ALGO-101

Week 7 - Dynamic Programming

Bilgenur Çelik

ITU ACM

MONTH 2023

Topics

Topics covered at week 7:

- Greedy Approach
- Dynamic Programming Approach
  - Memoization
  - Top-Down
  - Bottom-Up
- Common DP Problems
  - Coin Problem
  - Knapsack Problem
  - Longest Increasing Subsequence Problem
  - Longest Common Substring Problem
  - Tiling Problem

Greedy Approach

* solving a problem by selecting the best available option in a given situation
* assumes that: local optimal choice $==$ global optimum

Where to use:

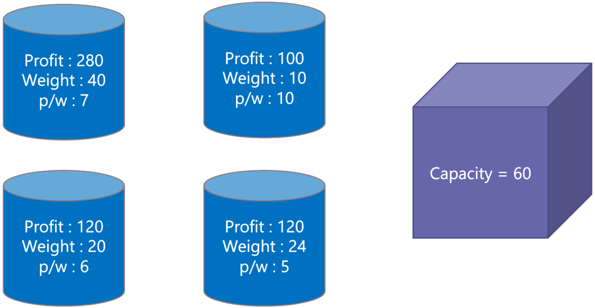
Finding the shortest path between two vertices using Dijkstra's algorithm.
Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.
Some optimization problems (faster than dp solution for those suitable questions)

Greedy Approach

ex.: FRACTIONAL KNAPSACK

With given n items (weigths and values) and the ability to choose the desired fraction of the item; fill the knapsack to reach total maximum value without exceeding the given capacity.

*Begin with max(value/weight)



Greedy Approach

ex.: INTERVAL SCHEDULING

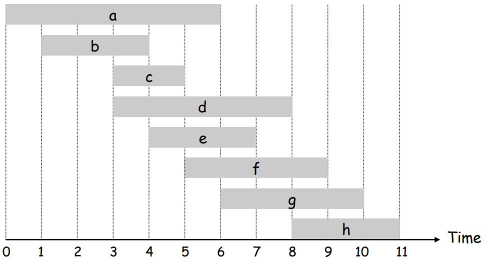
Maximize the amount of activities:
Every activity j starts at sj ends in fj, what is your algorithm?

A. [Earliest start time]
(Consider jobs in ascending order of sj.)

B. [Earliest finish time]
(Consider jobs in ascending order of fj.)

C. [Shortest interval]
(Consider jobs in ascending order of fj – sj.)

D. [Fewest conflicts]
(For each job j, count the number of conflicting jobs cj Schedule in ascending order of cj)



Greedy Approach

ex.: INTERVAL SCHEDULING

counterexample for earliest start time

counterexample for shortest interval

counterexample for fewest conflicts

Greedy Approach

ex.: CANDY

N children with integer ratings -greater or equal to zero- are standing in a line. As the school principle, you are going to distribute as minimum candies as possible with two precondition:

1. Each child must have at least one candy.
2. Children with a higher rating get more candies than their neighbors.

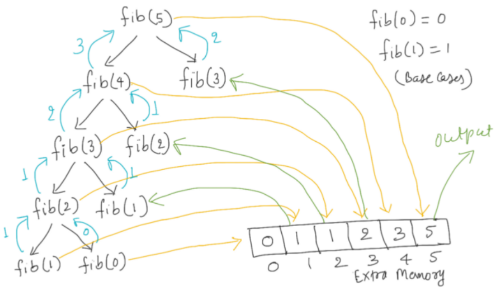
15 2 20 5 4 2 12 3

Dynamic Programming Approach

When sub-problems are dependent or repeated rather than solving each one repeatedly, solve the problem and store it in extra memory. Afterwards use the found variable if the same sub-problem appears later on. (Time-memory trade-off)
Caching $\rightarrow$ the process of storing to avoid redoing the same operations
Both top-down and bottom-up approaches of dp use extra memory to avoid resolving the overlapping sub-problems.

Top Down (Memoization)

Use recursion, but save the result of each sub-problem in an array or hash table.



Order of execution of recursive calls:

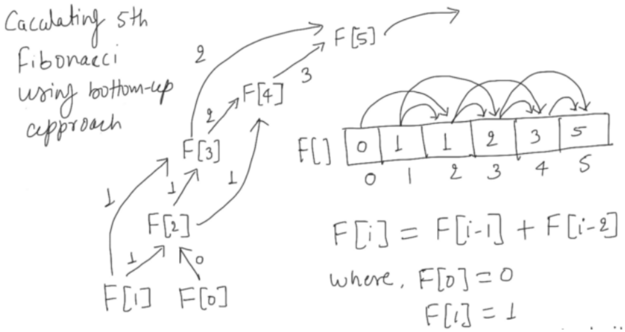
fib(n) $\rightarrow$ $fib(n-1) \rightarrow fib(n-2) ... fib(2) \rightarrow fib(1) \rightarrow fib(0)$

Order of storing the results in the table:

fib(0) and fib(1) $\rightarrow$ $fib(2) ... fib(n-2) \rightarrow fib(n-1) \rightarrow fib(n)$

Bottom Up (Tabulation)

It is iterative, starts from base cases. We solve the smallest sub-problem, than go through the complex ones with those results.



Top Down X Bottom Up

Both approaches are established to share the same algorithmic complexity.(Sometimes top-down doesn't need to recurse to all possible sub-problems, but it is neglected.)

- Top-down
  - Recursive
  - Easy to implement(short code)
  - Slower
  - Space for the recursion call stack(possibility to run out of stack space)
- Bottom-up
  - Iterative
  - Long code
  - Faster
  - Differs from question to question, but sometimes it can be optimized as time and space complexity. ex/ $O(N^2)$ to O(N) or O(N) to O(1). This optimization is easier to implement

Coin Change

Return the fewest coins that you will need to make up that sum from given units

For national currency units, the greedy approach is faster and as accurate to use. (Take Turkish lira as example)

But in a weird place using 1, 3, 4, 6; find the minimum change for 8.

Coin Change (cont'd)

{1, 3, 4, 6} -> 8

dp[0] = 0
dp[1] =1
dp[2] = 1+dp[1] = 2
dp[3] = 1+dp[2] || 1 =1
dp[4] = 1+dp[3] || 1+dp[1] || 1 = 1
dp[5] = 1+dp[4] || 1+dp[2] || 1+dp[1] = 2
dp[6] = 1+dp[5] || 1+dp[3] || 1+dp[2] || 1 = 1
dp[7] = 1+dp[6] || 1+dp[4] || 1+dp[3] || 1+dp[1] = 2
dp[8] = 1+dp[7] || 1+dp[5] || 1+dp[4] || 1+dp[2] = 2

Coin Change (cont'd)

if amount $==0$
    return 0
else if amount >0
    for every value smaller than amount
        find the minimum way to reach that state by considering **all possible coins**

Coin Change Code

```cpp
int coinChange(vector<int>& coins, int amount) {
    vector<int> cache(amount+1, amount+1);
    cache[0] = 0;

    for(int i=0; i<=amount; i++){
        for(int j=0; j<coins.size(); j++){
            if(coins[j] <= i)
                cache[i] = min(cache[i], 1 + cache[i-coins[j]]);
        }
    }

    return (cache[amount]!=amount+1) ? cache[amount] : -1;
}
```

0-1 Knapsack Problem

With given n items (weigths and values) maximize the value of the knapsack.
ex/ There are 5 items weight = 3, 4, 2, 1, 5, profit = 6, 4, 3, 2, 2 and knapsack capacity is 8.
What is the maximum value?

Profit:	Weight:		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
6	3	1									
4	4	2									
3	2	3									
2	1	4									
2	5	5									

0-1 Knapsack Problem (cont'd)

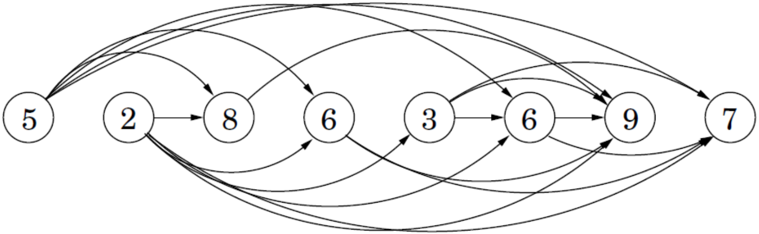
Profit:	Weight:		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
6	3	1	0	0	0	6	6	6	6	6	6
4	4	2	0	0	0	6	6	6	6	10	10
3	2	3	0	0	3	6	6	9	9	10	10
2	1	4	0	2	3	6	8	9	11	11	12
2	5	5	0	2	3	6	8	9	11	11	12

0-1 Knapsack Problem Code

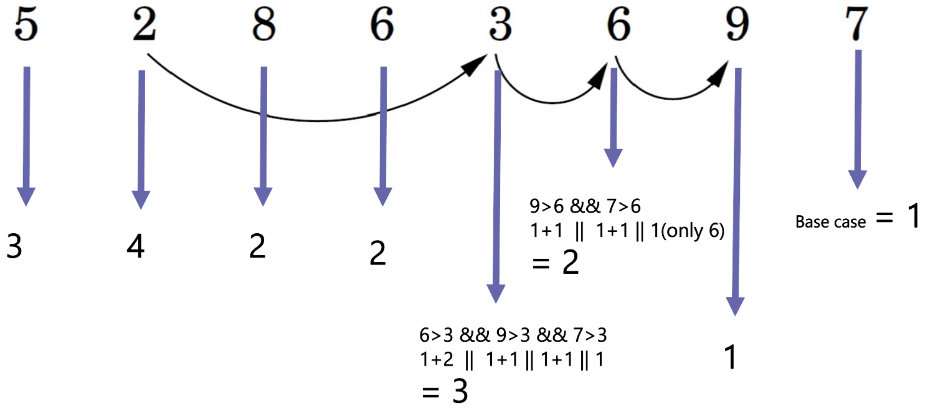
```
vector<vector<int>> dp_table(n+1, vector<int>(capacity+1, 0));

for(int item = 1; item <= n; item++){
  int w = weight[item-1], v = value[item-1];

  for(int c = 1; c <= capacity; c++){
    // if two controls can be made
    if(c >= w)                    // item not included--------item included
      dp_table[item][c]=max(dp_table[item-1][c],dp_table[item-1][c-w] +v);

    // if only one control can be made
    else
      dp_table[item][c] = dp_table[item-1][c];
  }
}
```

Longest Increasing Subsequence

Select the longest subsequence with elements sorted from lowest to highest in the given sequence.
ex.:



Longest Increasing Subsequence (cont'd)

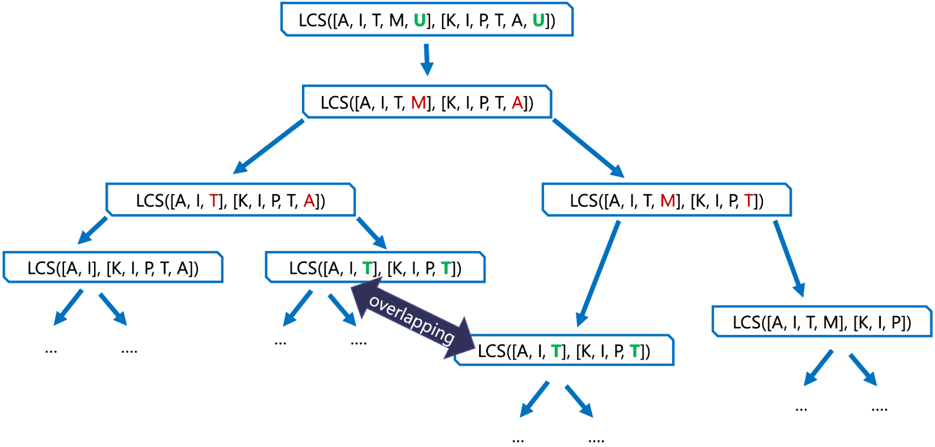


Longest Increasing Subsequence Code

```
vector<int> vec_cache(n, 1);

for(int i=n-1; i>=0; i--){
    for(int j=i+1; j<n; j++)
        // if increasing:
        if(sequence[j] > sequence[i])
            vec_cache[i] = max(vec_cache[i], 1 + vec_cache[j]);
}
for(int i : vec_cache){
    lis_len = max(lis_len, i);
}
```

Longest Common Subsequence

From given two strings, find the length of the longest common substring.



Longest Common Subsequence (cont'd)

Case 1:
Characters in the sequences match.
dplcs[i][j] $= 1 +$ dplcs[i+1][j+1];

Case 2:
Characters in the sequences do not match.
dplcs[i][j] $=$ max(dplcs[i+1][j], dplcs[i][j+1]);

	K	I	P	T	A	U	
A	3	3	2	2	2	1	0
I	3	3	2	2	1	1	0
T	2	2	2	2	1	1	0
M	1	1	1	1	1	1	0
U	1	1	1	1	1	1	0
	0	0	0	0	0	0	0
ITU							

Figure: lsc table

Longest Common Subsequence Code

```
int main(){
    string str1, str2;
    cin >> str1 >> str2;
    int len1 = str1.length(), len2 = str2.length();

    vector<vector<int>> dp_table(len1+1, vector<int>(len2+1));

    for (int i = len1-1; i>=0; i--){
        for (int j = len2-1; j>=0; j--){
            if (str1[i] == str2[j]){
                dp_table[i][j] = 1 + dp_table[i+1][j+1];
            }
            else {
                dp_table[i][j] = max(dp_table[i+1][j], dp_table[i][j+1]);
            }
        }
    }
```

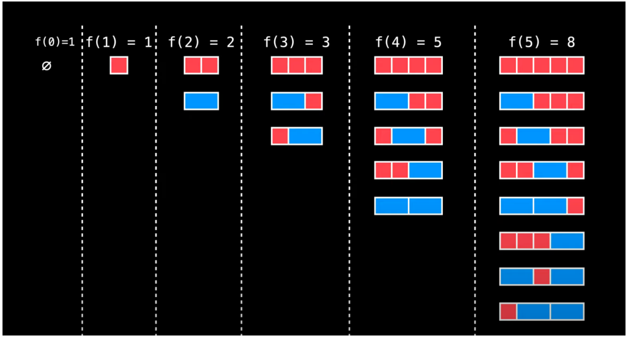
Longest Common Subsequence Code (cont'd)

```
    string answer = "";
    int i = 0, j = 0;
    while (i < len1 && j < len2){
        if (str1[i] == str2[j]){
            answer += str1[i];
            i++;
            j++;
        }
        else {
            if(dp_table[i+1][j] >= dp_table[i][j+1])
                i++; // down
            else
                j++; // right
        }
    }
    cout << answer << endl;
}
```

Tiling Problems

A type of dp problems, involving counting the possible ways of tilings on a grid.

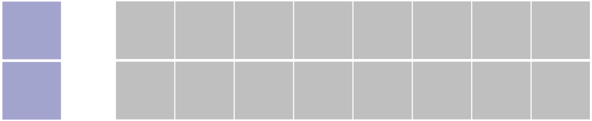
1: How many ways to tile 1*n grid with 1*1 and 1*2 tiles?



https://projecteuler.net/problem=117 Generalize this to 1*n and any number of 1*k tiles for fun.

Tiling Problems (cont'd)

There always is a pattern.
ex/ Tile a 2*n grid with 2*1 tiles.



$$count(n) = \begin{cases} n & \text{if } n = 1, 2 \\ count(n-1) + count(n-2) & \end{cases} \tag{1}$$

Tiling Problems (cont'd)

How many ways to tile m*n grid with 1*n tiles?

$$count(n) = \begin{cases} 1 & \text{if } 1 <= n < m \\ 2 & \text{if } n = m \\ count(n-1) + count(n-m) & \text{if } m < n \end{cases} \tag{2}$$

Tiling Problems (cont'd)

What about the harder tiling problems?

ex/ Tile a 3*n grid with 2*1 tiles.
`https://algoleague.com/problem/cafers_livingroom/detail7`
Pure math of tiling: `https://arxiv.org/ftp/arxiv/papers/2108/2108.08909.pdf`

Some Real-life Applications of Dynamic Programming

- sequence alignment
- document diffing algorithms
- plagiarism detection
- document distance algorithms
- speech recognition
- image processing
- economy