Scientific Programming: Algorithms (part B)

Programming paradigms

Luca Bianco - Academic Year 2020-21
luca.bianco@fmach.it
[credits: thanks to Prof. Alberto Montresor]

Problems and solutions

Given a problem:

- There are no "general recipes" to solve it
- Nevertheless, we can identify four phases:
  - Problem classification
  - Solution characterization
  - Selection of the algorithmic technique
  - Selection of the data structure
- These phases are not strictly sequential

Classification of problems

Decisional problems

- Does the input satisfy a given property?
- Output: the answer is yes/no
- Example: is the graph connected?

Search problems

- Research space: a set of possible "solutions"
- Admissible solution: a solution that does satisfy some conditions
- Example: position of a substring in the string

Classification of problems

Optimization problems

- Each solution is associated with a cost function
- We want to identify the solution with minimum cost
- Example: the shortest path between nodes in a graph

Approximation problems

- Sometimes, obtaining the optimal solution is computationally infeasible
- We may be satisfied by an approximate solution: low cost, but we are not sure that the cost is the smallest possible
- Example: the traveling salesman problem

Mathematical characterization

It is important to mathematically define the relationship between input and output

- Very often the mathematical characterization is trivial...
- ... but it could provide a first idea of the solution
- Example: given a sequence of n elements, a sorted permutation is given by the minimum followed by a sorted permutation of the remaining $n-1$ elements (Selection Sort)

The mathematical characterization can suggest a possible technique

- Optimal substructure $\rightarrow$ Dynamic programming
- Greedy choice $\rightarrow$ Greedy technique

Algorithmic techniques

Divide-et-impera

- The problem is subdivided in independent subproblems, that are solved recursively (in a top-down approach)
- Area of application: decision problems, search (ex. QuickSort)

Dynamic programming

- The solution is built in a bottom-up way from the solution of smaller problems (potentially repeated)
- Area of application: optimization problems

Memoization

- Top-down version of dynamic programming

Algorithmic techniques

Greedy

- Greedy approach: select the choice which appears "locally optimal"
- Area of application: optimization problems

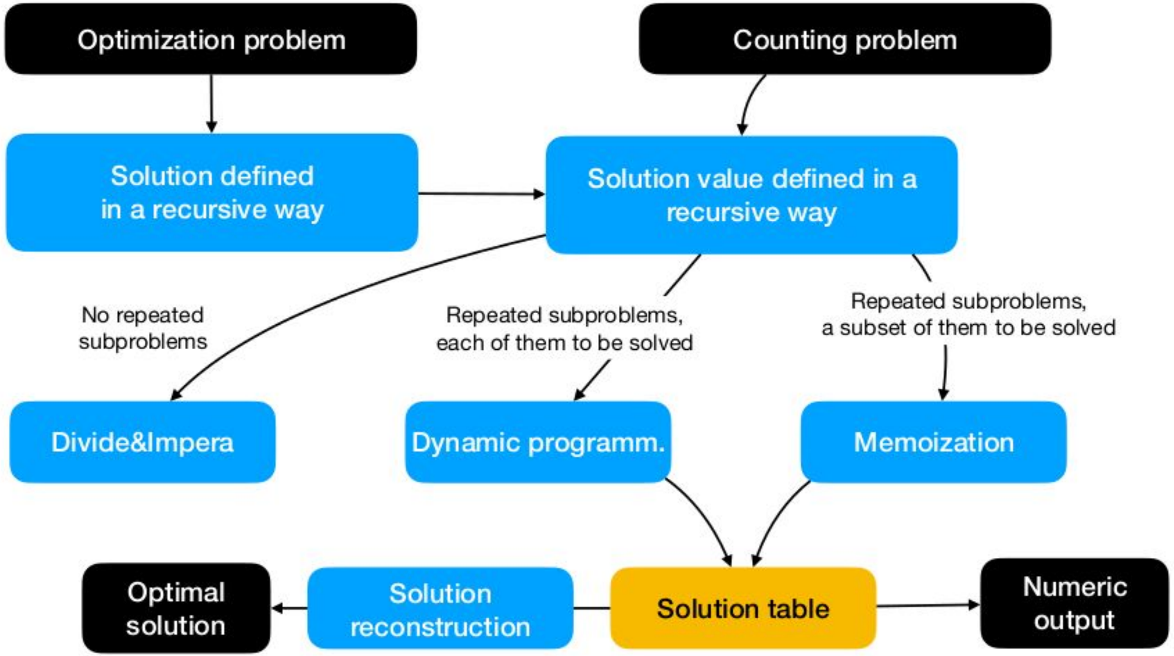
Backtrack

- Try something, and if does not work, try something else
- Area of application: search problems, optimization problems

Local search

- The optimal solution can be obtained by continuously improving sub-optimal solutions

General approach



1. Define the solution (better, the value of the solution) in recursive terms
2. Depending on if we can build the solution from repeated subproblems we apply different techniques
3. From DP and memoization (if we do not need to solve all the subproblems, but just a subset) we get a solution table that we need to analyze to get a numeric solution or to build the optimal solution

Dominoes

Definition

The dominoes game consists of tiles with size 2×1. Let us consider the arrangements of n tiles inside a rectangle $2 \times n$. Write an efficient algorithm that computes the number of possible arrangements and discuss its correctness. Compute an upper bound to its complexity.

Example

The cases below represent the five possible arrangements in a rectangle 2×4.

Any ideas on how to solve this problem?

Dominoes

Recursive definition

Let's define a recursive formula that computes the number of possible arrangements.

- If a vertical tile is placed, the problem of size $n-1$ must be solved.
- If an horizontal tile is placed, then another horizontal tile must be placed as well; the problem of size $n-2$ must be solved.

$$D(n) = \begin{cases} 1 & n \leq 1 \\ ? & n > 1 \end{cases}$$

n= 0, only one possibility: **no tiles.** n=1, only 1 possibility, **vertical tile**

2xn

n -1

2xn

n -2

Dominoes

Recursive definition

Let's define a recursive formula that computes the number of possible arrangements.

- If a vertical tile is placed, the problem of size $n-1$ must be solved.
- If an horizontal tile is placed, then another horizontal tile must be placed as well; the problem of size $n-2$ must be solved.

$$D(n) = \begin{cases} 1 & n \leq 1 \\ D(n-2) + D(n-1) & n > 1 \end{cases}$$

We sum because the two cases originate different solutions

2xn

n -1

2xn

n -2

Dominoes

$$D(n) = \begin{cases} 1 & n \leq 1 \\ D(n-2) + D(n-1) & n > 1 \end{cases}$$

The generated mathematical series is the following:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Does it sound familiar?

Fibonacci's numbers!

N = 4 (i.e. 2x4) → 5 possible dispositions (n+1)th Fibonacci's number

Dominoes: recursive algorithm

$$D(n) = \begin{cases} 1 & n \leq 1 \\ D(n-2) + D(n-1) & n > 1 \end{cases}$$

Write a recursive algorithm that solves the problem

```
def dominoes(n):
    if n <= 1:
        return 1
    else:
        return dominoes(n-2) + dominoes(n-1)


for i in range(10):
    print(dominoes(i), end = " ")
```

```
1 1 2 3 5 8 13 21 34 55
```

Complexity

```python
def dominoes(n):
    if n <= 1:
        return 1
    else:
        return dominoes(n-2) + dominoes(n-1)


for i in range(10):
    print(dominoes(i), end = " ")
```

```
1 1 2 3 5 8 13 21 34 55
```

What is the complexity of dominoes?

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n-1) + T(n-2) + 1 & n > 1 \end{cases}$$

cost of if and sum

Theorem

Let $a_1, a_2, \ldots, a_h$ be non-negative integer constants; let c and β real constant such that $c > 0$ and $\beta \geq 0$; let $T(n)$ be a recurrence defined as follows:

$$T(n) = \begin{cases} \sum_{1 \leq i \leq h} a_i T(n-i) + cn^\beta & n > m \\ \Theta(1) & n \leq m \leq h \end{cases}$$

Given $a = \sum_{1 \leq i \leq h} a_i$, then:

1. $T(n) = \Theta(n^{\beta+1})$, if $a = 1$,
2. $T(n) = \Theta(a^n n^\beta)$, if $a \geq 2$.

<u>Theorem not seen:</u>

Linear recurrences with constant order:

- $a_1 = 1, a_2 = 1, a = 2, \beta = 0$
- Complexity: $\Theta(a^n \cdot n^\beta)$

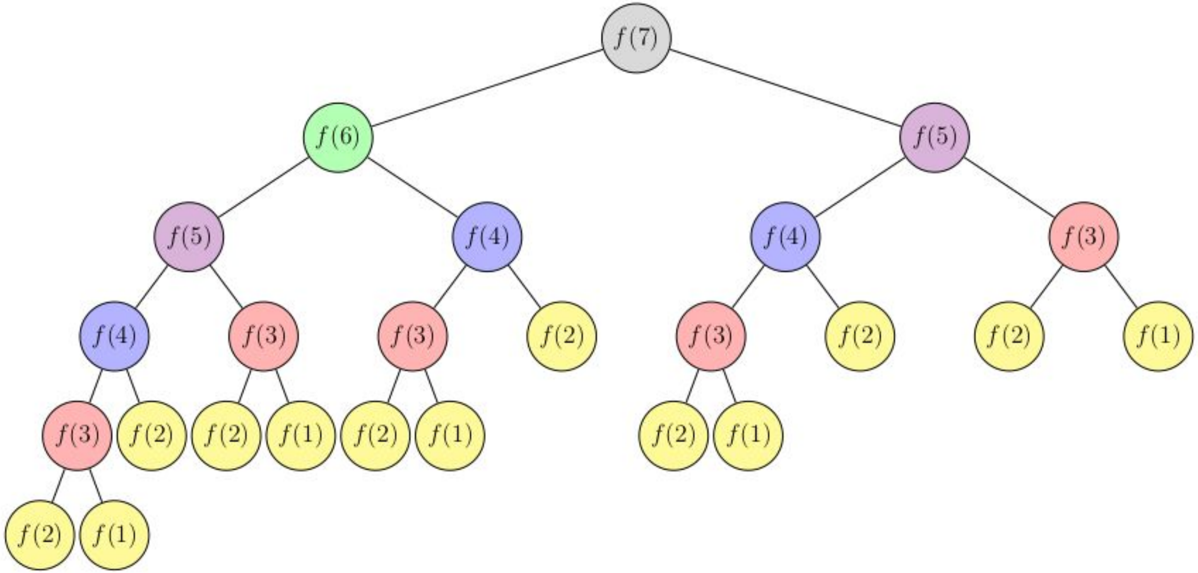
$$T(n) = \Theta(2^n)^*$$

Recursive tree

```python
def dominoes(n):
    if n <= 1:
        return 1
    else:
        return dominoes(n-2) + dominoes(n-1)


for i in range(10):
    print(dominoes(i), end = " ")
```

```
1 1 2 3 5 8 13 21 34 55
```



Several sub-problems are repeated!

How to avoid computing the same thing over and over again: Dynamic programming

DP Table

- We use a *DP* table (list, matrix, dictionary, etc.) to store results of sub-problems already solved
- The table contains an entry for each subproblem to be solved
- The table is indexed by a description of the input (e.g., size)
- When the same subproblem has to be solved again, we use the result stored in the table

How to avoid computing the same thing over and over again: Dynamic programming

Base cases

- The bases cases do not need to be computed, they can be stored immediately

Bottom-up iteration

- We start from problems that can be solved using only base cases
- We go up to larger and larger problems...
- ... up to the final goal

n	0	1	2	3	4	5	6	7
DP []	1	1	2	3	5	8	13	21

An iterative solution

$$D(n) = \begin{cases} 1 & n \leq 1 \\ D(n-2) + D(n-1) & n > 1 \end{cases}$$

Write an iterative algorithm that solves the Dominoes problem

```
def dominoes2(n):
    res = [0]*(n+1)
    res[0] = 1
    res[1] = 1
    for i in range(2,n+1):
        res[i] = res[i-1] + res[i-2]
    return res[n]
```

base cases, stored immediately

output

What is the computational complexity of `domino2(n)`?

$T(n) = \Theta(n)^*$

How about the space complexity?
What is the size of res?

$S(n) = \Theta(n)^*$

Ideas on how to improve this?

Another iterative solution

$$D(n) = \begin{cases} 1 & n \leq 1 \\ D(n-2) + D(n-1) & n > 1 \end{cases}$$

```python
def dominoes3(n):
    dp0 = 1
    dp1 = 1
    dp2 = 1
    for i in range(2,n+1):
        dp0 = dp1
        dp1 = dp2
        dp2 = dp0 + dp1
    return dp2
```

What is the space complexity of `domino3(n)`?

$$S(n) = \Theta(1)^{*}$$

n	0	1	2	3	4	5	6	7
DP []	1	1	2	3	5	8	13	21

Uniform vs Logarithmic cost model

$$D(n) = \begin{cases} 1 & n \leq 1 \\ D(n-2) + D(n-1) & n > 1 \end{cases}$$

Are you sure that our complexity formulas are correct?

*

Binet's Formula for Fibonacci's number

$$D(n-1) = F(n) = \frac{\phi^n}{\sqrt{5}} - \frac{(1-\phi)^n}{\sqrt{5}} = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

where

$$\phi = \frac{1+\sqrt{5}}{2} = 1{,}6180339887\ldots$$ golden ratio

Careful there: the Fibonacci's number grows exponentially!

How many bits are needed to store $F(n)$? $= \frac{\phi^n}{\sqrt{5}}$

Uniform vs Logarithmic cost model

$$D(n) = \begin{cases} 1 & n \leq 1 \\ D(n-2) + D(n-1) & n > 1 \end{cases}$$

Are you sure that our complexity formulas are correct?

Binet's Formula for Fibonacci's number

$$D(n-1) = F(n) = \frac{\phi^n}{\sqrt{5}} - \frac{(1-\phi)^n}{\sqrt{5}} = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

where

$$\phi = \frac{1+\sqrt{5}}{2} = 1{,}6180339887\ldots$$ golden ratio

How many bits are needed to store $F(n)$? $= \frac{\phi^n}{\sqrt{5}}$

$$\log F(n) = \Theta(n)$$

the complexity seen before needs to be multiplied by n

Careful there: the Fibonacci's number grows exponentially!

This affects the complexity of summing two consecutive Fibonacci numbers

Uniform vs Logarithmic cost model

$$D(n) = \begin{cases} 1 & n \leq 1 \\ D(n-2) + D(n-1) & n > 1 \end{cases}$$

Under the logarithmic cost model, the three versions have the following complexities:

Function	**Time complexity**	**Space complexity**
`domino1()`	$O(n2^n)$	$O(n^2)$
`domino2()`	$O(n^2)$	$O(n^2)$
`domino3()`	$O(n^2)$	$O(n)$

Uniform vs Logarithmic cost model

$$D(n) = \begin{cases} 1 & n \leq 1 \\ D(n-2) + D(n-1) & n > 1 \end{cases}$$

Under the logarithmic cost model, the three versions have the following complexities:

Function	Time complexity	Space complexity
domino1()	$O(n2^n)$	$O(n^2)$
domino2()	$O(n^2)$	$O(n^2)$
domino3()	$O(n^2)$	$O(n)$

```python
s = time.time()
for i in range(1,45):
    print(dominoes(i), end = " ")
e = time.time()
print("Elapsed time: {}s".format(e-s))
s = time.time()
for i in range(1,45):
    print(dominoes2(i), end = " ")
e = time.time()
print("Elapsed time: {}s".format(e-s))

s = time.time()
for i in range(1,45):
    print(dominoes3(i), end = " ")
e = time.time()
print("Elapsed time: {}s".format(e-s))
```

1 2 3 5 8 ... 1134903170
Elapsed time: 659.3645467758179s

1 2 3 5 8 … 1134903170
Elapsed time: 0.0007071495056152344s

1 2 3 5 8 … 1134903170
Elapsed time: 0.0011742115020751953s

Hateville

- Hateville is a strange village, composed of n houses, numbered 1-n and placed along a single road
- In Hateville, everybody hates his next-door neighbors, on both sides: thus a person living in house i hates the neighbors living in houses $i-1$ and $i+1$ (if they exist)
- Hateville wants to organize a festival; your task is to collect money to organize it.
- Each inhabitant i wants to donate a quantity $D[i]$ of money, but he will give nothing if any of his neighbors is donating.

Hateville

Consider the following problems:

- Write an algorithm that returns the largest amount of money that can be collected
- Write an algorithm that returns a subset of indexes $S \subseteq \{1, \ldots, n\}$ such that the total amount $T = \sum_{i \in S} D[i]$ is maximal.

remember the additional constraint that indexes must not be consecutive

Examples:

- Donation list: `D = [4, 3, 6, 5]`
- Maximum amount: 10
- Index set: $\{1, 3\}$

- Donation list: `D = [10, 5, 5, 10]`
- Maximum amount: 20
- Index set: $\{1, 4\}$

summing all even or all odds does not work!

Hateville

- Donation list: `D = [4, 3, 6, 5]`
- Maximum amount: 10
- Index set: $\{1, 3\}$

- Donation list: `D = [10, 5, 5, 10]`
- Maximum amount: 20
- Index set: $\{1, 4\}$

How would you solve the problem?

We re-define the problem

- Let $HV(i)$ be the set of numbers to be selected to obtain the maximum amount of donations from the first i houses, numbered $1 \ldots n$
- $HV(n)$ is the solution to the original problem

Hateville

- Donation list: `D = [4, 3, 6, 5]`
- Maximum amount: 10
- Index set: $\{1, 3\}$

- Donation list: `D = [10, 5, 5, 10]`
- Maximum amount: 20
- Index set: $\{1, 4\}$

Let's compute $HV(i)$ based on $HV(0) \dots HV(n-1)$ values

- What happens if I don't accept its donation?

$$HV(i) =$$

Hateville

- Donation list: `D = [4, 3, 6, 5]`
- Maximum amount: 10
- Index set: $\{1, 3\}$

- Donation list: `D = [10, 5, 5, 10]`
- Maximum amount: 20
- Index set: $\{1, 4\}$

Let's compute $HV(i)$ based on $HV(0) \ldots HV(n-1)$ values

- What happens if I don't accept its donation?

$$HV(i) = HV(i-1)$$

Hateville

- Donation list: `D = [4, 3, 6, 5]`
- Maximum amount: 10
- Index set: $\{1, 3\}$

- Donation list: `D = [10, 5, 5, 10]`
- Maximum amount: 20
- Index set: $\{1, 4\}$

Let's compute $HV(i)$ based on $HV(0) \ldots HV(n-1)$ values

- What happens if I don't accept its donation?

$$HV(i) = HV(i-1)$$

- What happens if I accept its donation?

Hateville

- Donation list: D = [4, 3, 6, 5]
- Maximum amount: 10
- Index set: $\{1, 3\}$

- Donation list: D = [10, 5, 5, 10]
- Maximum amount: 20
- Index set: $\{1, 4\}$

Let's compute $HV(i)$ based on $HV(0) \dots HV(n-1)$ values

- What happens if I don't accept its donation?

$$HV(i) = HV(i-1)$$

- What happens if I accept its donation?

$$HV(i) = HV(i-2) + D[i]$$

Hateville

- Donation list: `D = [4, 3, 6, 5]`
- Maximum amount: 10
- Index set: $\{1, 3\}$

- Donation list: `D = [10, 5, 5, 10]`
- Maximum amount: 20
- Index set: $\{1, 4\}$

$\boxed{\text{Let's compute } HV(i) \text{ based on } HV(0) \ldots HV(n-1) \text{ values}}$

- What happens if I don't accept its donation?

$$HV(i) = HV(i-1)$$

- What happens if I accept its donation?

$$HV(i) = HV(i-2) + D[i]$$

- How can I choose between the two cases?

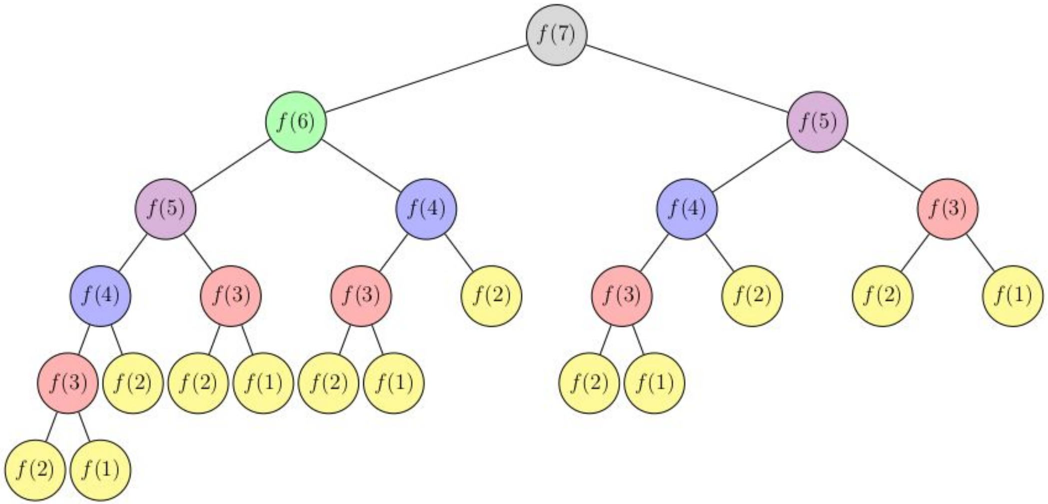
$$max(HV(i-1), HV(i-2) + D[i])$$

Hateville: recursive algorithm?

$max(HV(i-1), HV(i-2) + D[i])$

Write a recursive algorithm that solves Hateville?

Would it be a good idea?



DP Table

$max(HV(i-1), HV(i-2)+D[i])$

Value of the optimal solution

- Let $DP(i)$ be the value of the maximum amount of donation that we can obtain from the first i houses of Hateville
- $DP(n)$ is the value of the optimal solution

$$DP(i) = \begin{cases} 0 & \text{if i} = 0 \\ D[1] & \text{if i} = 1 \\ max(DP(i-1), DP(i-2)+D[i]) & \text{if } n \geq 2 \end{cases}$$

Iterative solution

$$DP(i) = \begin{cases} 0 & \text{if i} = 0 \\ D[1] & \text{if i} = 1 \\ max(DP(i-1), DP(i-2) + D[i]) & \text{if } n \geq 2 \end{cases}$$

Write an algorithm that solves the Hateville problem

```
def hateville(D, n):
    dp = [0]*(n+1)
    if n > 0:
        dp[1] = D[0]
    for i in range(2, n+1):
        dp[i] = max(dp[i-1],dp[i-2] + D[i-1])

    return dp[n]
```

```
D = [10,5,5,8,4,7,12]

print("Donations: {}".format(D))
for i in range(len(D)+1):
    print("Solution for {}: {}".format(D[0:i],hateville(D, i)))
```

```
Donations: [10, 5, 5, 8, 4, 7, 12]
Solution for []: 0
Solution for [10]: 10
Solution for [10, 5]: 10
Solution for [10, 5, 5]: 15
Solution for [10, 5, 5, 8]: 18
Solution for [10, 5, 5, 8, 4]: 19
Solution for [10, 5, 5, 8, 4, 7]: 25
Solution for [10, 5, 5, 8, 4, 7, 12]: 31
```

```
D1 = [10,1,1,10,1,1,10]

print("Donations: {}".format(D1))
for i in range(len(D1)+1):
    print("Solution for {}: {}".format(D1[0:i],hateville(D1, i)))
```

```
Donations: [10, 1, 1, 10, 1, 1, 10]
Solution for []: 0
Solution for [10]: 10
Solution for [10, 1]: 10
Solution for [10, 1, 1]: 11
Solution for [10, 1, 1, 10]: 20
Solution for [10, 1, 1, 10, 1]: 20
Solution for [10, 1, 1, 10, 1, 1]: 21
Solution for [10, 1, 1, 10, 1, 1, 10]: 30
```

Iterative solution

$$DP(i) = \begin{cases} 0 & \text{if } i = 0 \\ D[1] & \text{if } i = 1 \\ max(DP(i-1), DP(i-2) + D[i]) & \text{if } n \geq 2 \end{cases}$$

i	0	1	2	3	4	5	6	7
D		$\boxed{10}$	5	$\boxed{5}$	8	$\boxed{4}$	7	$\boxed{12}$
DP	0	$\boxed{10}$	10	$\boxed{15}$	18	$\boxed{19}$	25	$\boxed{31}$

i	0	1	2	3	4	5	6	7
D		$\boxed{10}$	1	1	$\boxed{10}$	1	1	$\boxed{10}$
DP	0	$\boxed{10}$	$\boxed{10}$	11	$\boxed{20}$	$\boxed{20}$	21	$\boxed{30}$

Problem

- We have the value of the optimal solution, but we don't have the solution!
  - Look in position $DP[i]$. From which cells this value has been computed?
    - If $DP[i] = DP[i-1]$, the house$i$ has not been selected
    - If $DP[i] = DP[i-2] + D[i \quad]$, house i has been selected

Build `solution(i)` recursively as:

`solution(i-2)` AND add `index i to a list`
`or`
`solution(i-1)`

Building the solution

- Look in position $DP[i]$. From which cells this value has been computed?
  - If $DP[i] = DP[i-1]$, the house$i$ has not been selected
  - If $DP[i] = DP[i-2] + D[i\quad]$, house i has been selected

```
def hateville(D, n):
    dp = [0]*(n+1)
    if n > 0:
        dp[1] = D[0]
    for i in range(2, n+1):
        dp[i] = max(dp[i-1],dp[i-2] + D[i-1])

    return build_solution(D,dp,n)

def build_solution(D, dp, i):
    if i == 0:
        return []
    elif i == 1:
        return [0]
    else:
        if dp[i] == dp[i-1]:
            sol = build_solution(D, dp, i-1)
        else:
            sol = build_solution(D, dp, i-2)
            sol.append(i-1)
    return sol
```

```
D = [10,5,5,8,4,7,12]
print("Donations: {}".format(D))
for i in range(len(D)+1):
    HV = hateville(D, i)
    print("Donors for {}: {}. Donations: {}".format(D[0:i],HV,sum([D[x] for x in HV])))
print("\n\n")

D1 = [10,1,1,10,1,1,10]
print("Donations: {}".format(D1))
for i in range(len(D1)+1):
    HV = hateville(D1, i)
    print("Donors for {}: {}. Donations: {}".format(D1[0:i],HV,sum([D1[x] for x in HV])))
```

```
Donations: [10, 5, 5, 8, 4, 7, 12]
Donors for []: []. Donations: 0
Donors for [10]: [0]. Donations: 10
Donors for [10, 5]: [0]. Donations: 10
Donors for [10, 5, 5]: [0, 2]. Donations: 15
Donors for [10, 5, 5, 8]: [0, 3]. Donations: 18
Donors for [10, 5, 5, 8, 4]: [0, 2, 4]. Donations: 19
Donors for [10, 5, 5, 8, 4, 7]: [0, 3, 5]. Donations: 25
Donors for [10, 5, 5, 8, 4, 7, 12]: [0, 2, 4, 6]. Donations: 31


Donations: [10, 1, 1, 10, 1, 1, 10]
Donors for []: []. Donations: 0
Donors for [10]: [0]. Donations: 10
Donors for [10, 1]: [0]. Donations: 10
Donors for [10, 1, 1]: [0, 2]. Donations: 11
Donors for [10, 1, 1, 10]: [0, 3]. Donations: 20
Donors for [10, 1, 1, 10, 1]: [0, 3]. Donations: 20
Donors for [10, 1, 1, 10, 1, 1]: [0, 3, 5]. Donations: 21
Donors for [10, 1, 1, 10, 1, 1, 10]: [0, 3, 6]. Donations: 30
```

Complexity

```python
def hateville(D, n):
    dp = [0]*(n+1)
    if n > 0:
        dp[1] = D[0]
    for i in range(2, n+1):
        dp[i] = max(dp[i-1],dp[i-2] + D[i-1])

    return build_solution(D,dp,n)

def build_solution(D, dp, i):
    if i == 0:
        return []
    elif i == 1:
        return [0]
    else:
        if dp[i] == dp[i-1]:
            sol = build_solution(D, dp, i-1)
        else:
            sol = build_solution(D, dp, i-2)
            sol.append(i-1)
    return sol
```

What is the complexity of build_solution?

$$T(n) = O(n)$$

What is the complexity of hateville?

$$T(n) = O(n)$$

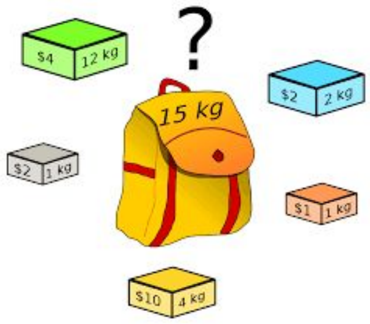
Exercise:

write hateville with $S(n) = O(1)$
(without reconstructing the solution)

Knapsack

Problem

Given a set of items, each of them characterized by a weight and a value, determine which items to include in a collection so that the total weight of the collection is less than or equal to a given "knapsack" capacity and the total value (or profit) is as large as possible.



Input

- List `w`, where $w[i]$ is the weight of the i-th item
- List `p`, where $p[i]$ is the value (or profit) of the i-th item
- The capacity `C` of the knapsack

Output

A collection $S \subseteq \{1, \ldots, n\}$such that:

- Total volume should be smaller or equal than the capacity: $w(S) = \sum_{i \in S} w[i] \leq C$
- Total profit is maximized: $p(S) = \sum_{i \in S} p[i]$ is maximal

Knapsack

Problem

Given a set of items, each of them characterized by a weight and a value, determine which items to include in a collection so that the total weight of the collection is less than or equal to a given "knapsack" capacity and the total value (or profit) is as large as possible.

Which are the best items for this example?

Item id	1	2	3
Weight	10	4	8
Profit	20	6	12

$C = 12$

S = {1}

Item id	1	2	3
Weight	10	4	8
Profit	20	7	15

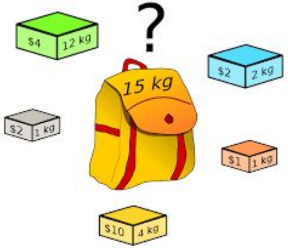
$C = 12$

S = {2,3}

A greedy approach would not work because in the second case we would pick item 1

Design an algorithm to solve the Knapsack problem

Knapsack



Definition: Sub-problem $DP(i, c)$

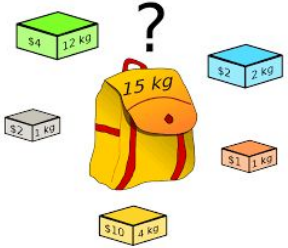
Given a knapsack with capacity C and n items characterized by weights w and profits p, we define $DP(i, c)$ as the maximal profit we can obtain from the first i items in a knapsack of capacity c.

$i \leq n$
$c \leq C$

Original problem

The maximal profit of the original problem corresponds to $DP(n, C)$.

Knapsack



Let us consider the last item of problem $DP(i,c)$

- What happens if you don't take it?

$$DP(i,c) = DP(i-1,c)$$

The capacity and profit do not change

- What happens if you take it?

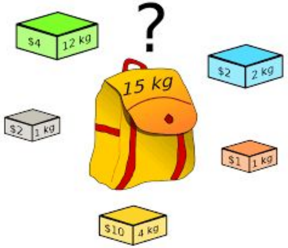
$$DP(i,c) = DP(i-1,c-w[i]) + p[i]$$

Subtract the weight of the item from the capacity and add its profit

How to select the best solution between the two?

$$DP(i,c) =$$

Knapsack



Let us consider the last item of problem $DP(i, c)$

- What happens if you don't take it?

$$DP(i, c) = DP(i-1, c)$$

The capacity and profit do not change

- What happens if you take it?

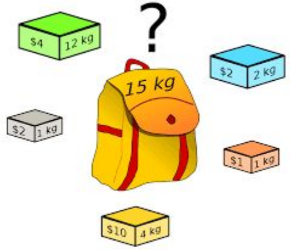
$$DP(i, c) = DP(i-1, c-w[i]) + p[i]$$

Subtract the weight of the item from the capacity and add its profit

How to select the best solution between the two?

$$DP(i, c) = \max(DP(i-1, c-w[i]) + p[i], DP(i-1, c))$$

Knapsack



What are the base cases for this recursive definition?

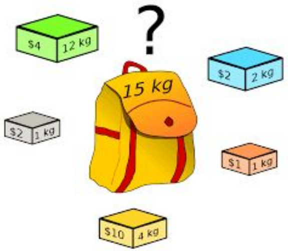
- What happens if you don't have any more items?
- What happens if you don't have any more capacity?
- What happens if your capacity is negative?

$$DP(i,c) = \begin{cases} 0 & i = 0 \textbf{ or } c = 0 \\ -\infty & c < 0 \\ \max(DP(i-1, c-w[i]) + p[i], DP(i-1, c)) & \text{otherwise} \end{cases}$$

to enforce NOT choosing objects that make capacity negative

Knapsack: the code

$$DP(i,c) = \begin{cases} 0 & i = 0 \textbf{ or } c = 0 \\ -\infty & c < 0 \\ \max(DP(i-1, c-w[i]) + p[i], DP(i-1, c)) & \text{otherwise} \end{cases}$$




```python
import numpy as np
import math


def knapsack(w, p, C):
    n = len(w)
    DP = np.zeros((n + 1, C + 1))
    for i in range(1, n+1):
        for c in range(1, C+1):
            not_taken = DP[i-1][c]
            if w[i-1]  > c:
                taken = -math.inf
            else:
                taken = DP[i-1][c - w[i-1]] + p[i-1]

            DP[i][c] = max( not_taken, taken)
    #print(DP)
    return DP[n][C]
```

⇦ inizialize a n+1 x C+1 matrix full of zeros

⇦ bottom-up

⇦ result is here!

```python
w = [4,2,3,4]
p = [10,7,8,6]
C = 9
print(knapsack(w,p,C))
```

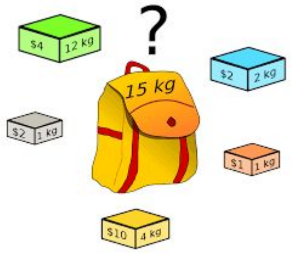
```
25.0
```

	c									
i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

DP[1][1]
not_taken = DP[0][1] = 0
taken = DP[0][1- w[0]] + p[0] → **4 > 1** → - ∞
max(0, -∞) = 0

Knapsack: the code

$$DP(i,c) = \begin{cases} 0 & i = 0 \textbf{ or } c = 0 \\ -\infty & c < 0 \\ \max(DP(i-1, c-w[i]) + p[i], DP(i-1,c)) & \text{otherwise} \end{cases}$$




```
import numpy as np
import math


def knapsack(w, p, C):
    n = len(w)
    DP = np.zeros((n + 1, C + 1))
    for i in range(1, n+1):
        for c in range(1, C+1):
            not_taken = DP[i-1][c]
            if w[i-1]  > c:
                taken = -math.inf
            else:
                taken = DP[i-1][c - w[i-1]] + p[i-1]

            DP[i][c] = max( not_taken, taken)
    #print(DP)
    return DP[n][C]
```

inizialize a n+1 x C+1 matrix full of zeros

bottom-up

result is here!

```
w = [4,2,3,4]
p = [10,7,8,6]
C = 9
print(knapsack(w,p,C))
```

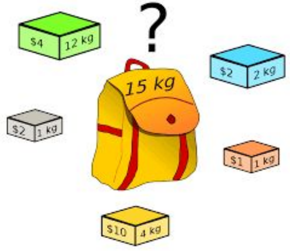
```
25.0
```

	c									
i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	$\boxed{10}$	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

DP[1][4]
not_taken = DP[0][4] = 0
taken = DP[0][4- w[0]] + p[0] → **4** ≤ **4** → 0 + p[0] = 10
max(0, 10) = 10

Knapsack: the code

$$DP(i,c) = \begin{cases} 0 & i = 0 \textbf{ or } c = 0 \\ -\infty & c < 0 \\ \max(DP(i-1, c-w[i]) + p[i], DP(i-1,c)) & \text{otherwise} \end{cases}$$




```python
import numpy as np
import math


def knapsack(w, p, C):
    n = len(w)
    DP = np.zeros((n + 1, C + 1))
    for i in range(1, n+1):
        for c in range(1, C+1):
            not_taken = DP[i-1][c]
            if w[i-1]  > c:
                taken = -math.inf
            else:
                taken = DP[i-1][c - w[i-1]] + p[i-1]

            DP[i][c] = max( not_taken, taken)
    #print(DP)
    return DP[n][C]
```

inizialize a n+1 x C+1 matrix full of zeros

bottom-up

result is here!

```python
w = [4,2,3,4]
p = [10,7,8,6]
C = 9
print(knapsack(w,p,C))
```

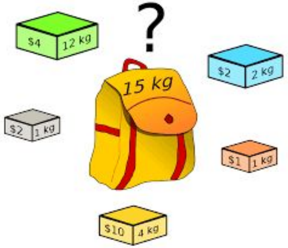
```
25.0
```

	c									
i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	$\boxed{7}$	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

DP[2][2]
not_taken = DP[1][2] = 0
taken = DP[1][2- w[1]] + p[1] → **2 ≤ 2** → 0 + p[1] = 7
max(0, 10) = 7

Knapsack: the code

$$DP(i,c) = \begin{cases} 0 & i = 0 \textbf{ or } c = 0 \\ -\infty & c < 0 \\ \max(DP(i-1, c-w[i]) + p[i], DP(i-1,c)) & \text{otherwise} \end{cases}$$




```python
import numpy as np
import math


def knapsack(w, p, C):
    n = len(w)
    DP = np.zeros((n + 1, C + 1))
    for i in range(1, n+1):
        for c in range(1, C+1):
            not_taken = DP[i-1][c]
            if w[i-1]  > c:
                taken = -math.inf
            else:
                taken = DP[i-1][c - w[i-1]] + p[i-1]

            DP[i][c] = max( not_taken, taken)
    #print(DP)
    return DP[n][C]
```

inizialize a n+1 x C+1 matrix full of zeros

bottom-up

result is here!

```python
w = [4,2,3,4]
p = [10,7,8,6]
C = 9
print(knapsack(w,p,C))
```

```
25.0
```

	c									
i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

DP[2][4]
not_taken = DP[1][4] = 10
taken = DP[1][4- w[1]] + p[1] → **2** ≤ **4** → 0 + p[1] = 7
max(7, 10) = 10

Knapsack: the code

What is the computational complexity of function `knapsack()`?

$$T(n) = O(nC)$$

2 for loops:
one of size n
one of size C

Is it a polynomial algorithm?

No, this is an example of pseudo-polynomial algorithm, because C is not the size of the input, is the input. Thus we need $k = \log C$ bits to represent it, and thus complexity is equal to:

$$T(n) = O(n2^k)$$

```
import numpy as np
import math


def knapsack(w, p, C):
    n = len(w)
    DP = np.zeros((n + 1, C + 1))
    for i in range(1, n+1):
        for c in range(1, C+1):
            not_taken = DP[i-1][c]
            if w[i-1]  > c:
                taken = -math.inf
            else:
                taken = DP[i-1][c - w[i-1]] + p[i-1]

            DP[i][c] = max( not_taken, taken)
    #print(DP)
    return DP[n][C]
```

	c									
i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

Memoization

Note (let's try a top-down approach!)

Not all elements of the table are actually needed to solve our problem.

```
w = [4,2,3,4]
p = [10,7,8,6]
C = 9
print(knapsack(w,p,C))
```

```
import numpy as np
import math


def knapsack(w, p, C):
    n = len(w)
    DP = np.zeros((n + 1, C + 1))
    for i in range(1, n+1):
        for c in range(1, C+1):
            not_taken = DP[i-1][c]
            if w[i-1]  > c:
                taken = -math.inf
            else:
                taken = DP[i-1][c - w[i-1]] + p[i-1]

            DP[i][c] = max( not_taken, taken)
    #print(DP)
    return DP[n][C]
```

	c									
i	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

c- w[n-1]
= 9 - 4

Memoization

Note

Not all elements of the table are actually needed to solve our problem.

```
import numpy as np
import math


def knapsack(w, p, C):
    n = len(w)
    DP = np.zeros((n + 1, C + 1))
    for i in range(1, n+1):
        for c in range(1, C+1):
            not_taken = DP[i-1][c]
            if w[i-1]  > c:
                taken = -math.inf
            else:
                taken = DP[i-1][c - w[i-1]] + p[i-1]

            DP[i][c] = max( not_taken, taken)
    #print(DP)
    return DP[n][C]
```

```
w = [4,2,3,4]
p = [10,7,8,6]
C = 9
print(knapsack(w,p,C))
```

	c									
i	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

5- w[n-2]
= 5 - 3

9- w[n-2]
= 9 - 3

Memoization

Memoization

Programming techniques that merge the tabular aspect of dynamic programming with the top-down approach of divide-et-impera

- Whenever we need to solve a sub-problem, we check in the table first, to see if the problem has already been solved in the past
  - If not, we compute the result and store it in the table
- We use the result stored in the table
- In any case, each subproblem is compute only once as in the bottom-up version

Memoized-knapsack using a table (np array)

$$DP(i,c) = \begin{cases} 0 & i = 0 \textbf{ or } c = 0 \\ -\infty & c < 0 \\ \max(DP(i-1, c-w[i]) + p[i], DP(i-1,c)) & \text{otherwise} \end{cases}$$

```
import numpy as np
import math

def knapsack_mem(w, p, C):
    n = len(w)
    DP = -np.ones((n+1, C+1))
    return knapsackRec(w, p, DP, n, C)

def knapsackRec(w, p, DP, i, c):
    if c < 0:
        return -math.inf
    if i == 0 or c == 0:
        #DP[i][c] = 0
        return 0
    if DP[i][c] < 0:
        #the solution has not been computed already!
        not_taken = knapsackRec(w,p,DP,i-1,c)
        taken = knapsackRec(w,p,DP,i-1,c-w[i-1]) + p[i-1]
        DP[i][c] = max(not_taken, taken)
    return DP[i][c]
```

-1 if value not computed yet

top-down

```
w = [4,2,3,4]
p = [10,7,8,6]
C = 9
print(knapsack_mem(w,p,C))
```

	c									
i	0	1	2	3	4	5	6	7	8	9
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	0	0	10	10	10	10	-1	10
2	-1	-1	7	-1	-1	10	17	-1	-1	17
3	-1	-1	-1	-1	-1	15	-1	-1	-1	25
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	25

very easy: we are implementing the formula above, with a top-down approach checking if we already computed intermediate solutions

Note: remember that NOT all elements of the table are actually needed to solve our problem.

Memoized-knapsack using a table (np array)

	c									
i	0	1	2	3	4	5	6	7	8	9
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	0	0	10	10	10	10	-1	10
2	-1	-1	7	-1	-1	10	17	-1	-1	17
3	-1	-1	-1	-1	-1	15	-1	-1	-1	25
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	25

- Table initialization
  - The initialization cost is equal to $O(nC)$
  - Applied in this way, there is no advantage in using the memoization technique
  - The real advantage is that it makes easy to translate the recursive formula into an algorithm
- Using a dictionary
  - Instead of using a table, we may use a dictionary
  - No pre-initialization is necessary
  - The execution cost is equal to $O(\min(2^n, nC))$

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n-1) + 1 & n > 1 \end{cases}$$

in the worst case is w[i] = 1

Memoized-knapsack using a dictionary

```
import math

def knapsack_mem(w, p, C):
    n = len(w)
    DP = dict()
    return knapsackRec(w, p, DP, n, C)

def knapsackRec(w, p, DP, i, c):
    if c < 0:
        return -math.inf
    if i == 0 or c == 0:
        #DP[(i,c)] = 0
        return 0
    if (i,c) not in DP:
        #the solution has not been computed already!
        not_taken = knapsackRec(w,p,DP,i-1,c)
        taken = knapsackRec(w,p,DP,i-1,c-w[i-1]) + p[i-1]
        DP[(i,c)] = max(not_taken, taken)
    return DP[(i,c)]
```

```
w = [4,2,3,4]
p = [10,7,8,6]
C = 9
print(knapsack_mem(w,p,C))
```

	c									
i	0	1	2	3	4	5	6	7	8	9
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	0	0	10	10	10	10	-1	10
2	-1	-1	7	-1	-1	10	17	-1	-1	17
3	-1	-1	-1	-1	-1	15	-1	-1	-1	25
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	25

Dictionary:
{(1, 9): 10, (1, 7): 10, (2, 9): 17, (1, 6): 10, (1, 4): 10, (2, 6): 17, (3, 9): 25, (1, 5): 10, (1, 3): 0, (2, 5): 10, (1, 2): 0, (2, 2): 7, (3, 5): 15, (4, 9): 25}

Longest common **subsequence**

Definition: subsequence

- A sequence P is a subsequence of T if P is obtained from T by removing one or more of its elements
- Alternatively: P is defined as the subset of indexes in $\{0, \dots, n-1\}$ describing the elements of T that are also in P
- The remaining elements are listed in the same order, although they do not need to be contiguous

Examples

- T = "AAAATTGA"
- P = "AAATA"

Note

The empty sequence is a subsequence of every sequence

Longest common subsequence (LCS)

Definition: common subsequence

- Given two sequence P and T, a sequence Z is a common subsequence of P and T if Z is subsequence of both P and T
- We write $Z \in \mathcal{CS}(P, T)$

P: ACAATACT
T: ATCAGTC

Z: ACA

Definition: longest common subsequence

- Given two sequence P and T, a sequence Z is a longest common subsequence of P and T if $Z \in \mathcal{CS}(P, T)$ and there is no other sequence W such that W is longer than Z ($|W| > |Z|$) and W is common subsequence of P and T ($W \in \mathcal{CS}(P, T)$).
- We write $Z \in \mathcal{LCS}(P, T)$

P: ACAATACT
T: ATCAGTC

Z: ACATC

Longest common subsequence (LCS)

Problem: LCS

Given two sequences P and T of length n and m, respectively, find either the length of the longest common subsequence or one of the longest common subsequences.

Examples:

P: ACAATAT
T: ATCAGTC
Out: 4

P: ATATATATAT
T: ATGATAAT
Out: 6

P: ATATATATAT
T: ATGATAAT
Out: 6

P: AAAAA
T: CTGCTC
Out: 0

Any ideas? Naive idea (“brute force”): generate all subsequences of P, all subsequences of T, compute the common ones and return the longest.

Problem: all subsequences of a sequence with length n are **2^n** (think about strings of n 0 or 1 : 1 means keep the character, 0 do not keep it…)
To check if a string is a substring of another one I need to read them both: O(m + n)

Computational complexity: $T(n) = \Theta(2^n(m+n))$

Longest common subsequence (LCS)

Prefix

Given a sequence P composed of the characters $p_1 p_2 ... p_n$, $P(i)$ will denote the prefix of P given by the first i characters, i.e.:

$$P(i) = p_1 p_2 ... p_i$$

Examples

- $P = \texttt{ABDCCAABD}$
- $P(0) = \emptyset$ (empty subsequence)
- $P(3) = \texttt{ABD}$
- $P(6) = \texttt{ABDCCA}$

Longest common subsequence (LCS)

Goal

Given two sequences P and T of length n and m, write a recursive formula $DP(i, j)$ that returns the length of the LCS of the prefixes $P(i)$ and $T(j)$.

$$DP(i, j) = \begin{cases} ? & \text{Base case} \\ ? & \text{Recursive case} \end{cases}$$

Longest common subsequence (LCS)

Goal

Given two sequences P and T of length n and m, write a recursive formula $DP(i,j)$ that returns the length of the LCS of the prefixes $P(i)$ and $T(j)$.

$$DP(i,j) = \begin{cases} ? & \text{Base case} \\ ? & \text{Recursive case} \end{cases}$$

Case 1:

Consider the two prefixes $P(i)$ and $T(j)$ such that their last characters are the same: $p_i = t_j$.

How would you compute $DF[i,j]$?

$$DF[i,j] = DF[i-1, j-1] + 1$$

Ex.

P : TACGCA
T: ATCGA

A is part of the LCS

Longest common subsequence (LCS)

Goal

Given two sequences P and T of length n and m, write a recursive formula $DP(i,j)$ that returns the length of the LCS of the prefixes $P(i)$ and $T(j)$.

$$DP(i,j) = \begin{cases} ? & \text{Base case} \\ ? & \text{Recursive case} \end{cases}$$

Case 2:

Consider the two prefixes $P(i)$ and $T(j)$ such that their last characters are different: $p_i \neq t_j$.

How would you compute $DF[i,j]$?

Hint: either p_i or t_j are useless for the LCS.

$$DF[i,j] = max(DF[i-1,j], DF[i,j-1])$$

Ex.

P : TACGC
T: ATCG

either C or G is useless (removing C seems the most reasonable choice)

Longest common subsequence (LCS)

Goal

Given two sequences P and T of length n and m, write a recursive formula $DP(i,j)$ that returns the length of the LCS of the prefixes $P(i)$ and $T(j)$.

$$DP(i,j) = \begin{cases} ? & \text{Base case} \\ ? & \text{Recursive case} \end{cases}$$

Base cases:

What if i = 0 or j = 0?

$$DF[i,j] = 0$$

Ex.

P : TACGC
T:

length of LCS is 0

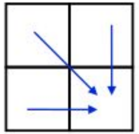
Putting it all together:

$$DP(i,j) = \begin{cases} 0 & i = 0 \textbf{ or } j = 0 \\ DP(i-1, j-1) + 1 & i > 0 \textbf{ and } j > 0 \textbf{ and } p_i = t_j \\ \max\{DP(i-1,j), DP(i,j-1)\} & i > 0 \textbf{ and } j > 0 \textbf{ and } p_i \neq t_j \end{cases}$$

LCS: example

$$DP(i,j) = \begin{cases} 0 & i = 0 \textbf{ or } j = 0 \\ DP(i-1, j-1) + 1 & i > 0 \textbf{ and } j > 0 \textbf{ and } p_i = t_j \\ \max\{DP(i-1, j), DP(i, j-1)\} & i > 0 \textbf{ and } j > 0 \textbf{ and } p_i \neq t_j \end{cases}$$

P: CTCTGT
T: ACGGCT



arrows specify where the values come from

i \ j		0	1	2	3	4	5	6
			C	T	C	T	G	T
0		0	0	0	0	0	0	0
1	A	0	↓0	↓0	↓0	↓0	↓0	↓0
2	C	0	↘1	→1	↘1	→1	→1	→1
3	G	0	↓1	↓1	↓1	↓1	↘2	→2
4	G	0	↓1	↓1	↓1	↓1	↘2	↓2
5	C	0	↘1	↓1	↘2	→2	↓2	↓2
6	T	0	↓1	↘2	↓2	↘3	→3	↘3

result in DP(n,m)

Memoized LCS

$$DP(i,j) = \begin{cases} 0 & i = 0 \textbf{ or } j = 0 \\ DP(i-1,j-1)+1 & i > 0 \textbf{ and } j > 0 \textbf{ and } p_i = t_j \\ \max\{DP(i-1,j), DP(i,j-1)\} & i > 0 \textbf{ and } j > 0 \textbf{ and } p_i \neq t_j \end{cases}$$

```python
def LCSrec(P,T,DP, i,j):
    if i == 0 or j == 0:
        return 0
    if (i,j) not in DP:
        if P[i-1] == T[j-1]:
            DP[(i,j)] = LCSrec(P,T, DP, i-1, j-1) + 1
        else:
            DP[(i,j)] = max(LCSrec(P,T, DP, i-1,j),
                            LCSrec(P,T, DP, i,j-1)
                            )
    return DP[(i,j)]

def LCS(P,T):
    n = len(P)
    m = len(T)
    D = dict()
    LCSrec(P,T,D,n,m)
    print(D)
    return D[(n,m)]
```

```python
T = "CTCTGT"
P = "ACGGCT"

print(LCS(P,T))
```

i \ j		0	1	2	3	4	5	6
			C	T	C	T	G	T
0		0	0	0	0	0	0	0
1	A	0	↓0	↓0	↓0	↓0	↓0	↓0
2	C	0	↘1	→1	↘1	→1	→1	→1
3	G	0	↓1	↓1	↓1	↓1	↘2	→2
4	G	0	↓1	↓1	↓1	↓1	↘2	↓2
5	C	0	↘1	↓1	↘2	→2	↓2	↓2
6	T	0	↓1	↘2	↓2	↘3	→3	↘3

DP: {(1, 1): 0, (1, 2): 0, (1, 3): 0, (1, 4): 0, (2, 3): 1, (2, 4): 1, (2, 1): 1, (2, 2): 1, (3, 1): 1, (3, 2): 1, (3, 3): 1, (3, 4): 1, (4, 5): 2, (4, 1): 1, (4, 2): 1, (4, 3): 1, (4, 4): 1, (5, 3): 2, (5, 4): 2, (5, 5): 2, (6, 6): 3}

Result:
3

Memoized LCS: where is my string?

j / i		0	1	2	3	4	5	6
			C	T	C	T	G	T
0		0	0	0	0	0	0	0
1	A	0	↓0	↓0	↓0	↓0	↓0	↓0
2	C	0	↘1	→1	↘1	→1	→1	→1
3	G	0	↓1	↓1	↓1	↓1	↘2	→2
4	G	0	↓1	↓1	↓1	↓1	↘2	↓2
5	C	0	↘1	↓1	↘2	→2	↓2	↓2
6	T	0	↓1	↘2	↓2	↘3	→3	↘3

```python
def subsequence(DP,P,T,i,j):
    if i == 0 or j == 0:
        return []
    if P[i-1] == T[j-1]:
        S = subsequence(DP,P, T, i-1, j-1)
        S.append(P[i-1]) #or T[j-1]
        return S
    else:

        if DP[(i-1,j)] > DP[(i,j-1)]:
            return subsequence(DP,P,T, i-1, j)
        else:
            return subsequence(DP,P,T,i, j-1)

def LCSrec(P,T,DP, i,j):
    if i == 0 or j == 0:
        return 0
    if (i,j) not in DP:
        if P[i-1] == T[j-1]:
            DP[(i,j)] = LCSrec(P,T, DP, i-1, j-1) + 1
        else:
            DP[(i,j)] = max(LCSrec(P,T, DP, i-1,j),
                            LCSrec(P,T, DP, i,j-1)
                            )
    return DP[(i,j)]

def LCS(P,T):
    n = len(P)
    m = len(T)
    D = dict()
    for i in range(n+1):
        D[(0,i)] = 0
    for j in range(m+1):
        D[(j,0)] = 0


    LCSrec(P,T,D,n,m)
    return subsequence(D,P,T,n,m)
```

```python
T = "CTCTGT"
P = "ACGGCT"

print("LCS: {}".format("".join(LCS(P,T))))
```

```
LCS: CCT
```

travel back up to build the substring...

Longest common subsequence (LCS)

What is the computational complexity of `subsequence()`?

$$T(n) = O(m + n)$$

we “consume” one element of either of the two sequences at each step

What is the computational complexity of `LCS()`?

$$T(n) = O(mn)$$

that is the size of the matrix

Automatic memoization in python

```python
import time

def fib(n):
    if n<2:
        return 1
    return fib(n-1) + fib(n-2)

s=time.time()
print(fib(45))
e=time.time()
print("elapsed time: {:.3}s".format(e-s))
```

```
1836311903
elapsed time: 3.04e+02s
```

```python
##Automatic memoization in python

from functools import wraps

def memo(func):
    cache = {} # Stored subproblem solutions
    @wraps(func) # Make wrap look like func
    def wrap(*args): # The memoized wrapper
            if args not in cache: # Not already computed?
                cache[args] = func(*args) # Compute & cache the solution
            return cache[args] # Return the cached solution
    return wrap # Return the wrapper


@memo
def fib(n):
    if n<2:
        return 1
    return fib(n-1) + fib(n-2)

s=time.time()
print(fib(45))
e=time.time()
print("elapsed time: {:.3}s".format(e-s))
```

```
1836311903
elapsed time: 0.000436s
```

Exercise: palindrome

A string is said palindrome if it reads indetically if read from left to right and from right to left.
Write an algorithm that returns the minimum number of characters to be inserted in a string to make it palindrome.
For example, input: “casacca”:

- $n = 7$ caratteri: “casaccaACCASAC”
- $n = 6$ caratteri: “casaccaCCASAC”
- $n = 3$ caratteri: “casaccaSAC”
- $n = 2$ caratteri: “ACcasacca”

Please note that characters may be inserted in the middle of the string as well; for example, “anta” $\rightarrow$ “antNa”.

Exercise: palindrome

- If the string $s = \mathrm{a}s'\mathrm{a}$ has the same initial and final character "a" then:

$$f(s) = f(s')$$

- If the string $s = \mathrm{a}s'\mathrm{b}$ has two different initial and final characters:
  - add a b character at the beginning or a a character at the end
  - count this added character as an insertion
  - consider the two subproblems given by the first and last (equal) characters removed
  - choose the minim

$$f(s) = \min\{f(\mathrm{a}s'), f(s'\mathrm{b})\} + 1$$

Exercise: palindrome

$$DP(i,j) = \begin{cases} 0 & i \geq j \\ DP(i+1, j-1) & i < j \wedge s[i] = s[j] \\ \min(DP(i+1,j), DP(i,j-1)) + 1 & i < j \wedge s[i] \neq s[j] \end{cases}$$

Exercise: palindrome

$$DP(i,j) = \begin{cases} 0 & i \geq j \\ DP(i+1, j-1) & i < j \wedge s[i] = s[j] \\ \min(DP(i+1,j), DP(i,j-1)) + 1 & i < j \wedge s[i] \neq s[j] \end{cases}$$

```python
def palindrome(in_str):
    DP = dict()
    out = palindrome_rec(DP, in_str)
    #print(DP)
    return out

def palindrome_rec(DP, in_str):
    if len(in_str) < 2:
        DP[in_str] = 0
        return 0
    else:
        if in_str[0].upper() == in_str[-1].upper():
            if in_str[1:-1] in DP:
                return  DP[in_str[1:-1]]
            else:
                DP[in_str] = palindrome_rec(DP, in_str[1:-1])
        else:
            begin_add = palindrome_rec(DP, in_str[-1] + in_str)

            end_add = palindrome_rec(DP, in_str + in_str[0].upper())
            if begin_add >= end_add:
                DP[in_str] = end_add + 1
            else:
                DP[in_str] = begin_add + 1
            #DP[in_str] = min(begin_add, end_add) + 1
    return DP[in_str]


input = "casacca"
print(palindrome(input))

input = "anta"
print(palindrome(input))
```

```
2
1
```

Shortest common supersequence

- A is a supersequence of B if B is a subsequence of A
- The shortest common supersequence (SCS) of P, T is the shortest sequence that is supersequence of both P, T
- SCS problems appear in the sequencing projects, when the genome of the individual is broken into several pieces. These pieces are sequenced individually, and then the whole genome is constructed as the shortest common supersequence of the sequences
- For the more general problem of finding a string S which is a supersequence of a set of strings S_1,S_2, . . . ,S_k, the problem is NP-Complete

problems for which there is no polynomial time algorithms known. IF there was, then all NP problems would be solved polynomially