11 INTRODUCTION TO DYNAMIC PROGRAMMING

Introduction

Dynamic Programming (**DP**) is a method for solving a complex problem by

- breaking it down into subproblems,
- solving each of those subproblems just once,
- and storing their solutions using a memory-based data structure (array, map, etc.)

We usually use this method to solve the **decision problems** (if possible), **optimization problems** (minimum cost, maximum profit vs.) and **combinatorial problems** (in how many different ways …, what is the number of …). **Bellman-Ford** and **Floyd-Warshall** shortest paths alogrithms are two examples of DP algorihtms.

Dynamic programmig is **a clever brute force** where we **trade space for time**.

DP Keywords

- **Overlapping subproblems** (smaller version of the original problem)
- **Optimal substructure**
- **Recursion** (backtracking) with **memoization** (top-down design - deduction)
- Redundant data
- **Tabulation** / table filling (bottom-up design - induction)
- Pre-calculation
- Forward or Backward solution

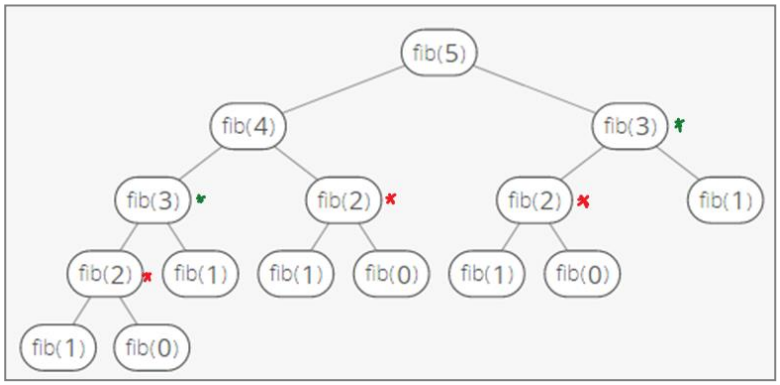
Overlapping Subproblems

A problem is said to have overlapping subproblems if the problem can be broken down into subproblems which are **reused several times** or a recursive algorithm for the problem **solves the same subproblem** over and over rather than always generating new subproblems.

N*th* Fibonacci number problem is a typical example to understand the overlapping subproblems property.

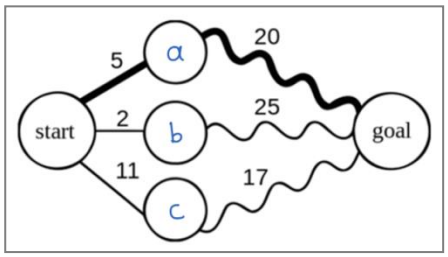
```
fib(0) = 0 and fib(1) = 1        (base case)

fib(n) = fib(n-1) + fib(n-2)     (recursive relation)
```

The following illustration shows the recursion tree of *fib(5)* function. As you notice, *fib(3)* and *fib(2)* functions are called multiple times. Therfore, *fib(3)* and *fib(2*) are overlapping subproblems.



Optimal Substructure

A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems. Consider the shortest path problem between two cities: Suppose that the shortest path passes trhough the *city a*. Since the shortest path problem has a optimal substructure property, the length of the paht from the source city to the destination city will be the sum of the distances from the source city to *city a* and form *city a* to the destination city.



On the other hand, the longest paht probem on a graph does not have the optimal substructure property. Therefore, we cannot solve it by dividing the origianal problem into some smaller subproblems.

Top-Down Design vs Bottom-Up Design

There are two approaches of implementing a dynamic programming program. The first one is the top-down approach and the second is the bottom-up approach. In both approaches, you would have to determine the recurrence relation and the base case. However, top-down is implemented recursively; bottom-up method is implemented iteratively.

Top-Down Design

Top-down design of dynamic programming is a recursive solution with memoization (a word derived from memo for this). **Recursion** and the **memoization** are two keywords of top-down design. **Memoization** is the technique to avoid repeated computation on the same problems. It just uses a data structure (array, set, map etc.) to store the solutions of subproblems.

We make a **separate recursive function call for each subproblem** of the current problem. Since the original problem is the largest problem, the first recursive call is made for the original problem. Eventually we will get the smallest subproblems (**base case**) where we stop recursive function calls and just return its solution.

In other terms, it can also be said that we just hit the problem in a natural manner and hope that the solutions for the subproblem are already calculated and if they are not calculated, then we calculate them on the way.

You may follow theese steps in top-down design:

- Define subproblems (subtree, prefix/suffix of an array, subsets, partitions of plane etc.)
- Determine what to calculate for each subproblem and what variables you will need in this calculation.
- Determine the base case and its answer. Base case is a subproblem that cannot be further divided into some other subproblems.
- Formulate the solution. It should contain the base case and the recursive realtion.
- Determine the memoization technique. It should be time and space efficiency.

Recursive DP Function

- Most of the time it returns the solution of the current subproblem. Not a *void* function.
- If the current subproblem is in memo, return the result directly. Avoid recalculation.
- Otherwise calculate the current subproblem, memoize and return the result.
- Do not modify the global variables other than the memo.
- Remove the redundant local variables and function parameters.

Example 1: Sum

In how many different ways you can obtain N as sum of a, b, and c? N, a, b and c are positive integers. Repetitions are allowed.

Sample Input

```
5
1 3 4
```

Sample Output

```
6
```

Output Details

```
1 + 1 + 1 + 1 + 1
1 + 4
4 + 1
1 + 1 + 3
1 + 3 + 1
3 + 1 + 1
```

We can formulat this problem with the following recurrence. *F(n)* is the solution for *n*. To calculate *F(n)* we should know the answers of three subproblems: *F(n-a)*, *F(n-b)* and *F(n-c)*. if *n* is equal to 0 the answer is 1; if n is smaller than zero, the answer is 0.

$$F(n) = \begin{cases} 1 & \text{if } n == 0 \quad \text{//base case} \\ 0 & \text{if } n < 0 \quad \text{//base case} \\ F(n-a) + F(n-b) + F(n-c) \end{cases}$$

```
// Sum DP Top Down.cpp
#include <iostream>
#include <vector>

using namespace std;

int N, a, b, c;
vector<int> memo;
//-------------------------------------------------
int go(int n)
{
	//base case
	if (n == 0)
		return 1;
	if (n < 0)
		return 0;

	//Curent subproblem has been already calculated
	if (memo[n] > 0)
		return memo[n];

	//Call this function recursively for the three subproblems.
	//Calculate and store the solution in memo and return it.
	return memo[n] = go(n - a) + go(n - b) + go(n - c);
}
//-------------------------------------------------
int main()
{
	N = 5;
	a = 1, b = 3, c = 4;
	memo.resize(N + 1, 0);
	cout << go(N) << endl;
}
```

```
Output

6
```

Bottom-Up Design (Tabulation)

The other way we could implement a dynamic programming program is by starting from the bottom (leaves) of the recursion tree and traversing up to the root where we obtain the solution of the original problem.

We use a term **tabulation** for this process because it is like filling up a table from the start (base case).

Example 2: Solution of Sum Probelem with Bottom-Up-Design

```
// Sum DP Bottom Up.cpp
#include <iostream>
#include <vector>

using namespace std;

int N, a, b, c;
//-------------------------------------------------
int go()
{
```

```
	vector<int> memo(N + 1, 0);
	//base case
	memo[0] = 1;

	for (int i = 1; i <= N; i++)
	{
		if (a <= i)
			memo[i] += memo[i - a];
		if (b <= i)
			memo[i] += memo[i - b];
		if (c <= i)
			memo[i] += memo[i - c];
	}

	return memo[N];
}
//-------------------------------------------------
int main()
{
	N = 5;
	a = 1, b = 3, c = 4;

	cout << go() << endl;
}
```

```
Output

6
```

The *int go()* function may also be implemented as following:

```
int go()
{
	vector<int> memo(N + 1, 0);
	//base case
	memo[0] = 1;

	for (int i = 0; i <= N; i++)
	{
		if (i + a <= N)
			memo[i + a] += memo[i];
		if (i + b <= N)
			memo[i + b] += memo[i];
		if (i + c <= N)
			memo[i + c] += memo[i];
	}
	return memo[N];
}
```

Exercise 1: Bottom-Up Design For a Sparse Memo

Re-implement the iterative *Sum* problem to skip the elements of the memo array if they remain 0. Consider the input where *N* is 10^6, *a* is 10^4, *b* is 2*10^4 and *c* is 5*10^4.

The "Wines" Problem

Imagine you have a collection of N wines placed next to each other on a shelf. For simplicity, let's number the wines from left to right as they are standing on the shelf with integers from 1 to N, respectively. The price of the i-th wine is pi (prices of different wines can be different).

Because the wines get better every year, supposing today is the year 1, on year y the price of the i-th wine will be y*pi, i.e. y-times the value that current year.

You want to sell all the wines you have, but you want to sell exactly one wine per year, starting on this year. One more constraint - on each year you are allowed to sell only either the leftmost or the rightmost wine on the shelf and you are not allowed to reorder the wines on the shelf (i.e. they must stay in the same order as they are in the beginning).

You want to find out, what is the maximum profit you can get, if you sell the wines in optimal order.

So for example, if the prices of the wines are (in the order as they are placed on the shelf, from left to right): p1=1, p2=4, p3=2, p4=3

The optimal solution would be to sell the wines in the order p1, p4, p3, p2 for a total profit 1*1 + 3*2 + 2*3 + 4*4 = 29

Analyzing The Problem

The original problem has an array of integers (prices of wines form left to right) with *N* elements in year 1. In every year we can sell the left-most or righ-most element. Thus, the problem is getting smaller in size every year by one element. Finally, there will bo no wine to sell.

When there is no more wine (subproblem with zero element), the profit will be zero. This will be our base case. We can define each subproblem with its range (index of the left-most element and index of the riht-most element) and the current year.

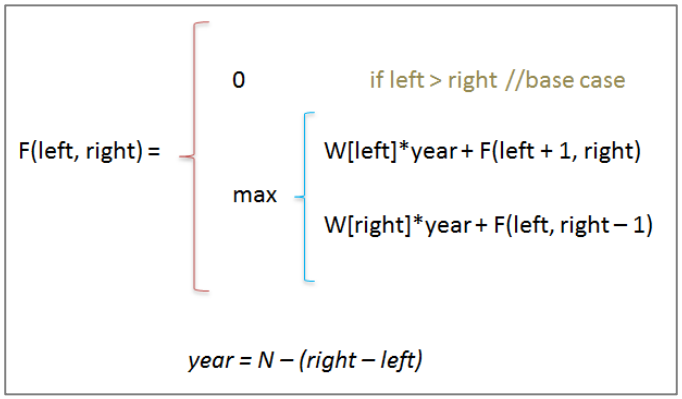
The **initial recursive relation** can be expressed as following. For the original problem *F = (1, N, 1).*

$$F(left, right, year) = \begin{cases} 0 & \text{if } left > right \text{ //base case} \\ \max \begin{cases} W[left]*year + F(left+1, right, year+1) \\ W[right]*year + F(left, right-1, year+1) \end{cases} \end{cases}$$

In the initial recurive relation, we have three parameters: *left*, *right* and *year*. As the *year* increases by 1, the range of the problem decreases by the same amount. In other words, we can calculate the year parameter using the left and right parameters for each sub-problem. The year parameter actually is redundant.

```
year = N - (right - left)
```

The refined recursive relation:



Before implementing the program, we shoud determine the memoizaton method. We need to store the best profit for a range of array for each subproblem. We can represent all possible ranges with a 2D array. Since profit is an integer, a 2D array of integer will be our memo.

Top-Down (Recursive) Implementation

```
// Wines.cpp : DP Top-Down Design
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

ifstream cin("wines.in");
ofstream cout("wines.out");

int N;
vector<int> wines;
vector<vector<int>> dp; //for memoization
//-------------------------------------
void readInput()
{
	cin >> N;
	wines.resize(N+1);
	for (int i = 1; i <= N; i++)
		cin >> wines[i];
}
//-------------------------------------
int go(int left, int right)
{
	if (left > right) //base case
		return 0;

	if (dp[left][right] > 0) //already calculated
		return dp[left][right];

	int year = N - (right - left);
	int temp1 = year * wines[left] + go(left + 1, right);
	int temp2 = year * wines[right] + go(left, right - 1);

	return dp[left][right] = max(temp1, temp2);
}
//-------------------------------------
int main()
{
	readInput();
	dp.resize(N+1, vector<int>(N+1, 0));
	cout << go(1, N) << endl;
}
```

Bottom-Up (Tabulation) Implementation

In the tabulation (table filling) implementation, we start to fill the table from the base case. The smallest possible subproblem(s) is the base case. In the recursive implementation, the size of the base case subproblem was zero. We do not have any zero-sized subproblem in the solution table. The smallest subproblem has the size 1 (left == right). Therefore, we will start to fill the table with subproblems of size 1.

dp[i][j] stores the best profit in the ange *i … j*. The elements on the main diagonal (where *i* == *j*) are the solutions of the base case problems. We will directly fill the main diagonal and complete the rest of the table in diagonal manner (year by year) starting with year 2.

`dp[i][j] = max(dp[i+1][j] + W[i]*year + dp[i][j-1] +W[j]*year, where i < j`

```
// Wines.cpp : DP Bottom-UP Design
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

ifstream cin("wines.in");
ofstream cout("wines.out");

int N;
vector<int> wines;
//-------------------------------------
void readInput()
{
	cin >> N;
	wines.resize(N + 1);
	for (int i = 1; i <= N; i++)
		cin >> wines[i];
}
//-------------------------------------
void go()
{
	vector<vector<int>> dp(N + 1, vector<int>(N + 1, 0));

	//Calculate base case
	for (int i = 1; i <= N; i++)
		dp[i][i] = wines[i];

	//Complete the rest of the table
	for (int year = 2; year <=N; year++)
		for (int i = 1; i + year - 1 <= N; i++) //i is the left end of the current range
		{
			int j = i + year - 1; //j is the right end of the current range
			int temp1 = dp[i + 1][j] + wines[i]*year;
			int temp2 = dp[i][j - 1] + wines[j]*year;
			dp[i][j] = max(temp1, temp2);
		}
	cout << dp[1][N] << endl;
}
//-------------------------------------
int main()
{
	readInput();
	go();
}
```

Sample Input

```
5
2 3 5 1 4
```

Sample Output

```
48
```

```
The dp matrix

2  8  23 28 48
0  3  13 20 36
0  0  5  11 24
0  0  0  1  9
0  0  0  0  4
```

Exercise 2: Wines with BFS Implementation

Implement the bottom-up Wines problem with BFS. Add the main diagonal values into the queue beforehand.

Time Complexity of DP

In Dynamic programming problems, **Time Complexity** is the number of unique states/subproblems * time taken per state (transition time).

Most of the time size of the demo in element is proportional with the number of subproblems. For example N^{th} Fibonacci Number program uses an array of N elements to store the partial solution and we make two recursive calls for each subproblem. So it runs in O(N) time and O(N) space. On the other hand, the Wines program occupies O(N*N) space again in a constant time we calculate each subproblem. It rund in O(N*N) time and in O(N*N) space.

Dynamic Programming, Divide and Conquer, Greedy

If any problem has the optimal substructure property but does not have overlapping subproblems, it can be solved by **divide and conquer** technique instead of dynamic programming. Divide and Conquer works by dividing the problem into sub-problems, conquer each sub-problem recursively and combine these solutions. **Merge-sort** is a divide and conquer algorithm.

Greedy approach deals with forming the solution step by step by choosing the local optimum at each step and finally reaching a global optimum.The greedy approach is suitable for problems where local optimality leads to an optimal global solution. **Dijkstra** shortest path and **Prim's** MST algorithms are two typical greedy algorithms.

Useful Links and References

https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm
https://en.wikipedia.org/wiki/Overlapping_subproblems
https://www.freecodecamp.org/news/follow-these-steps-to-solve-any-dynamic-programming-interview-problem-cc98e508cd0e/
https://www.codesdope.com/course/algorithms-dynamic-programming/
https://www.quora.com/Are-there-any-good-resources-or-tutorials-for-dynamic-programming-DP-besides-the-TopCoder-tutorial/answer/Michal-Danil%C3%A1k?share=1&srid=3OBi
https://afteracademy.com/blog/dp-vs-greedy-algorithms
https://www.quora.com/What-is-the-time-complexity-of-dynamic-programming