * w)$ where $n$ is the number of items and $w$ is weight limit.

# D. Ski

Submission#: 177938119

In this program, I start off with allocating 2 *int* to get row and column size, two 2D *vector* for input values(heights) "Resort Matrix" and for path to solve at each point "Matrix to Solve". I start by taking input heights for "Resort Matrix", then I go and compute longest path at each point save path value to "Matrix to Solve". I implemented a function that computes all possible paths at each point, checking 4 possible paths using $if\ [i,j] > [(i-1,j)(i-1,j)], [(i+1,j)(i+1,j)], [(i,j-1)(i,j-1)], [(i,j+1)]$ *then go and compute path.* This is done recursively till all points in "Resort Matrix" are computed. Display longest path at the end. Time complexity is $O(n * n) = O(n^2)$.

# References


[1] geeksforgeeks *0-1Knapsack* https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

[2] geeksforgeeks *Unbounded Knapsack* https://www.geeksforgeeks.org/unbounded-knapsack-repetition-of-items-allowed-set-2/

[3] geeksforgeeks *Longest Path in a Matrix* https://www.geeksforgeeks.org/longest-increasing-path-matrix/