ALGO-101

Week 7 - Dynamic Programming

Bilgenur Çelik

ITU ACM

November 2022

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 Subsequence 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:

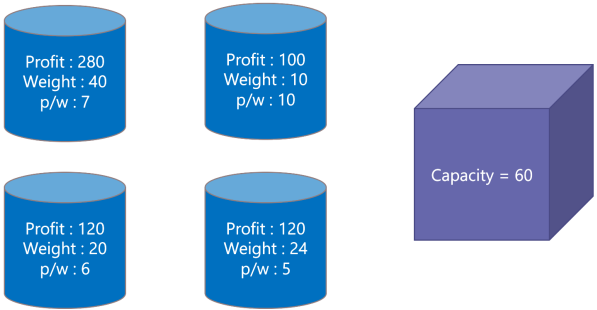
Finding the shortest path between two vertices using Dijkstra's algorithm.
Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.
Some optimization problems (faster then dp solution for those suitable questions)

Fractional Knapsack

With given n items (weigths and values) and the ability to choose the desired fraction of the item.

Goal: Fill the knapsack to reach total maximum value without exceeding the given capacity.

*Begin with max(value/weight)



Fractional Knapsack Code

```
static bool cmp(pair<double,double> &a, pair<double,double> &b) {
  return a.first/a.second > b.first/b.second;
}

int main(){
  //given value and weight;
  vector<pair<double,double>> steal {{280, 40}, {100, 10}, {120, 20}, {120, 24}};
  // max weight the knapsack can carry;
  int w = 50;

  sort(steal.begin(), steal.end(), cmp);

  double total_price = 0;
  for(auto item : steal){
    // if item can be selected as a whole
    if(w - item.second > 0){
      w -= item.second;
      total_price += item.first;
    }
    // if item can be selected as some part
    else if(w>0){
      total_price += w*(item.first/item.second);
      w = 0;
    }
    else break;
  }
  cout << total_price << endl;
```

Tasks and Deadlines

We are given n tasks with the durations and deadlines and our task is to choose an order to perform the tasks.
For each task, we earn d - x points where d is the task's deadline and x is the moment when we finish the task.
Goal: What is the largest possible total score we can obtain?

$$\{task, duration, deadline\}$$

$$\{A, 4, 2\}, \{B, 3, 5\}, \{C, 2, 7\}, \{D, 4, 5\}$$

In this case, an optimal schedule for the tasks is C, B, A, D. In this solution, C yields 5 points, B yields 0 points, A yields -7 points and D yields -8 points, so the total score is -10.

Correct greedy strategy is to simply perform the tasks **sorted by their durations in increasing order.**

Tasks and Deadlines (cont'd)

How to achieve the algorithm:
Let's have 3 tasks: $g_1 = t_1,\ d_1\ g_2 = t_2,\ d_2\ g_3 = t_3,\ d_3$

to achieve the score: (s is the initial time, 0 in the beginning)
$= d_1$ - (s + t_1) + d_2 - (s + t_1 + t_2) + d_3 - (s + t_1 + t_2 + t_3)

As we can see in the equation, we have to include all deadlines once no matter the order.
If we take the 3-time appeared t1 small as possible the score is smaller.

Dynamic Programming Approach

In divide and conquer $\rightarrow$ *non overlapping subproblems*.
In greedy and DP $\rightarrow$ *overlapping subproblems*

Greedy $\rightarrow$ *local optimum* $=$ *global optimum*

DP $\rightarrow$ *don't have this characteristic*

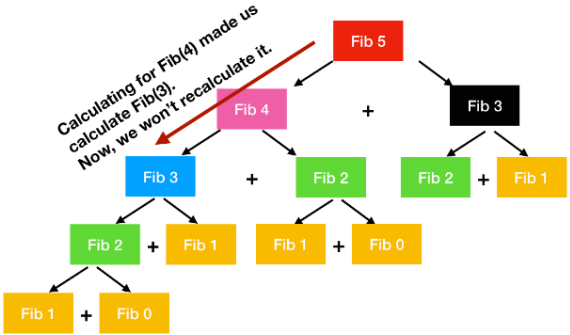
$+$ Top-Down

$+$ Bottom-Up

The idea is to store the results of sub-problems in extra memory in order not to recompute them when needed later on. (Caching) This simple optimization reduces time complexities from exponential to polynomial.

Top Down (Memoization)

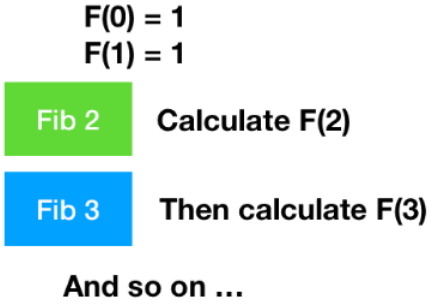
Use recursion, but **save the result of each sub-problem** in an array or hash table.



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 due to recursive calls and returns
  - Space for the recursion call stack(possibility to run out of stack space)
- Bottom-up
  - Iterative
  - Long code
  - Faster with direct access to elements in table
  - 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

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

(Greedy approach: Subtract the larger money unit each step. Return steps.) For national currency units, the greedy approach is faster and as accurate to use. (Take the Turkish lira as an example: to get 65: 50+10+5, 3 is correct and sufficient.)
But in a weird place using 1, 3, 4, 6; find the minimum change for 8.

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**
// Time comp.:O(coins.size()*amount) Space comp.:O(amount)

Coin Change (cont'd)

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

dp[0] = 0
dp[1] = 1+dp[0] = 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 Code

```
1  int coinChange(vector<int>& coins, int amount) {
2      vector<int> cache(amount+1, amount+1);
3      cache[0] = 0;
4
5      for(int i=0; i<=amount; i++){
6          for(int j=0; j<coins.size(); j++){
7              if(coins[j] <= i)
8                  cache[i] = min(cache[i], 1 + cache[i-coins[j]]);
9          }
10     }
11
12     return (cache[amount]!=amount+1) ? cache[amount] : -1;
13 }
```

0-1 Knapsack Problem

A knapsack has limited weight capacity.
Goal: 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.

Base Cases:
· If the capacity of the sack is 0 kg , then no item can be added to the knapsack.
· If zero items are available for filling the knapsack, then none can be put into the knapsack

If the available capacity of the knapsack is greater than the weight of the chosen item to be put, check what would give a higher value to the knapsack (A knapsack not containing the chosen item: Thus the number of items now available is reduced by 1. *OR* A knapsack containing the chosen item: Thus the capacity of the knapsack is reduced by the weight of the item, but the value of the knapsack is increased by the value of the chosen item.)

0-1 Knapsack Problem

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

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									
3	2	3									
2	1	4									
2	5	5									

0-1 Knapsack Problem

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									
2	1	4									
2	5	5									

0-1 Knapsack Problem

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									
2	5	5									

0-1 Knapsack Problem

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									

0-1 Knapsack Problem

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

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

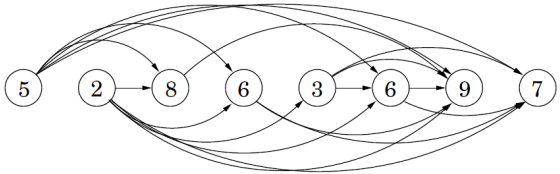
0-1 Knapsack Problem Code

```
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];
  }
}
```

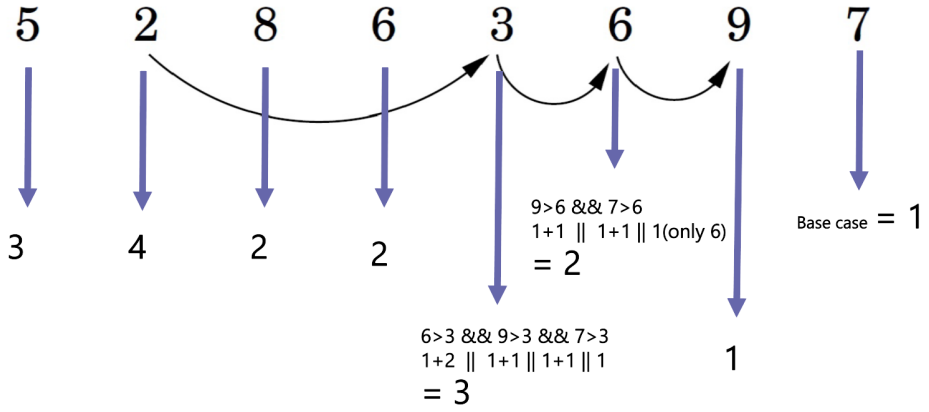
Longest Increasing Subsequence

Goal: Select the longest subsequence with elements sorted from lowest to highest in the given sequence.
(*Naive approach:* We could look at every permutation : $O(2^n)$)
ex.: arr[] = {3, 10, 2, 1, 20} LIS = 3 ({3, 10, 20})

From last element to first one(arr[i])
  scan from that element+1 to last one(arr[j])
    if greater(arr[j] > arr[i])
      dp[i] = max(dp[i], 1+dp[j])



Longest Increasing Subsequence



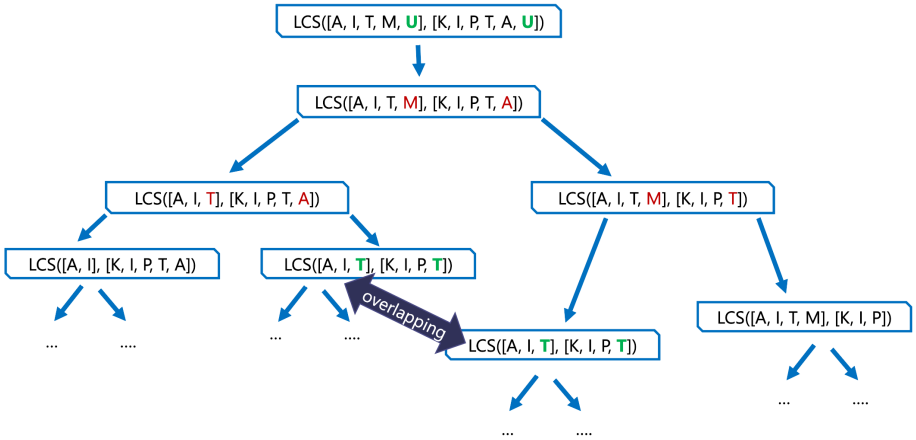
Longest Increasing Subsequence Code

$O(N^2)$

```
1   // n = sequence.size()
2   vector<int> vec_cache(n, 1);
3
4   for(int i=n-1; i>=0; i--){
5       for(int j=i+1; j<n; j++)
6           // if increasing:
7           if(sequence[j] > sequence[i])
8               vec_cache[i] = max(vec_cache[i], 1 + vec_cache[j]);
9   }
10
11  int lis_len = 0;
12  for(int i : vec_cache){
13      lis_len = max(lis_len, i);
14  }
15  cout << lis_len << endl;
```

Longest Common Subsequence

Goal: From given two strings, find the length of the longest common subsequence.
LCS for input sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.



Longest Common Subsequence (cont'd)

Base Case:
Elements representing empty list are 0 (last row and column)

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]);

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]);
            }
        }
    }
```

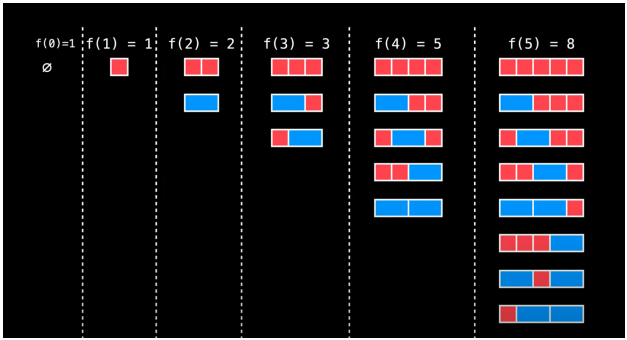
Longest Common Subsequence Code (cont'd)

```
    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.

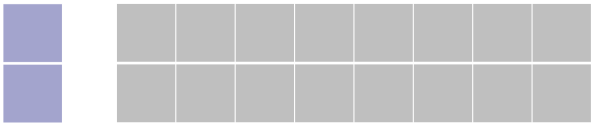
1: How many ways to tile $1 \times n$ grid with 1×1 and 1×2 tiles?



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

Tiling Problems (cont'd)

There always is a pattern.
ex/ Tile a $2 \times n$ grid with 2×1 tiles.



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

Tiling Problems (cont'd)

How many ways to tile $m \times n$ grid with $1 \times n$ tiles?

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

Tiling Problems (cont'd)

What about the harder tiling problems?

ex/ **Cafers Livingroom:** Tile a $3 \times n$ grid with 2×1 tiles.

A link to pure math of tiling

Questions:

- Uncrossed Lines
- Two City Scheduling
- Minimum Path Sum
- Candy
- Jump Game
- Shortest Common Subsequence
- AtCoder - Educational DP Contest

Three important articles!

- ABCs of Greedy
- Dynamic Programming Patterns
- Thief with a knapsack, a series of crimes

References:

https://www.algotree.org/algorithms/dynamic_programming/

https://leetcode.com/discuss/general-discussion/458695/Dynamic-Programming-Patterns

https://leetcode.com/discuss/general-discussion/1061059/abcs-of-greedy

https://www.cs.princeton.edu/~wayne/kleinberg-tardos/ https://github.com/inzva/Algorithm-Program/blob/master/bundles/05-dp-1/05_DP1.pdf

https://www.geeksforgeeks.org/tabulation-vs-memoization/

Some Real-life Applications of Dynamic Programming

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