School of Computer Science

Dynamic Programming

ALGORITHMICS

Vicente García Díaz
garciavicente@uniovi.es


Contents

DYNAMIC PROGRAMMING

Basic concepts

Problems with Divide and Conquer

- The idea **was** to divide the original problem in subproblems and combine them to solve the original problem

- Drawbacks:
  - Not suitable when the **number** of subproblems is very **high** and then the complexity is not polynomial
  - Not suitable when generating a number of subproblems that are **repeated** and therefore are solved several times in the same execution

Dynamic Programming

- The idea **is** to divide the original problem in subproblems and combine them to solve the original problem

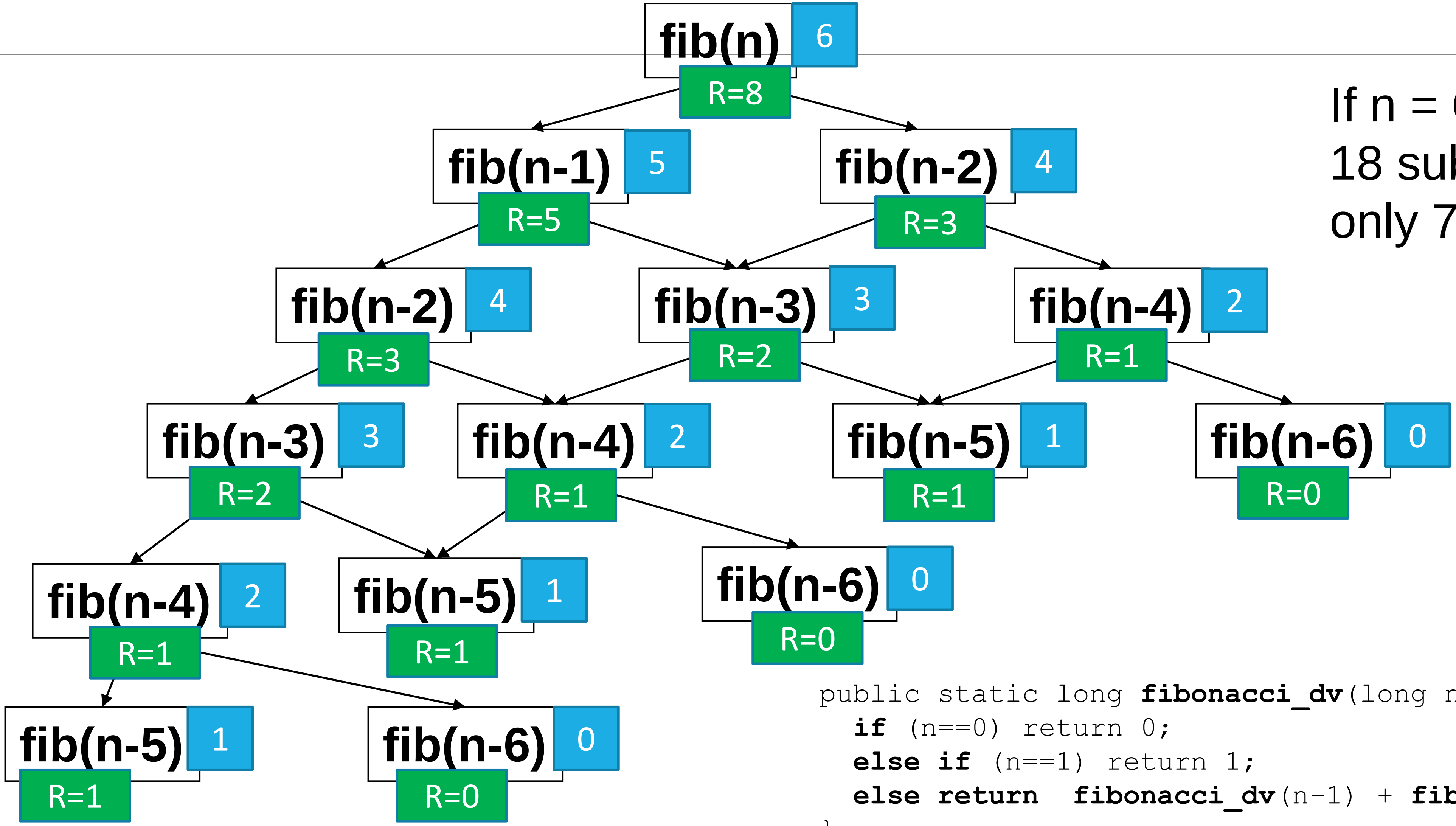
- Improvement:
  - We can solve each subproblem once and store the solution for later use
  - The idea is to **avoid calculating the same subproblem twice**, usually maintaining a table of known results

Pseudocode (Divide and Conquer)

- Fibonacci (example)

```
long fibonacci_dv(int n)
    if (n==0) return 0
    else if (n==1) return 1
    else return fibonacci_dv(n-1) + fibonacci_dv(n-2)
```

Call tree (Divide and Conquer)



If n = 6
18 subproblems
only 7 differ

```
public static long fibonacci_dv(long n){
  if (n==0) return 0;
  else if (n==1) return 1;
  else return  fibonacci_dv(n-1) + fibonacci_dv(n-2);
}
```

Pseudocode

- Fibonacci (example)

```
public static int fibonacci_dp(int n){
  int[] f = new int[n+1]; //0, 1, 2, 3, 4, 5, 6

  f[0]= 0; f[1]= 1; //we know it
  for (int i=2; i<n+1; i++)
      f[i]= f[i-1]+f[i-2];
  return f[n];
}
```

f[2] = f[1] + f[0] = 1+0 = 1
f[3] = f[2] + f[1] = 1+1 = 2
f[4] = f[3] + f[2] = 2+1 = 3
f[5] = f[4] + f[3] = 3+2 = 5
f[6] = f[5] + f[4] = 5+3 = **8**

fib2()?

Divide and Conquer vs Dynamic Programming

Divide and Conquer

- Descending technique (progressive refinement)
- We start with the whole problem
  - We divide it into subproblems

Dynamic Programming

- Ascending technique
- We start with the subproblems
  - We compose solutions until reaching the solution for the whole initial problem

DYNAMIC PROGRAMMING

Examples of use

Fibonacci series

Goal

- Calculate the Fibonacci function (0,1,1,2,3,5,8,13,21,34,55,89, ...)

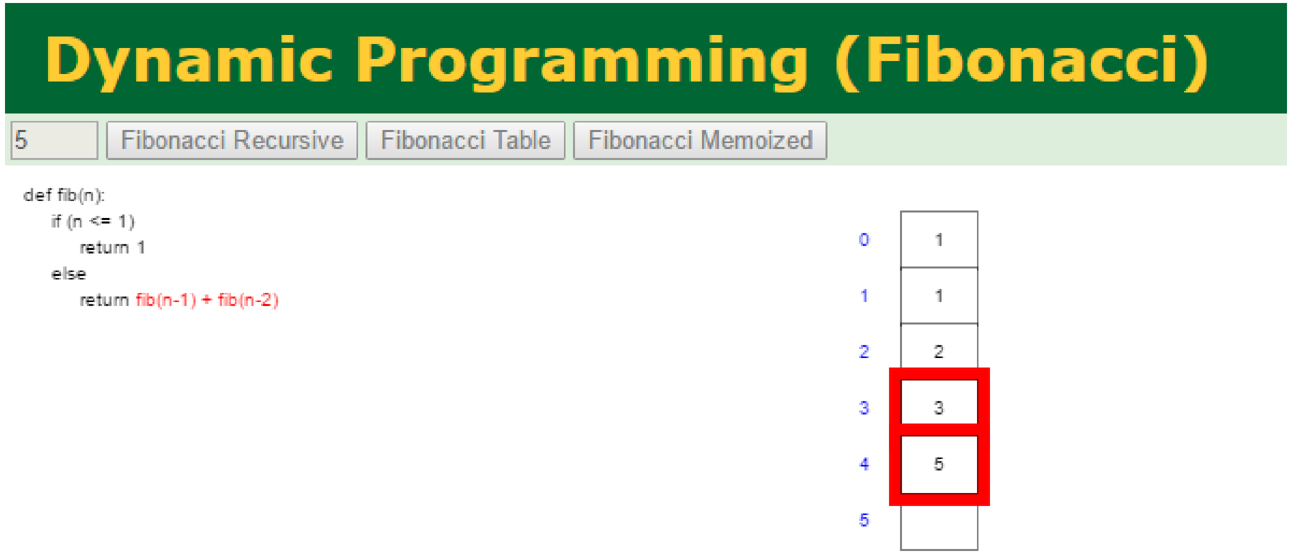
$$F = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n>1 \end{cases}$$

Complexity comparison

- Divide & Conquer → $O(1.6^n)$
- Dynamic Programming → $O(n)$

visualize it

- http://www.cs.usfca.edu/~galles/visualization/DPFib.html



Combinations

$$\frac{50!}{6!(50-6)!} = \frac{50!}{6!(44!)}$$
$$= \frac{50 \times 49 \times 48 \times 47 \times 46 \times 45}{6 \times 5 \times 4 \times 3 \times 2}$$
$$= 15{,}890{,}700$$

- In mathematics, a **combination** is a way of selecting several things out of a larger group, where (unlike *permutations*) order does not matter

- In smaller cases it is possible to count the number of combinations
  - For example, given **three fruits**, say an apple, orange and pear, **there are three combinations of two** that can be drawn from this set

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{si } 0 < k < n, \quad \binom{n}{0} = \binom{n}{n} = 1$$

source: https://goo.gl/Bn0jp3

Join at vevox.app

Or

Search Vevox in the app store

ID: 120-620-870

GET IT ON Google Play | Download on the App Store | Available as a Web App

What is the result if we want to calculate the number of combinations of 9 elements taken 5 by 5?

POLL OPEN

1. 100
2. 115
3. 125
4. 126
5. 130
6. 117

A possible solution with DP

$$\frac{50!}{6!(50-6)!} = \frac{50!}{6!(44!)}$$
$$= \frac{50\times 49\times 48\times 47\times 46\times 45}{6\times 5\times 4\times 3\times 2}$$
$$= 15{,}890{,}700$$

	0	1	2	3	…	k-1	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
…	…	…	…	…	…		
…	…	…	…	…	…	…	
n-1						C(n-1,k-1) +	C(n-1,k)
n							C(n,k)

combinationsDivideAndConquer()?

Implementations

$$\frac{50!}{6!(50-6)!} = \frac{50!}{6!(44!)}$$
$$= \frac{50 \times 49 \times 48 \times 47 \times 46 \times 45}{6 \times 5 \times 4 \times 3 \times 2}$$
$$= 15{,}890{,}700$$

- Complexities?
  - Divide and Conquer
  - Dynamic programming

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{si } 0 < k < n, \qquad \binom{n}{0} = \binom{n}{n} = 1$$

```
public long combinationsDivideAndConquer(int n, int k) {
    if (n==k)
        return 1;
    else
        if (k==0)
            return 1;
    else return combinationsDivideAndConquer(n-1, k-1) +
            combinationsDivideAndConquer(n-1, k);
}
```

```
public int combinations(int[][] table, int n, int k) {
    for (int i = 0; i <= n; i++) table[i][0] = 1;
    for (int i = 1; i <= n; i++) table[i][1] = i;
    for (int i = 2; i <= k; i++) table[i][i] = 1;
    for (int i = 3; i <= n; i++)
        for (int j = 2; j <= k; j++)
            table[i][j] = table[i-1][j-1] + table[i-1][j];
    return table[n][k];
}
```

The knapsack problem

- n objects and a backpack to transport them
- Each object $i = 1,2, ...n$ has a weight of w_i and a value of v_i
- The backpack can carry a total weight not exceeding W
- **The idea is to maximize the value of objects, while respecting the weight limitation**
- Objects **cannot be fragmented**; we take an entire object, or we leave it

Data for a specific problem

- Number of objects: `n=3`
- Weight limit of the backpack: `W=10`

Object	1	2	3
w_i	6	5	5
v_i	8	5	5

Strategy

- Table `V`
  - Rows: `i` objects
  - Columns: maximum weight of the backpack

- `V[i,j]` → maximum value of the items we would carry
  - We include only until object `i` for each case
  - The weight limit is `j`

- Solution to our problem `V[n,W]` → `V[3,10]`

Strategy (II)

- Function that calculates values in the matrix:

$$V(i,j)=\begin{cases} -\infty \text{ if } \ j<0 \\ 0 \text{ if } \ i=0 \,\&\, j\geq 0 \\ \max(V(i\text{-}1,j),V(i\text{-}1,j\text{-}w_i)+v_i) \ \text{ other case} \end{cases}$$

- `i`, is the number of objects we try to put in the backpack
- `j`, is the maximum weight of the backpack

Table of values

- For $n = 3$ (objects), $W = 10$ (maximum load)

Maximum weights

Objects

	0	1	2	3	4	5	6	7	8	9	10
1											
2											
3											

V[i,j]

V[n,W]

Table values. Cell in

- For $n = 3$ (objects), $W = 10$ (maximum load)

Maximum weights

Objects

i \ j	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0					
2	0	0	0	0	0	5					
3	0	0	0	0	0						

Obj.	1	2	3
w_i	6	5	5
v_i	8	5	5

V[i,j] Max(V(i-1,j),V(i-1,j-w_i)+v_i)

Table values. Cell out

- For `n = 3` (objects), `W = 10` (maximum load)

Maximum weights

Objects

i \ j	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0								
2	0	0	0								
3	0	0									

Obj.	1	2	3
w_i	6	5	5
v_i	8	5	5

V[i,j] $Max(V(i-1,j),V(i-1,j-w_i)+v_i)$

Join at vevox.app

Or

Search Vevox in the app store

ID: 120-620-870

GET IT ON Google Play | Download on the App Store | Available as a Web App

POLL OPEN

What is the maximum value we can carry in the backpack with a W = 6

Obj.	1	2	3	4
w_i	3	2	1	4
v_i	6	4	5	10

1. 13
   0%
2. 14
   0%
3. 15
   100%
4. 16
   0%
5. 17
   0%
6. 18
   0%
7. 19
   0%

Implementation

- Complexity?

```
public float knapsack01(int maxWeight, float[]values, int[]weights) {
    int n = weights.length;
    //Creates the table [different types of objects][value we need to deal with + 1 because we start in zero]
    float[][]v = new float[n][maxWeight+1];

    float notInsertingNewObject = 0;
    float insertingNewObject = 0;
    for (int i=0; i<=maxWeight; i++)
        if (i >= weights[0]) //We only insert the first element when we have capacity
            v[0][i] = values[0];

    for (int i=1;i<n;i++)
        for (int j=0; j<=maxWeight; j++) {
            notInsertingNewObject = v[i-1][j]; //The value from the previous row
            if (j >= weights[i]) //If we can get an object from weights[i] and we still have objects to insert
                insertingNewObject = values[i] + v[i-1][j-weights[i]];
            else insertingNewObject = Integer.MIN_VALUE; //It is not reachable
            //We always choose the most valuable object => we want much value
            v[i][j] = Math.max(notInsertingNewObject, insertingNewObject);
        }

    return v[n-1][maxWeight];
}
```

The problem of the change

- Design an algorithm to pay a certain amount of money, using the fewest possible coins

- Example:
  - We have to pay €2.89
  - Solution: 1 coin of €2, 1 coin of 50 cents, 1 coin of 20 cents, 1 coin of 10 cents, 1 coin of 5 cents, 2 coins of 2 cents
    - → Optimal solution ☺

- **Greedy heuristic**: Take the coin of the biggest possible value without exceeding what we have left to return → It does not work for all the cases

The problem of the change (II)

- **Can you do it** *(using the dynamic programming technique)*?

- Key differences with “Knapsack problem”
  - Main methods:
    - `float knapsack01(int maxWeight, float[]values, int[]weights)`
    - `int change(int amount, int[]coins)`
  - Now we don’t look for the greatest value, we look for the lowest
  - We should sort the coins from the smallest to the biggest (the first one should have a value of 1)

change()?

Join at vevox.app

Or

Search Vevox in the app store

ID: 120-620-870

GET IT ON Google Play | Download on the App Store | Available as a Web App

POLL OPEN

How many coins do I need to pay 15 using coins of 1, 4, 5, 12, 20, 50, 100 and 200?

✓ 1. 3

100%

2. 4

0%

3. 5

0%

4. 6

0%

5. 7

0%

6. 8

0%

7. 9

0%

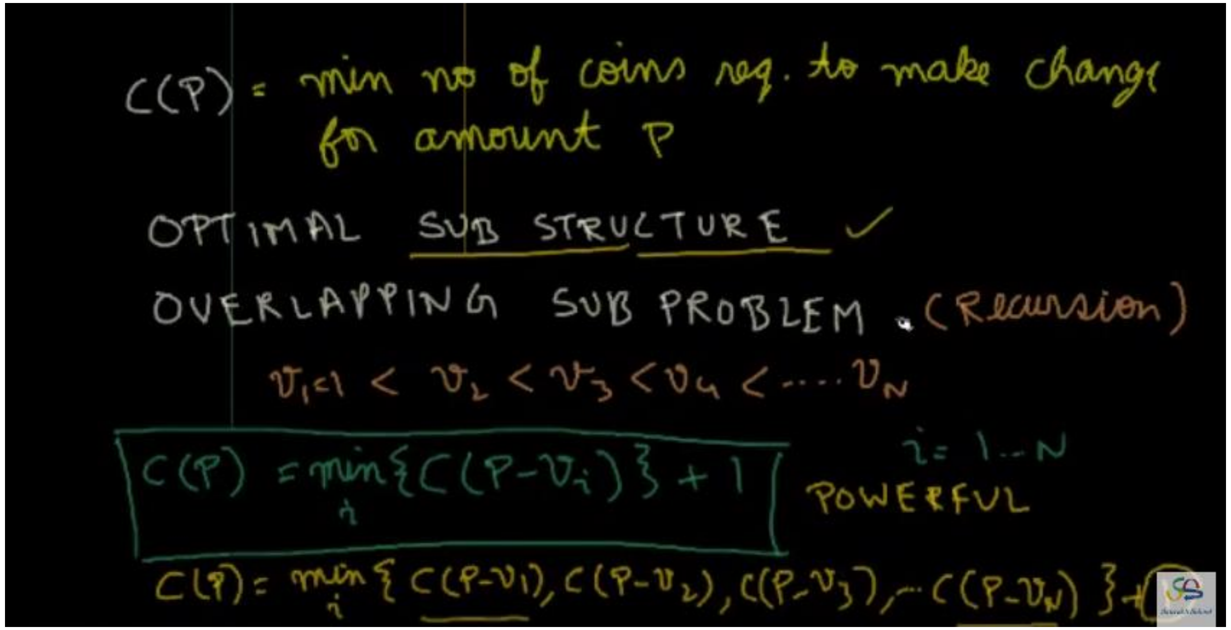
Implementation

- Complexity?

```
public int change(int amount, int[]coins) {
    int types = coins.length; //The different types of money we have for a specific problem
    int[][]sol = new int[types][amount+1]; //Creates the table
    //[different types of coins][value we need to deal with + 1 because we start in zero]
    int notPickingNewCoin = 0;
    int pickingNewCoin = 0;
    for (int i=0; i<= amount; i++)
        sol[0][i] = i; //It saves the value of the money (from 0 to the given value)

    for (int i=1; i<types; i++)
        for (int j=0; j<=amount; j++) {
            notPickingNewCoin = sol[i-1][j]; //The value from the previous row
            if (j >= coins[i]) //If we can get a coin from coin[i] and we still have money to refund
                pickingNewCoin = 1+sol[i][j-coins[i]];
            else pickingNewCoin = Integer.MAX_VALUE; //It is not reachable
            sol[i][j] = Math.min(notPickingNewCoin, pickingNewCoin); //We always choose the smallest coin => we want few coins
        }

    return sol[types-1][amount]; //It returns the last value of all
}
```



More at:

- Dynamic Programming: Coin Change Problem
  https://www.youtube.com/watch?v=GafjS0FfAC0

Cheaper travel on the river

- We are in a river that has `n` docks
- In each of them you can rent a boat to go to any other dock downstream (it is impossible to go upstream)
- There is a fee table that indicates the cost of traveling from dock `i` to dock `j` (`i<j`)
- It may happen that a trip from `i` to `j` is more expensive than a succession of shorter trips, in which case we would take a boat from `i` to a dock `k` first and a second boat to go from `k` to `j`
- Our problem is to design an efficient algorithm to determine the minimum cost for each pair of docks `i,j (i<j)`
  - Indicate, in function of `n`, the time used by the algorithm

riverTravel()?

vevox
Audience Engagement

Join at vevox.app

Or

Search Vevox in the app store

ID: 120-620-870

GET IT ON Google Play | Download on the App Store | Available as a Web App

POLL OPEN

What would be the best cost for going from E1 to E5?

	E1	E2	E3	E4	E5
E1	X	3	8	9	20
E2	X	X	5	5	2
E3	X	X	X	3	6
E4	X	X	X	X	2
E5	X	X	X	X	X

1. 3
   0%
2. ✓ 5
   100%
3. 12
   0%
4. 19
   0%
5. 20
   0%
6. 29
   0%

What would be the best cost for going from E1 to E5? (II)

	E1	E2	E3	E4	E5
E1	X	3	8	9	20
E2	X	X	5	5	2
E3	X	X	X	3	6
E4	X	X	X	X	2
E5	X	X	X	X	X

Fee table

	E1	E2	E3	E4	E5
E1	X				
E2	X	X			
E3	X	X	X		
E4	X	X	X	X	
E5	X	X	X	X	X

c

POLL OPEN

Is Divide and Conquer a descending technique?

1. Yes
2. No

30

POLL OPEN

Is Dynamic programming a descending technique?

1. Yes
2. No

30

POLL OPEN

Is Divide and Conquer suitable when the number of subproblems is high?

1. Yes
2. No

30

POLL OPEN

When we have a great number of subproblems that are repeated several times in the same execution, it is better:

1. To use Divide and Conquer
2. To use Dynamic Programming

30

POLL OPEN

What is the complexity for solving the Fibonacci problem with Dynamic Programming?

1. $O(2^n)$
2. Between $O(n^{n/2})$ and $O(2^n)$
3. $O(n)$
4. $O(1.6^n)$

30

POLL OPEN

What is the complexity for solving the Combinations problem (given `n` elements and taken them `k` by `k`) with Dynamic Programming?

1. O(nlogn)
2. O(n^2)
3. O(n)
4. O(n*k)

30

POLL OPEN

What is the complexity for solving the Backpack01 problem with Dynamic Programming?

1 O(n)

2 O(nlogn)

3 O(n^3)

✓ 4 O(maxWeight*objects)

30

POLL OPEN

What is the complexity for solving the River Travel problem with Dynamic Programming?

1. $O(n^4)$
2. $O(n^3)$
3. $O(n)$
4. $O(n^2)$

30

Bibliography