Lecture 16 - Asymptotic Notation & DP

CSE 2202 - Design & Analysis of Algorithms

Contents

We use **Asymptotic Notations** to describe the **performance** of an algorithm

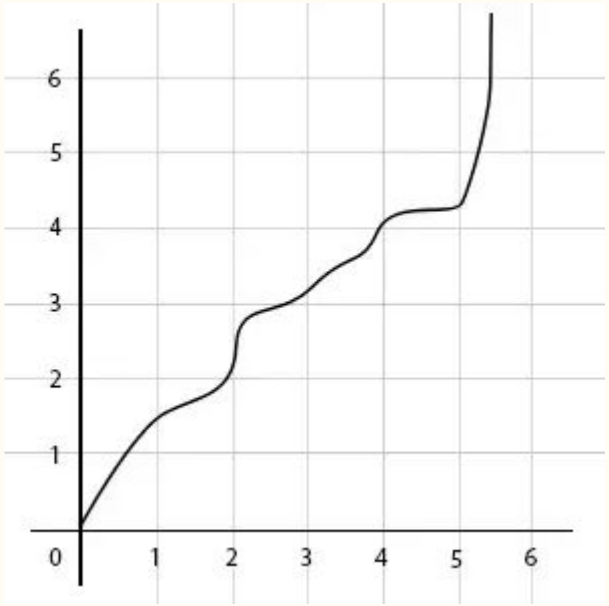
Asymptotic Notations

For example, we wrote an algorithm, and when we plot the graph of its runtime, it looks like this:

By looking at the graph, we cannot say if the algorithm's time complexity is linear or quadratic.

If we try to calculate exact mathematical notation of the algorithm, we will get some weird functions; studying and explaining this notation is very difficult.

This filter of "dropping all factors" and of "keeping the largest growing term" as described above is what we call asymptotic behavior.



Asymptotic Notations

O(1) – Accessing an element in an array by index.

O(log n) – Binary Search on a sorted array.

O(n) – Find the maximum element in an array.

O(n log n) – Merge Sort

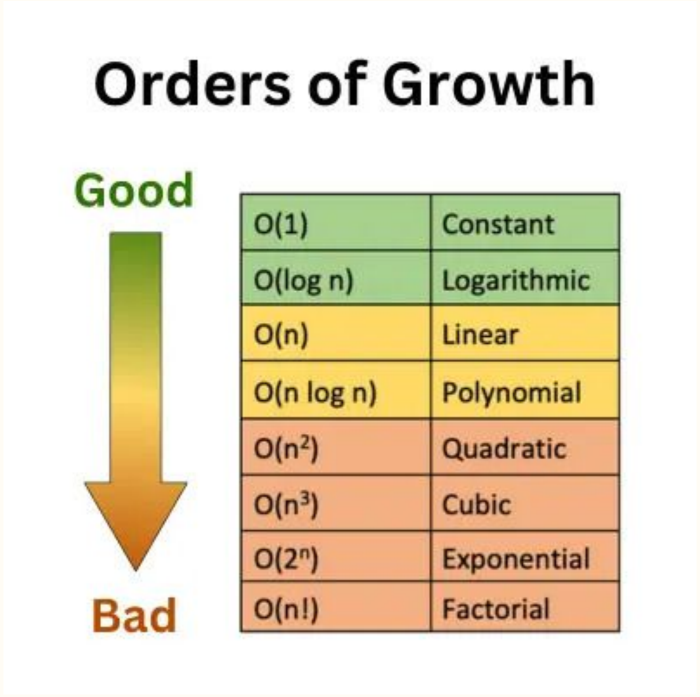
O(n ^ 2) – Bubble Sort

O(2 ^ n) – Solving the Tower of Hanoi problem

O(n!) – Generating permutations of a string.

O(m+n) - Merging two sorted arrays.

O(mn) - Matrix multiplication.



Orders of Growth

O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Polynomial
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
O(n!)	Factorial

Asymptotic Notations

Big O (O): Upper bound (asymptotic maximum rate).
Big Theta (Θ): Tight bound (exact asymptotic rate).
Big Omega (Ω): Lower bound (asymptotic minimum rate).

If f(n) is the runtime of an algorithm and g(n) is a growth function:
Big O: f(n) never grows faster than g(n) beyond a certain point.
Big Theta: f(n) grows at the same rate as g(n) beyond a certain point.
Big Omega: f(n) never grows slower than g(n) beyond a certain point.

https://discrete.gr/complexity/

O (Big O) Notation

The O-notation provides an upper bound on the asymptotic behavior of a function, indicating that the function does not grow faster than a certain rate determined by its highest-order term.

Given two functions, f(n) and g(n), we say that f(n) is O(g(n)) if there exist constants c and n_0 such that:

$0 \le f(n) \le c*g(n)$ for all $n \ge n_0$

Before proving this summation, let's clarify these notations and attributes used in the above summation.

$f(n) \in O(g(n))$ = f of n is Big-O of g of n.

Notations

f(n) - This is the function representing the actual complexity of an algorithm, typically expressed as a function of the size of the input n.

For example, for the algorithm that checks whether a number is prime, f(n) could be the number of divisions the algorithm has to perform.

g(n) - This is the function that we are using to compare with f(n). It's a simplification of the actual complexity that gives us a general idea of how the complexity grows with respect to the input size n.

For example, we might say that the complexity of sorting n items is O(n log n), where g(n) is n log n.

Notations - Continued

n_0

This is a constant representing the point from where the given condition (whether it's for Big O, Big Omega, or Big Theta) holds true.

This is typically used because, for smaller n, certain irregular behaviors might be observed that don't represent the general trend of the function.

For example, Car A has a speed of 50 mph and takes 6 hours to cross a distance of 300 miles. On the other hand, Car B has a speed of 60 mph and takes 5 hours to cover the same 300-mile distance. There is only one hour difference, but when the distance is 30,000 miles, Car A will take 600 hours, and Car B will take 500 hours. See how the time change when input grows.

Notations - Continued

c

This is a positive constant multiplier. Depending on the context (Big O, Big Omega, Big Theta), we use this to bound the function f(n) by multiplying it with g(n).

For Big O, we're saying f(n) is no more than c times g(n) for sufficiently large n.

For Big Omega, we're saying f(n) is at least c times g(n) for sufficiently large n.

In the case of Big Theta, we use two constants, c1 and c2, to say f(n) is bounded by c1 times g(n) and c2 times g(n) for sufficiently large n.

Proving the summation!

$0 \le f(n) \le c*g(n)$

$0 \le 3(n^2) + 2n + 1 \le 6(n^2)$
$0 \le 3(n^2) + 2n + 1 \le 3(n^2) + 3(n^2)$ (since $3(n^2)$ is the largest term)
$0 \le 3(n^2) + 2n + 1 \le 6(n^2)$

Since the inequality holds for all $n \ge 1$, we have successfully proven that f(n) is O(n^2).

Ω(Omega) Notation

The Ω notation represents a **lower bound** on the asymptotic behavior of a function, indicating that the function grows at least as fast as a certain rate determined by its highest-order term.

For example, consider the function 3(n^2) + 2n + 1. Since the highest-order term in this function is 3(n^2), it grows at least as fast as n^2.

Therefore, we express its growth rate as Ω(n^2). Furthermore, this function is also Ω (n). We can generally express it as Ω(n^c) for any constant c <= 3.

Ω(Omega) Notation

Given two functions, f(n) and g(n), we say that "f(n) is Ω(g(n))" if there exist constants c and n_0 such that:

$0 \leq c*g(n) \leq f(n)$ for all $n \geq n_0$

Let's use the same function $f(n) = 3n^2 + 2n + 1$ and claim that f(n) is $\Omega(n^2)$.

Choosing c = 1 and n_0 = 1, we need to show that for all n greater than or equal to 1, the inequality holds:

$0 \leq c*g(n) \leq f(n)$

$0 \leq n^2 \leq 3n^2 + 2n + 1$

$0 \leq n^2 \leq 3n^2 + 3n^2$ (since $3n^2$ is the largest term)

$0 \leq n^2 \leq 6n^2$

Θ(Theta) Notation

The Theta (Θ)-notation represents a tight bound on the asymptotic behavior of a function. It specifies that a function grows exactly at a certain rate based on its highest-order term.

In other words, Theta-notation captures the growth rate of a function within a constant factor from above and below.

It's important to note that these constant factors may not be equal. If a function is both O(f(n)) and Ω(f(n)) for a certain function f(n), it can be concluded that the function is Θ (f(n)).

Θ(Theta) Notation

For example, let's consider the function 3(n^2) + 2n + 1. Since this function is both O(n^2) and Ω(n^2), it can also be denoted as Θ(n^2). This means that the growth rate of the function precisely matches the growth rate of n^2 within a constant factor from above and below.

Given two functions, f(n) and g(n), we say that "f(n) is Θ(g(n))" if there exist constants c_1, c_2, and n_0 such that:

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$

We previously chose $c_1 = 1$, $c_2 = 6$, and $n_0 = 1$. Now, we need to show that for all n greater than or equal to 1, the inequality holds:

Θ(Theta) Notation

$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

0 ≤ n^2 ≤ 3n^2 + 2n + 1 ≤ 6n^2
0 ≤ n^2 ≤ 3n^2 + 2n + 1 ≤ 3n^2 + 3n^2 (since 3n^2 is the largest term)

Since the inequality holds for all n ≥ 1, we have successfully proven that f(n) = 3n^2 + 2n + 1 is Θ(n^2) using the chosen constants c1 = 1, c2 = 6, and n_0 = 1.

This shows that the function f(n) has an upper bound and a lower bound that are multiples of n^2, confirming that it grows at a similar rate to n^2 within a constant factor for all n greater than or equal to n_0. Hence, f(n) is Θ(n^2).

Summary Using Driving Analogy

Think of analyzing your driving speed:

- Big-O: “At most, I’ll go 60 mph.” (Upper bound)
  Big-O is like saying, “This function won’t grow faster than this.”

- Big-Theta: “I usually drive at exactly 50 mph.” (Exact rate)
  Big-Theta is like saying, “This function grows exactly like this.”

- Big-Omega: “At least, I’ll drive 30 mph.” (Lower bound)
  Big-Omega is like saying, “This function grows no slower than this.”

This is how we describe how fast an algorithm grows (in terms of the time or steps it takes) as the input size gets really large!

More Resources on Asymptotic Notations

Bangla Blog - https://www.shafaetsplanet.com/?p=1313

Difference between Big-O and Little-O Notation -
https://stackoverflow.com/questions/1364444/difference-between-big-o-and-little-o-notation

Understanding Big O -
https://dev.to/princem/big-o-notation-a-simple-explanation-with-examples-1egh

Theory -
https://www.stat.rice.edu/~dobelman/notes_papers/math/big_O.little_o.pdf
https://pages.cs.wisc.edu/~vernon/cs367/notes/3.COMPLEXITY.html
https://discrete.gr/complexity/

Non-optimal substructure

Dynamic programming (DP) is highly effective for problems where the **optimal substructure** property holds—that is, the optimal solution to a problem can be constructed from optimal solutions to its subproblems.

However, there are problems where this property does not hold, making DP unsuitable.

Example Problem: The Longest Simple Path in a Graph

Non-optimal substructure

Problem Description
Given a directed graph $G = (V, E)$ with weights on the edges, find the longest simple path (a path with no repeated vertices) between two vertices s and t.

Why DP Fails
To apply dynamic programming, we would need to solve subproblems such as "What is the longest path from s to a vertex u?" and combine these subproblems optimally. However, the longest simple path problem lacks the optimal substructure property because:

- A solution to a subproblem might not lead to the overall optimal solution due to constraints on visiting nodes only once (the "simple path" constraint).
- Revisiting a node in a path might result in a longer path, but doing so would violate the "simple" constraint, complicating the reuse of solutions to subproblems.

Non-optimal substructure

Vertices: A, B, C, D
Edges:
A → B (weight 2)
B → C (weight 3)
C → D (weight 4)
A → D (weight 7)

If you decompose the problem into subproblems, the longest path from A to B is 2, and the longest path from A → C (via B) is 2 + 3 = 5 and A→D is 9 (via B and C).

Now, let's add another edge D → C (weight 4), then calculate A → C
If you try to construct the longest path from these subproblems, you might end up reusing A → B → C → D , which contradicts the "simple path" constraint, and the solution no longer reflects the constraints.

Non-optimal substructure

Explanation

In this problem, the constraint that the path must be simple breaks the ability to combine subproblems optimally. The lack of the optimal substructure property means dynamic programming algorithms cannot guarantee a correct solution.

How It Is Solved Instead

The longest simple path problem is solved using techniques like backtracking or exhaustive search. It is NP-hard, so efficient exact solutions are generally infeasible for large inputs. Approximation or heuristics may be used in practice.

Non-optimal substructure

Graph Coloring
Problem: Assign colors to the vertices of a graph such that no two adjacent vertices have the same color, using the minimum number of colors.
Why DP Fails: The coloring of one vertex affects the options available for adjacent vertices, leading to dependencies that cannot be represented by independent subproblems.

Knapsack with Conflicting Items
Problem: In the standard knapsack problem, items are chosen to maximize value within a weight limit. In the conflicting items variant, certain pairs of items cannot be chosen together.

Why DP Fails: The conflict constraints create dependencies between item choices, so the solution to a subproblem (e.g., choosing items for a smaller capacity) may not be valid if a conflicting item is added later.

Memoization vs Tabulation

```
// Initialize the memoization table
long long memo[100];

// Initialize memoization table with default values
void initializeMemo() {
   for (int i = 0; i < 100; i++) {
      memo[i] = -1;
   }
}

// Calculate Fibonacci number using memoization
long long fib(int n) {
   if (memo[n] != -1) {
      return memo[n];
   }

   if (n <= 1) {
      return memo[n] = n;
   }

   return memo[n] = fib(n - 1) + fib(n - 2);
}
```

```
long long dp[100];

void initializeDP() {
   dp[0] = 0;
   dp[1] = 1;
}

void fibTabulation(int n) {
   for (int i = 2; i <= n; i++) {
      dp[i] = dp[i - 1] + dp[i - 2];
   }
}
```

https://dev.to/ankursheel/dynamic-programming-series---introduction

Longest Common Subsequence

The Longest Common Subsequence (LCS) problem is a classic dynamic programming problem that involves finding the longest subsequence that is common to two given strings.

A subsequence of a string is a sequence of characters that appears in the same order in the string, but not necessarily consecutively.

Uses the formula:

- If characters match: dp[i][j] = dp[i-1][j-1] + 1
- If characters don’t match: dp[i][j] = max(dp[i-1][j], dp[i][j-1]) .

Longest Common Subsequence

```
function findLCSLength(str1, str2):
    m = length(str1)
    n = length(str2)

    // Create a 2D array dp of size **(m+1) x (n+1) [Note: RxC]**
    dp = array of size (m+1, n+1) filled with 0

    for i from 1 to m:
        for j from 1 to n:
            if str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1 // Characters match
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]) // Take the best of ignoring a character

    return dp[m][n] // The length of the LCS
```

Knapsack Problem

The Knapsack problem is a classic optimization problem that involves finding the optimal subset of items to pack into a knapsack with a finite capacity, so as to maximize the value of the items packed.

Given:

- n items, each with a weight $w[i]$ and value $v[i]$,
- A maximum capacity W for the knapsack.

Objective:
Maximize the total value of items included in the knapsack without exceeding the weight capacity W .

Knapsack Example Problem

Take a scenario where you're given a backpack that can carry about 100 kg of items and you are given six items to sell with price in dollars 120, 200, 150, 350, 100, 90 and weights for each item in kg as 15, 20, 40, 50, 20, 10 respectively.

```
values = [120, 200, 150, 350, 100, 90]
weights = [15, 20, 40, 50, 20, 10]
capacity = 100
n = len(values)
```

https://dev.to/freddthink/dynamic-programming-0-1-knapsack-problem-22d5

Knapsack Problem

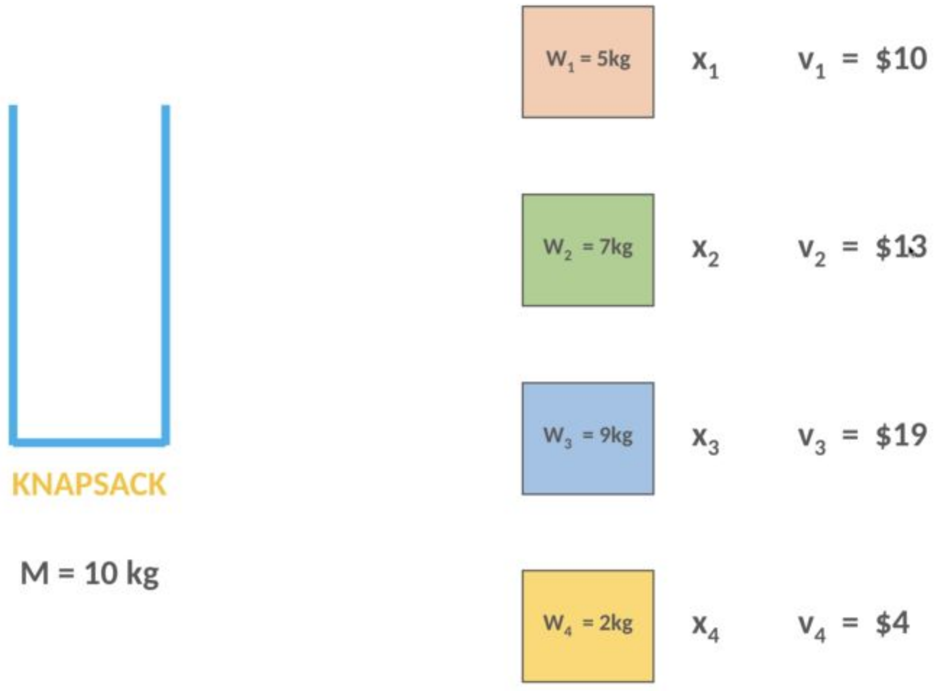
```
function knapsack(values, weights, W):
    n = length(values)

    dp = array of size (n+1, W+1) filled with 0

    for i from 1 to n:
        for w from 0 to W:
            if weights[i-1] <= w:
                // Item can be included: take the max of including or excluding it
                dp[i][w] = max(dp[i-1][w], dp[i-1][w - weights[i-1]] + values[i-1])
            else:
                // Item cannot be included: exclude it
                dp[i][w] = dp[i-1][w]

    return dp[n][W] // Maximum value that can be achieved
```

Knapsack Problem



Knapsack Problem

x_i	whether we take item **i** or not (**1** or **0** accordingly)
v_i	value of the **i**-th item
w_i	weight of the **i**-th item
M	maximum capacity of knapsack

---

maximize $\sum_{i=1}^{N} v_i * x_i$ **subject to** $\sum_{i=1}^{N} w_i * x_i \leq M$

Knapsack Problem

- we have to define subproblems: we have **N** items so we have to make **N** decisions whether to take the item with given index or not
- subproblems: the solution considering every possible combination of remaining items and remaining weight
- **S[i][w]** the solution to the subproblem corresponding to the first **i** items and available weight **w**
- **S[i][w]** is the maximum cost of items that fit inside a knapsack of size (weight) **w**, choosing from the first **i** items
- we have to decide whether to take the item or not

0-1 Knapsack Problem

If we consider all subsets of the items – there may be **2** cases:

1.) the given item is **included** in the solution (optimal subset)
2.) the given item is **not included**

So the maximum value (solution) can be reduced to smaller and smaller subproblems – and these subproblems overlap

1.) the **i**-th item is not included which means that the max value is obtained by the previous **N-1** items (and M total weights)
2.) the **i**-th item is included – max value is $\mathbf{v_i}$ plus the values obtained by the previous **N-1** items (and $\mathbf{M\text{-}w_i}$ total weights)

0-1 Knapsack Problem

S[i][w]	=	Math.max(	S[i-1][w]	;	v_i + S[i-1][w-w_i]	)
the maximum profit that fit inside a knapsack of weight **w**, choosing from the first **i** items			*do not take* ***i****-th item*		*we take* ***i****-th item*	

- we have to use this **S[][]** two-dimensional array (list)
- we are only considering **S[i-1][w-w_i]** if it can fit **$w > w_i$**
- if there is not room for it: the answer is just **S[i-1][w]** !!!

0-1 Knapsack Problem

N = 3 items	M = 5kg capacity of knapsack	
item #1	w_1 = 4kg	v_1 = \$10
item #2	w_2 = 2kg	v_2 = \$4
item #3	w_3 = 3kg	v_3 = \$7

		0	1	2	3	4	5
no items	**0**						
[item #1]	**1**						
[item #1, item #2]	**2**						
all items	**3**						

weights [kg]

S[i][w] = Math.max(S[i-1][w] ; v_i + S[i-1][w-w_i])

0-1 Knapsack Problem

N = 3 items	M = 5kg capacity of knapsack	
item #1	w_1 = 4kg	v_1 = \$10
item #2	w_2 = 2kg	v_2 = \$4
item #3	w_3 = 3kg	v_3 = \$7

		0	1	2	3	4	5
no items	0	0	0	0	0	0	0
[item #1]	1	0	0	0	0	10	10
[item #1, item #2]	2	0	0	4	4	10	10
all items	3	0	0	4	7	10	\$11

weights [kg]

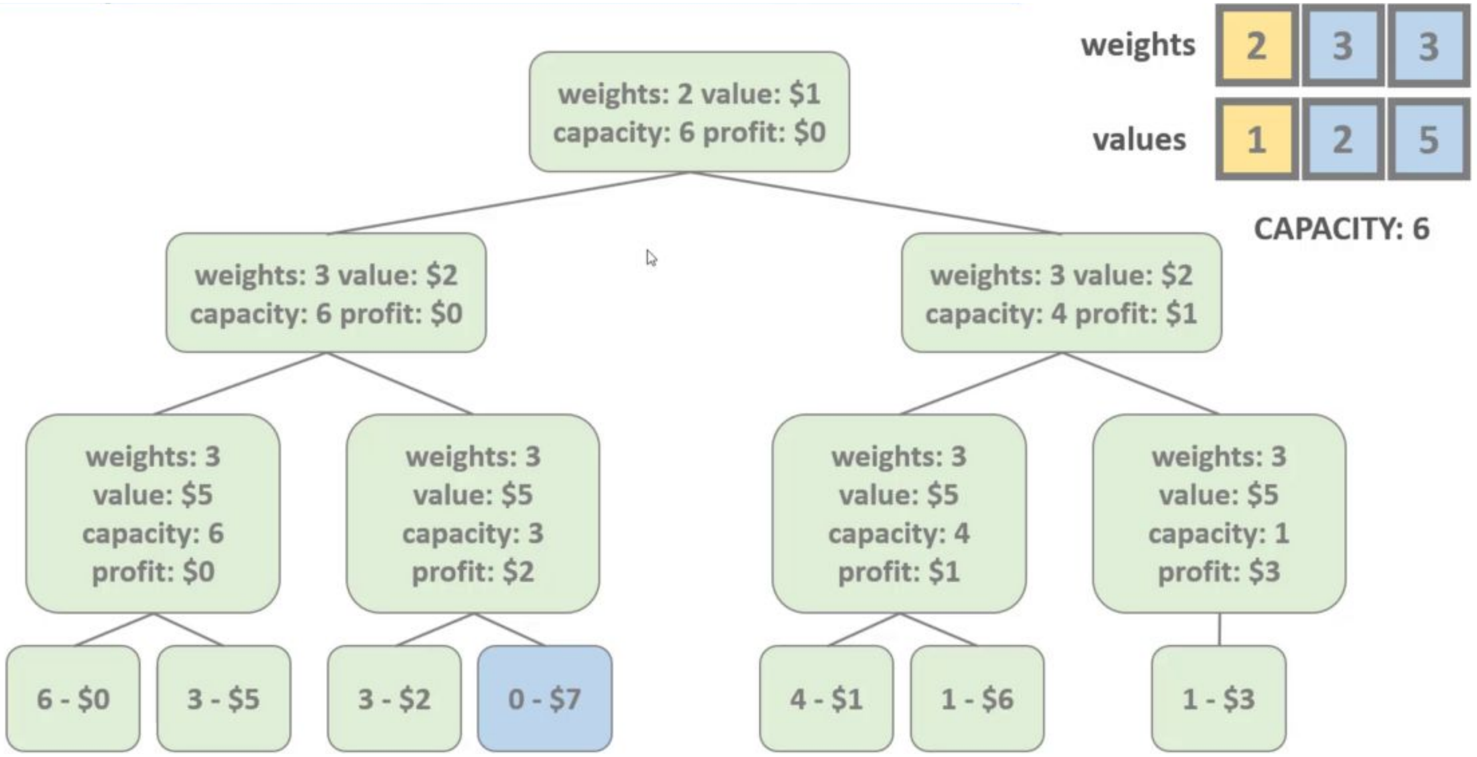
we know that the maximum profit is ***\$11*** *but how*
wo achieve this result – what items to include?
we have to start with the result and has to check the rows above
with the decremented weights accordingly

Knapsack with Recursion

```
knapsack(m, weights, values, n)
    if m==0 or n==0
        return 0

    if weights[n-1] > m
        return knapsack(m, weights, values, n-1)
    else:
        n_include = values[n-1] + knapsack(m-weights[n-1], weights, values, n-1)
        n_exclude = knapsack(m, weights, values, n-1)
        return max(n_include, n_exclude)
```

Knapsack with Recursion


weights
2
3
3
values
1
2
5
CAPACITY: 6
weights: 2 value: $1 capacity: 6 profit: $0
weights: 3 value: $2 capacity: 6 profit: $0
weights: 3 value: $2 capacity: 4 profit: $1
weights: 3 value: $5 capacity: 6 profit: $0
weights: 3 value: $5 capacity: 3 profit: $2
weights: 3 value: $5 capacity: 4 profit: $1
weights: 3 value: $5 capacity: 1 profit: $3
6 - $0
3 - $5
3 - $2
0 - $7
4 - $1
1 - $6
1 - $3

More DP Problems

Edit Distance

Maximum Subarray

Coin Change

Matrix Chain Multiplication

Longest Increasing Subsequence

Traveling Salesman Problem

0-1 Integer Programming

Perfect Squares

Longest Palindromic Substring

Maximum Product Subarray

Challenge from last class!

"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 ith wine is p_i. (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 $y*p_i$, 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?"

https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/

Questions?

More Resources

https://dev.to/nikolaotasevic/dynamic-programming--7-steps-to-solve-any-dp-interview-problem-3870

https://leetcode.com/problems/min-cost-climbing-stairs/

https://leetcode.com/problems/fibonacci-number

https://leetcode.com/problems/n-th-tribonacci-number/

https://leetcode.com/problems/pascals-triangle-ii/

https://leetcode.com/problems/get-maximum-in-generated-array/