Scientific Programming: Algorithms (part B)

Programming paradigms

Luca Bianco - Academic Year 2019-20
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

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

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

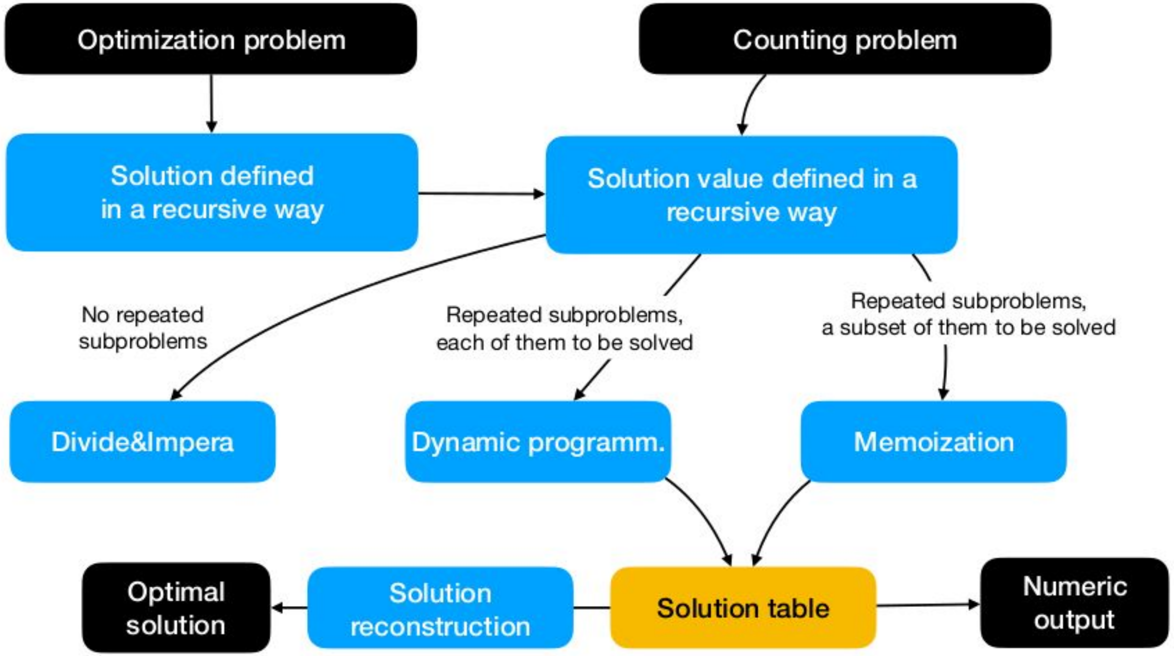
Backtrack

- 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



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

2xn

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

2xn

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

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

What is the complexity of dominoes?

```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
```

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

<u>Theorem not seen:</u>

Linear recurrences with constant order:

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

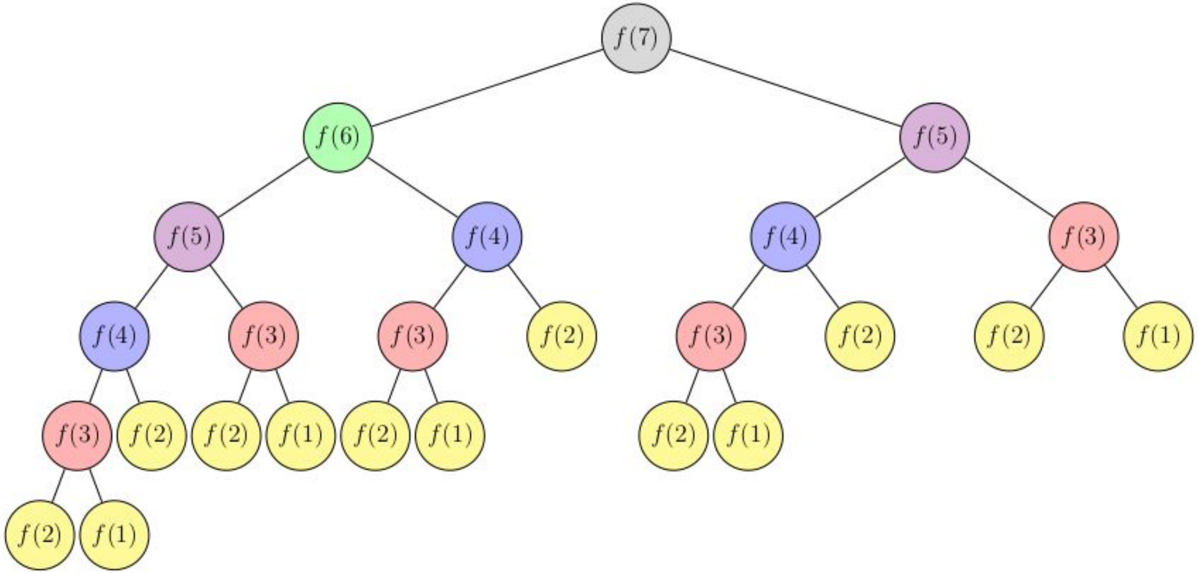
$$T(n) = \Theta(2^n)$$

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

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

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]
```

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

Watch out: the Fibonacci's number grows exponentially!

How many bits are needed to store $F(n)$?

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

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

How many bits are needed to store $F(n)$?

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

complexity seen before needs to be multiplied by 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()	$O(n^2)$	$O(n)$

```
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

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()	$O(n^2)$	$O(n)$

```
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 $S \subseteq \{1, \dots, n\}$ 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: $\{1, 4\}$

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 $HV(0) \ldots HV(n-1)$ 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 $HV(0) \ldots HV(n-1)$ values

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

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

- What happens if I accept its donation?

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

- How can I choose between the two cases?

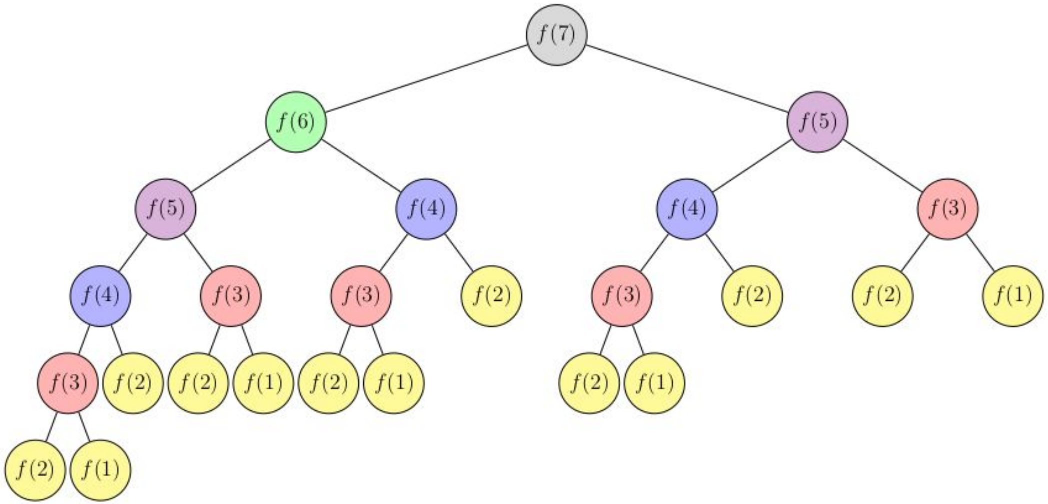
$$max(HV(i-1), HV(i-2) + D[i])$$

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		10	5	5	8	4	7	12
DP	0	10	10	15	18	19	25	31

i	0	1	2	3	4	5	6	7
D		10	1	1	10	1	1	10
DP	0	10	10	11	20	20	21	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 $DP[i] = DP[i-2] + D[i-1]$, house i has been selected

Build `solution(i)` recursively as:

`solution(i-2)` + index `i`
`or`
`solution(i-1)`

Iterative solution

```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
```

```python
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?

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

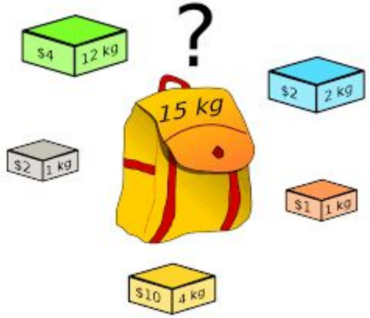
What is the complexity of hate_ville?

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

Exercise:

write hateville with S(n) = O(1)

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}

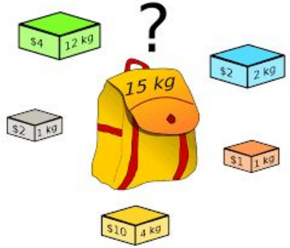
Item id	**1**	**2**	**3**
Weight	10	4	8
Profit	20	7	15

$C = 12$

S = {2,3}

Design an algorithm to solve the Knapsack problem

Knapsack



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

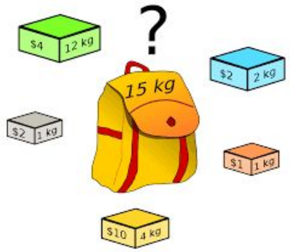
Given a knapsack with capacity C and n items characterized by weights w and profits p, we define $DP(i, c)$ as the maximal profit we can obtain from the first i items in a knapsack of capacity c.

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

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?

$$DP(i, c) =$$

The capacity and profit do not change

- What happens if you take it?

$$DP(i, c) =$$

The capacity and profit do not change

Exercises

DNA sequence comparison

Problem

Given two DNA sequences, find how "similar" they are.

Examples

- One substring of the other?
  `CCTT` $\subseteq$ `AGACCCTTAA`
- Edit distance:
  `AGACCCTTAA` can be changed into `AGACTCTTAA` by substituting a `T` with a `C`
- Longest common subsequence:
  the longest common subsequence of `TCGCA` and `TTGCCA` is `TGA`

Longest common subsequence

Problem

Given two DNA sequences, find how "similar" they are.

Examples

- One substring of the other?
  CCTT $\subseteq$ AGACCCTTAA
- Edit distance:
  AGACCCTTAA can be changed into AGACTCTTAA by substituting a T with a C
- Longest common subsequence:
  the longest common subsequence of TCGCA and TTGCCA is TGA