$$T(n) = 4 \cdot T(n/2) + O(n)$$

$$\left.\begin{array}{l} d = n^{\log_2 4} = n^2 \\ c = 1 \end{array}\right\} O(n^2)$$

$$(a_2 b_1 + a_1 b_2) \cdot 10^{n/2}$$

$$(a_1 + a_2)(b_1 + b_2) = a_1 b_1 + a_1 b_2 + a_2 b_1 + a_2 b_2$$

$$a \cdot b = a_2 b_2 + a_1 b_1 \cdot 10^n + \left((a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2\right) \cdot 10^{n/2}$$

But — are the same

So:

$$T(n) = 3 \cdot T(n/2) + O(n)$$

$$\left.\begin{array}{l} d = \log_2 3 \\ c = 1 \end{array}\right\} O(n \cdot \log_2 3)$$

are very small -> so it still takes time

-> In the Median String Problem -> we can compute the total hamming distance of the word => if the partial is already > than the best seen so far, continuing to explore useless.

↳ B&B MEDIAN SEARCH

è praticamente l'opposto.

If the prefix distance > ~~bst~~ best distance:
cut

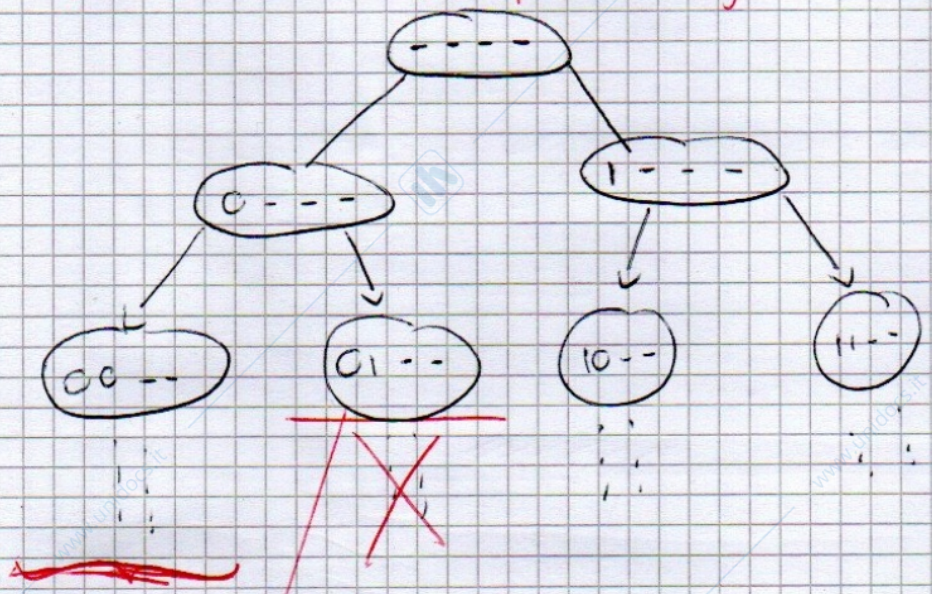
else:
I can try to see if I get a better hamming distance (shorter)

ofc here not sure to find a better one but I cannot cut.

· NOW WE APPLY THIS TO THE KNAPSACK PROBLEM: (Just to give an intuition)

- $\{1, \dots, n\}$; $\{P_1, \dots, P_n\}$; $\{w_1, \dots, w_n\}$; T

objects ~~weights~~ prices weights



if already for the first choice we're exceeding the max weight

we are not interested in knowing how much it exceeds the max weight, we can simply cut.

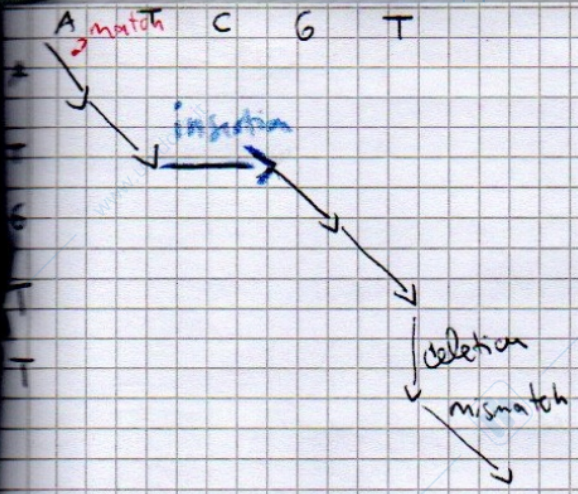
-> one band we have is P -> think a better another one.

P | w

prezzi | peso

-> If we're not looking for the best solution -> a good can work.

it can become much faster.



con colori in più e molto chiaro

- it gives us info about the path through that grid

bc we're summarizing all the possible ways of aligning the first & 2nd string

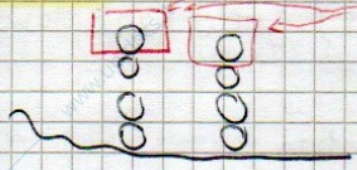
for establishing which alignment is better, we need a scoring function

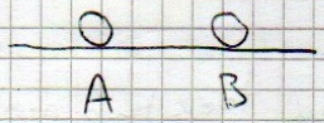
= computing the best path is equivalent to compute the best alignment in the grid.

A function that assigns ≠ scores to the paths → defining a scoring function is equivalent to assign weights to the path.

[redacted]

MOVES: take one rock or 2 rocks, one of each from each pile.

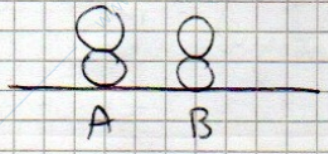




→ you need to play a strategy so that you'll be the last one to take the rocks.

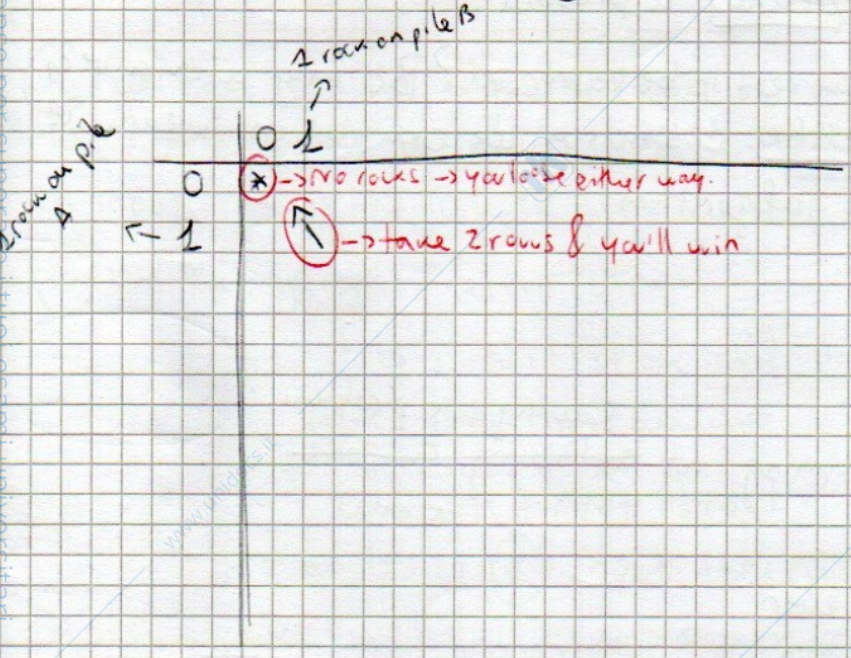
- Let's say we have 10 rocks for each pile. If Alice starts, if we assume that Bob plays at his best, does Alice have a strategy for always being the winner?

- Simple case:



→ 2 options for Alice:
① takes 2 rocks → Bob wins
② takes 1 rock → Alice wins

in realtà T in slide è libro il 2+2 gioco vinto da Bob.



A

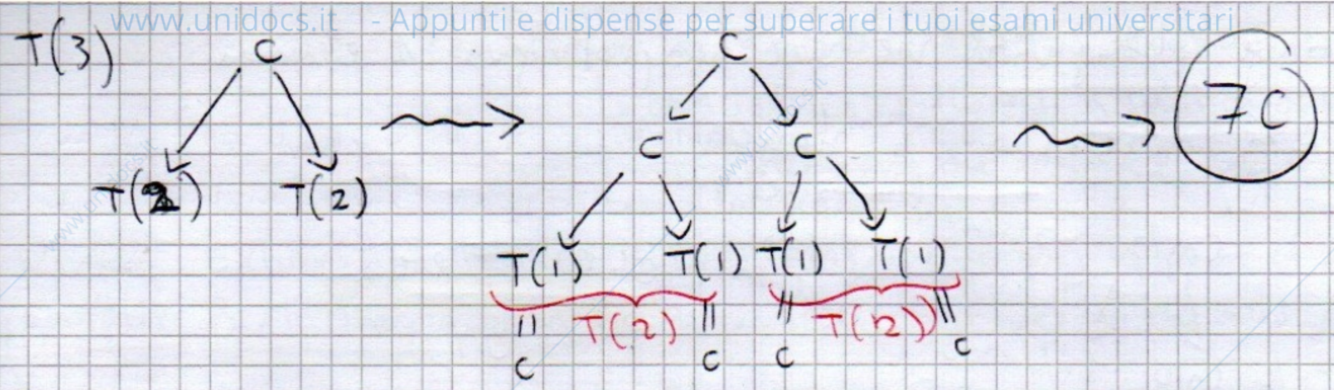
- The point of all this is that in simple cases we can come up w/ a situation, BUT if problems are more difficult we need a strategy.
  ↳ that's why we want to invent solutions for our problems.

- Combinatorial problem → def → slide.

ex. For us a problem is a set of pairs → made by input & output

ex. SUMMING UP 2 NUMBERS:
(x, y) → x is the INPUT
→ y is the OUTPUT

→ for ex. $(\{2,3\}, 5)$ $(\{1,4\}, 5)$ →



! 3c & 7c are just the n° of nodes of trees we have.

$2^n - 1 \Rightarrow$ # NODES.

Perhaps:

$T(n) \leq (2^n - 1) \cdot c$ — Is this TRUE? We verify by induction.

· BASE CASE: $n = 1$ → we know that $T(1) = c$

is $T(n) \leq (2^n - 1) \cdot c$ → yes $c \leq c$ ✓

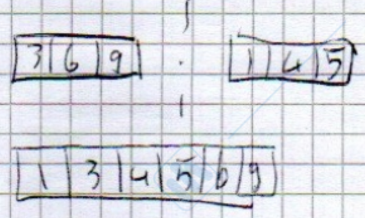
· INDUCTIVE STEP: (generic n)

$T(n) = 2 \cdot T(n-1) + c \leq 2 \cdot ((2^{n-1} - 1) \cdot c) + c$

$2^n \cdot c - 2c + c = 2^n \cdot c - c =$ ~~finisci copiare dagli appunti che carica~~

$(2^n - 1) \cdot c$ (OK)

MERGE-SORT(a)
 if len(a) = 1 : then | array of 1 el. → return it.
  return
 else:
  MERGE-SORT(first-half(a))
  MERGE-SORT(second-half(a))
  MERGE(the 2 halves)

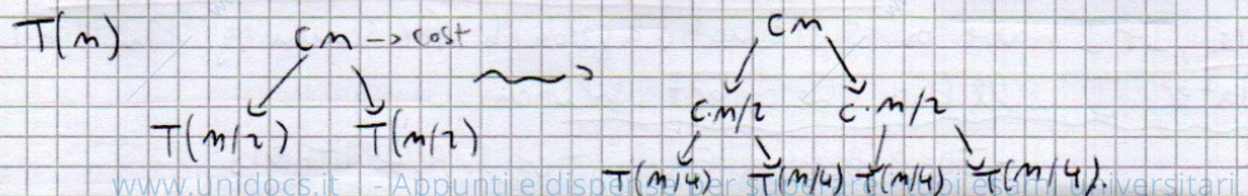


$T(1) = c$

$T(n) = 2 \cdot T(n/2) + c \cdot n$ — for each element we will look / chiamiamo ricorsivo con metà

| $2 \cdot T(n-1)$ ← Hanoi → similar |
| $2 \cdot T(n/2)$ → this case. |

→ does it make a difference



→

$f(n)$.

formal way to say it: $f(n)$ is $O(g(n))$ if there exists a point n_0, after which $f(n) \leq c \cdot g(n)$ (i.e. there's a constant c, which multiplied by $g(n)$, will make $g(n)$ dominate $f(n)$).

EX.

$f(n) = 3n^2 + 4n$ → $O(n^2)$, bc we looked at "the most powerful" eleme which dominates. BUT it is also dominated by n^3, bc $O(n^2) < O(n^3)$, it is even more powerful ⇒ $O(2^n)$, $O(n^3)$, $O(n^2)$ → all dominate $f($

BUT we're interested in the smallest upper bound. → upper bo

we want the closest possible to $f(n)$ → they bound from ab

$O(n)$ is NOT an upper bound.

Limit $f(n)$ fro above

- Big O notation is just a way for expressing this concept of upper bound.

! We could also bind functions from below.

$f(n) = \Omega(g(n))$ → lowerbound of $f(n)$

INTUITION: If n goes to infinity, $g(n)$ will be $< f(n)$.

DEF: $g(n)$ is a lower bound of $f(n)$ if there exists n_0, suc that $g(n)$ scaled by c, will be lower than $f(n)$.

$f(n) = 3n^2 + 4n$ → $\Omega(n^2)$ & also $\Omega(n)$ but NOT $\Omega(n^3)$.

- We can compare the 2 concepts:

$f(n)$ is $\Theta(g(n))$ if both $O(g(n))$ & $\Omega(g(n))$.

both a lower & upper bound

- HANOI TOWER → COMPLEXITY.

$T(1) = c$ ——→ (we assume they're the same)

(time for moving 1 disc)

$T(n) = 2T(n-1) + c$

$T(2)$ c ⇝ $3c$

$T(1)$ $T(1)$

c c

→

solution at polynomial time will never be found.

BOOLEAN FORMULA:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_3 \vee x_4 \vee x_5)$$

(and)

"x_1 or x_2 and not x_3 ..."

→ sol: $x_1 = x_2 =$ true $x_3 = x_4 = x_5 \rightarrow$ false

$$(\text{true} \vee x_2 \vee \neg x_3) \wedge (\underbrace{\text{not false}}_{\text{true}} \vee \text{false} \vee \text{false})$$

(or or) (or or)

TRUE so 1 enough so true ∧ true ✓

∴ gli altri casi nelle slide

It is an NP-hard problem bc there's no clear path to follow in order solve it.

---

0 - 1 KNAPSACK with integer weights → N-P hard Problem

- INPUT: n objects with integer weights

$\underbrace{w_1 \dots w_n}_{\text{band w on weight}}$ and $v_1 \dots v_n$ values

- OUTPUT: A subset $S \subseteq \{1, \dots, n\}$ s.t.

$\sum_{i \in S} w_i \leq W$ $\sum_{i \in S} v_i$ is maximum

$OPT_{w,i}$: optimal value when we consider only the objects $\{1, \dots, i\}$ and their weight is exactly w.

→ (A) object i is part of the optimal solution

(B) object i is NOT part of the optimal solution

(A) $OPT_{w,i} \longrightarrow OPT_{w-w_i, i-1}$ → = specific weight that we consider to build solution

(B) $OPT_{w,i} = OPT_{w,i-1}$ → $OPT_{w,i}$ can be obtained from $OPT_{w,i-1}$

$(\dots w \dots x_i) \longrightarrow (\dots w - w_i \dots)$

(x_i = I remove the i-th element)

$OPT_{w,i} \longrightarrow OPT_{w,i} - v_i$

ALGORITHM = a sequence of steps for solving a problem. It is still not a program. The program is the one implemented w/ a programming language. Basically also the pasta recipe is an algorithm.

↳ If many steps → humans are error prone. Instead a machine is faster, less errors prone.

→ PROGRAM = description of an algorithm in a language understandable by a computer

Qual è la differenza tra programma & algoritmo?

"sigma star"

Σ^* indicates the set of all finite sequences of elements from Σ

- alphabet → a defined set of symbols

If binary strings are $\Sigma = \{0, 1\}$

ε : ε 0101 0000 1 01101001 → same elements

- the edit distance $\delta(s, t)$ is the minimum number of changes necessary to turn s into t.

s: 0001010 t: 1000000

0001010 → 000010 → 000000 → 1000000

removed; changed to 0 (from 1); added 1

3 steps

$\delta(0001010, 1000000) \leq 3$

This only means that there's a way of transforming one into the other in 3 steps.

↳ at most 3! There could be other ways more efficient

- finite structures = domains w/ a limited (finite) number of symbols.

For this often these problems are called "Combinatorials"

05/03/2021

The Rock Pile Game

- 2 piles of rock.

No such space, you take another box & see if it fits.

FIRST-FIT (I)

$(U_1, \dots, U_n) \leftarrow (\emptyset, \dots, \emptyset)$ → we take n boxes & initialize them as empty.

for $i = 1$ to n — for each object $O(n)$

$j \leftarrow$ is the smallest index such that U_j has space for i → the box j has space for — $O(n)$

$U_j.add(i)$

return the PREFIX of $(U_1, \dots, U_n)$ of non-empty boxes.

- The overall complexity of FIRST-FIT is $O(n^2)$.
- For this algorithm we'll never have more than 1 less than half full box, bc even if we had 2 less than half full boxes, their contents would be put in the very same box.

→ LEMMA: FF returns an allocation in which AT MOST one box is full for less than 50%. → cool property considering that it does NOT waste space.

$$\bar{H} = \{U_j \mid \sum_{i \in S_j} s_i > 0.5\}$$

U_j ↓ allocation

$\bar{H}$ ↓ set of boxes full for more than 50%.

$$\sum_{U_j \in \bar{H}} \sum_{i \in U_j} s_i > \sum_{U_j \in \bar{H}} 0.5 \geq (FF(I) - 1) \cdot 0.5 = \frac{FF(I) - 1}{2}$$

$\sum_{U_j \in \bar{H}}$: for all boxes in $\bar{H}$

$\sum_{i \in U_j}$: for every object in each box we want to summesize.

$\sum_{U_j \in \bar{H}} 0.5$: $FF(I) - 1$ → n° of boxes returned by FIRST FIT. → that are at least for more than 50%

$\frac{FF(I) - 1}{2}$ → [illegible] of boxes [illegible] full [illegible]

$\sum_{U_j \in \bar{H}} \sum_{i \in U_j} s_i$ → size of the objects contained in each box of $\bar{H}$.

$$\sum_i s_i \geq \sum_{U_j \in \bar{H}} \sum_{i \in U_j} s_i$$

$\sum_i s_i$: this may be greater than that.

$$OPT \geq \sum_i s_i > \frac{FF(I) - 1}{2}$$

OPT ↓ optimal n° of

$$OPT > \frac{FF(I) - 1}{2} \longrightarrow 2OPT > FF(I) - 1 \longrightarrow 2OPT + 1 > FF(I) \longrightarrow$$

optimal value

$$2OPT \geq FF(I) \longrightarrow \frac{OPT}{OPT} \quad 2 \geq \frac{FF(I)}{OPT}$$

→ optimal value. → APPROXIMATION VALUE.

slide 21 -> spiega la complessità

-> Rule of thumbs to evaluate recurrences -> NOT EASY !!!ooo

Let's assume we have a divide-and-conquer algorithm

$$T(n) = a \cdot T(n/b) + f(n)$$

recurrence

$a \cdot T(n/b)$: a recursive calls, performs calls on size n/b

$f(n)$: for reconstructing the final result.

a recursive calls: $O(n^{\log_b a})$

$f(n)$: $O(n^c)$

INTUITION: these 2 pieces contribute for a piece of the complexity; if $O(n^{\log_b a}) > O(n^c)$, then complexity is $O(n^{\log_b a})$, ~~otherwise~~ & viceversa $O(n^c)$. If comparable, a combination of the two. $d = \log_b a$

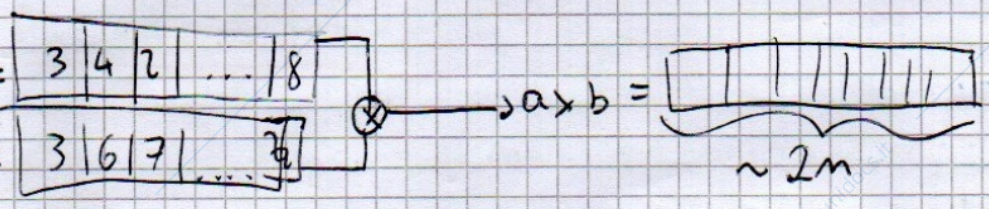
-> if $f(n)$ is $O(n^c)$ AND $c < d$, $T(n) = O(n^d)$

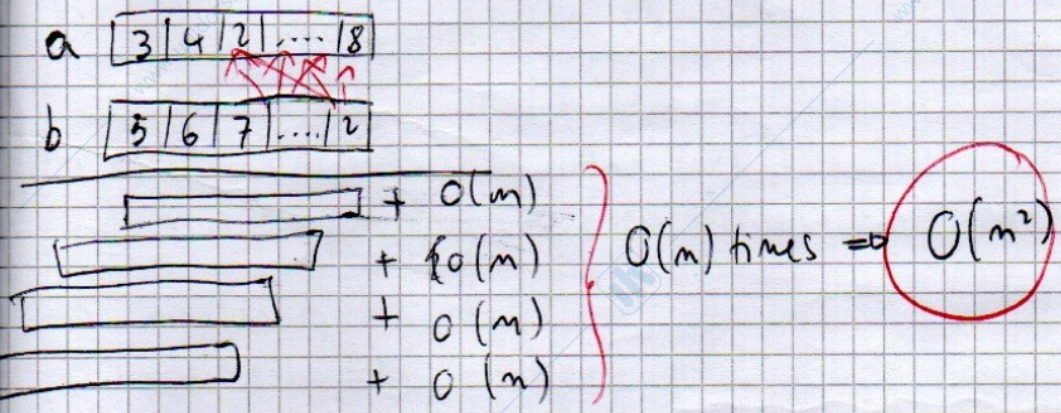
-> if $f(n)$ is $\Omega(n^c)$ and $c > d$, $T(n) = O(n^c)$

-> if $f(n)$ is $\Theta(n^d)$, $T(n) = O(n^d \cdot \log n)$

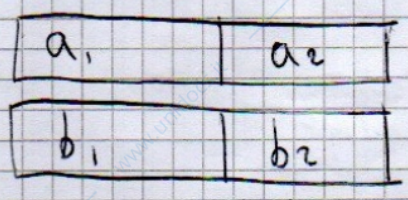
Problem:

$a, b \longrightarrow a \times b$





Now:



$$a = a_2 + a_1 \cdot 10^{n/2}$$

$$a \cdot b = (a_2 + a_1 \cdot 10^{n/2})(b_2 + b_1 \cdot 10^{n/2}) =$$

$$= a_2 \cdot b_2 + a_2 \cdot b_1 \cdot 10^{n/2} + a_1 b_2 \cdot 10^{n/2} + a_1 \cdot b_1 \cdot 10^{n}$$

4 moltiplicazioni

->

- we create a griglia (graph), where edges have a n° indicating how many attractions we can see from there (= by crossing that street)

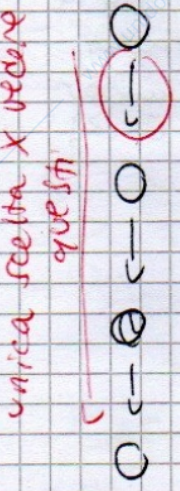
-> (0,0) the starting vertex -> source vertex (bc we cannot go upper nor more to left

-> the last -> the sink / target vertex (n,m)

- One drawback is that choices are done, if we're following a greedy approach, we would simply choose at each target the edges with highest weight, but it could be that this prevents us from seeing many attractions down to something

IDEA: we could build the best path by building best solutions for smaller parts of the grid

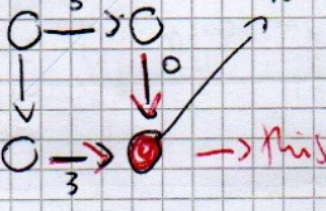
stessa cosa in cui →



- So we can get the best path from source to sink for first column & first row.

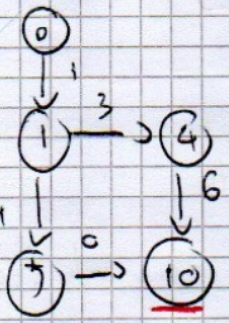
now:

If we focus on a vertex.



only thraugh these two:

so 1+3 or 3+c



3+c or 4+c

idea: let's focus on smaller instances & not directly from the first to the last.

↓

PROGRESSIVE APPROACH

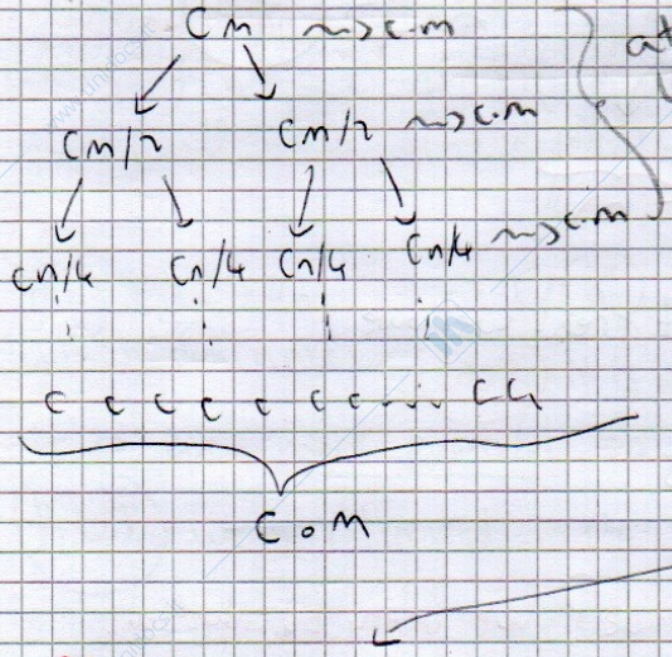
-> input for algorithm: 1 matrix w/ weights of the edges (orizontal) $\vec{w}$

1 matrix " " (vertical) $\vec{w}$

2 3 |-> weights ↑

4 5 |-> weights →

we continue splitting until we have arrays of just 1 element



$c \cdot n \cdot \#levels$

↓

GUESS:

$c \cdot n \cdot (\log n + 1)$

↓ since we're always dividing by 2.

PROOF:

$T(n) \; c \cdot n \cdot \log n + cn$

- BASE CASE:

$T(1) = c \leq c \cdot cn \cdot \log_2 n + cn = c \cdot n = c \rightarrow$ (yes)

- INDUCTIVE:

$T(n) = cn + 2T(n/2) = cn + 2\left(\frac{cn}{2} \cdot \log \frac{n}{2} + c\frac{n}{2}\right)$

$cn + cn \log \frac{n}{2} + cn = cn + cn(\log n - \log c) \; cn =$

$cn + cn \log - cn + cn = cn + cn \cdot \log n$

$O(n \cdot \log(n))$

much smaller than $O(n^2)$

15/03/2021

EXHAUSTIVE SEARCH ALGORITHMS

- Some problems in which we cannot avoid to look at the all possible search space; exhaustive search algorithms are the best candidates for these types of prob

↳ I don't know the solution but I do know that the solution is in the search space, If I have a systematic approach for scanning this spa Sooner or later, I will find the correct solution.

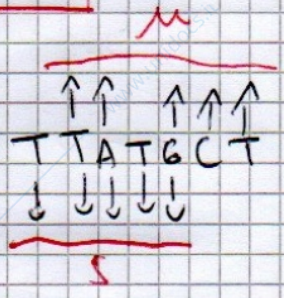
↳ IDEA BEHIND EXHAUSTIVE S. ALGORITHMS.

SHORTEST COMMON SUPSEQUENCE

27/04/21

$S = TTATG$

$\mu = TAGCT$ (two strings)



- the problem of largest common supersequence would be done by just combining the 2 strings, so it is kinda trivial.
  - i.e. attaching them.

- INPUT: S, μ strings
- OUTPUT: SCS -> a string t that is the shortest common supersequence

s' | prefixes of S & μ
μ' |

we know that $S = s_1 \dots s_{len(s)}$

$\mu = v_1 \dots v_{len(v)}$

t' SCS of s' & v'

=> a is the last character of t' -> 3 possible cases

(A) the character a is the last character of s' & u'

(B) the character a is the last character of s' but NOT of v'

$s''a = s'$ and $\mu''b = \mu$ $\quad b \neq a$

(C) the character a is the last character of μ' but not of s'

$s''b = s'$ and $\mu''a = \mu$ $\quad b \neq a$

if $t' = t''a$

(A) in this case $SCS(s'', \mu') = t''$

(B) " " $SCS(s'', \mu') = t''$ -> bc the last character of t' a is coming from s'.

(C) " " $SCS(s', \mu'') = t''$ ->

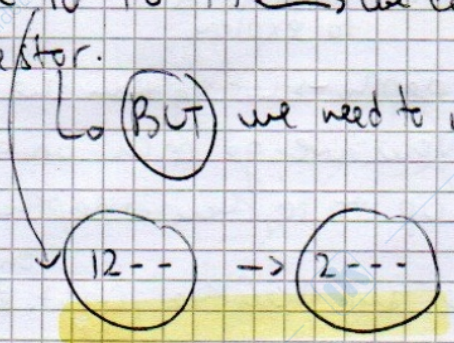
-> we have a recursion of optimality.

for each of these cases we can characterize from t' the optimal solution for the smaller cases: to produce t', we simply need to take the best among these cases to start from.

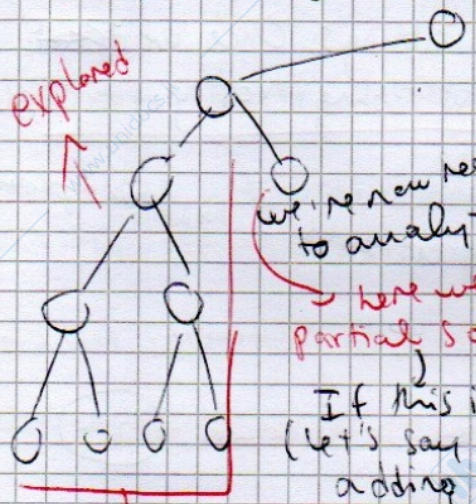
THE IDEA: we have seen that an optimal solution allows us to find

-> Uninteresting subtrees -> slide spiega bene

- For example let's assume we want to & bypass and go directly fra node 10 to 17 -> we can do as before, to a node to a sibling of ancestor.
  - BUT we need to know that exploring nodes 11 - 16 is useless.



- BYPASS -> simply as the 2nd part in next vertex but not applied to a leaf but a node in the middle.



If this is very low (let's say 10 & even by adding the best estimation score, the best alignment w/ what's below) & it is < 100, we can cut & go to the sibling of its ancestor ~~of the~~ length

ex. 100.

$$Score(S, i, DNA) + (t-1)\cdot \ell < bestScore$$

i = initial; $(t-1)\cdot\ell$ = best alignment. hit for everything

then we can safely avoid to explore that node

BRANCHANDBOUNDMOTIFSEARCH.

6 we perform an optimistic score
7 if optimistic < bestscore | we cut it, we don't ~~want~~ need to explore it
Bypass
else
NextVertex

↓
Even in the best case we're not able to do better.

- This is a branch & bound technique
  - = the optimistic score

- However ~~this~~ this is an improvement, but NOT amazing -> i.e. sometimes cut

HEURISTICS

\section{Conclusion}
In section 2 we have seen how to write some mathematical formulas.
Here we conclude that, blah blah \dots
↳ displayed "...".

The n° of sections is MANAGED BY LaTeX.

17/03/2021

optional -> assignment (1).
↳ using InsertionSort -> bc good for small lists.
↳ the function merging the 2 lists. If all the elements are < than the ones of the other, it is not necessary to do it linearly, but you can do it in constant time by appending one to the other

-> The fact that the most gains depends on (1) how the code is implemented (2) the hardware in Alessandro's code in InsertSort is between 20-70.

APPROACHES: -> for measuring Performances.

Analitical	Experimental	
• Asymptotic (it could tell us MergeSort performs better BUT NOT the threshold). ↓ gives us the result for BIG inputs.	Precise (bc we take timing of how long it takes the algorithm to perform)	-> we can choose lengths of lists / inputs needed
• Language & Machine Independent	Language & Machine Dependent	e.g. C++ has direct access to hardware ↑ faster. -> the language used the machine
• Worst case	Average case	

-> Some algorithms, once fixed the length of the input, even if these is change perform always the same (ex. ~~Insertion~~ Selection Sort)
Instead there are many algorithms that, even if fixed the size of input, real depend on what the input is (i.e. is the input sorted? Then perform faster. An example of this is InsertionSort).
↳ So we need to perform many tests, for understanding well how the algorithm performs. →

- If the pairs are a finite number -> we can just list them.
- BUT in cases like the previous one (summing up two numbers) we'd need to list infinite combinations
  - → we create a precise characterization.
    - How THE OUTPUT DEPENDS ON THE INPUT
    - we just specify a relationship, not how to obtain it.

the ROLE of the algorithm will be of finding an automatic way that executes & obtains the output

Ex. The rock problem took two numbers as input.

(n, m)

↳ here the characterization is wrt the example of summing up 2 numbers

Key concept: important to describe precisely a problem:
- what is the input
- what is the output

} why do we need to be so precise? Bc only with a very specific description, we can design a capable algorithm.

ALGORITHM

↳ Lo def. -> slide

↳ simply: sequence of steps aiming at ~~achieving a goal~~ solving a problem

- finite -> it cannot take forever
- unambiguous

- For describing algorithms we choose a language (unambiguous & precise enough)
  - we will use pseudocode -> it has a flavor of programming languages but easier for us to understand.
- We want algorithms that always give us an answer.

DIFFERENCE w/ PYTHON:

for loops

for i IF a to b → here a included & b also

B

↓ in Python a included & b excluded

would we need to arrive $b(\pi) = \sigma\rho$ $b(\pi)$ steps. But in the worst case scenario, we're not able to reduce by 1 at every step → you need to reverse the increasing to decreasing & then you can reduce by 1 ⇒ 2 steps each time.

So in the worst case scenario: $2 \times b(\pi)$.

→ Can the greedy approach be applied to

- we optimize only for the first 2 sequences, & then for all the remaining you follow your choice. → we find the most similar only for the first 2 sequences.

! no one knows the Approximation ratio

APPROXIMATION ALGORITHM → takes less time but doesn't find the optimal solution.

ALGORITHM:

lines 3-4 ⇒ we're considering the first 2 DNA sequences

line 12 ⇒ once fixed the first 2, all the other DNA sequences are taken & looked for similarity with the match between 1 & 2.

→ there's then slide on the complexity. → we pay efficiency w/ correctness

here concluded the part about this on the book. He now shows another algorithm

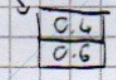
BIN PACKING PROBLEM → very frequent.

↳ we need to pack objects, each of which characterized by a ~~weight~~ size

- INPUT: $U = \{1, \dots, n\}$ objects

$i \longrightarrow s_i \in (0, 1]$ → size for each object.

Ex. $\{1, 2, 3\}$ $\{0.7; 0.4; 0.6\}$



- OUTPUT: $\{U_1, \dots, U_k\}$ $U_i \in U$ → we want to find a set of boxes such that:

$U_i \cap U_j = \emptyset \quad \forall i, j$ → one object only in 1 box, no more.

$\bigcup_i^k U_i = U$ → all objects are placed in a box

$\sum_{i \in U_j} s_i \leq 1 \quad \forall j$ → for each box no more than 1

- WORST case: for n objects, n boxes.

First fit algorithm:

- You put the object in the first box having enough space for it → If there

This is correct if the smaller solutions are correct.

In order to show it, we must show it satisfies a PROPERTY via ~~induction~~ inductiai:

1. Identify the property → the most difficult part, bc it may not be directly to algorithm correctness.
2. Show that it is satisfied in the base case
3. Assume that it is satisfied in all cases until "n-1"th one (inductive hypothesis)
4. Show that it is satisfied for the n^{th} case, based on the correctness of the inductive hypothesis

ex. with hanoi tower → property: the algorithm correctly moves the top most n disks from frompeg to topeg

This works easily for the base case but doesn't for the inductive case, it has too little preconceptions so we need a stronger property.

We add "if all disks in all pegs are correctly stacked and the topmost n disks of frompeg are smaller than the top disks, anyway, in all other pegs".

↓

fibonacci problem — input: an integer n

output: the n^{th} fibonacci number $f_n = f_{n-1} + f_{n+2}$,

considering $f_n = f_2 = 1$

This can be solved through a recursion or in an iterative way

its correctness is simply proven, it uses only the definition. This version is less efficient, it would take more time.

to prove its correctness we need a property (called invariant bc it's true before the first iteration, during the for loop and after it). In this case an array is created so that in ~~proprio~~ position n of the array we always find the n^{th} fibonacci number.

KNAPSACK PROBLEM → combinatorial. → fatto in programming

- we're a thief → we have a maximal amount of weight we can carry. We want to steal the maximum worth.
  ↳ MAXIMIZING the max worth w/ max weight I can carry

- INPUT: $\{1, \dots, n\}$; (n objects)
  $P_1, \dots P_n$; (prices)
  $w_1, \dots w_n$; (weight)
  $T = \max w$

- OUTPUT: $X \subseteq \{1, \dots n\}$ (a set of objects) SUCH THAT
  $\sum_{i \in X} w_i \leq T$ and $\sum_{i \in X} P_i$ is MAXIMUM
  (the sum of weights of taken objects does NOT exceed T.)

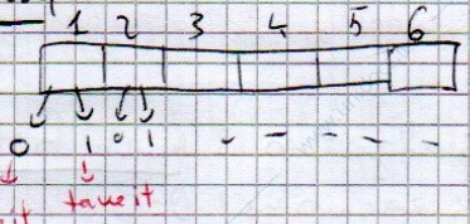
(A) Provide an algorithm
(B) Prove the algorithm complexity.
(C) Evaluate its complexity.

A

```
knapsack(w, P, T)
best_value ← 0
best_x ← {}
for each subset X ⊆ {1, ..., n}:
    if Σ_{i∈X} w_i ≤ T:
        current_value ← Σ_{i∈P} P_i   (current value = sum of values)
        if (current_value > best_value):
            best_value ← current_value
            best_x ← X
return best_x
```

B It is correct bc in the for we're looking at all possible possibilities, some point we'll find which is the best one.
↳ i.e. It is easy

C $\{1, \dots, n\}$ for the for cycle.



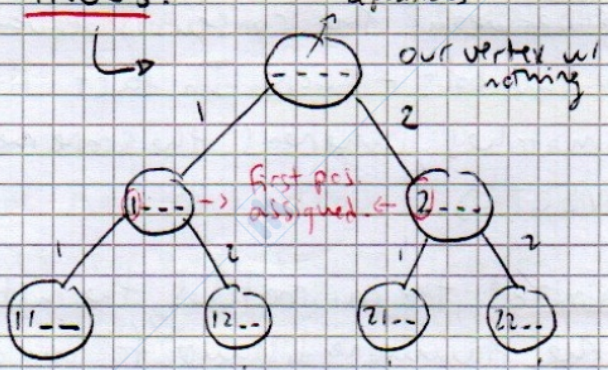
$2 \cdot 2 \cdot 2$ (6 times) → if $n = 6$.

$O(2^n \cdot c \cdot n)$ (for, for)
$O(2^n)$ → EXPONENTIAL → slow

Slow, when the input is big, it has to try many things.

· In this case we have the molecules (could be whatever symbol) that can be seen as tuples of numbers.

↳ TO LOOK IN A TUPLE-ORGANIZED SEARCH SPACE we use TREES.



} and so on until the 4 positions are occ

· We're interested in the last layer ⇒ the leaves of our tree.

→ Trees are very common in computer science.

→ the DEPTH → = how many layers we have → can also be called "HEIG

→ we interested in trees with a fixed & constant nº of children (can be 2, 3, 4...

→ BRANCHING FACTOR = the nº of children departing from a node.

· EXPLORING THE TUPLE SPACE:

If we look at the ~~leaves~~ leaves it is like a counting:

1122 → 1211 → reset.

this implemented

it is similar to ~~odometer~~ odometer.

or the same:

1222 → 2111

We have a code that given a leaf

NextLeaf(a, L, K) (combinations given a leaf → depth; branching factor)

(in the previous (ex) L = 4, K = 2)

```
1 for i ← L to 1          we increment symbols from the last one to 1.
2
3     if a_i < K          if the last symbols ... has this reached the last availa
4         a_i ← a_i + 1   you implement if no
5         return a
6     a_i ← 1             → you put 1 if has reached the (q)
7 return a
```

ALL LEAVES → just the idea that will be used in the next finding problems

- Although characterized in a ≠ way, these 2 problems are computationally equal

→ OUR AIM NOW: finding relations in this 2 problems.

- We look for the Hamming distance between the consensus sequence w/ the alignment matrix is all the symbols - score. In fact Hamming distance measures the n° of mismatches, whereas the score measures the matches. ⇒ HIGHLY CONNECTED.

↳ the consensus sequence minimizes the mismatches, the distance.

The consensus string minimizes the Hamming distance for an alignment matrix

→ We want to look for the string v minimizing the total distance with DNA ↳ OUTPUT of median string problem

$$\min_{v}\min_{s} d_H(v,s) = \min_{s}\min_{v} d_H(v,s) = \min_{s} d_H(w_s, s) = \min_{s}(l \cdot t - Score(s, DNA))$$

(v, s can be exchanged bc both minimization.)

(distanza va minimizzata in Hamming e massimizzata lo score)

(l·t is a constant so)

(we want to minimize a − operation ↳ maximize.)

$$= l \cdot t - \max_{s} Score(s, DNA)$$

↳ THIS IS EXACTLY THE MOTIF PROBLEM

apart from constants that we can ignore.

→ In the median string algorithm we use the same approach: we loop from AAA to TTT, so that we examine all possible combinations

↳ you optimize line by line → therefore this can be done in $O(n \cdot l \cdot t \cdot 4^l)$

→ We've produced 2 Bruteforce algorithms, both of which are exponential. → but in the last 4^l is the exponential & $4^l << n^t$

22/03/2021

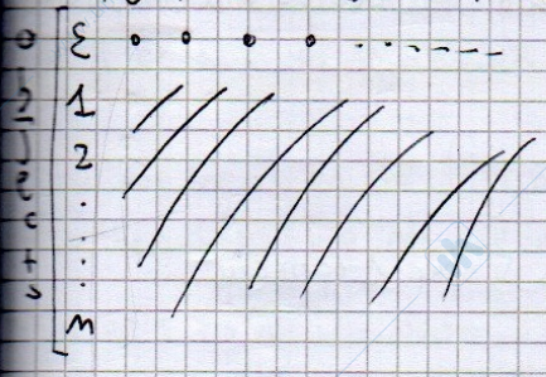
- Today we're going to see how to structure the algorithm, to avoid the exploration of useless paths in the search space.

↳ we have to find an ORDER for looking at the search space

TUPLES → a more formal way for saying arrays. →

$OPT_{w,i} = \max \begin{cases} OPT_{w,i-1} & \text{(B) not added} \\ OPT_{w-w_i,i-1} + v_i & \text{(A) added} \end{cases}$

w_i = weights



(A) → aggiungi valore e togli peso della capienza

$O(n \cdot W)$

↳ ? This is tricky: THIS IS THE TRICKY POINT: the complexity of this looks polynomial but it considers the whole W.

size to represent W is $\log_2 W$ bits

The number of bits is much smaller than the number W. So it would be $O(n \cdot 2^{|w|})$ → the number of bits required for representing the W (see numeric values)

It is said pseudopolynomial: polynomial in the value of the input BUT exponential in the size of the input. (required for representing it).

↳ It is not polynomial, bc we need to consider the size, NOT the value of the input

28/04/21

DIVIDE & CONQUER ~ divide et impera

a bit similar to the idea of dynamic programming.

we have seen an example of this, the mergesort algorithm.

The idea: if we're not able to compute the whole instance, we divide it.

MERGESORT(C) → array → we want to sort the elements of this

codice slide, chiaro

→ It is usually with the goal of gaining smth in terms of computation time or space usage.

1 n ← size etc

2
3 | base case → if length n=1 I just return it, one element always ordered

The interesting part is line 4 → recombining of the lists, bc we are ~~merging~~ merging two

WE DO NOT VERIFY ALL OF THEM SIMULTANEOUSLY, but one by one. So having them all in memory is a waste of space, bc we don't need to access them all at one time. A way for having in memory at the time the one needed can allow me to save a lot of space.

↳ trick for exploring a huge search space.

↓
we compute pieces of these data structures when needed.

keyword = yield

↳ ~~usable~~ in our (ex.) (slide) we can assume yield i means like return.

- it returns the value i; (BUT) it remembers at which point of the computation we are.

For ex. for i in rangeGenerator(0,4,1):

↳ dopo aver stampato 0, i è 1, quindi richiama la funzione rangeGenerator e glielo dice di iniziare da i a quel punto

we resume the computation from where we left.

- nell'esempio di 100'000'000 -> alla fine stampa così, ma the data structures are computed "on the fly" -> quindi non sta creando una lista con 100'000'000 ma crea un elemento alla volta

-> we can use generators w/ other generators => esempio rangeGeneratorsFiltered.

19/04/21

ASSIGNMENTS

↳ MARKING: 0-1-2-3-4

↳ (2) assignments -> da 0 a 4 -> da 0 a 8 punti nell'esame.

8 voti - assignments

22 voti -> scritto + orale

! sempre entrambi !!!

LEGEND:
0 -> missed submission / poor quality -> se non mandi 1 file (pdf/code)
1 -> solution has major issues - big mistakes / the quality of the report poor.
2 -> more or less acceptable -> solution can have some issues BUT ok report.
3 -> solution has minor issues / good exposition, might lack some details. ->

09/03/2021

How to find a property for correctness? Not easy, the trick: try to abandon the idea that we can find it at first => continuous search.

↓

very tricky to prove smth by induction.

we're dealing with proving correctness

-> For iterative problems we make use of invariants i.e. some variables are kept TRUE

↓

we ~~keep~~ see an example.

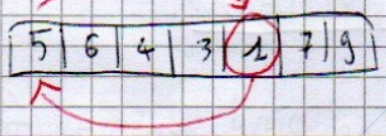
SELECTIONSORT(a,n) (we have an array, we scan it, looking for the smallest number once found, we take it & put in the first position)

1 for $i \leftarrow 1$ to $n-1$

2 $\quad a_j \leftarrow$ smallest element among $a_i, a_{i+1}, \dots a_n$.

3 $\quad$ swap a_i and a_j

4 return a



can be "enriched" by IndexOfMin, a function which helps in finding the minimum element.

INVARIANT property: once the initial part = and the elements of the first part are smaller than those in the remaining part are all sorted

$$P(i) = (a[1:i] \text{ is sorted}) \wedge (a[1:i] \leq a[i+1:n]) \wedge$$

(a is reordering of the original array) -> tells us that no loss, just exchange

COMPLEXITY : -> def. -> slide.

We need to do some assumptions (to simplify):

- Complexity is simply the n° of basic steps executed
- each step takes the same amount of time.

slice

-> It is just for us to have a rough idea of the time needed for that algorithm to perform.

ASymptotic Notation -> $f(n) = O(g(n))$ -> in this case what we mean is that when n grows a lot, g(n) dominates

So any reversal $\rho(i,j)$ can decreak at most $b(\pi)$.

$$d(\pi) \geq \frac{b(\pi)}{2}$$

DISTANCE of the permutation π from the identity.

minimal computation necessary for deleting all breakpoints

→ (if always best case).

ALGORITHM:

1

2 we consider all possible breakpoints & chose which reversal ensures to delete the most n° of breakpoints.

↳ ASSUMPTION: we're always capable of getting rid of breakpoints?

↳ it works only if we've the guarantee that at each step we can find a reversal reducing the n° of breakpoints

IS IT POSSIBLE? ←

THEOREM → slide

↳ PROOF: k is the smallest n° appearing in a decreasing step, in our case $k=3$, therefore:

- $k-1$ must be in an increasing strip (2)
- $k-1$ must be at the end of the strip.

↳ By reversing, $3(k)$ will be close to $2(k-1)$.

if we've a decreasing strip we're capable of decreak n° of breakpoints at least by 1 then

It may be the case that 0 1 2 → is on the right side & not as here:

0 1 2 7 6 5 8 4 3 9 | it doesn't change, it works the same.

IMPROVEDBREAKPOINTREVERSALSORT(π)

(line 5): in the else condition

- in case we have only increasing strip BUT w/ breakpoints, what we do, is to reverse an increasing, so that later we're sure we can find a reversal.

APPROX RATIO = 4 → we can say that in the worst scenario every 2 steps we're capable of decreasing the n° of breakpoints by 1.

$\frac{2b(\pi)}{d(\pi)}$ → our solution.

$d(\pi)$ → optimal sol.

we record that: $d(\pi) \geq \frac{b(\pi)}{2}$

$$AR \leq \frac{2b(\pi)}{\frac{b(\pi)}{2}} = 4$$ → it is constant (the error).

Bc $b(\pi)$ = n° of breakpoints present at the beginning; the possibilities are 2: there's a decreasing strip (we can find the reversal to reduce the n° of breakpoints by 1). If this were always the case, how many steps →

line: ↑ add 1 coin to the best n° of coins

→ if numCoins+1 < bestNumCoins

& I store it in a variable

→ Each time I need to recompute things however → waste of time

↳ IDEA: since going from bigger to smaller n° we have to recompute, we have to go from smaller to bigger numbers: ↘ like seen in the Fibonacci case

ex. I want 1 → 1 coin of 1

I want 2 → 2 coins of 1

I want 3 → I could have 3 coins of 1 BUT the best is 1 coin of 3

we use an array to memorize the best sol. ← so that we can have access to them & how everytime I need it I read it from memory ← trick: we're memorizing the optimal solution for the smaller cases

DPC change

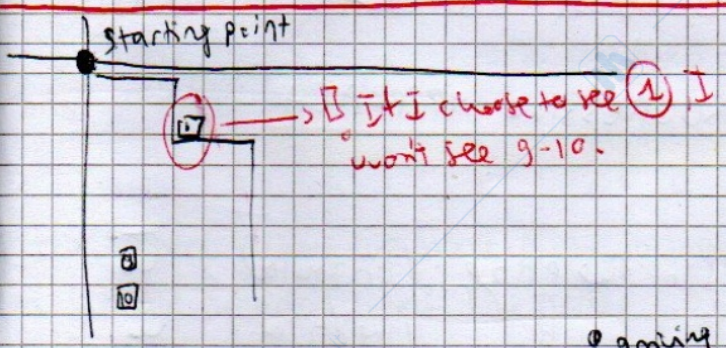
1. bestNumCoins$_0$ ← 0
2. per tutte le quantità di m.
3. ~~for~~
5. If m (the value considering now) ≥ c_i:

if bestNumCoins$_{m-c_i}$ + 1 < bestNumCoins$_m$ | if better of what seen up to now, we memorize it.

& return bestNumCoins$_m$.

PRICE: the memory → you memorize what's done up to now

THE MANHATTAN TOURIST PROBLEM



If I choose to see (1) I won't see 9-10.

- we want to traverse the city

↓

Constraints: - horizontal road only from left to right
- vertical only from north to south.

→ During our travel, we want to maximize the n° of attractions that we see. We need to find the best path to do so.

→

· We copy the subsequences in an alignment matrix. I look at it colum by column ⇒ I basically count how often the nucleotides appear in ea column.
↳ "PROFILE MATRIX"

→ the CONSENSUS STRING is formed by the most frequent symbols column column. ↳ we obtain the most similar to each of the substrings.

To measure how good this is we define a SCORE: the sum of the highe frequencies for each column. (In this case (42): it corresponds to the match it has with the alignment matrix).

the higher it is, the higher the nº of matc
& the higher it is, the lower the variance

· this is a deterministic procedure.

once we pick the starting positions, this table is determined & so is the consensus sequence.

BRUTEFORCEMOTIFSEARCH(DNA, t, n, l)
1 bestScore ← 0 we define this variable
2 for each $(s_1 \dots s_t)$ from $(1, \dots, 1)$ to $(n-l+1, \dots, n-l+1)$ → ossia ogni co
⋮

viene poi spiegata la complexity $O(t \cdot l \cdot n^t)$ | very slow
line 3 · line 2
↓
So other perspective

other perspective: → (we will reach a faster algorithm).

Median strings

↳ we get the Hamming distance → the nº of positions at which the (2 strings) dif

· We then define the total distance between a string v with a DNA matrix is the minimum distance that we can achieve by considering all the candidate starting positions

basically we're trying to find the string repeated more times w/ the small nº of variations

MotifFinding problem & MedianString problem are actually the same.
we want to find the positions w/ less variations in DNA. ↳ we want to find the string repeated often w/ less variation. →

Dynamic Programming

14/04/21

→ up to now → we dealt w/ EXHAUSTIVE SEARCH, which are good bc they give us correct results, BUT pretty slow & with high complexity.
↳ In real scenarios, NOT much used, (BUT) can be improved

→ Greedy algorithms → each time has to perform a choice, they tend to choose the solution looking the best at the moment.

DYNAMIC P.:

- IDEA: for building the optimal solution we partition the instances in sub-instances, we solve each of them in polynomial time
  - (ex.) the change problem

→ If in input we have N, the best solution can be obtained from a number n < N? If yes, we can use dynamic programming approach.

(ex) n = 77, if somebody tells us the best solution for 76, as there are no coins = 1, we can obtain the optimal for 77.
↳ intuition: usually optimal solutions are kinda recursive; we can build solutions starting at from optimal solutions of smaller cases

(ex.) $c = (1, 3, 7)$ (possible coin sizes)

if $(i_1 - 1, i_2, i_3)$ not optimal sol for 76 (we assume by contradiction) there would be a solution

$$J_1, J_2, J_3 < i_1 - 1 + i_2 + i_3$$

& by adding 1 we'd get:

$$J_1 + 1 + J_2 + J_3 \leq i_1 + i_2 + i_3$$

↳ contradicting i. (slide)

→ we will ask the best sol. for cases 76, 74 & 70 $(-1, -3, -7)$. We ask to someone else, we receive the best solutions for these 3 subcases we know that the best sol. for 77 will be adding 1 coin to some of these 3, by taking the best one.

- What does it mean to ask to someone else? A recursive call, we essentially launch the algorithm to a smaller instance.

- In alto a destra -> new project
- scrivi a sinistra e vedi a destra cosa puoi vedere.

-> You describe in LaTeX how to write -> sort of programming languages.

\documentclass[a4paper]{article}
- \documentclass: deve sempre essere messo
- [a4paper]: format
- {article}: Lo type of writing
-> meth sempre

\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
non ci frega cosa è -sneth.

\usepackage[english]{babel}
Lo definisci la lingua d'uso.

∇ LaTeX is able to automatically andare a capo

breaks the word -> so come suddivide in syllabus.

DEVE SAPERE LA LINGUA CHE SI USA

\usepackage{a4wide}
aggiunge un po' di spazi, viene meglio.

\usepackage{lmodern}

\usepackage{authblk}

TUTTE QUESTE METTILE, YOU DON'T CARE WHAT THEY ARE.

\title{An example document in \LaTeX}
\author{name}
\author{name}
\affil{DISI, University of Bologna, Italy}
\date{10 March 2021} %\date{\today} -> se non vuoi aggiungi \date()
\begin{document}

\maketitle

\section{Introduction}
This is an example of introduction.

\section{Body}\label{sec_body} -> in altre sezioni puoi richiamarlo \ref{sec_body}

It is easy to write Mathematics here
For example: -> se lasci linea empty va a capo, altrimenti dà solo spazio.

\[
\sum_{i=1}^{n} \frac{n(n+1)}{2};
\]

mostrerà:

$$\sum_{i=1}^{n} \frac{n(n+1)}{2}$$

lists that are sorted. -> So it can be done in linear time ! If not ordered it would take us quadratic time.

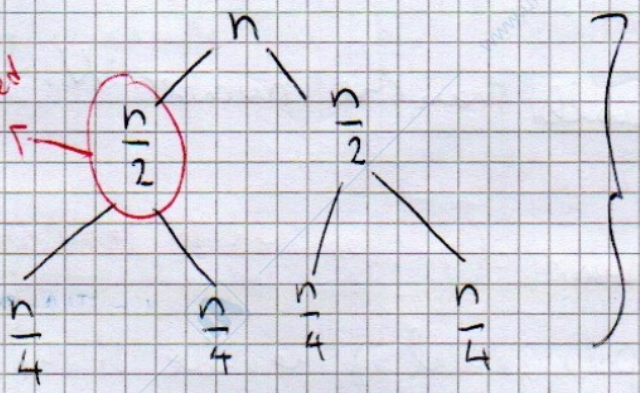
-> To be sure the complexity gains smth, we need to compute a complexity analysis

↓ For the recursion:

$T(1) = 1$ -> if size = 1 => takes simply 1 operation

$T(n) = \underbrace{2T(n/2)}_{\text{OR}} + cn$ -> time for merging the two.

-> A recursion tree is a tree depicting how execution is carried out.



-> The overall complexity for this is $O(n \log n)$

! Merge Sort is an optimal ~~problem~~ solution, bc there are mathematical proofs that show that there's no better complexity for ordering an array.

-> If we focus again on the global alignment problem: If we are interest in the n°, we need quadratic time bc we have to run but we can use linear space, bc only the last column is needed

-> If we want the actual alignment, can we do better than quadratic space? We need to exploit the trick that the score of the distance can be calculated in linear space.

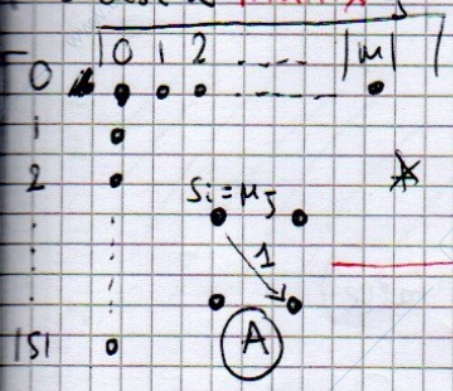
Clide: the complexity of PATH

-> We want to compute the alignment from source to sink (using a limited amount of space.); any path has to go through the middle column at some point. So also the best pa has to traverse it. However we don't know exactly when.

OUR AIM: try to compute where is the point in which the best path crosses

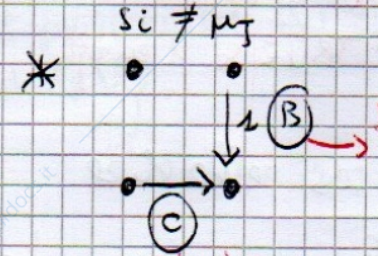
optimal solutions for the subcases.

-> As seen for previous problems, we may need an helper data structure, in this case a matrix



$c_{i,j}$ = length of the scs of a i-prefix of s & a j-prefix of μ.

-> Since the last character of s & μ is the same, we just add one more character in both.



ex.

$c_{i,0} = i$

$c_{0,j} = j$

$$c_{i,j} = \min \begin{cases} c_{i-1,j-1} + 1 & \text{if } s_i = \mu_j \\ c_{i-1,j} + 1 & \text{if } s_i \neq \mu_j \\ c_{i,j-1} + 1 & \text{if } s_i \neq \mu_j \end{cases}$$

Complexity: $O(|s| \cdot |\mu|)$ -> polynomial.

ex.

$s = TGC$

$t = TGC$ needs to be a superSequence for s & μ; an empty one can't be a superstring for μ but not for s!

$\mu = \varepsilon$ -> empty

-> The difference with exhaustive search is that in dynamic programming we do reuse things -> the work for computing sub-instances is reused. In exhaustive search scenario each time the optimality is checked from scratch.

-> The NP-hard problems -> for them at the current state of art, the only known solution is at esponential time. Up to now no polynomial time algorithm has been shown BUT still not proven that it is NOT possible to find a solution. (i.e. exhaustive search).

live in a limbo

to Security, cryptography are based on the common assumed idea that no

last content lecture

04/05/21

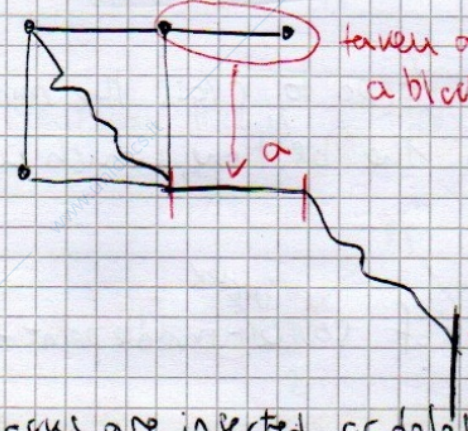
→ for the global alignment we've seen that we can reduce the space complexity; what we want to do today is to find out if we're capable of reducing the time complexity through Divide-and-conquer.

↳ For the global alignment still researches about if we can reduce the $O(n^2)$ complexity to a linear one!

→ But we can look at a similar problem, the BLOCK ALIGNMENT

→ Let's consider a string $v = v_1 \dots v_n$, a t block string is so if n is a multiple of t.

ex.



- a string $v = v_1 \dots v_n$ is a t-block string if n is a multiple of t. So we can divide it in $\frac{n}{t}$ blocks of length t.

! So the length of the strings in this problem are under this costrain!!

- t-blocks are inserted or deleted as a whole, when you align them, they don't have to be the same. So:

$$s_{i,j} = \max \begin{cases} s_{i-t,j} - \delta \text{ block} & \text{del} \\ s_{i,j-t} - \delta \text{ block} & \text{insert} \\ s_{i-t,j-t} + \beta_{i,j} & \text{align} \end{cases}$$

the recursion which allows us to fill the dynamic programming data structure

these values should be computed befor for all possible pairs of blocks / so that they are ready to be used in the matrix filling

→ A lookup table → where we store the values for aligning 2 blocks of length t. In this way we don't have to compute it everytime

! The values in the table must be stored in an alphabetical order, bc in this way we can perform binary search in the array (we don't use hash tables bc in the case of long & unbalanced hash clashes, we would need to perform a linear search in that list & we want to avoid this).

we put a strong bound in the worst case (in the worst case we take logarithmic time).

↳ (In the worst case, so not likely but we don't have the guarantee).

we guarantee that we remain in logarithmic time.

→ After this pre-computation we simply perform a change.

DECISION PROBLEM -> the solution of these is YES or NOT.
Lo can tell us if there's a solution or NOT.

OPTIMIZATION PROBLEM -> you look for the best, the optimal solution.

Usually EXHAUSTIVE SEARCH ALGORITHMS are from a complexity point of view very complex, BUT they give us the guarantee that we find the answer.
↳ in real life -> slow, so practically never used, BUT we're interested be it's an easy-idea method, which can constitute the base for further algorithms

RESTRICTION MAPPING

Lolui fa esempio con motorways -> in DNA same for finding restriction sites for restriction enzymes

Complete vs partial digest -> we're NOT 100% sure that the enzymes cut in all restriction sites -> usually they cut in ~~successive~~ consecutive points BUT we could obtain fragments with length 4, ignoring one site.
↳ what we obtain is cuts w/ ≠ lengths, not necessarily in 2 consecutive points. ↳ If length = 6 -> this tells us that there are 2 points at distance = 6, BUT it may be that these are NOT consecutive -> just a distance between 2 points (ex. 4-10)

- MULTISET = elements in a set can have multiplicity (ex. 2 has multiplicity 2, 3 also 2)
↳ (one in slide).

Partial Digest Problem

• the 2nd biggest element 8 tells us that there are 2 points whose distance is 8. So it could be a point in 2(2-10) or in 8(0-8). THIS IS THE IDEA -> we don't know which of the 2 pts is the right one, we will now test it.

INCREMENTAL STRATEGY

$L = \{2,2,3,3,4,5,6,7,8,10\}$ $\quad X = \{\}$

$L = \{2,2,3,3,4,5,6,7,8\}$ $\quad X = \{0, 10\}$ → I need to justify a distance of 10.

bidirezione ⇒ we encounter a decision point for 8 (distance) -> bc you can take it from left or right.

Then for 7 again:

from left:

$X = \{0,2,3,10\}$ but $3-2=1$

so $1 \in L$ not satisfied -> WRONG, I take the other. (the right)

Then again the same for distance (6).

IDEA: By looking at the biggest distance & based on that I have 2 possible eligible options -> I test them.

Going on, we could arrive at a point where none of left/right options satisfy the condition $\in L$ ⇒ THIS means that before we made the wrong choice (i.e. both were fine, we chose the wrong ⇒ WE NEED TO BACKTRACKING).

• BACKTRACKING -> If we arrived at a dead point, we need to go back to previous attempts to modify them. It goes hand-in-hand with exhaustive search algorithms.

PLACE -> algorithm. -> segue proprio questo ragionamento.

If L empty -> return ⇒ abbiamo finito.

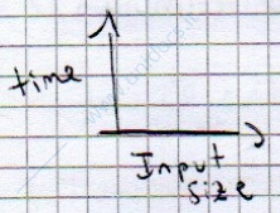
Remove y from X & add to plus $\Delta(y, X)$ to L

↳ stiamo togliendo y & lo raggruppiamo a quelli da analizzare.

▽ Experimentally, this algorithm is much faster than the previous ones (at maximum 2 options each time) BUT:

WORST CASE SCENARIO: again exponential, BUT in pratica it runs faster than the others →

Data fitting → experimental analysis



- Suppose we analytically know that the complexity is linear $O(n)$.
  ↳ How to find a linear function $f: \mathbb{N} \to \mathbb{N}$?

We can use a statistical approach: **least-square method**

INTUITION: We want to find a & b for the linear function.

We want to minimize the squared errors → Numpy & Scipy are libraries having it.

```
xdata = numpy.array([10.0, 20.0, 30.0, 40.0, 50.0])
ydata = numpy.array([7.3, 8.46, 10.2, 11.4, 13.08])
```

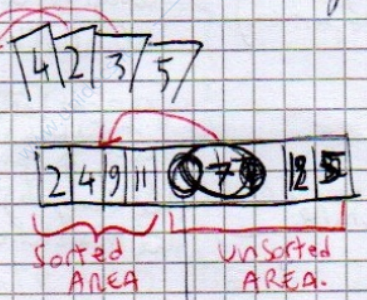
(time associated to input size)

the estimate of our curfit → Calcola l'algoritmo.

→ We will then plot it in LaTeX.
↳ we will use pgf

~~Write, compile & plotting~~

- Why in InsertionSort there's return & e capo l'altro comando?
  Let's assume we're given a set of cards.



focusing on the first element of the unsorted area & moving it in the right position

$$t = a \cdot x^2 + b \cdot x$$

$$4.1279e^{-6} \cdot x^2 + (-1.2548e^{-3}) \cdot x$$

- since in some we have 20 & 25 coins -> the better to have 40 would be 2x20 & not 1x25 + 1x10 + 1x5

↓ so

NOT CORRECT because the choices are optimal now BUT NOT always.

↳ it's a greedy algorithm -> when you have a choice point you take the greedy (= the best now).

In this case it's better to start from the trivial solution -> we consider all the possible (no logic) combinations of coin that reach M & find the best solution.

↳ BRUTEFORCECHANGE -> you don't go completely blind (in line 2 there is some logic)

↳ I focus on a small space: the combinations that do NOT go more than M

← we give an UPPER BOUND

↓

& since we go "blindly" there's NO change we miss any values

↓

EXHAUSTIVE SEARCH

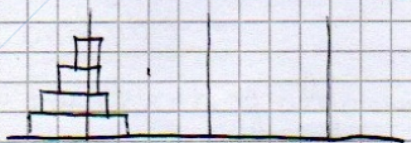
↳ it is correct bc I spot the result I want & note it down.

08/03/2021

RECURSION = when the algorithm calls itself (partitioning the input into smaller parts which can be solved, then recombine the solutions)

↓

HANOI PUZZLE → move all disks from peg 1 to peg 2, moving only one disk at the time without putting a bigger one on a small one.



Intuition: move the first n-1 disks on the 2nd peg and the last one on the 3rd. The solution with 3 disks is easy, if we do it with 4, we can reconduct (particolare) the problem to the solution with 3 disks.

- So, what is the base CASE? Part executed without (recursions) recursive calls, it is correct when we only have one disk.

unused peg ← 6 - from peg - to peg (mathematical fraction to determine which of the 3 is actually not used)

- What is the recursive case? part which resorts the recursive calls.

ALGORITHMS & DATA STRUCTURES

04/03/2021

- lectures will be held online -> lui è a Londra, insegna al King's College

2 MODULES:

1. THEORY -> 16 lectures of 2H
2. PRACTICALS -> "hands-on" -> we need to invent ways on how to solve problems

- ABSTRACTION -> understanding the structure of a problem from an abstract point of view, so that we can tackle the design.

We will to design how to solve problems -> we will have a context of bioinformatics

- Structural complexity -> the complexity of a problem, NOT of an algorithm.
  - to understand if
    - our algorithm is not fast bc of the implementation
    - the problem is intrinsicly long to solve, no much I can do.

The point is that we don't want only "an answer", we want the best one.

EXAMS

1) module: written (exercises) & oral (on how you reason)
2) exercises during the practicals -> 2/3 assignments -> 10/14 days to solve it.
   - scrivi soluzione & then short report

- We will see what problems & algorithms are, the complexity of algorithms (this is important for deciding the algorithm to be used).
- How to design algorithms -> following some design principles

- Algorithms are ways for solving problems.
- "Why should I learn Algorithmics?" -> Slide 7
  - Usually we encounter problems for which people have already provided solutions (you simply import) -> Sometimes, especially in difficult scenarios, it NOT like that -> you need to deal w/ it. ->

- There are $\neq$ sets of points X, whose ΔX is the same.
  - INTUITION: we could shift all the values by a $n \in$ (ex. 2), & the pairwise distance would be kept.
    - bc we're given 2 points, defining a distance but we're not told where these do start.

- With this algorithm we generate all the possible combinations $\binom{n-1}{n-2}$ So it
  - very slow.
  - $\binom{n-1}{n-2}$ = from a set of $n-1$ elements, need to take n
  - We can improve a bit the performance
    - Would it make sense to choose a point 1 in (1)? The correctness would be impacted (BUT) the point is that we're wasting time trying conditions that FOR SURE are wrong (bc distance = 2 lost)
      - So we can update the brute force algorithm, to produce another one. It is the same (BUT) difference in line 2, that the point must be contained in L.

$$\binom{|L|}{n-2} = O(n^{2n-4}) \quad \Big| \quad O(n^{n-2})$$

combinations I generate — still exponential — for the previous.

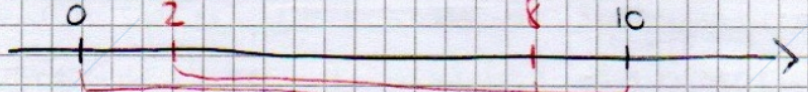
we choose this bc in principle $n \ll m$.

- Theirs example: $L = \{2, 998, 1000\}$
  - $n = 3$
  - $m = 1000$

- These algorithms are BOTH slow, even though the 2nd one is a bit less slow.

→ Before we started defining the extreme points (the initial ato) & $m = 10$. Then we said it would not make sense to put a point in 1 bc if so we'd observe a distance not present in the multiset input; last one can guide us in choosing the points.

what we can do: we ignore 10 (we already dealt w/ it) & we consider the 2nd bigger $h \in$ ⇒ i.e. (8) → Since the largest remaining distance is 8, there must be a point at distance 8 (0-8) but if



$L = \{10, 8, \ldots\}$ → biggest element → To Justify: there are 2 points of 10.

- In Python arrays are expressed as lists.
  - here elements accessed as elements in list. -> the index = 0 of the first element.

In pseudocode -> just indexing -> the index of first element = 1

Problem -> Change Problem

quarters -> 25 cents
dimes -> 10 cents
nickels -> 5 cents
pennies -> 1 cent

INPUT: M -> amount of money in cents.

OUTPUT: we want to give an amount of cents, whose sum is M but minimized in numbers.

(1) We start giving as many quarters as we can ($\leq M$).
Then we pass to a less valuable coin.

HOW TO CHECK CORRECTNESS?

(1.) we can test the algorithm on a huge set of inputs. The problem is that we're sure it works on those -> so likely correct, BUT there may be an input, not tested, which gives an error

(2.) we can prove the correctness.
-> it is correct if on all possible inputs, the algorithm always gives the right answer.
↳ when is it not correct? the negation! & the negation is that at least (1) input does not give the right answer.

generalized:
BetterChange(M, c, d): (c: array w/ values in decreasing order)
1 r ← M
r: remainder ——> is distinct from what we've done before.
2 for k ← 1 to d
3 ik ← r/ck -> integer division ex. $\frac{67}{50} \to 1$
4 r ← r − ck · ik -> this enters again
67 67 50 1
5 return (i1, i2 ... in)

-> r = 17 -> cannot be divided by coin of value 20, but yes with 10.

!! NOT CORRECT bc in some situations it does not work ——>

LAB PART

10/03/2021

- Practical point of view

EXPERIMENTAL APPROACHE

- slides.
  - How can we do it? By profiling a program -> in slide spiegato.

To use it we will use cProfile (ya simply import it).

- In the example of Fibonacci we have:
  - the import statement
  - cProfile.run('fibonacci(12)') -> through this we're telling the system to perform Fibonacci passing the value '12' & keep track of the timing & statistics.
    - -> We have 468 calls & the time 0.00 bc it was very fast.

In the next example:
483156 33 -> Fibonacci calls
483,56 36 ->

we call the statistics for Fibonacci basically.

tells you how many times each function was called

We're interested in Cumtime -> computational time: the time spent to execute from the first call.

ASSIGNMENTS (some will be marked)

1. Describe the situation (streamline & formalize) -> loro te lo danno scritto in generico -> leggila descrizione, come posso renderlo -> what is the input/output / how I proceed.
2. You will have to implement in Python -> loro possono chiederti anche la complessità -> you draw a graph & 2 lines of conclusions.

explain why correct & what you think of the complexity / draw a graph / spiega. -> PAROLE IN CONCLUSIONE

-> DA USARE LaTeX. -> tailored for scientific research
Lo X scrivere il report

---

www.overleaf.com

Crea profilo -> FREE ACCOUNT ->

- BRUTE FORCE METHOD SEARCH AGAIN -> basically here we integrated the previous algorithms, but we're looking at all leaves of the tree. Still we haven't cut parts of it -> to do so, we need to figure out how to search in the tree. to explore

-> DEPTH FIRST SEARCH: we start exploring in depth -> we go from the root (1) to the 5 -> the leaf -> then the sibling; Now we go back on our way to 7, then again in depth (8 & 9) & then up to 10, then in depth.

First we explore a node, then left side sub-tree and then the right side sub-tree.

We go to the maximum depth as we can; once we arrive to a leaf (5). The following node analyzed is the sibling; after this there are no more siblings. Once we ~~arrive~~ arrive at 9, the next node to analyze is the first sibling NOT yet explored of an ancestor of the current case. 9 -> 10 case.

16 -> 17
to all the left end side has been explored

current node level at the moment

```
NEXT VERTEX(a, i, L, K)
1  if i < L          -> if so, we're not on a leaf, we want to go down.   (12--)
2      a_{i+1} ← 1                                                        (12+-)
3      return (a, i+1)
4  else -> we're in a leaf.
5      for j ← L to i   (from last position, to first last one.)
6          if a_j < K
7              a_j ← a_j + 1
               return(a, j)
8  return (a, 0)
```

we look at the last position & ask: is this the last symbol? If not, it means it has a sibling

1121 -> has 1122 -> to this has no sibling

we will have ignored 1127 -> 12--
increment this

once we've finished, we return 0 as level of exploration.

has it reached the last symbol?

1122 — 2 has reached? yes — 2 has not reached? No — 2 has reached? yes

- SIMPLE METHOD SEARCH

line -> 5 | 6 -> manage the exploration when we're at mid-level -> we simply generate the next vertex.

So basically we're not doing anything →

ENUMERATING SETS in Python

31/03/2021

- Before to implement the algorithm -> try drawing ideas on paper.
- IDEA -> we have to build a subset of a given set -> you start from an empty set -> you encounter the 1st element -> to add or not? & so on -> & you draw all possible combinations.

16/04/21

LAB 5 We're going to learn a trick for not running out of memory! -> GENERATORS

How to deal with sets in Python

- can be presented with curly braces {...} -> ! we cannot use {} because it means empty dictionary
- operator in
- the difference w/ lists is that even if we have more occurrences of the same element, in the representation (as in the mathematical concept), the element is printed only once.

set() -> for the empty set

UNION:

{0, 1, 2, 3, 4} | {1, 3, 5, 7}

INTERSECTION:

&

DIFFERENCE:

{0, 1, 2, 3} - {1, 3, 5, 7}

{0, 2}

↳ basically we remove from the first set the elements in the second.

non modifying operators

↳ the operands are not modified, the operator generates a new s

- .add()
- .remove()

|= = like plus equal

-> Sometimes it is useful to convert sets into lists & viceversa: When don't know the order of elements for sets

- We have .copy()
- we have the command len()

GENERATORS -> In the ex. we did, when we ~~allo~~ generated all the possible combinations; BUT THE CONCEPT IS THAT

- APPROXIMATION RATIO → how much our greedy algorithm is far from the optimal solution.

→ the bigger the n size, the more the approx. ratio ⇒ for our algorithm, so approx. ratio not fixed in our case.

Today we'll see an algorithm whose ALGORITHM RATIO is constant. (it will be 4), that is not as an approx. ratio of error!

what we do:

GIVING NAMES

- when we consider a permutation $\Pi = \Pi_1, \dots \Pi_2$, we introduce 2 more numbers $\Pi_0 = 0$ & $\Pi_{n+1} = n+1$ (tail)
- adjacency → one element is the successive of the other → we don't care if 1-2 or 2-1 (bigger first of later) ⇒ $|\Pi_i - \Pi_{i+1}| = 1$
- breakpoint → 1-3 or 5-8, when they're not consecutive

to essentially intuitively tell us about problems

0 2 6 9

if breakpoints in Π_0 & Π_{n+1}, we know there's a problem there

INTUITIVELY: the more breakpoints we have, the farest from the sol. we are

the only thing is that Π_0 & Π_{n+1} CAN NOT counted for the reversal

added first & last

- a strip = a sequence of adjacences → any interval between 2 breakpoints

3 4 5 → increasing strip

8 7 6 ← decreasing strip.

If many ~~reversals~~ breakpoints we need to have many reversals, in the end there will be NO breakpoints.

MAX N°: n+1 if completely unsorted seq.

→ n° of breakpoints: measure of how far we are.

What does it happen to a breakpoint when we reverse smth?

- Assume we're going to reverse 8 7 6, what happens is that everything in the interval is reversed → everything that was an adjacency remains an adjacency. We're just changing the order of things BUT NOT the order of elements in the segment. Whatever it was a breakpoint, will remain a breakpoint as well.

When we do a reversal, the n° of adjacencies & breakpoints remains the same WITHIN the segment.

! What changes is what happens at the borders → in the best scenario the n° of breakpoints could diminish by 2. → (! at most → in the best scenario)