ALGORITHMS AND DATA STRUCTURES IN BIOLOGY

Giacomo Castagnetti - Genomics 2023

INDEX

                - ★ Brute force approach
                - ★ Backtracking approach
            - ➢ Knapsack problem
                - ★ Exhaustive search approach
            - ➢ Search for regulatory motifs in DNA sequences
                - ★ Finding the starting positions of repeated sequences approach
                - ★ Median string approach
    - Speeding up exhaustive search
    - Branch and bound technique
        - ➢ Knapsack problem
- Greedy paradigm
    - Idea behind greedy approaches
        - ➢ Genome rearrangement problem
    - Approximation ratio
        - ➢ Pancake flipping problem
    - Bounded approximation ratio
        - ➢ Genomic rearrangement problem
        - ➢ Motif finding problem
        - ➢ Bin packing problem
            - ★ First-fit approach
- Dynamic programming
        - ➢ Change problem
        - ➢ Manhattan tourist problem
            - ★ Exhaustive search approach
            - ★ Greedy approach
            - ★ Dynamic programming approach
        - ➢ Variation to manhattan tourist problem
        - ➢ DNA alignment
            - ★ Longest common subsequence problem
            - ★ Edit distance problem
            - ★ Global alignment problem
            - ★ Local alignment problem
        - ➢ Shortest common supersequence problem
        - ➢ Satisfiability problem
        - ➢ 0/1 knapsack with integer weights problem
- Divide-and-conquer algorithms
    - Space complexity of an algorithm
        - ➢ Block alignment problem
    - Recursion master theorem
        - ➢ Multiplication problem

INTRODUCTION

Algorithms are just procedures to solve problems. While a program is a series of instructions given to a computer to execute an algorithm. We will learn how to solve problems through algorithms.

Two modules:

- Theory: how to invent algorithms. Theory behind the development of algorithms. How to abstract problems, different approaches can solve the same problem. Easy, hard, efficient.
- Laboratory: attempt to solve the exercises. Performance evaluation of algorithms as implemented in the python programming language.
  Learn how to write a scientific report by way of the latex systems.

Textbook: An Introduction to Bioinformatics Algorithms by Neil C. Jones and Pavel A. Pevzner

Exam:

- Optional course works: solve a problem, write a report of the solution
- Written test: programming exercises, write algorithms in a more abstract way (pseudocode). Then open end or closed end questions. At most 30. If more than 18 the vote can be accepted
- Optional oral exam: to increase or decrease the grade. Compulsory for the lode (also compulsory at least 25 in the written test).

Workflow

We will analyse the problems trying to understand their structure, then we will invent solutions (algorithms) to such problems. Coding an algorithm into a program is a matter of translation, it is quite easy, it won't be the focus of the course.

Problem in genomics → combinatorial problems → algorithms → python program.

With combinatorial problems we refer to a problem expressed with combinatorial structures: vectors and matrices.

We have to understand what the problem is made of, there are some things on which we are not interested. Inventing an algorithm to solve a problem is peculiar to humans, computers cannot do it.

Example

Assume we have a dna sequence **S**, then we have another string **T**, which is a pattern (T=AACTTCGG). The problem is to find where T appears in S, even with a slight change, we have to find the similar versions of T in S.

The interesting point is the word "similar", it is vague, until we provide a meaning, until we invent a meaning. When we describe a problem we have to be precise, it means that we have to overcome the ambiguity of terms.

To formalise the similarity stated in the problem we can invent a system like this:

Given an alphabet, a set of symbols (e.g. the italian alphabet), a string over this alphabet is a series of occurrences of its symbols (e.g. AGT).

We can define the distance between two strings as the number of changes that we need to perform to transform one string into another (such changes are: substitution, deletion, addition).

Ex.

abc → arfc. Change b into r, add f. Distance = 2.

We will see that all problems are characterised by an input and an output. In this case the input is the string S and the output is T.

Why are we doing this?

- We need to have a precise description.
- We want to define an algorithm (it needs a precise description).

PROBLEM

Imagine a game played by player X and Y. We have two piles of rocks, pile A and pile B. Each player in turn decides between two possible moves.

1. Take two rocks one from each pile
2. Take only one rock from either pile A or B

The player taking the last rock is the winner.

Problem: given a situation like this, we want to come up with a move that will lead us to win the match. Given the current situation, what is the best move?

The first intuitive approach is to try to simplify the problem:

Two piles with two rocks each. X starts: if it takes two rocks it will grant the win to Y that will take again two rocks. For this reason X picks 1 rock. Y at this point can take a rock from the pile with 1 rock remaining, forcing X to take just 1 rock and winning the game.

We have to come up with an algorithm to figure out the best move. The algorithm solving the general case of this problem is the same that aligns DNA sequences.

Sometimes when we analyse problems we will find that the intrinsic structure is equal to problems that have been already solved. Once we extract the structure we might find out that the problem has been already solved. We can go online and take a solution that already exists. In cases in which structure is similar to other problems we have to create a new solution.

	0	1	2	3	4	5	6	7	8	9	10
0	*	←	*	←	*	←	*	←	*	←	*
1	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
2	*	←	*	←	*	←	*	←	*	←	*
3	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
4	*	←	*	←	*	←	*	←	*	←	*
5	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
6	*	←	*	←	*	←	*	←	*	←	*
7	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
8	*	←	*	←	*	←	*	←	*	←	*
9	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
10	*	←	*	←	*	←	*	←	*	←	*

In the rows we have the number of rocks in the first pile, in columns we have the number of rocks in the second pile.

Look at coordinates 2,2. We have information on the situation. The **asterisk** shows that the player that plays first will lose.

Consider now the point 3,2. It has an **arrow**. The first player to play will win because he will take one rock from pile A, in order to leave the other player with the first move in a 2,2 situation.

The up arrow means: reducing the number of rocks of pile A of 1.

The left arrow means: reducing the number of rocks of pile B of 1.

The diagonal arrow means; picking two rocks, one from each pile.

This table can be filled simply by layers, each layer is filled on the bases of the information of the previous layer, pointing the arrows toward the asterisks.

This table works well, but what if the number of rocks per pile was higher than 10? What if there were 3 piles of rocks? With the right algorithm we will develop generic ways to solve problems with common characteristics.

A **combinatorial problem**: a problem is an **unambiguous** and **precise** problem concerning the production of some outputs from some inputs.

- The class of possible inputs must be clearly specified.
- Which output one gets from each input must itself be specified without ambiguity.
- Specifying how to obtain the output from the input is not part of the problem's definition.

Why does it have to be unambiguous and precise? Let's go through the definition of a problem to find this out.

The easiest way to interpret this is by defining a problem through a list of pairs, input-output.

Ex.

Define the summation problem: given two numbers define their sum.

(x,y)

(<3,2>,5)

(<1,4>,5)

…

x is the input, y is the output.

This kind of definition is very precise and unambiguous, we are telling what output corresponds to which input. The issue with this problem representation, although it is very precise, is that it is not concise, it is not finite.

Since we cannot use this we have to use language, we overcome the issue of describing the problem in this way, by the use of language. This has a side effect: ambiguity.

Describing a problem consists in defining an output for given input, so the simpler way to do it is to give a list of output for each input (pair of input-output values) and in some cases this type of definition can be effective.

When defining a problem we need to describe what the inputs are, what the outputs are and how they are related to the inputs. Since we cannot list them infinitely. We have to describe them via language in an unambiguous and precise way. We should be able to virtually retrieve the list of input-output pairs from the language description.

Now let's see some examples of problem definition, remember that we have to define what is the input and what is its meaning.

Sum problem

Input: 2 numbers (a,b)

Meaning: whatever number

Output: 1 number (c)
    Meaning: addition
Relation: the output is the sum of the two inputs
Note that we are just describing what is the output, not how it is obtained.

Rock pile problem (RPP)

Input: 2 numbers(n,m) the inputs are the same!!!! Two different problems with the same input.
    Meaning: number of rocks for each pair.
Output: move
    Meaning: pick one or two rocks.
Relation: the output has to be the best move for the input situation

Why do we need to be so specific? Because an algorithm is something that goes in between input and output, it tries to go from an input to an output. It is a finite sequence of unambiguous steps that, starting from the input, produces the output.

An important characteristic of algorithms is correctness: an algorithm is correct when the procedure produces the expected output. We can tell if it is correct only if we know the correct output. We need to know where we want to go.

Important: when we describe a problem we need to give: input, meaning of input, output, meaning of the output and the relationship between input and output, we do not have to explain how to obtain the output. A description of a problem is a thing, the algorithm is the solution. The same problem could accept different solutions and therefore different algorithms.

Algorithm

Algorithm: **finite** seq of **unambiguous** steps tailored to **solve a problem**.

Solving a problem

An algorithm can be characterised as correct or not only by means of the result: an algorithm is correct if it solves the problem for which we designed it. An algorithm correct for a problem could be wrong for another.

This is an important concept, example: given problem A, does algorithm B solve it? The answer could be false even if B is a clever algorithm.

Finite

Algorithms must terminate in finite times. The procedures that do not guarantee termination are not algorithms. These infinite procedures are problematic.

Unambiguous

Every algorithm must be deterministic. At each step we have to know exactly what it does.

Pseudocode

During the course we will take the following assumption: once an algorithm will be described, we will be automatically able to write the python version.

We will work with pseudocode. It is a natural language that can be used to describe intermediate steps of the procedures, a high level description comprehensible for humans but hot computers. It still has to be precise enough to be able to create a program.

Assignment of variables

```
a←b #sets the variable a to the value b
```

Arithmetic

```
a+b, a-b, a*b, a/b, a^b #addition, subtraction, multiplication, division, and exponentiation of numbers.
```

Conditional

```
if A is true
        B
else
        C
```

If A is true, the block of instructions B is executed, otherwise block of instruction C is executed.

For loops

```
for i←a to b
        B
```

Sets i to a and executes instructions B. sets i to a+ 1 and executes instructions b again. Repeats for i=a+2, …, b-1, b (b included).

While loops

```
while A is true
        B
```

Checks the condition A. If it is true, then executes instructions B. Checks A again; if it's true it executes B again. repeats until A is not true

Array access

```
a_i
```

the ith number of array a=(a_1, …, a_i, …, a_n). For example, if F=(1,1,2,3), then F_3=2, and F_4=3. Arrays are 1 based, the first element of an array is indexed 1.

United states change problem

United States Change Problem:
Convert some amount of money into the fewest number of coins.

Input: An amount of money, M, in cents.

Output: The smallest number of quarters q, dimes d, nickels n, and pennies p whose values add to M (i.e., $25q + 10d + 5n + p = M$ and $q + d + n + p$ is as small as possible).

United States because we are assuming that we are using United States coins (25 cents, 10 cents, 5 cents and 1 cent). We are assuming that the value of the coins is fixed, it is defined in the problem definition.

An easy algorithm to solve this is to start from big coins.

```
while M>0
        c ← largest count that is smaller than or equal to M
        Give coin with denomination c to customer
        M ← M - c
```

This approach is called greedy approach (i do the best right now and then i will see). We will see that this approach has some wrong implications in a more general setting.

Correctness: how can we know whether an algorithm is correct or not? An algorithm is correct if whatever input, the algorithm returns the expected output.

How can we prove this?

- **Empirical approach**: code the algorithm, give the program some input, check if outputs are correct. This doesn't work if the input of a problem (usually called instances) is infinite (an instance is a case of a problem, an input).
  If problems have a finite number of instances this method is ok, even though if they are 3 billions it is still problematic.
- **Formal approach**: we need to analyse the structure of the algorithm and show formally, mathematically, that the way it is designed allows it to not make mistakes.

Checking for the incorrectness of an algorithm is much easier: to prove it theoretically we have just to find a case in which the algorithm doesn't provide the correct answer, this is a counterexample. Another way is to analyse the structure of the algorithm and figure out some problems. To show the correctness we would have to check all inputs, to show that it is wrong it is enough to have 1 case.

Change problem

Change Problem:
Convert some amount of money M into given denominations, using the smallest possible number of coins.

Input: An amount of money M, and an array of d denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, in decreasing order of value $(c_1 > c_2 > \cdots > c_d)$.

Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 i_1 + c_2 i_2 + \cdots + c_d i_d = M$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

Better change (M,c,d)

```
r←M #r stands for reminder
for k←1 to d
        ik←r/ck
        r←r-ck*ik
return(i1, …, id)
```

We cycle through d, starting from the biggest coin. We divide the money for the largest coin, we save the result in the variable i, and then we subtract from the total money the current coin times the result of the division.

This algorithm is not correct because we can find a counterexample.
If M = 40; c = (25, 20, 10, 5), the smallest number of coins (expected result) is 2 (20, 20), but the result of the algorithm is 3 (25, 10, 5).

With this kind of approach, once we make a choice in the current moment (we look for the bigger coin) this might create a problem afterwards. Sometime, if we are not sure about the correctness of the algorithm we have in mind, it is better to start with an algorithm which is trivially correct, although having perhaps other problems.

We should look at all possible combinations. Generate all possible combinations, explore them one after the other and select the best one.

```
BRUTEFORCECHANGE(M, c, d)
1  smallestNumberOfCoins ← ∞
2  for each (i_1, ..., i_d) from (0, ..., 0) to (M/c_1, ..., M/c_d)
3      valueOfCoins ← Σ_{k=1}^{d} i_k c_k
4      if valueOfCoins = M
5          numberOfCoins ← Σ_{k=1}^{d} i_k
6          if numberOfCoins < smallestNumberOfCoins
7              smallestNumberOfCoins ← numberOfCoins
8              bestChange ← (i_1, i_2, ..., i_d)
9  return (bestChange)
```

At the beginning we haven't seen anything so we set the smallest number to infinite.

With the second line we are trying (generating) all possible combinations. A computer can't understand this line, we will have to do some work to translate it to python.

The "i"s are the coefficients for each coin, we want to test for each coin, each possible quantity that can be useful for giving the change. E.g. if the change is 60 and the biggest coin is 25 we will try a combination in which we give only the smaller coins, one in which we give 1 coin of 25 and one in which we give 2 coins of 25. From 0 to 60/25 (=2).

Since we are trying all combinations, there will be many of them in which we give too little coins (e.g. 0,0,0,0,1) and many in which we will give too many coins (e.g. 2,4,6,16). For each combination we evaluate the sum of the coins, only the combinations that will add up to the correct number will be further processed.

The number of coins of each correct combination is computed and then a variable stores the combination which has the lowest number. This variable is updated only if a combination with a lower number of coins shows up.

To use this kind of approach we need to be sure that among these there is the solution. We have to be sure that we will see the correct solution. This type of algorithm is called exhaustive search algorithm.

This algorithm is correct because we are looking for a big number of solutions that will contain the right one. All exhaustive search algorithms are based like this. The correctness is proven by the fact that the algorithm doesn't miss anything.

The only downside of this algorithm is that it is really slow.

Brute force algorithm is correct: proof is that algorithm tries everything, if there is a solution at some point we will see it.

Induction principle

It is a mathematical formal tool to prove properties and theorems that need to be shown to hold for all numbers. We want to demonstrate that it is valid in an infinite number of cases in a finite amount of time.

The induction principle allows us to show that a property holds in infinite cases.

Intuition: assume we want to prove that property p is true for all possible numbers, from 0 onward.

The induction principle consists of 3 steps:

1. **Base of induction**: during the first step we prove that the property holds true for the first case (base case). p(0) is true.
2. **Inductive hypothesis**: we assume that the property holds for all numbers up to k. p(0).....p(k) is true. In this step we don't prove anything, we just assume stuff to be true.
3. **Induction step**: prove that p(0….p(k) is true, implies that p(k+1) is true.

We have to prove a case and then prove that the case+1 is true, in this way we consider case+1 true and by the same proof also case +2 will be true.

Ex.

Show that every number n>2 admits a prime factorisation:

1. Show that the property holds for the base case:
   Factorisation of 2: 2→2 is a prime number.
2. Assume that all numbers up to k admit a prime factorisation
3. From assumption 2 we have to conclude that k+1 admits factorisation
   There are two cases:
   a. K+1 is a prime number: itself it is a prime factorisation
   b. K+1 is not prime: it is a composite number, it can be obtained as product of two numbers. Different from 1.
      n1*n2
      n1<1, n2>1
      We can understand that these two numbers are smaller than k+1.
      They are at most k, n1<k, n2<k.
      Since they are at most k they admit a factorisation.
      The factorisation of k+1 is the factorisation of n1 times n2

We have proven a property that holds for infinitely many cases.

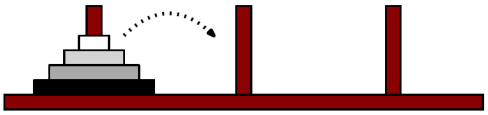
Ex.

3^k-1 is even k>1

1. 3^1.1=2
2. Assume that for all numbers up to k 3^k -1 is even

3. 3k+1 -1 is even
   3*3^k -1= (2+1)*3^k -1 = 2*3k +3^k-1= (2*3^k) + (3^k -1)
   3^k -1 is even by assumption.

Hanoi puzzle problem



How can we move the disks from the first peg to the third one by following some rules:

1. We can move only 1 disk at a time. It must be a top-most disk, we cannot take disks from the middle of a pile.
2. When we are moving a disk we can place it on a peg that is either empty or that has a disk which is bigger than the disk that we are moving.

Towers of Hanoi Problem:
Output a list of moves that solves the Towers of Hanoi.

Input: An integer n.

Output: A sequence of moves that will solve the n-disk Towers of Hanoi puzzle.

The hanoi puzzle with 3 disk has the following solution:

1. Place the smallest disk in the third peg
2. Place the middle disk in the second peg
3. Place the smallest disk on the middle disk
4. Place the largest disk on the third peg
5. Place the smallest disk on the first peg
6. Place the middle disk on top of the largest one
7. Place the smallest disk on top of the middle one

To solve the 4 disks we have to move the 3 top most disks to the second pole, then we move the fourth, and then we move the three disks on top of the largest one.

Algorithm:

```
HANOITOWERS(n, fromPeg, toPeg)
1  if n = 1
2      output "Move disk from peg fromPeg to peg toPeg"
3      return
4  unusedPeg ← 6 − fromPeg − toPeg
5  HANOITOWERS(n − 1, fromPeg, unusedPeg)
6  output "Move disk from peg fromPeg to peg toPeg"
7  HANOITOWERS(n − 1, unusedPeg, toPeg)
8  return
```

We want to solve hanoi tower for whatever n, with n>0. We can easily identify two cases: n=1 or n>1. In line 1 to 3 we deal with the case in which n=1. From line 4 to 8 we deal with the cases in which n>1.

Base case: the part of the algorithm that deals with such an easy case of the problem that can be solved without recursion. If n=1 we just need to move one disk from the initial peg (left) to the final one (right).

The second part uses recursion, in line 5 we want to move n-1 disks from “fromPeg” to “unusedPeg”. In the matter of this problem, fromPeg is the initial position of the disks (1), toPeg is the final position of the disks (3) and unusedPeg is the auxiliary peg (2).

By numbering the pegs, we can find a formula that always gives the number of the unusedPeg. This is a trick that works only for this problem.

6 - fromPeg - toPeg

E.g. 6-1-3 = 2

Once the auxiliary peg has been identified, the hanoi function is called with n-1, giving as starting peg (fromPeg) the starting position of the n-1 disks and as arrival peg (toPeg) the unusedPeg. Eventually we will just have to move one disk from a fromPeg to a toPeg.

With this algorithm we are not giving much details, we are just proposing the general idea.

We are capable of moving the n disk if someone else is able to solve the n-1 case. This is repeated until the solution needed is so easy that it can be computed, moving a single disk. The chain of calls will solve the problem.

Correctness

In order to be sure that this algorithm is correct we have to prove it by induction.

We use the proof by induction because the induction principle and the recursive algorithms are similar. The Induction principle has a base of induction, while a recursive algorithm has a base case.

In order to show that algorithm works for generic case n, we have to show that it works for n-1.

When we need to prove that a recursive algorithm is correct by induction we need to spot a property that, if satisfied, makes the algorithm correct.

Assume this is the property we want to test: HanoiTowers(n, fromPeg, toPeg) correctly moves the top-most n disks from fromPeg to toPeg.

If this property was true the algorithm would be correct. We will show that this is true by induction principle.

1. Base of induction: Does it hold for n=1?
2. Inductive hypothesis: we assume that the algorithm satisfies the property for all recursive cases up to n-1.

In this case there is a problem, we have to be more precise. If our algorithm starts from the wrong state it will do wrong stuff. We need to specify details to be sure that we produce good results. The property cannot be proved by induction if we start from wrong cases.

We have to refine the characteristics that our algorithm has, allowing us to show that it is correct. If all disks are correctly stacked, then our algorithm does things properly. If the algorithm starts from a consistent state, it works well and leaves the towers in a correct state.

We need a new (more precise) property:

If all disks in all pegs are correctly stacked and the top-most n disks of fromPeg are smaller than the top disks, if any, in all the other pegs, then HanoiTowers(n, fromPeg, toPeg) correctly moves the top-most n disks from fromPeg to toPeg.

1. Base of induction: when n=1 we just move a disk to the toPeg.
2. Inductive hypothesis: assume that algorithm is right up to n-1.
3. Inductive step: considering the inductive hypothesis true, does the algorithm work for n? Via induction hypothesis, line 5 correctly moves n-1 disks and leaves the game in a consistent way. We can move the big disk from fromPeg to toPeg. Everything will be left in a consistent case.

Note that the moves proposed by the algorithm are just strings, they are not organised in data structures.

Fibonacci sequence problem

Fibonacci Problem:
Calculate the nth Fibonacci number.

Input: An integer n.

Output: The nth Fibonacci number $F_n = F_{n-1} + F_{n-2}$ (with $F_1 = F_2 = 1$).

Recursive approach

```
RECURSIVEFIBONACCI(n)
1  if n = 1 or n = 2
2      return 1
3  else
4      a ← RECURSIVEFIBONACCI(n − 1)
5      b ← RECURSIVEFIBONACCI(n − 2)
6      return a + b
```

Computes the number just by following the definition: if the input is 1 or 2 we return 1, if the input is > 2 we compute the fibonacci number of n-1 and of n-2, then we sum them and return their sum.

Property: RecursiveFibonacci(n) returns the n_{th} fibonacci number

1. Base of induction: we have two base cases. They are easily verifiable: If the input is 1 or 2 the algorithm returns 1, which is the n_{th} fibonacci number.
2. Inductive hypothesis: RecursiveFibonacci(n) up to n-1 returns the n-1_{th}** fibonacci number
3. Inductive step: is the property true if we assume the second step to be true?
   Yes, it is true because when we give in input a number n > 2 the algorithm computes first

the fibonacci numbers for “n”s smaller than n. By inductive hypothesis the value returned by the calls at lines 4 and 5 are correct, and therefore the result is correct. We are returning the correct value, which proves that the property holds.

Iterative approach

```
FIBONACCI(n)
1  F_1 ← 1
2  F_2 ← 1
3  for i ← 3 to n
4        F_i ← F_{i−1} + F_{i−2}
5  return F_n
```

Different algorithms solve the same problem. A problem is a thing, an algorithm is another. Algorithms are iterative if they are not recursive, they do not need to have loops.

This algorithm performs the following instructions: we have an array F, we put in position 1, 1, and in position 2, 1. Then we cycle starting from 3 to n, for each cycle, we compute the i_{th}-fibonacci number on the basis of the previous numbers of the list. Then we return the value in position n of the array.

To prove the correctness we will use the approach of **invariance**. An **invariant property** is a property that the algorithm fulfils **before entering into the loop**, **at the end of every iteration** of the loop, and when it gets **out of the loop**. An invariant property is something that doesn’t change.

We have to think creatively about this property. E.g. property = in position 1 of the array F, there is number 1, this is a useless but true invariant property.

The property that we need is: Fi is the i_{th} fibonacci number. This property is true before the cycle: we can consider i smaller than 3, when i is 1 the number of the array is 1, and when i is 2 the number of the array is 1. This is a rough reasoning but it works for our purposes.
At the end of each loop we will put in position i the i_{th} number. The i gets bigger and bigger, until it reaches n, this means that in position n we will put the n_{th} fibonacci number.

Since in position i there is the i_{th} fibonacci number and we return the n_{th} position of the array, we are actually returning the n_{th} fibonacci number.

Sorting problem

Sorting Problem:
Sort a list of integers.

Input: A list of n distinct integers $\mathbf{a} = (a_1, a_2, \ldots, a_n)$.

Output: Sorted list of integers, that is, a reordering $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ of integers from $\mathbf{a}$ such that $b_1 < b_2 < \cdots < b_n$.

We are going to use the selection sort approach: we traverse the list, from the beginning to the end, looking for the smallest value. Once we find it we move the smallest element in the first position and the first number in the position of the smallest.

Then once we have the smallest element in the first position we are not interested about it anymore. We consider the second element, we swap it with the lower among all of the remaining.

We do this for all i's from 1 to n-1.

```
SELECTIONSORT(a, n)
1  for i ← 1 to n − 1
2      a_j ← Smallest element among a_i, a_{i+1}, ... a_n.
3      Swap a_i and a_j
4  return a
```

To be more precise we need to call an additional function that can find the smallest element among the remaining.

```
SELECTIONSORT(a, n)
1  for i ← 1 to n − 1
2      j ← INDEXOFMIN(a, i, n)
3      Swap elements a_i and a_j
4  return a
```

```
INDEXOFMIN(array, first, last)
1  index ← first
2  for k ← first + 1 to last
3      if array_k < array_index
4          index ← k
5  return index
```

We want to prove that this algorithm works by invariance.

Property: the initial portion of the array up to i_{th} position is sorted and contains the smallest number.

Before entering the loop an empty portion of the array is sorted, then each cycle an element of the array is sorted and the sorted portion increases. At the end of the cycle all of the array is sorted.

Why should we pick a solution or another?

COMPLEXITY

If there are different correct algorithms solving the same problem, why should we prefer one over another? The complexity of an algorithm will allow us to make a choice.

There exist two kinds of complexities.

Time complexity

Time complexity is the **amount of steps that the algorithm performs before giving the answer**.

What are the steps? Traditionally, the complexity of an algorithm was defined on a Turing machine; it is an abstract machine that allows counting the steps.

In modern computers we have operations. We don't want the details, we'll be rough. This kind of measure gives an idea of how much time an algorithm takes to compute. It is enough to characterise if an algorithm is slow or fast.

We can measure the time complexity in two ways (as for correctness):

- Empirically (doesn't always work)
- Analytically

When we want to understand the time complexity of an algorithm empirically we have to implement the algorithm in an actual programming language, then perform a few experiments. We give inputs and we measure how much time the algorithm takes for producing the output. We will figure out which of the available algorithms are faster.

As you can imagine this procedure is very machine-dependent. The amount of time in a machine will be different from another machine.

We will check the time complexity by implementing an algorithm and then take some time measurement. We can estimate if it is a good algorithm or not. On the other hand as this is machine dependent we can also carry out an analytical way.

If we want to work analytically, the time complexity is the amount of steps, so to estimate it we can try to understand how many steps (operations) need to be performed before stopping.

Few principles of analytical analysis:

1. The number of steps is the number of basic operations performed by the algorithm (basic operations are: adding, subtracting, assigning a variable, adding a value to an array)
2. All basic operations cost the same, they take the same amount of time to be executed.
3. Ignore multiplicative or adding constant to the complexity.
4. We are interested in the worst case scenario

Assume we are analysing the complexity of selection sort algorithm:

```
SELECTIONSORT(a, n)
1  for i ← 1 to n − 1
2      j ← INDEXOFMIN(a, i, n)
3      Swap elements a_i and a_j
4  return a
```

```
INDEXOFMIN(array, first, last)
1  index ← first
2  for k ← first + 1 to last
3      if array_k < array_index
4          index ← k
5  return index
```

Sorting an array of 10 elements and sorting 100 elements doesn't take the same amount of time. More values to sort corresponds to a larger amount of time to spend.

The amount of time taken by the algorithm to provide output given input, depends on the input, the bigger the input the longer the time. The time complexity cannot be defined in absolute values, it has to be expressed as a function of the input.

The actual definition of the time complexity of an algorithm is the **amount of steps or basic operations that algorithm performs before output with respect to the size of the input**.

Input size

What is the size of input? The amount of bits needed to represent the input, for example, numbers can be represented in binary. Usually we use 32 bits for each number, we keep the number of bits constant (we could represent 4 billion characters).

When we have arrays in input, their dimension is proportional to the number of elements. When dealing with arrays, their size will be the number of values. Each additional value will correspond to additional bits to process.

Addition problem

Input : n, n integers
Output: n+m
Relation: two integers as input, the output is their sum.

What is the size of the input in this case? 2*32 bits.

Assume we are in a context in which n and m are represented by a variable number of bits. We would need log2(n) bits for each number (ex. 16: 1000).

Worst case complexity

When computing the complexity, we are always interested in the worst case scenario. Assume we have this in input: (4,2,1,5,7).

How long does it take to sort this array using selection sort?

In the first iteration, to find the smallest value, we have to check all the numbers, we need n operations. In the second operation n-1 values will be checked. Consider that even if the first element is already the smallest (or even if the array is already completely sorted) we do not save time, we still have to check all of them.

The total number of operations is n+n-1+n-2+...+1: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$

For the particular algorithm SelectionSort, even if the array is already sorted, the time is all the same. The **shape** of the input doesn't influence the algorithm complexity of selection sort. In certain algorithms s.a. bubble sort, if we input an array which is already sorted, the computation time is n, just the time to check if it is already sorted.

During the analysis we reasoned on the worst complexity possible because we are interested in the worst case scenario. We have to remember also that, after having found the complexity function, we have to imagine to fix the dimension of the input and try to find the worst case. The worst time complexity can be also related to the shape of the input.

During the course we will also talk about the average time complexity and best case complexity.

Asymptotic notation

We want a rough estimate of the time complexity of an algorithm. E.g. SelectionSort works like a quadratic algorithm. The asymptotic notation is a system that allows us to extract from a complex function only information that we need.

O notation

Intuition: the function $f(n)$ is $O(g(n))$ intuitively, if $g(n)$ is an upper bound of $f(n)$. $g(n)$ has to be a roof, it has to limit $f(n)$ from above.

Formally: $f(n)$ is $O(g(n))$ if there exist a number n_0 from which $g(n)$ multiplied for a constant c is bigger than $f(n)$. If there

$f(n)\ is\ O(g(n))\ if\ \exists n_0,\ c\ such\ that\ \forall n \geq n_0\ f(n) \leq c * g(n)$

There exist a number n_0 and a constant c such that , for all n bigger or equal than n_0, $f(n)$ is dominated by $c * g(n)$.

Example: $f(n) = 3n^2 + 4n$ is $O(n^2)$ and also $O(n^3)$, and certainly $O(2^n)$ however it is not $O(n)$. If we consider the third case, we can take whichever constant c we want but at a certain point $f(n)$ (that has a n^2) will overtake $c * g(n)$ (which is n).

When computing the O of a function, we are interested in the **smallest upper bound**, in this example it is $O(n^2)$.

Ω notation

Together with O notation we also have Ω, it the specular notation. O is for upper bounds, Ω is for lower bounds.

$f(n)\ is\ \Omega(g(n))\ if\ \exists n_0,\ c\ such\ that\ \forall n \geq n_0\ f(n) \leq c * g(n)$

A function $f(n)$ is $\Omega(g(n))$ if there are constants c and n_0 such that, for all values of $n \geq n0,\ f(n) \geq c \cdot g(n)$. We know that the function won't go lower than big omega.

Θ notation

A function $f(n)$ is $\Theta(g(n))$ if f is both $O(g(n))$ and $\Omega(g(n))$.

We are saying that the $f(n)$ and $g(n)$ have a similar rate of growth.

Overestimating

Among various simplifications one thing we have to be aware of is overestimating. When we analysed the selection sort we said that we need n operations at first round, then n-1 and so on until we reach 1. To simplify we can say that each cycle needs n operations, in this way it is easier to reach the result, which is n^2.

Examples of complexity

Hanoi puzzle

Now we will see the complexity of the recursive algorithm to solve the hanoi towers puzzle. $T(n)$ **is the time complexity** of the algorithm solving hanoi towers (in the theoretical explanation that we proposed above T(n)=f(n)).

```
HANOITOWERS(n, fromPeg, toPeg)
1  if n = 1
2      output "Move disk from peg fromPeg to peg toPeg"
3      return
4  unusedPeg ← 6 − fromPeg − toPeg
5  HANOITOWERS(n − 1, fromPeg, unusedPeg)
6  output "Move disk from peg fromPeg to peg toPeg"
7  HANOITOWERS(n − 1, unusedPeg, toPeg)
8  return
```

T(1) is the number of operations performed by the algorithm when the input is 1.

$T(1) = c$

T(n) is the time complexity for a generic input n.

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

c is the cost of line 4 and 6. Then we have line 5 and 7, they are calls to the function with a different input (smaller), so we can put the time complexity of the function with a smaller input in the formula.

This time complexity function has an issue, in its own definition it appears itself, the definition of T depends on T. If we consider the function up to now, we are not able to determine its exponent and we cannot define $g(n)$ s.t. $f(n) = O(g(n))$. We have to **solve the recursion** first, this is a tough task.

A way to solve this issue is to find a workaround to the analytical expression of the complexity. We have to try, by observing the behaviour of the algorithm in several cases, to extract the time complexity function. Then we will have to test that the function that we found is an upper bound of T(n), we are not interested in the actual formula of T(n), we just want to find an upper bound to be able to use the O notation.nIn order to do this we will use the induction principle.

Firstly, we need to invent a property by reasoning on the behaviour of the algorithm.

- When n=1, the complexity is c: $T(1) = c$
- When n=2, the complexity is:
  $T(2) = c + t(n-1) + t(n-1) = c + t(1) + t(1) = 3c$.
- When n=3:
  $T(3) = c + t(2) + t(2) = c + (c + t(1) + t(1)) + (c + t(1) + t(1)) = 7c$

We are basically building a tree, you can try to draw it.

The total number of nodes is $(2^n - 1) * c$. The time complexity can be approximated as the number of nodes.

We are going to show that the function T(n) (=f(n)) is bound to g(n).

$g(n) = (2^n - 1) * c$
$f(n) = O(g(n))$?

1. Base of induction: n=1
   $T(1) \leq (2^1 - 1) * c = c$
   true
2. Assume that this bound is valid for all cases up to n-1.
   $T(n-1) \leq (2^{n-1} - 1) * c$
3. $T(n) \leq (2^n - 1) * c$
   Is this inequality true? We can base our reasoning on the inductive hypothesis.
   $T(n) = 2 * T(n-1) + c$ (this is obtained by the definition)
   By inductive hypothesis we have a bound over this quantity (coming from our intuition).
   $T(n) = 2 * T(n-1) + c$ (subst. T(n-1) with the inductive hypo., adding the inequality)
   $T(n) \leq 2 * ((2^{n-1} - 1) * c) + c$

Hence, we can conclude that the big O notation of the algorithm is 2^n.

Exponential and polynomial

The selection sort algorithm is $O(n^2)$, it is a polynomial function, while the hanoi towers algorithm is $O(2^n)$, it is an exponential function. In an exponential function the size of input appears at exponent, while in polynomial function the size is at the base of the exponent. A polynomial function can never overtake an exponential one, no matter how big is the base, eventually, the exponential function will catch up with the polynomial.

Merge sort

This is another sorting algorithm: it takes an array, it divides it into two parts and then it recursively sorts both parts. After the two subarrays have been ordered, the two parts can be merged to reconstruct the original array.

In order to merge we look at the smallest value at the beginning of each subarray, we place it at the first position and we remove it from the subarray from which it was taken (we won't consider the added elements anymore).

MergeSort:

```
if |a|=1 then return a
else
        mergesort(first half of a)
        mergesort(second half of a)
        merge(the two halves)
```

Complexity:
$T(1) = c$
$T(n) = 2 * t(\frac{n}{2}) + c * n$
We have a recursive definition, it is not very helpful. We have to find an upper bound.
We try to understand the generic behaviour of the algorithm by intuition.

- n=1: $T(1) = c$
- n=2: $T(2) = t(1) + t(1) + c * 2 = c + c + c * 2 = 4 * c$

Generic: $T(n) = t(n/2) + t(n/2) + c * n$

Again we can build a tree. Each half can be divided in half until we will have that each operation costs c.

$T(n) = c * n * (log(n) + 1)$

How did we obtain it?

How many nodes at level 1: 2. At level i we have: 2^i nodes. We have to find the level h at which the number of children is n. How many nodes will there be at level h? 2^h

We want to find h such that:
$2^h = n$
$log(2\text{^}h) = log(n)$
$h = log(n)$

+1 because there is level 0

The complexity of the algorithm is: $O(n * log(n))$

Merge performs much faster even with relatively small arrays.

Now we will prove this result through the induction principle, it will allow us to state the bonding between the two functions: the time complexity of a recursive procedure and f(n) an inductive function that we have to discover.

In the recursive complexity of merge sort we have the merging cost c, plus the cost of sorting the two halves.
T(n)=c*n+t(n/2)+t(n/2) if we substitute we have: c*n+(c*n/2+t(n/4)+t(n/4))+(c*n/2+t(n/4)+t(n/4))

The cost of the merge is smaller because the size of the array is smaller. At each level we will need smaller operations to perform the merging until we reach c*n/n+T(1)+T(1). The overall cost is the sum of all the merges, and each level adds up to c, so the total number of operations is c*the depth of the tree.

The depth is log(n+1). We add one because we need to count the first one.

So the function that we find f(n)=c*n*(log(n+1))

Base of induction:
Is it true that T(1)<= f(n)
1<= c*1*(log(1)+1)
1<=c

Inductive hypothesis:
By induction we assume that t(n-1)<=f(n-1) from zero up to n-1.
Therefore: $T(n-1) \leq c * (n-1) * [log(n-1)+1]$

Inductive step:
By induction hypothesis, since t(n/2)<=f(n/2) (n/2 is just n minus something)

$T(n) = 2 * t(n/2) + cn$

Thanks to the inductive hypothesis: $T(\frac{n}{2}) \leq c\frac{n}{2} * [log(\frac{n}{2})+1]$

Therefore: $T(n) \leq 2 * \{c\frac{n}{2} * [log(\frac{n}{2})+1] + cn$

$T(n) \leq cn * [log(\frac{n}{2})+1] + cn$

$T(n) \leq cn * [log(\frac{n}{2})+2]$

$T(n) \leq cn * [log(n) - log2 + 2]$
$T(n) \leq cn * [log(n) - 1 + 2]$
$T(n) \leq cn * [log(n) + 1]$

We proved that the inductive equation that we induced from the structure of the problem is an upper bound for the complexity of the algorithm. Therefore merge sort has O(n*logn). Merge sort is asymptotically better than selection sort (O(n^2))

Structural complexity theory

Is this possible to optimise whatever problem? Can we always go as low as we want with algorithm complexity?

If we design an exponential algorithm for a problem, after some time we may be able to simplify it a little but it could still remain exponential. Some problems cannot be solved faster than a certain threshold.

We are moving our focus **from algorithms to problems**. There are problems that can be solved by fast algorithms and others that cannot.

Structural complexity theory is a branch of computer science that tries to understand the intrinsic complexity of problems, classifying them on the basis of how hard they are. The tricky point is that there is no structural property that tells us if a problem is hard or easy.

A problem is defined as **easy** if there exists a polynomial algorithm that solves it, if there exists a fast way of solving it.

A problem is defined as **hard** if there is no fast way for solving it (all the possible algorithms that can be developed take exponential time).

We can define the **complexity of a problem** as how difficult it is to be solved, hard or easy; this concept is different from the complexity of an algorithm, which concerns how intricate the algorithm is, how many steps it requires.

However the notion of algorithm and problem complexity are deeply linked. The complexity of a problem is based on the complexity of the algorithm that solves it.

Asymptotic notation:

The time complexity upper-bound of a problem P is $O(f(n))$, if there exists one algorithm capable of solving P with time complexity of $O(f(n))$. When defining the upper-bound we are looking at the fastest algorithm that has ever been implemented; in the case that we would have to solve the problem today, how much time would we need?

The time complexity lower-bound of a problem P is $\Omega(f(n))$ if all algorithms for P have time complexity $\Omega(f(n))$. The lower bound is defined on the basis of the complexity of all algorithms solving a problem, in order to say something like this we would have to look at all algorithms, but, given a problem, there are infinite algorithms solving it. This is not a viable way to find the lower bound complexity of a problem.

In order to retrieve the lower bound we need to look at how problems are made, understand whether the structure allows us to have an algorithm that solves the algorithm in the shortest possible way.

The upper bound is a relative measure, based on the algorithms available, we take the best one because O(f(n)) is the smallest function above the solution of the problem. The lower bound is an absolute measure, it can be defined only if we know that there is no algorithm faster than the ones that we are using.

There exists two techniques to determine the time complexity lower-bound of a problem:

Direct approach

We look at the structure of the problem, we try to think which is the minimal amount of operations to solve it.
E.g. We need to sum two numbers: is it possible to sum two numbers without looking at the whole input? No, so the lower bound will be n.

Think again about the problem of sorting an array, suppose the array has no repetitions as input; the output is a sorted array.

From an array, we need to generate another array, whose content is the same but the order is different, it is just a permutation of the initial array. More specifically it is the permutation in which the elements are sorted. An algorithm capable of sorting arrays, a general algorithm, needs to find out what is the right permutation of the input array to return as output.

Suppose an input array of 3 elements (8,7,9)(a1,a2,a3), its sorted version is (7,8,9)(a2,a1,a3). The elements were moved from (a1,a2,a3) to (a2,a1,a3), they were shuffled to obtain a sorted version.

In this imaginary sorting algorithm we have to compare the elements, why do we do this? How is this linked to the right permutation?

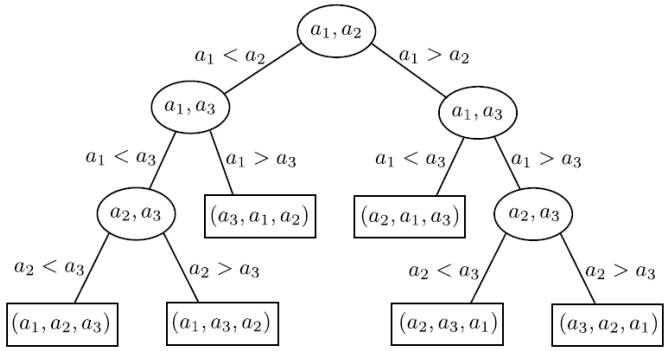
Assume we compare the first element (8=a1) with the second one (7=a2), what kind of information can we get out of this comparison?

We are capable of saying which element has to come before the other in the output array. We have some insights on how the results should look like. We have some conditions on the output.

In the example above, after the first comparison we have 3 possible outputs: a2, a1, a3; a2, a3, a1; a3, a2, a1. The intuition is that, after one comparison, we have lowered the amount of candidate permutations, we have halved it.

Now we can compare a3; by carrying out other comparisons, we will be able to select the correct one. This argument we are following is generic, it is valid for all types of sorting algorithms. We compare a1 with a3 and so on, until we manage to spot the right permutation.

We are in front of a binary tree:



The circles are the comparisons between the elements, the squares are possible outputs. We are capable of spotting the right permutation when we will arrive at the leaves. Before spotting the right permutation we need to perform comparisons in the tree until we reach a leaf.

The question is, how many nodes we need to visit from the top to the bottom before we arrive at a square leaf. The amount of circles that we traverse is the amount of comparisons.

How many are the possible permutations of a list of elements? n!, which in the example above corresponds to 3.

> To remember this we can think: how many choices to choose the first element? 3; how many to pick another one? 2 and then 1. 3*2*1.

Before hitting one of the leaves, how many levels do we have to go down? We can over-approximate this number to $2^h \geq n!$, where h is the number of levels.

$$
\begin{aligned}
k &\geq \lg(n!) \\
&= \lg(n \cdot (n-1) \cdot \cdots \cdot 2) = \lg(n) + \lg(n-1) + \cdots + \lg(2) \\
&= \sum_{i=2}^{n} \lg(i) = \sum_{i=2}^{\lceil n/2 \rceil - 1} \lg(i) + \sum_{i=\lceil n/2 \rceil}^{n} \lg(i) \geq 0 + \sum_{i=\lceil n/2 \rceil}^{n} \lg(n/2) \\
&\geq \frac{n}{2} \lg\left(\frac{n}{2}\right) = \frac{n}{2}(\lg(n) - \lg(2)) = \frac{1}{2}(n \lg(n) - n) \\
&\in \Omega(n \lg n)
\end{aligned}
$$

The depth is the number of comparisons we need to compute to state which is the correct permutation. We cannot do better than this, this is the lower bound.

Reduction approach

Assume we need to solve problem A, and we don't know how to do it, but, we have a library, a py library, in which we have a function that is able to solve problem B which is similar.

We can use a shortcut to solve A:

Given an input x for A, we want to find the output x_s, instead of solving this problem directly, we study the relations between problem A and B. If the two problems are compatible and we are smart enough, we can transform x in a related input y for B, then we find the output y_s through the algorithm of B. By applying the inverse transformation we can retrieve x_s.

The question is: what can we say about the relative complexity of problems A and B?

It depends on the type of transformations between the two inputs: if we have a transformation that happens in polynomial time then we have that **A<=B**. The complexity of problem A is not bigger than problem B. B cannot be simpler than A.

If we are able to transform instances of A in instances of B and then exploit the algorithm of B to solve the problem, we can say something about the complexity of the two problems.

If we are capable of transforming x to y, we can say that **problem A reduces to B**, or that **there is a reduction from problem A to B**.

Example: information system to manage football teams online

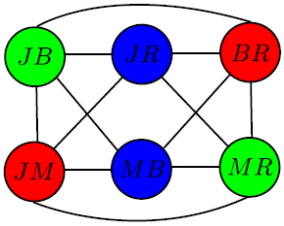
We want to create a function capable of developing the match schedule of a football tournament. If we struggle in solving this problem, we could transform it to something we are able to solve.

We decide to transform it into the problem of colouring a graph. The problem is: given a graph with vertices and edges, we want to colour the vertices of the graph s.t. vertices connected by one edge have a different colour. The number of colours depends on the intricacy of the graph.

We have to transform the teams into a graph. In order to do so we have to reason on the nature of the graph problem: when we have a graph and we colour the vertices, there will be vertices with the same colour. The characteristic common to the vertices belonging to each colour is that they are never linked.

When generating a match schedule, the main problem to solve is that the same team cannot play two matches the same day. By intuition we can think that colours might be the days of the matches. And the matches could be the vertices.

Assume we have 4 teams (J, B, R, M), the whole set of matches is the following: JB, JR, BR, JM, MB, MR. We can represent them in a graph, by connecting the matches that cannot be played the same day.



If we colour the graph, the vertices with the same colour are matches that can be played on the same day. At this point we simply list the matches following the colours.

If we create the match schedule in this way, colouring graphs cannot be easier than generating fixtures. If the procedure of transforming the matches to the vertices takes polynomial time, we can say that A reduces to B.

Computation and decision problems

Up to now we have seen **computation, or search, problems**. Problems for which to get the output we need to compute something, the possible solution is one among many.

E.g. Given a map that can be represented as vertices and edges, where the vertices are the cities and the edges are the roads, we want to go from a vertice to another. A computational algorithm would provide a solution.

A **decision problem** is one in which the answer is yes or no. Decision problems are usually easier, we are only interested in a yes or no answer.

A decision variant of the example above would be: there exists a path from A to B?

There is a relationship between search problems and decision problems. Search problems generally have a decision version as we saw before.

E.g. Is this array sorted? Or, given n, m and k is it true that k is the sum of n and m?

Why are we looking at this? Studying complexity of decision problems is easier.

Since there is a link between decision and computation it makes sense to focus on the decision problems for the complex procedures and then transfer the results to the computation problems.

If we look at decision problems, we identify two big classes of decision problems.

P

P is the class of decision problems that can be solved in Polynomial time. These are all easy decision problems.

NP

Superset of P, class of problems whose solution can be verified quickly, if somebody gives us a solution, we can certify this quickly.

We have a map of cities and roads to go from one city to another: finding the path to go from one city to another is very easy. If we want to find a path that visits all cities and passes only once for each city (hamiltonian problem) is much more difficult. If someone gives us a solution, we can quickly verify if it is ok, but coming up with an actual solution is very hard.

Problems in NP are theoretically more difficult. An interesting point is that nobody knows if the inclusion of these two sets is strict. The issue is that problems that are in NP, are problems for which finding the candidate is very intricate.

The algorithms for all problems in NP, are almost every time exponential. We have useful problems in this class. S.a. protein folding. Note that there are many problems that we like to be

hard: modern cryptography is based on these problems. If NP hard problems were easy solvable we would be able to clear cryptography

The connections of NP problems with the notion of reduction are the following:

- A problem P is NP-hard if there exists a polynomial reduction from any problem inside NP.
- A problem P is NP-complete if P is NP-hard and P is in NP.

A solution to an NP-hard problem will be an approximation.

An additional class of decision problems is R, which represents the set of solvable problems. Unfortunately there are problems outside R, for which there is no algorithm to solve them. S.a. verify if an algorithm provides the correct solution to a problem.

ALGORITHMS DESIGN PRINCIPLES

We will explore some universally valid approaches to algorithm creation.

Exhaustive search

When we define a problem we need a relationship between input and output. When we look at how output is designed we can get an idea on how the solution is.

In the money change problem we wanted to minimise the combination of coins; the output was an array of coins. In order to solve this problem we looked at the solution search space.

Once we know the description of the problem we try to understand which is the bigger domain within which we might find the solution.

In the money change problem the solution was a list of numbers, therefore the way in which the solution looks like is a list of numbers, the domain of the solution is just a list of numbers; this domain contains all possible solutions, i.e. all the possible lists of numbers.

If we can be sure that somewhere in the set the result appears, then we can commit to spotting if some conditions are met and find the solution among all of the possible ones.

Since we are looking at a set it has to have some characteristics.

1. Finite: if it has infinite elements we will never reach the end.
2. Contains the solution: we cannot reduce the set too much, we still want the guarantee that it contains the solution.
3. Allows to explore all candidates: once we have a search space we need to look at the candidates, so we need a structure that allows us to traverse all candidates, from the beginning to the end, so that we are sure that we don't miss anything. We mandatorily need an order, if we go randomly we could bounce in the solution but we could also miss it. We need an order to traverse all elements while being sure that we don't miss anything.

These kinds of approaches are actually neither fast or slow, the complexity depends on the search space. Small search space means a small time required to find the solution, large search space means a long time to find the solution. Some small spaces can be explored in polynomial times. For some reason easy problems can be solved with more efficient techniques, while NP hard problems have no other solution than a very time consuming exhaustive search algorithm. These very hard problems usually have a combinatorial solution, therefore generally we have that exhaustive searches are slow, but it doesn't depend intrinsically on them.

Exhaustive Search algorithms, also called brute force algorithms, are a sort of algorithms: that typically have high (most often, exponential) complexity; but that are often relatively easy to prove correct.

The idea at the base of an exhaustive search algorithm is that, whenever the problem can be seen as the problem of looking for an element in a finite search space such that: the element has certain properties, or the element is the best according to some criteria, or combinations therefore we find the solution by (blindly) exploring the search space.

Restriction mapping problem

On DNA strands there are points where restriction enzymes attach, when there was no informatic procedure to find those sites they performed experiments. Each sequence was cut at each site by nucleases, the length of the sequences corresponded to the position of the restriction site. The experiment was never completely precise, they didn't cut the segment at all points, there were some missing cuts. They needed to reconstruct from the data containing the length of various fragmented sequences the position of the sites.

This problem can be generalised: out of some distances we want to find where the points are on the segment.

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

The first thing to notice is that if we shift the points on the segment the relative position of the points doesn't change. We locate the first point always at position 0 arbitrarily, since the points can be shifted in whatever direction.

Partial Digest Problem:
Given all pairwise distances between points on a line, reconstruct the positions of those points.

Input: The multiset of pairwise distances L, containing $\binom{n}{2}$ integers.

Output: A set X, of n integers, such that $\Delta X = L$

Multiset: set in which the element can be repeated.

We want to compute a set x of points, s.t. the relative distance of these points reflects the distance of the values given in input.

Brute force approach

A first exhaustive search approach is the following:

```
BRUTEFORCEPDP(L, n)
1  M ← maximum element in L
2  for every set of n − 2 integers 0 < x_2 < ··· < x_{n−1} < M
3       X ← {0, x_2, ..., x_{n−1}, M}
4       Form ΔX from X
5       if ΔX = L
6            return X
7  output "No Solution"
```

If we look at the set, is there a point that we can set in the right position for sure: the zero (arbitrary) and the biggest number (M).

Then we can generate all possible combinations of n-2 points between the first and the last.

For each combination we compute ΔX (the relative distances): if this corresponds to input data, we return the combination. If no solution is found we return “no solution”.

This is a solution to a decision problem, it just returns if there is a solution or not. In a computation problem the exhaustive search algorithm needs to keep track of the solution found so far. There might be no solution but if we see something we have to save it.

We look at it and we check if this solution is better than the previous one. Once we have seen all of them, the best one will be stored in the temporary variable

Correctness: the correctness of the brute force algorithms is easily proven: surely, among the (many) sets of points considered, there is one generating L.

Complexity: intuitively we can say that this algorithm takes exponential time, it tries all possible combinations. The number of all possible combinations of n-2 elements taken out from a set of M-1 is their binomial coefficient: $\binom{M-1}{n-2}$.

Given that $\binom{a}{b} = O(a^b)$, the algorithm has exponential complexity equal to $O(M^{n-2})$.

Does it make sense to try out and set a point at position 9? No, we don’t have distance 1. This algorithm, although it can find the solution, has a search space which is huge and it has many candidates that we know from the beginning to be wrong. In the search space there is a lot of stuff that doesn’t make sense to be looked at.

A second smarter approach to the problem is the following:

```
ANOTHERBRUTEFORCEPDP(L, n)
1  M ← maximum element in L
2  for every set of n − 2 integers 0 < x_2 < ··· < x_{n−1} < M from L
3       X ← {0, x_2, ..., x_{n−1}, M}
4       Form ΔX from X
5       if ΔX = L
6            return X
7  output "No Solution"
```

In order to locate a point at a general distance p, the p must appear in the input. Instead of generating everything we take only the numbers from the input.

Correctness: this algorithm is correct because we try everything, the only candidate solutions that are discarded are those for which there is no solution.

Complexity: The difference is just a line, instead of all numbers we use only the significant one. We have that from a set of size L, we take combinations of n-2 elements taken from the set L.

Hence the complexity is: $\binom{|L|}{n-2} = \binom{\frac{n(n-1)}{2}}{n-2} = O(n^{2n-4})$.

Both algorithms are exponential, they are both slow. But we can do a consideration on the base: in the second algorithm it is n (the number of elements in list l), while in the second first algorithm it is M, which is the biggest value in the input.

If we have list like this={2,998,1000}, the first algorithm would run 1000^{3-2}, while the second one would run 3^{6-4}. The first one is slower because it is related to the value of the input, not on the size.

Backtracking approach

Another exhaustive search approach to generate all possible candidates is backtracking. It is a technique that can be used in various contexts.

Assume a solution is composed of various parts; we can try to build a solution by putting all pieces together, during this process we will have choice points, here or there. In backtracking we try all possibilities for each choice. If a solution deriving from a particular choice doesn't work we retract and we change our choice.

Think again at the digest problem: as we said we can confidently place the first (0) and last element (10). Then we can start placing the other numbers, starting with the bigger one (8); it can be placed just in two different ways: either it represents a distance from the left or from the right.

We don't know which is correct, so we assume that the distance refers to the right and we place the point in position 2.

The idea now is to replicate the reasoning, with the next highest value. Imagine we have distance 7: it can represent a point in position 3 or 7. If we place the point in position 3, now we will have distance 1 between 2 and 3, this is not possible because this number doesn't appear in the input. So we backtrack to the last choice and we change it, positioning the point in position 7.

We are tossing a coin, what could happen is that we arrive in a situation in which things are inconsistent. The idea of backtracking is that we backtrack, we go back level by level, changing the very last decision that we made. We go back to the previous choice point, and see if there is any path not explored, if yes we travel it. If again it is not consistent we go back, if there are no other options for the last choice we go back to the one before and we try another path.

If the tree has been designed in a way that it contains the solution, then we will find the solution, on the other hand if there is no solution we will traverse the whole tree without finding anything.

Recursive procedure:

```
PARTIALDIGEST(L)
 1  width ← Maximum element in L
 2  DELETE(width, L)
 3  X ← {0, width}
 4  PLACE(L, X)

PLACE(L, X)
 1  if L is empty
 2      output X
 3      return
 4  y ← Maximum element in L
 5  if Δ(y, X) ⊆ L
 6      Add y to X and remove lengths Δ(y, X) from L
 7      PLACE(L, X)
 8      Remove y from X and add lengths Δ(y, X) to L
 9  if Δ(width − y, X) ⊆ L
10      Add width − y to X and remove lengths Δ(width − y, X) from L
11      PLACE(L, X)
12      Remove width − y from X and add lengths Δ(width − y, X) to L
13  return
```

Base case: by updating the list L, we will arrive at a point in which the list is empty, in this case we output the sequence of numbers X that we have generated.

In the recursive loop, we start by taking in y the maximum element of L, we need to y either at distance y from left or from right, and then we check whether the set of distances we generated by placing y on the left is compatible with X. In line 6 we are placing y on the left, we don't know if it is the correct one, but we take this choice and we go deeper in the recursion, looking at the next element. The next recursion will have to check the correctness of the choice: if the return does not generate anything useful we go to line 8, we remove y from X and we restore L as the previous step has never happened.

If the path is correct the function will just output X, it won't return up in the recursion, and line 8 will not be executed.

To compute the complexity we have to consider that when we try where to locate points we have to try 2 possibilities for each level. This reminds us of hanoi tower.
It is an exponential algorithm, on paper it is not faster than the other ones. It has been shown that generally these algorithms run faster, at each level only one of the two choices has to be tried when there are nasty problems.

Knapsack problem

There are various descriptions of this problem. We are a thief, we want to steal from an apartment, we have a backpack (knapsack) to fill, with a weight limit. Each item has a weight. This problem asks us to find the combinations of objects to steal whose total value is maximum and we are not exceeding the maximum weight.

The input for knapsack is just a list of items, from 1 to n, each object is associated with a value v_i. We also have another list, the weights, in which w_i is the weight of object i. Then we have t: threshold, above which we cannot go.

The output is a set s (subset of 1 to n), which contains the objects that we chose to steal s.t. The sum of their weights does not exceed t and their total value is maximum.

This is an optimization problem which is NP hard. Which means that for this problem there is no known polynomial solution. All the correct procedures solving knapsack are exponential procedures, they take exponential time. The structure of this problem is so intricate that to solve it we cannot' do better than try all the possibilities.

Exam question: this situation is described and we are asked to model the problem, provide an algorithm for its solution, prove the algorithm correctness and the complexity. We will describe the solution with pseudocode.

Exhaustive search approach

We generate all the possible solutions of the problem (search space), we look at them all in turn and memorise the best one.

ExhaustiveSearchKnapsack (v, w, t, n)

```
{1, 2, …, n}
v <- (v1, v2, …, vn)
w <- (w1, w2, …, wn)
t <- threshold
best value <- 0
best set <- empty
for each subset S of {1,...,n}:
        if the sum(wi) <= t:
                val <- sum(vi)
                if the val > best value:
                        best value <- val
                        best s <- s
return best s
```

Correctness: this algorithm is correct because we are trying all possibilities and we are storing the best one, we will for sure find the correct combination in the search space and we will return it.

Complexity: inside the for we have a sum, it costs as the number of elements that are summed, in this case we can consider it as O(n). The comparison itself costs 1, then we evaluate another sum, again O(n) and some other comparisons and assignments, whose complexity is constant. The complexity of the operations inside the for is around 2n. We have to multiply this complexity for the number of times we cycle in the for: we have 2^n combinations of objects so the total complexity is $2^n * 2n$, which can be approximated to $O(2^n)$. As we expected, this algorithm is exponential.

Let's say we have a set of 5 elements, all possible subsets of the set are generated like this: do we include or not the first element? for the second same thing and so on. We have 2 choices for each element, which means: 2*2*2*2*2=2^5.

Search for regulatory motifs in DNA sequences

In DNA there are some repeated sequences that allow the interaction with enzymes. This problem can be subdivided into two kind of problems:

- Find the location of these motifs (where they appear)
- Find out the motif itself

Assume we have regulatory motifs in disjoint parts of the DNA sequence. We can look at distinct sequences of DNA strands that possibly contain the motif and in these sequences we want to find a motif that is often repeated.

If we consider the motif to be always exactly the same, this would simplify our life: once we know the exact sequence, we need just to look for that in all database sequences. Finding exact substrings can be done with a smart linear time algorithm.

The problem is that motifs can appear slightly modified in the string. Since we are admitting changes, it might be that a sequence is repeated only once in that shape, and then it is changed. When things appear changed inside the DNA sequence then we don’t know what to look for.

So there is even an issue in formalising the problem: if we say that we want to find a sequence that is repeated in the sequence a bit changed. What does it mean?

What we are going to do is understand the problem. We are not going to see the algorithm solving the problem until the end of the lecture.

We have to analyse the problem, understand the structure to formalise it. If we don’t know what the output is, how can we generate it?

We want to find in T sequences, the location at which a motif of length L appears. In each sequence we want to find the starting point of the regulatory motifs. The sequence of length L should show low variance.

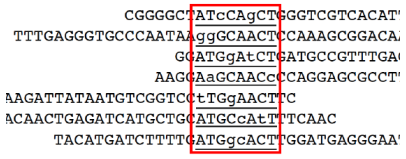
We need something to estimate the variance of strings.

Finding the starting positions of the repeated sequences approach

The interesting point is that measuring the low variance is not something that is set in stone. We need to come up with an idea that makes sense. We might even define, given two strings, that the variance is a random number between 0 and 1, but this doesn't help us at all, it is not proportional to similarity. We need to invent something that makes sense.

Definition of **variance**: something related to what appears in the T sequences, something that relates to the repeated sequences in the database.

If we put each of the initial points of the motif in T in a vector S, we can directly retrieve the sequences of length L associated to them.



At this point we can just put the motif of each sequence in a matrix, to try to identify how similar they are. This matrix is called alignment matrix, and it has T rows and L columns. This matrix is used to compute a statistic: we count, column by column, the occurrences of the characters in the strings; this count fills a matrix called profile matrix; it will have as many rows as the number of characters of the alphabet considered and L columns (with this matrix we will be able to retrieve the consensus sequence, namely the sequence that reports all of the most frequent characters for L positions).

Alignment		A	T	C	C	A	G	C	T
		G	G	G	C	A	A	C	T
		A	T	G	G	A	T	C	T
		A	A	G	C	A	A	C	C
		T	T	G	G	A	A	C	T
		A	T	G	C	C	A	T	T
		A	T	G	G	C	A	C	T
Profile	**A**	5	1	0	0	5	5	0	0
	T	1	5	0	0	0	1	1	6
	G	1	1	6	3	0	1	0	0
	C	0	0	1	4	2	0	6	1
Consensus		A	T	G	C	A	A	C	T

Intuitively, column by column, the highest numbers give a measure on how many times the strings are similar in that position. If the number is very high it means that almost every string has that character. On the other hand if it is low it means that a lot of sequences will be different in that position

Therefore, we can decide that the score of the alignment is simply the sum of the biggest number in the profile matrix taken column by column. Given an alignment matrix, we compute the frequencies, then for each column we take the biggest numbers, we sum them, and we call the resulting number score.

Given a tuple S of locations, we associate to it a number, score of S: Score(S, DNA).

The score is computed by taking strings of length L from the starting positions, then use them to fill the alignment matrix, then compute the profile matrix and sum the biggest numbers.

Is this definition precise enough to create an algorithm? Yes.

We are claiming that we can use this number to define the variance of the string. Is it related to the variance of the string?
If the strings are similar the score will be higher: if the strings are identical the score is T*L. if the strings are different the score will be lower: in the worst possible case, each column of the profile matrix (it has to add up to T) has all its elements with value T/4, the score would be T/4*L. So there is a relation of similarity between score and similarity.

Definition of the problem:

Motif Finding Problem:
Given a set of DNA sequences, find a set of l-mers, one from each sequence, that maximizes the consensus score.

Input: A $t \times n$ matrix of DNA, and l, the length of the pattern to find.

Output: An array of t starting positions $\mathbf{s} = (s_1, s_2, \ldots, s_t)$ maximizing $Score(\mathbf{s}, DNA)$.

We have a set of T DNA sequences, each of length N. S is a list of numbers identifying in the input DNA database, starting positions s.t. the score at these positions is maximised. We use the score function defined before.

Exhaustive search algorithm:

We need to return a list of numbers, going from 1 to N. Among all combinations, we need one maximising the score.

```
BRUTEFORCEMOTIFSEARCH(DNA, t, n, l)
1  bestScore ← 0
2  for each (s_1, ..., s_t) from (1, ..., 1) to (n − l + 1, ..., n − l + 1)
3      if Score(s, DNA) > bestScore
4          bestScore ← Score(s, DNA)
5          bestMotif ← (s_1, s_2, ..., s_t)
6  return bestMotif
```

We initialise the best score to a number that doesn't make sense, 0. Then we generate all combinations, then we compute the score for each combination, if it is higher than the best score, we override it.

This algorithm is correct: it tries every combination. Invariant: in the best score there is always the best score found so far, so at the end we will have the best overall score.

Complexity: from line 3 to 5 we have only one expensive operation, the computation of the score. How much does it cost? We need to look at all elements of the alignment matrix. Then we will be able to compute the statistics. The elements are T*L so the complexity is O(T*L).

How many times do we execute the for (how many combinations)? n*n*n*n…. We have N^T combinations, the complexity is $O(N^T)$.

The overall complexity of the algorithm is $O(T * L * N^T)$, this is an exponential algorithm, it is so slow that nobody uses it.

Median string approach

Because we found such a shitty algorithm we can tackle the problem in a different way: find the string that is mostly repeated in among the DNA sequences.
We need to define a measure of distance and similarity of two strings. What we invent is the **hamming distance**: we have two strings of the same length, the hamming distance between them is the number of positions on which the two strings differ.

We can generalise this notion of hamming distance between two strings to the notion of hamming distance between a given string U and S: $D_H(U, S)$, where S is a list of starting positions in a database of DNA sequences, from which we can retrieve T sequences of length L. The hamming distance between U and S is the sum of the hamming distance between U and all the strings starting at the starting position listed in S. If we want to compute this quantity we look at strings of length L starting at S and then we sum the distances. This distance gives info on how different is U, compared with strings starting at position S.

We would like to find a string U that in some way is more similar to the strings located at starting position S. Since we are looking for a string that is repeated often with little changes, we are looking for a string that minimises its differences from its occurrences in the dna sequence.

Define total distance: tD(U, DNA)

With this number we want to measure how good is string U with respect to how many times it appears inside DNA sequences slightly changed. We are computing the distance for all possible combinations of starting positions and we are minimising it to obtain the distance of U from the entire dataset.

$$TotalDistance(u, DNA) = \min_{s} d_H(u, s)$$

The combination of S guaranteeing us the minimal distance between U and S, where U is a sequence among all the possible ones that minimises the distance, is the one that we are looking for.

This definition works in this way: the lower the number is the better U is. If tD is very small, it means that somewhere in the input dna string there are strings that are very similar to U. Among all possible U, we pick one that minimises the quantity. The median string is defined as the string U, minimising the total distance. U is the sequence that appears in the dna sequence with least variation possible.

With this approach we are not discussing how to find the repeated sequences, we are focussing on the string that is most repeated in the database (U). We need a function that estimates how good the repetition is.

We have U in input, we want to find portions that are similar to U. We have to try the distance of U with all the substrings of the database. Therefore the total distance is a minimisation function.

Among all possible Us, we pick the one that minimises the total distance, it is a double minimisation.

The U minimising the total distance is the median string. The median string is related to the answer to the problem that we saw before.

The definition of the median string problem:

Median String Problem:
Given a set of DNA sequences, find a median string.

Input: A $t \times n$ matrix DNA, and l, the length of the pattern to find.

Output: A string v of l nucleotides that minimizes $TotalDistance(v, DNA)$ over all strings of that length.

Exhaustive search approach:

```
BRUTEFORCEMEDIANSEARCH(DNA, t, n, l)
1  bestWord ← AAA···AA
2  bestDistance ← ∞
3  for each l-mer word from AAA...A to TTT...T
4      if TOTALDISTANCE(word, DNA) < bestDistance
5          bestDistance ← TOTALDISTANCE(word, DNA)
6          bestWord ← word
7  return bestWord
```

Correctness: We generate all combinations of nucleotides of length L, for each of them we estimate the distance for the currently considered word, if it is lower we memorise it. The algorithm is correct.

Complexity: lines 5 and 6 are constant. Line 4 is a minimisation problem (the comparison is constant), to execute this line we need to try the u (given by the for loop) with all the possible sequences: min (sum(distance of u from starting point s)).

We have to compute all combinations to compare u with. We again are looking for all possible starting positions (n^t) compute hamming distance and minimise it.

More efficient approach: We go line by line and compute the minimum starting number. This is correct because **each string is independent from the others**. If we use this approach, how much does it cost? O(n*l*t).

This is the complexity of the operations inside the for. The for generates all possible words (u): O(4^l). The total complexity is O(n*l*t*4^l).

Why does this approach serve to solve the approach of before?

When we defined the profile matrix. The consensus string is the string that is composed, position by position, of the most frequent base. If we look at the meaning of consensus string is the string that is more similar, Is the string maximising the similarity between to the strings in the alignments.

Connection between motif finding problem and median string problem:

In the motif finding problem the consensus string is the string maximising the similarity with strings in the alignment matrix. In the median string problem we are searching for the string that minimises the differences with the database.These two concepts are very related.

Observations:

- If, for a given set of starting position if we want to minimise the minimum distance on any word. This is obtained via consensus string (w), since it maximises the similarity it also minimises differences.

$$\min_u d_H(u, \boldsymbol{s}) = d_H(w_{\boldsymbol{s}}, \boldsymbol{s}) = l \cdot t - Score(\boldsymbol{s}, DNA)$$

- In the second problem we wanted another kind of minimisation. We expand t

$$\begin{aligned}\min_u \min_{\boldsymbol{s}} d_H(u, \boldsymbol{s}) &= \min_{\boldsymbol{s}} \min_u d_H(u, \boldsymbol{s}) = \min_{\boldsymbol{s}} d_H(w_{\boldsymbol{s}}, \boldsymbol{s}) \\ &= \min_{\boldsymbol{s}}(l \cdot t - Score(\boldsymbol{s}, DNA)) \\ &= l \cdot t - \max_{\boldsymbol{s}} Score(\boldsymbol{s}, DNA)\end{aligned}$$

These two problems are structurally equivalent. We are observing this because the two algorithms have different complexity: 4^l and n^t. 4^l is much smaller than the other, so we can run the median string algorithm to find the most common repeated sequence with slight variation; then we can locate the sequence into the string.

Speeding up exhaustive search

All exhaustive search algorithms explore all research space, since the space is large they usually require a lot of time. The question is, can we use any trick to **safely** skip parts of the space to save time?

We cannot skip parts or regions if we are unsure whether we can find a solution, we need to invent techniques to safely skip parts only if we are sure that we won't find the solution there.

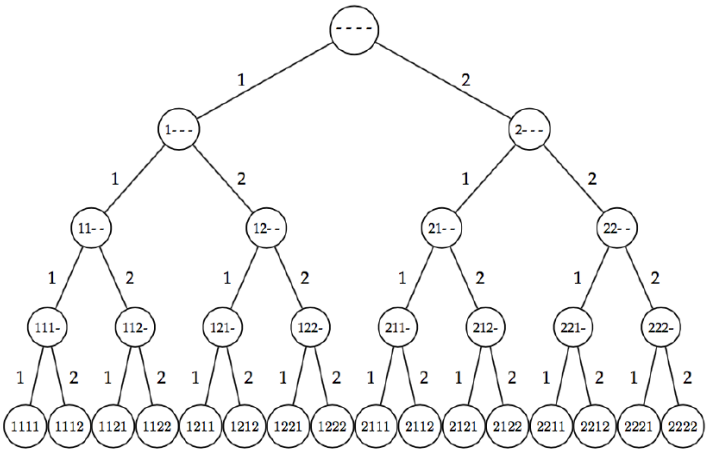
In order to achieve this goal, we need to organise the search in some way, if we explore the space randomly we can't do much. We have to find an order in which we can explore the space to skip regions.

So we will invent a way to explore the search space. The way we are going to explore the search space is not related to this specific problem, it is a technique that can be used in many circumstances. We will overimpose to the problem another structure, we will restructure the solution search space as a **tree** (this has nothing to do with DNA similarity and so on).

We will apply this approach to the motif finding problem: we have t DNA sequences, all of length n, and a motif of length l. We want to find an array of starting positions s such that their similarity, according to a function, is maximised.

We have to try all combinations of starting points to select the one with the highest similarity score. Generating all combinations requires us to generate all tuples of starting positions from the start to the end. We have to structure this space of candidate solution via a tree.

The number of combinations is related to the symbols we are working with. Assume we have to generate tuples of length 4, for which at each position we have 2 symbols:



The leaves of the tree contain all the combinations that we are interested in. All other nodes contain dashes, which means that we haven't decided a character for that position yet.

Note that the subtree rooted at the node "1 - - -", has the leaves that have fixed in first position the first symbol.

This could be done also with tuples of the DNA problem, this tree structure doesn't come from the problem. This structure can allow us to ease the solution but it is not related to the problem, it is just a metaphor that we overimpose the problem to reach the solution faster.

Tree nomenclature:

- First vertex of a tree is called root.
- Last layer of nodes are the leaves of the tree.
- The nodes in between are intermediate nodes or mid nodes.
- Two nodes sharing a common ancestor, are called siblings
- The nodes attached to the same node are the children of the node which is the parent.
- We assume that every node has the same number children and that the leaves have no children
- **Branching factor**: number of children for each node
- **Depth of a tree (h)**: length of the longest path from the root to the leaf.
- Number of leaves in a balanced tree in a tree with branching factor k and depth h, is **k^h**. It is the number of combinations we want to explore.

What can we do with this structure?

We have to explore all the leaves, as we said they are all combinations of solution search space that we are interested in exploring.

If we want to instantiate the algorithm described above, we need a way for generating all combinations, so we can try to find a way of generating all the leaves of this tree.

To do so, we have to focus on how they look like, so that we can invent a way to display them.

They look like a base two count. We can explore all possible leaves of this tree just by counting in base k (branching factor). We increment symbols starting from the right, when we arrive at the last one we increment the one on the left.

We need to invent an algorithm that is able to count numbers in a generic base k.

```
NEXTLEAF(a, L, k)
1  for i ← L to 1
2      if a_i < k
3          a_i ← a_i + 1
4          return a
5      a_i ← 1
6  return a
```

The combinations have to be explored by incrementing the number on the right. L is the number of the considered position in the combinations that we are exploring. We start by considering the last position of the combination, and we go to the first element. We have to increment the value in the considered position only if it is smaller than k, if we increment we return a. If not we reset it (K0) and we consider the position on the left, trying to increment it.

```
AllLeaves(L, k)
1  a ← (1, ..., 1)
2  while forever
3      output a
4      a ← NextLeaf(a, L, k)
5      if a = (1, 1, ..., 1)
6          return
```

This can generate all possible leaves. If the combination is the last one we exit, otherwise we keep iterating to the next leaf.

```
BruteForceMotifSearchAgain(DNA, t, n, l)
1  s ← (1, 1, ..., 1)
2  bestScore ← Score(s, DNA)
3  while forever
4      s ← NextLeaf(s, t, n − l + 1)
5      if Score(s, DNA) > bestScore
6          bestScore ← Score(s, DNA)
7          bestMotif ← (s_1, s_2, ..., s_t)
8      if s = (1, 1, ..., 1)
9          return bestMotif
```

At this point we can explore all of the search space but we are not really exploiting the tree structure. We want to exploit the tree structure more to get a real advantage. Looking at the middle nodes of the tree will allow us to cut parts of the search space. Via observing the middle nodes we can skip. Cut parts of search space.

We need to invent a smart way to explore the tree: we want to explore the tree in depth.

We start from the root, then we recursively explore the left subtree and right subtree. If there are no children, we do nothing, the recursion has reached the max depth and it goes back. The following node is the deepest sibling of an ancestor that has not been explored yet.

The definition is intricate but the algorithm is easy.

What are the differences and similarity of two subsequent vertices (9 and 10) ? They share the first element. The last two elements that were present in 9 are ignored, they are deselected, the first symbol that can be incremented is incremented.

```
NextVertex(a, i, L, k)
1  if i < L
2      a_{i+1} ← 1
3      return (a, i + 1)
4  else
5      for j ← L to 1
6          if a_j < k
7              a_j ← a_j + 1
8              return (a, j)
9  return (a, 0)
```

The first part deals with middle nodes, the second with leaves. If the current node (i) is not a leaf (its depth does not correspond to the height of the tree) we need to assign the first symbol to a new position (i+1), then we return the node. If we are at a leaf, we have to generate all siblings before going up. We just increment the rightmost element until we reach k.

J is the depth of the two nodes, we have to regress in the depth until we find a position that can be implemented, we will restart from there to explore the tree going down. We do the same thing as before, but instead of resetting we cancel, we ignore the positions that are already at k. We need to generate a new node, in which the last symbol has been incremented and all the successive nodes on the right are cancelled.

In line 7 we update the right most symbol, then in the next element we consider up to position j.

We can integrate this with the previous algorithm:

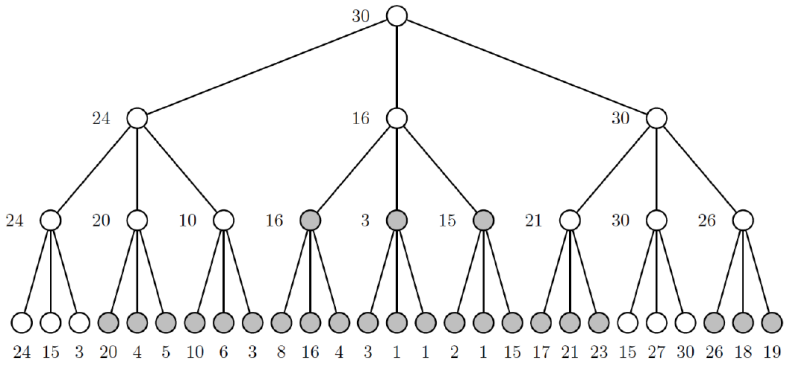
```
SIMPLEMOTIFSEARCH(DNA, t, n, l)
 1  s ← (1, ..., 1)
 2  bestScore ← 0
 3  i ← 1
 4  while i > 0
 5      if i < t
 6          (s, i) ← NEXTVERTEX(s, i, t, n − l + 1)
 7      else
 8          if Score(s, DNA) > bestScore
 9              bestScore ← Score(s, DNA)
10              bestMotif ← (s_1, s_2, ..., s_t)
11          (s, i) ← NEXTVERTEX(s, i, t, n − l + 1)
12  return bestMotif
```

This algorithm is exploring all nodes, and intermediate nodes are skipped. This algorithm produces the same result as the previous one, the difference is that this one is actually slower, we are also checking intermediate nodes for no reason.

Branch and bound technique

Assume that the **leaves** of a tree are **associated with a numerical value**. We would like to **spot the leaf with maximum value**. What we would do is to look at all the leaves before finding the best one.

Imagine that we have the possibility, once we arrive at a node, to estimate what is the best value that we can find in the subtree rooted in that node. Imagine we already know what is the estimation of the best leaf in the specific subtree we are considering. Once we know the highest score, we will compare it with the other estimations. If another estimation is higher (in a maximisation problem), then there is no point in exploring this subtree, we should explore the other one.
If we are able to estimate how good the tree is and the value is not good enough, then we can skip that part of the tree.



Assume we have arrived at a mid node we know that there is no point in exploring the subtree, we cut it. We have simply to consider the mid node as a leaf.

```
BYPASS(a, i, L, k)
1  for j ← i to 1
2      if a_j < k
3          a_j ← a_j + 1
4          return (a, j)
5  return (a, 0)
```

We can exploit all previous algorithms only if we have an estimation of the score of a subtree.

We will focus on the **motif finding problem**. The idea is that we want to spot the right positions for which we have the maximum score.

We take the segments and put them in an alignment matrix. Once we have it we can compute the statistic, based on that we compute the score: the sum of the number of times the most frequent symbols show up.

Imagine that we have decided two starting positions. We consider the score of this partial solution, obtained considering only two indices. We prepare an alignment matrix from which we obtain a score for the intermediate alignment.

The idea is that we know that the **score of a complete solution** - in which we have decided all positions - is **found starting from the partial, adding something**.

If we **consider a partial solution and its score**, and this score is **so low** - compared with what we have seen up to now - that **even in the best scenario** it is still lower than what we have seen in the previous possible solutions, it makes no sense to continue checking for a complete solution that starts from this partial solution.

If the first part of the solution is already bad, there is no point in seeing how bad we can do.

The partial score is $Score(s, i)$ with i indexes fixed. If we start from this, which is the best score that we can achieve? The **best case** is the case in which we **find the consensus sequence**, they would **add up for** $l * (t - i)$**.**

In the best case we would be able to find points in DNA whose sequence is the consensus. From the partial solution, if we only find consensus, we would find $score + l * (t - i)$.

```
BRANCHANDBOUNDMOTIFSEARCH(DNA, t, n, l)
 1  s ← (1, ..., 1)
 2  bestScore ← 0
 3  i ← 1
 4  while i > 0
 5      if i < t
 6          optimisticScore ← Score(s, i, DNA) + (t − i) · l
 7          if optimisticScore < bestScore
 8              (s, i) ← BYPASS(s, i, t, n − l + 1)
 9          else
10              (s, i) ← NEXTVERTEX(s, i, t, n − l + 1)
11      else
12          if Score(s, DNA) > bestScore
13              bestScore ← Score(s)
14              bestMotif ← (s_1, s_2, ..., s_t)
15          (s, i) ← NEXTVERTEX(s, i, t, n − l + 1)
16  return bestMotif
```

We can update the algorithm: for the middle nodes we estimate the optimistic score. If this score is smaller than the best score registered so far, we skip the subtree.

We use this **bound**, because if we know that it is very low, we skip it.

If our algorithm has to be correct we cannot skip the parts in which the estimation is enough to be higher than the maximum.

Median string problem: this is a symmetric problem of motif finding.

We compute similarly as before, the measure of how good the distance is in the case of a substring. If the hamming distance of a partial substring is higher than the best case we have seen so far we can discard the subtree. The best case that can happen during the completion of a partial solution is that the additional part doesn't increment the distance. Essentially if our partial solution is already more different than a complete solution we can discard it.

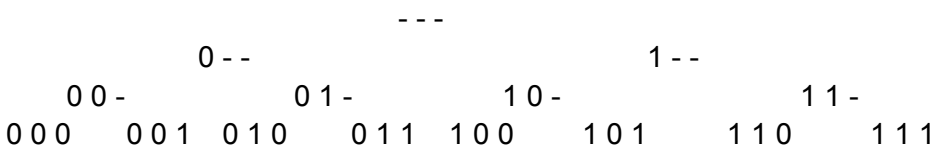
```
BRANCHANDBOUNDMEDIANSEARCH(DNA, t, n, l)
 1  s ← (1, 1, ..., 1)
 2  bestDistance ← ∞
 3  i ← 1
 4  while i > 0
 5      if i < l
 6          prefix ← nucleotide string corresponding to (s_1, s_2, ..., s_i)
 7          optimisticDistance ← TOTALDISTANCE(prefix, DNA)
 8          if optimisticDistance > bestDistance
 9              (s, i) ← BYPASS(s, i, l, 4)
10          else
11              (s, i) ← NEXTVERTEX(s, i, l, 4)
12      else
13          word ← nucleotide string corresponding to (s_1, s_2, ... s_l)
14          if TOTALDISTANCE(word, DNA) < bestDistance
15              bestDistance ← TOTALDISTANCE(word, DNA)
16              bestWord ← word
17          (s, i) ← NEXTVERTEX(s, i, l, 4)
18  return bestWord
```

Knapsack problem

Input: set n of elements, set of values for the elements, set of weights, threshold for the weight.
Output: subset of n, s.t. the sum of weights is smaller than the threshold and the value is maximum.

For each element we have to choose whether to take it or not. Imagine we have 3 items: a television, a hairdryer and a fork. We have to organise all possible combinations of objects as a tree, for each of the objects, we simply have to decide, we take it or not. Assume 0 is not taken and 1 is taken

The tree is the following:



In the algorithm we will traverse the tree and generate all combinations, storing the best one. As we explore the tree, there are two possible cuts that we can perform:

- weight: if a sub-solution has a weight higher than the threshold we can cut the subtree.
- value: if we arrive at a node in which the value given by the partial combination plus the value of all remaining objects not considered so far, is still smaller than the maximum value, then there is no point in continuing the combinations.

There are problems for which there aren't better options than exploring the whole search space. For these problems the only correct approach to solve them is an exhaustive search. All these techniques are going to speed up the problem, but just in the average case, in some cases the solution still requires a lot of time and the complexity of the algorithm is still exponential.

Greedy paradigm

We looked at exhaustive search, in that case we had to look at the output, define a search space, then explore the entire space looking for a solution. This approach is easy to design, we generate all possibilities and select the best one. These approaches are slow because they are used in NP hard problems that have big search spaces. If the search space was smaller they would take less time. Remember that the definition of too much time depends on the context. Maybe for certain operations an hour is too much.

NP hard problems have a combinatorial solution and there are no polynomial algorithms able to solve them. Formally speaking there is no way of quickly solving tough problems. So we are forced to use exponential algorithms in problems with combinatorial solutions, that's what makes exhaustive searches long.

An interesting approach to this problem is to use faster algorithms that are not exact. We can decide to have a **quick solution at the expense of the precision of the solution**, it is a **trade-off**. If we want a super optimal solution we have to wait. If we don't want to wait and we are willing to lose precision in the solution, we can use greedy techniques. The idea is that we are facing a tough problem, we cannot wait too long, so we buy a quick solution at the cost of the precision of the solution.

Idea behind greedy approaches

Greedy algorithms proceed by **making choices which are locally optimal**.

In the last practicals we needed to generate all combinations of the suppliers. We had to discard or admit each supplier and check the score of each combination.

In a greedy approach, we need to build candidate solutions, we look at the option that looks best at that moment of the choice. In the problem of suppliers we could look at the supplier that gives the maximum substance.
At each choice point, while we build the solution, we will pick something that looks good in that moment, we will **never dispute the choice again**. Once we choose something we never change the solution.

For example in suppliers we could take the supplier providing us the maximum of the substance, or the one having the minimum number of incompatibility.
Any criteria doesn't guarantee optimality, we base on local optimality criteria.

This kind of approach is fast because we perform a choice that can be taken in polynomial times, and then it is never changed. We can build up a set of suppliers, compatible, and provide something that we think is a good amount of substance.

Are we sure that what we obtain is optimal? Once we carry out this algorithm we are **not sure that we will get the global optimal**, we are just taking **local optimal solutions**.

We are discarding part of the structure of the problem. We look at the best thing in the moment and we never go back.

Can a greedy algorithm provide the global optimal solution? It can happen, by chance, but we will never know that it is the optimal solution.

The advantage of using a greedy approach is that we can do things quickly. If we need a fast answer this is the only approach that we can consider.
Another super cool feature of greedy approaches is that for certain problems - easier than NP hard - greedy approaches can provide us the best solution. Is it always the case that greedy algorithms provide a non-optimal solution? No.

Ex. We can implement a greedy approach in the problem of finding the shortest path given a source and a destination. It can be proved that this kind of problem has such structure that a greedy approach can solve it optimally.
There are problems whose best solution can be built with greedy approaches. These are easy problems. Nobody has ever managed to prove that it is possible or impossible to optimally solve a NP hard problem quickly.

We want to use greedy approaches on hard problems, because they take exponential times. We cannot wait for the best solution so we give up on the optimality of solution.

Genome rearrangement problem

We have two sequences of genomes. In order to understand how far genomes of different species are, we need to measure. To compute this measure we can consider the steps to transform a genome into another.

Design an **algorithm capable of finding the minimum amount of rearrangement to transform a genome into another**. The rearrangements that we are allowed to perform are only reversals (a portion of the genomic sequence is reversed). Given two genomic sequences we need to compute the minimum amount of reversals to transform one into the other.

Abstraction of the problem:

We need to introduce the concept of permutation: when considering a list of numbers, a permutation σ (sigma) is a function going from the original list of numbers to another list with a different order of numbers. The permutation function receives a list of numbers from 1 to n in input and gives in output another list of numbers from 1 to n.

Ex. 32415 is a permutation of 12345. A permutation is a sequence of numbers.

Among all possible permutations there is one, which we call identity permutation (12345). If we put numbers in natural order we have an identity permutation.

Given a permutation $\pi = \pi_1, \pi_2, \ldots, \pi_n$ with $\pi_i \in \{1, \ldots, n\}$, a reversal of a portion of π from index i to j is: $\pi_1, \ldots, \pi_i, \ldots, \pi_j, \ldots, \pi_n \rightarrow \pi_1, \ldots, \pi_j, \ldots, \pi_i, \ldots, \pi_n$ which we indicate as $\pi * \rho$ (rho).

We could start from the permutation $\pi = 1243756$ and apply to π the reversal $\rho(3, 6)$, obtaining the sequence $\pi * \rho(3, 6) = 12(5734)6$

We can define this problem:

Reversal Distance Problem:
Given two permutations, find a shortest series of reversals that transforms one permutation into another.

Input: Permutations π and σ.

Output: A series of reversals $\rho_1, \rho_2, \ldots, \rho_t$ transforming π into σ (i.e., $\pi \cdot \rho_1 \cdot \rho_2 \cdots \rho_t = \sigma$), such that t is minimum.

We will consider a variant of this problem. Often the sequence σ is assumed to be the identity of the permutation. The input is only the permutation π, we want to compute the sequence of reversal to get σ.

Sorting by Reversals Problem:
Given a permutation, find a shortest series of reversals that transforms it into the identity permutation.

Input: Permutation π.

Output: A series of reversals $\rho_1, \rho_2, \ldots, \rho_t$ transforming π into the identity permutation such that t is minimum.

To transform the permutation π in the identity permutation we have to solve a sorting problem. Can we use selection sort? No, although the input and output are the same, in this problem we want to sort it only by reversal. With selection sort we would do it one by one.

We cannot use merge sort either. Although it looks like a sorting problem we cannot use any sorting algorithm that we described before. Just reversal operations are allowed.

A **first approach** similar to selection sort is the following:
We look for the first element: is it true or not that the smallest element is in first position? No, then we perform the reversal of the first part of the permutation up to the smallest element. Then we exclude the first element and we search for the smallest among the remaining ones.

Ex.
Input: 32541(σ=12345)

We revert the whole permutation to get the first element in first position: 14523. Now we consider the first element fixed.
Is it true or not that the second-smallest element is in second position? No. So we perform a reversion from position 2 to position 4: 12543. Finally we revert the last 3: 12345.

We consider the elements in ascending order, if the i_{th} element is not in position i (say that it is in position j), we simply apply a reversal to the sequence, from the position i to the position j. We perform the reversal in the portion i-j, it means from i, to j, the position of the element we are considering.

```
SIMPLEREVERSALSORT(π)
1  for i ← 1 to n − 1
2      j ← position of element i in π (i.e., π_j = i)
3      if j ≠ i
4          π ← π · ρ(i, j)
5          output π
6      if π is the identity permutation
7          return
```

Does this algorithm provide a **good answer**? Since we are putting each time the i_{th} element in position i, the sequence of reversal is meaningful for the problem.

Is this algorithm **correct**? **No**, it doesn't do anything to optimise the sequence of reversals. It doesn’t provide the output we are expecting. This algorithm provides meaningful answers but not necessarily the optimal (best) one. If the problem was defined as: provide any sequence of reversals, this algorithm would be correct. But if we have to minimise the sequence we cannot rely on this algorithm. This algorithm is an **approximate algorithm**.

An approximation algorithm is an algorithm that provides a meaningful answer but not necessarily the best one.

Imagine we are trying to solve a decision problem in which the answer is yes or no. in this case a greedy algorithm is more than enough, if we find a possible solution we can already provide an answer. In the case of a computation problem that requires optimality we cannot use this algorithm.

Is it true that all greedy algorithms are approximation algorithms? No, there are greedy algorithms that provide the true answer.

Approximation ratio

We introduced the lecture by saying that we need a fast way to solve hard problems. This concept is a bit ambiguous. If we want to decide whether we want to get a bad answer fast, we need to know how bad the answer is. If the answer is too wrong, maybe it is not worth it to save time. We need to introduce a concept of **approximation ratio**. We need to estimate how good the answer of a greedy algorithm is, then we will be able to decide if we want to trade precision for time.

Pancake flipping problem

We have a pile of pancakes with different diameters, in mixed order. We want to turn pancakes to build a pyramid with the biggest on the bottom and the smallest at the top. We want to perform some reversals, we are not allowed to reverse pancakes in the middle. We can only reverse pancakes from a given level to the top. We decide where to put the “turner” and we turn all of the elements up to the last.

The pancake flipping problem and the identity combination problem have the same input and the same output. The thing that is different is the relation between them. The input of the pancake flipping problem is a permutation, the output is the shortest sequence of reversals to transform the input permutation into the identity permutation.
The only difference is that in the pancake flipping problem the reversals that we are allowed to perform are reversals starting only from the first position. Only in the prefix reversals are allowed. An algorithm to solve this problem is a variation of what we saw before.

In order to bring the biggest pancake to the bottom we need two flips. We bring it to the top and then we flip everything to put it at the bottom. Then to put the second-biggest in the second-last position we need again 2 flips, we bring it to the top and then we flip everything up to the second-last position.

For each initial pancake array we obtain a sequence of reversal building the pyramid. Since we need two flips each time, in the worst case scenario we need 2*(n-1) reversal operations.

A smarter way would require 5/3(n+1), it was discovered by Bill Gates. We don’t care about this better algorithm, the question is: How good can we get? Are there better solutions? In order to deal with this concept we need to introduce the concept of approximation ratio.

Recap: we were describing greedy algorithms, we said that in various circumstances they provide approximate solutions in this case they are considered approximate algorithms. Since we are using approximate algorithms to get answers for problems that generally are very tough to save time, we need to know how bad are the answers that we get, to decide if we are willing to accept it. We need to measure how far is the approximate solution proposed compared with the optimal one.

Example: knapsack problem
Assume we know that the best combination of objects has value 100 and assume we have a greedy algorithm and we run it. The greedy algorithm gives in output a combination with value 50.
The intuition of the notion of approximation ratio is measuring how distant is the proposed solution from the optimal. We say that on this input the approximation ratio of the algorithm is 2,

because the optimal value is twice as big as the approximated one.
If we have a problem for which the optimum solution is 100 and the approximate solution is 33 we have an approximation ratio of 3.

The approximation ratio of the algorithm depends on the input. Sometimes it might provide better or worse answers.

This is the definition just for knapsack, which is a maximisation problem. The approximation ratio can also be defined for minimisation.

Example: reversal problem
The shortest sequence of reversal if 5. The approximate algorithm gives 10 as result. In this case the approximation ratio of the algorithm is 2, because the approximate solution is twice as big as the true one.

The definition of approximation ratio varies whether we are considering maximisation or minimisation.

The approximation ratio is a concept linked to algorithms, not the ratio of a problem. It is a characteristic of an algorithm.

Definition: Assume we have a problem and an algorithm A to solve it. Let us consider X as an instance (input) of the algorithm. We denote as OPT the value of the best answer when the input instance is X.

The approximation ratio of algorithm A on input X is defined as:

$$AR_{\mathcal{A}}(x) = \frac{OPT(x)}{\mathcal{A}(x)}, \text{ if we consider a maximization problem}$$

$$AR_{\mathcal{A}}(x) = \frac{\mathcal{A}(x)}{OPT(x)}, \text{ if we consider a minimization problem}$$

In maximisation problems we measure how much smaller we are from the optimal solution, in minimisation problems it is the opposite, we measure how much bigger we are with respect to the optimal solution.

These definitions are provided in a way s.t. the approximation ratio is a number >=1, the bigger the number the worse solution it gives. If an algorithm has an approximation ratio of 3, and another has 5, it means that the one with 3 performs better than the one with 5.

The **approximation ratio is defined with respect to specific input instances that we are considering**. An algorithm can have different performances with different input instances. With some it may perform well and with others not so well. The approximation ratio may vary depending on the input instance. We need to generalise this concept. We want to have a generic definition, we want to free ourselves from considering a specific input instance.

Approximation ratio of an algorithm on an input of size n:

$$AR_{\mathcal{A}}(n) = \max_{|x|=n} AR_{\mathcal{A}}(x)$$

We consider all the possible inputs of length n, among those we pick the one that performs worse, this is the approximation ratio of the algorithm. The normal approximation ratio is defined

with respect to specific instances. To know the general approximation ratio of an algorithm we need to take the worst case scenario, in an input of a certain size.

We can use this concept to evaluate the approx ratio of the algorithm for sorting by reversal.

If the algorithm returns 5 reversals when instead they would be 2, the algorithm has an approximation ratio of 5/2. This is the approx ratio of the algorithm in this specific case, but what if the input was longer?
Assume we are running this algorithm on an instance like this: N, 1, 2, …, N-1. We can see by eye that the minimum number of reversal would be 2, flip all of the sequence and then flip the first n-1 numbers. If we run the greedy algorithm on the input permutation, it will propose us N-1 reversals: it would flip each couple of numbers to bring N in the last position.
On a generic input of size N. The approx ratio of this algorithm would be N-1/2

The approx ratio of this algorithm depends on the size of the input, the bigger the input the worse is the algorithm. With increasing input the mistake of the algorithm increases, linearly. We can say that the approx ratio of the algorithm is at least the one we found. It can even be worse.

There are algorithms whose approx ratio is constant. They guarantee that their error is a specific value. They do not perform worse as the size of the input increases, in these cases we talk about bounded approximation ratio.

Exercises

1

Say we have a problem that is solved by algorithm A. Algorithm A has an approximation ratio of 4 for an input π.

Let's imagine that the output of A is 12, what is the optimal solution for π?

$\frac{optimal(\pi)}{A(\pi)} <= 4$

$optimal(\pi) <= 4 * A(\pi)$

$optimal(\pi) <= 4 * 12$

2

What is the approximation ratio of the algorithm that solves the better change problem? We didn't take into account the fact that we might have to start from smaller coins. Take the biggest and then go to the smaller one until the total change is reached.

Imagine we have in input: (25,20,10,5,1) and n=40.
Our algorithm will propose the following solution: 25*1, 10*1, 5*1.
While the optimal solution is: 20*2.
The approximation ratio is 3/2.

There are cases in which this can be even worse. We need to find situations in which this algorithm really performs badly.

If we invent strange monetary system: (51, 50, 1) and we have n=100, the optimal solution is: 50*2. But our algorithm will give: 51*1, 1*49.
The ratio is: 50/2=25, this is really bad.

We can generalise to something like this: if we have input with (n+1, n, 1) and an amount of money 2*n, the optimal solution is always 2, and the algorithm always gives n. The ratio will be n/2.

Bounded approximation ratio

We would like to introduce an algorithm that has a bounded approx ratio, that is to say, an algorithm whose solution, although we don't have guarantee of optimality, has a distance from the true value that is always the same, it doesn't increase with input size.

Genomic rearrangement problem

Assume we have a particular permutation, we give few names:

- **Adjacent elements**: two consecutive numbers of the permutation of which value differs by one.
- **Breakpoint**: two consecutive elements that have a value that differs more than one make a breakpoint.
- **Strips**: portion of permutation between two breakpoints. Increasing or decreasing depending on whether the numbers increase or decrease.

Our algorithm will solve this problem via a series of reversals.
We can use breakpoint as an estimation of how far we are from the identity. This is a criteria among others. The more breakpoints in a permutation, the more distance we are from identity. If there are no breakpoints we are at the identity perm.

The intuition is: we are looking for reversals that give a decreasing number of breakpoints.

First of all we need a way to understand if the first element is in first position and the bigger in the last, for this reason we add two dummy elements: 0 and n+1, at the beginning and at the end. In this way, if the first element has a breakpoint with zero, then it is not 1. Same at the end.

Say we are working with this permutation: 0|21|345|876|9. Consider the portion: 45|87, if we reverse this portion of the permutation the number of breakpoints will increase, we are disrupting the strip 345. Reversing in between strips doesn't make sense, it makes the situation worse.

We will reverse portions with breakpoints as borders. Doing so breakpoints will decrease or remain the same.

What's the maximum number of breakpoints we could get rid of in the best case through a reversion? Consider the portion |21| in the example above, if we revert it, the number of breakpoints decreases by 2.

For a permutation π, let $b(\pi)$ denote the number of breakpoints in π; we have that $0<=b(\pi)<=n+1$. In the identity permutation $b(\pi)=0$. If there is a breakpoint between any two contiguous positions, we have n+1 of them.

Any reversal p(i,j) in π can decrease $b(\pi)$ by at most 2, because: since p(i,j) reverses everything in between the ith and jth positions, all breakpoints and adjacencies between these two positions are maintained. In the best scenario, the two breakpoints at the border of the reversal become adjacencies after the reversal.

If d(π) denotes the minimum number of reversals to sort π, d(π)>=b(π)/2, because we have to get rid of all breakpoints to arrive at the identity, and each reversal eliminates at most two breakpoints.

We have that b(π)/2 is the minimum number of reversals in order to get from π to the identity permutation. We are not sure about this, we may have to do more passages than this, this is a **theoretical limit**.

The reasoning followed so far brings to this algorithm:

```
BREAKPOINTREVERSALSORT(π)
1  while b(π) > 0
2      Among all reversals, choose reversal ρ minimizing b(π · ρ)
3      π ← π · ρ
4      output π
5  return
```

If b(π) is higher than zero it means we are not at identity. Among reversals available we take one which maximises the decreasing of the number of breakpoints. We apply the reversal and then we repeat with the new permutation.

Does this algorithm terminate? Can we give an estimation of n of reversals?
In this way the algorithm doesn't work, we are not sure that there will be a reversal that will decrease the number of breakpoints.

In order to create a reliable algorithm we need a theorem and an assumption:
Theorem: if a permutation π contains a decreasing strip, then there is a reversal p that decreases the number of breakpoints in π, that is b(π*p)<b(π).
Whenever there is a decreasing strip we will always find a reversal that decreases the number of breakpoints.
Assumption: every string of length one is considered a decreasing one (except for the dummy elements, if they are alone they are considered increasing).

Example and proof:
0127658439
Let K be the smallest number appearing in a decreasing strip. In the example the smallest is 3.
Focusing on the element K-1, in this case it is 2, we can say that:
- K-1 appears in an increasing strip, we are assuming that the smallest number appearing in a decreasing strip is K, it cannot be the case that k-1 is in a decreasing strip.
- K-1 terminates its strip, otherwise K would belong to the same increasing strip containing K-1.
Hence, K-1 and K correspond to two breakpoints: the former at the end of an increasing strip, and the latter at the end of a decreasing strip.

If we revert everything between k and k-1, we will remove at least one break point. We can bring the element k close to the element k-1 and they will be part of the same increasing strip.

If we have no decreasing strip there is no way of decreasing the number of breakpoints. But we can create a decreasing strip for the next step: we pick any increasing strip, we revert it, and at the following step we will have a decreasing strip.

Based on this, we can build a new algorithm and estimate its approximation ratio.

```
IMPROVEDBREAKPOINTREVERSALSORT(π)
1  while b(π) > 0
2      if π has a decreasing strip
3          Among all reversals, choose reversal ρ minimizing b(π · ρ)
4      else
5          Choose a reversal ρ that flips an increasing strip in π
6      π ← π · ρ
7      output π
8  return
```

In the worst case scenario this algorithm reduces the number of breakpoints in π by one every two steps. It can be the case that if π has a number of breakpoints, b(π) , this algorithm might need 2*b(π) reversals before arriving at the identity permutation.

This is a minimisation algorithm. We said that the approximation ratio of minimisation algorithms is the output of algorithm over the optimal value:

$AR <= \frac{A(x)}{OPT(x)}$

$AR <= \frac{2b(\pi)}{b(\pi)/2}$

$AR <= 4$

This means that this algorithm can perform badly. But the approximate solution is never 4 times bigger than the optimal value. It doesn't depend on the input.

Motif finding problem

How to apply greedy approaches to the motif finding problem?

Problem: find in a set of T dna sequences the starting position of the motif in each string.

For this problem we saw exhaustive search approaches, which considered all possible combinations of starting positions.

Greedy approach: focus on the first 2 sequences. For these two only we try all possible combinations of starting positions. Generate all possible pairs of starting positions only for the first two sequences. Then we find the best starting position for these two.
Once we have identified the best starting position for those sequences, we never change them again. We consider the remaining DNA sequences, looking at the position maximising the partial score.
Starting from the two initial sequences, we compute sequence W, the consensus string. Then we check for all the other strings: what is the string that is most similar to the two that we previously found? Once w has been computed we look at the other sequences to find the portion most similar to w.

The algorithm works in polynomial time, but no bound is known on its approximation ratio. It is however a useful algorithm in practice, it is a good compromise between performance and accuracy.

```
GREEDYMOTIFSEARCH(DNA, t, n, l)
 1  bestMotif ← (1, 1, ..., 1)
 2  s ← (1, 1, ..., 1)
 3  for s_1 ← 1 to n − l + 1
 4      for s_2 ← 1 to n − l + 1
 5          if Score(s, 2, DNA) > Score(bestMotif, 2, DNA)
 6              BestMotif_1 ← s_1
 7              BestMotif_2 ← s_2
 8  s_1 ← BestMotif_1
 9  s_2 ← BestMotif_2
10  for i ← 3 to t
11      for s_i ← 1 to n − l + 1
12          if Score(s, i, DNA) > Score(bestMotif, i, DNA)
13              bestMotif_i ← s_i
14      s_i ← bestMotif_i
15  return bestMotif
```

With the first part of the algorithm we find the consensus in the first two sequences. After we find it we start searching in the other sequences, one by one, and we find the best starting position in those sequences. Looking at the score, which is maximised.

In the branch and bound techniques we cut portions of search space when we are sure that there we cannot find anything interesting. But, even if we were able to cut half of the combination, the number of combinations that we needed to check was still exponential. The complexity was still exponential. This algorithm is much faster, because we are not cutting combinations that we are sure to be bad. We are cutting a lot to find a non-optimal solution.

In GreedyMotifSearch, we need to scan all possible combinations of two starting positions in the first two DNA sequences, and scan the l symbols of the motif. Therefore this operation requires $l(n - l + 1)^2$, which is $O(ln^2)$.

In the second part, for each of the t-2 remaining DNA sequences we select the l-mer that is the most similar to the one fixed in the previous part. We have $(n - l + 1) * l$ combinations, with t sequences, which is $O(n * l * t)$.

Hence, the overall complexity is $O(ln^2 + lnt)$ that is much better than the complexity $O(ln^t)$ of the exhaustive search algorithm. Generally l is much smaller than n, so we can approximate to $O(n^2)$.

Bin packing problem

We want to move from our house, and we have found a room. We have to bring some objects, they have a size bigger than zero and smaller than one. We need to find a way to pack our stuff using boxes of size one.

$U = \{1,\ldots, n\}$ objects

$S = \{Si,\ldots, Sn\}$ sizes $0 < s <= 1$

We can put whatever number of objects in a box if they do not exceed size 1. We want to find an allocation of objects into boxes, s.t. the number of boxes is minimised.

$U_i = \{U_1, \ldots, U_k\}$ boxes, subsets of U. K is minimised

An object must go in just one box.

$U_i \subseteq U$

$U_i \cap U_j = 0$

$\bigcup_{i=1}^{k} U_i = U$ We take everything with us.

This is a NP hard problem, it requires an exponential algorithm for its exact solution. We need an approximate algorithm to solve it in polynomial time. There exist various approaches to solve this problem.

First-fit approach

We open a box, we put stuff inside, and as soon as it is full we open another one. Once we pick an object, we put it in the first box that has the space to contain it. If there is no such box, we open a new one.

Pseudocode:

```
(S1, …, Sn) #array of sizes
(U1,..., Un) <- (0,..., 0) #n empty boxes, surely n boxes will contain n objects, worst case.
for i=1 to n:
        j<-the lowest index such that Uj has enough space to contain i
        Uj.add(i)
return the prefix of (U1,..., Un) of non empty boxes #select for boxes that are full and return them
```

We just consider the order of the object in input, we don't care if it is not optimal for the solution, this is an approximate solution.

Complexity: n iterations, for each iteration we traverse n boxes to find the one that has enough space space, this algorithm requires $O(n^2)$

How approximate is this solution?

Approximation ratio:
FF(I) number of boxes returned by algorithm
OPT(I) real optimal value of boxes
In order to understand which is the approx ratio of this algorithm we need to carry out some considerations: Can this algorithm provide some boxes that are very empty? No, the maximum number of boxes filled for less than half is 1 (if there were more than 1, they could be merged, the algorithm would have found it).
Lemma 50%: FF returns an allocation in which at most one box is filled for less than half of its capacity.

We denote by H the set of boxes Uj s.t. the sum of the size occupied by the objects i inside Uj is greater than 0.5. H are all boxes filled more than 50%

$$H = \{U_j | \sum_{i \in U_j} S_i > 0.5\}$$

We want to know what is the total size of object contained in H:

$$\sum_{U_j \in H} \sum_{i \in U_j} S_i$$

Since by definition of H the sizes of the objects inside the boxes is at least 0.5, we can rewrite this is this way:

$$\sum_{U_j \in H} \sum_{i \in U_j} S_i > \sum_{U_j \in H} 0.5$$

We know by lemma 50% that the boxes in H are all the boxes proposed by the algorithm minus 1, so we can write:

$$\sum_{U_j \in H} \sum_{i \in U_j} S_i > (FF(I) - 1) * 0.5$$

The total size of objects in H is at most equal to the size of all the objects:

$\sum_{U_j \in H} \sum_{i \in U_j} S_i \leq \sum_i S_i$ (it means that: $\sum_i S_i > (FF(I) - 1) * 0.5$)

The optimal value in an instance (I) must be bigger or equal than the total size of the object. If we take all objects and sum them up, we obtain that the total value is x, and each box is able to carry 1, at most each box is completely full, we need at least x boxes. The amount of boxes must be bigger than the total size of the objects.

$$OPT(I) \geq \sum_i S_i$$

We can relate these two:

$$OPT(I) \geq \sum_i S_i > (FF(I) - 1) * 0.5$$

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

$$2OPT(I) > FF(I) - 1$$

$$2OPT(I) \geq FF(I)$$

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

This is the approx ratio of the algorithm. This is a bounded approximation ratio: whatever the input is, the result returned will never be bigger than twice the optimal number of boxes.

Dynamic programming

We started investigating algorithm approaches with exhaustive search: very easy, we have to understand the output, and then look at the entire space of the candidate solutions by designing an algorithm that searches in the domain, sooner or later we will find the solution.
We discussed how exhaustive search is sometimes slow, it is not slow by itself, it depends on the size of the search domain.

Then we talked about combinatorial problems that do not admit a quick solution because they are hard. they can only be solved optimally with exhaustive search. We can only cut useless parts of the search space but the algorithm will still be exponential.

With greedy approaches we perform a local optimal choice to reach the solution. Each time we take a local optimal choice we don't reconsider it. We are not sure that building up a solution by a local optimality will give a good global solution. These algorithms are approximation algorithms because they don’t give the optimal solution. Greedy approaches are usually quicker. Because we chose and we do not reconsider our choice.

Actually in some cases we can get the optimal solution from the local optimum. And in some problems we might get sub optimal solutions.

Especially with NP hard problems, the only way to get the optimal solution is to try all of them. We cannot be sure that the solution proposed by a greedy algorithm is the optimal one. We introduced the approximation ratio to know how far is the estimation from the optimal.

This is an engineering problem to solve: sometimes we don’t have time to reach the optimal solution. For example streaming services need to compress the video to send it live on the internet. Video compression is something really expensive in terms of computational power. When we have to compress videos at 24 fps, we need to compress each frame in less than 1/24 of a second. For this reason sometimes greedy approaches are used.

Similarly to how exhaustive searches are not hard by themselves, the greedy approaches are not bad by themselves. If we apply greedy approaches to problems with a good structure (many of them) we will get good solutions.

In the middle between exhaustive search and greedy approaches there are fast algorithms that provide correct results. Dynamic programming approaches are greedy approaches, but they are executed in problems that have a structure that allow to get an optimal solution from a greedy algorithm. The approximation ratio of these algorithms is 1.

How is this possible? Only because the problems on which these approaches are applied, are problems with a good structure, that admit a solution.

Change problem

We will see this problem again. We have an amount of money for which we have to give a change. We saw a couple of solutions: exhaustive search approach that was very inefficient and a greedy approach, which was a generalisation of the american problem, and it was imprecise in some cases.

Now we will discover that we are very lucky because this problem has a particular structure that allows us to use a dynamic programming approach. Optimal solutions of instances of this problem can be built by looking at optimal solutions of subinstances of the problem.

Assume we have a monetary system (7,3,1), we need to develop an algorithm that returns the best combination of coins.
What is best combination when we need to give back 0 change: (0,0,0)
1: 1 coin of value 1.
2: 2 coins of value 1.
3: 1 coin of value 3.
4: 2 coins, 1 of value 3 and 1 of value 1.

We can generalise these results. When we want to find the best change for a given amount of money we need to give the combination of multiple changes for smaller amounts.

Ex. M=5
When we have to give the change we could give the amount of coin resulting fromthe solution for 4 plus a coin of value 1 or from the solutions for 2 plus a coin of value 3. How do we pick the best? we consider both, we take the smallest.

The intuition is that the best comb for a generic amount n, can be obtained by looking at the best combinations for smaller amounts.
If the number is 10, the intuition is that we need to consider **3 candidates**: 3, 7, 9. They are all the possible ways in which we can reach 10: 10-7, 10-3, 10-1.
We can take the minimum among M-7, M-3, M-1. If this works we will have a way to get the good solutions. We are not sure that this works. We can build the best solution for M by looking at the best solution for these values.

The proof is by contradiction:
Let's call i_M the minimum number of coins that are needed to reach the amount of money M. I_M is the minimum between i_{M-7}+1, i_{M-3}+1 and i_{M-1}+1. We build the best value of M starting from the best values for M-7, M-3 and M-1, this is our claim (this can be generalised, but we will stick to the example).

For the sake of contradiction, assume this is not true, we cannot obtain the best combination by looking at the best previous results. Assume we have a better way: $\bar{i}_M$.

If $\bar{i}_M$ is better than i_M, it means that $\bar{i}_M$ is smaller than all the 3 possible values for i_M. Assume that the minimum in our example is $i_M=i_{M-7}+1$.

$\bar{i}_M < i_{M-7}+1$ #i tilde is a better solution for M than the one we found

$\bar{i}_M <= i_{M-7}$ #since we are working with integers we can remove 1 and put smaller or equal

Now we have a relation between the best number of coins for M and the best number of coins for M-7. If we look at the best combination of coins for M, and we remove 1 coin of value 7, we obtain a combination of coins for M-7, we obtain a number related to the solution for M-7.

$\bar{i}_M - 1 < i_{M-7}$ #if we remove 1 from i tilde we should get a better solution for M-7

We have a number related to the best combination of coins for M-7 that is smaller than the best combination of coins that we have taken by assumption. We have taken as assumption that i_{m-7} is the best, we cannot have a value i~ for m-7 which is smaller than i_{m-7}.

```
RECURSIVECHANGE(M, c, d)
1  if M = 0
2      return 0
3  bestNumCoins ← ∞
4  for i ← 1 to d
5      if M ≥ c_i
6          numCoins ← RECURSIVECHANGE(M − c_i, c, d)
7          if numCoins + 1 < bestNumCoins
8              bestNumCoins ← numCoins + 1
9  return bestNumCoins
```

RecursiveChange's complexity can easily be shown exponential. This algorithm is extremely slow. The recursive version is very slow. This recursive algorithm is redoing a lot of work. We are not actually exploiting the nice property. What could we do to improve this?

Instead of starting from i_m, we have to start from i_0! We start from the bottom rather than from the top.

```
DPCHANGE(M, c, d)
1  bestNumCoins_0 ← 0
2  for m ← 1 to M
3      bestNumCoins_m ← ∞
4      for i ← 1 to d
5          if m ≥ c_i
6              if bestNumCoins_{m−c_i} + 1 < bestNumCoins_m
7                  bestNumCoins_m ← bestNumCoins_{m−c_i} + 1
8  return bestNumCoins_M
```

We have an array to store the minimum amount of coins we need to give as change (we don't store the best combination) for each number from 1 to M, the array has M positions and in position M we find the solution of the problem. We iterate from 1 to M, at each iteration we have to fill the position in the array corresponding to m.

We take as the initial best amount of coins 0. And we iterate through the available coins. If m is bigger or equal than the considered coin, we look at the best result for m minus the coin that we are considering (by querying the array) and we add 1. If this is smaller than the best number of coins we update the variable, otherwise we try with another coin. The possibilities depend on the number of coins in the system.

We need to store in memory all previous results, we are exchanging computation time with memory.

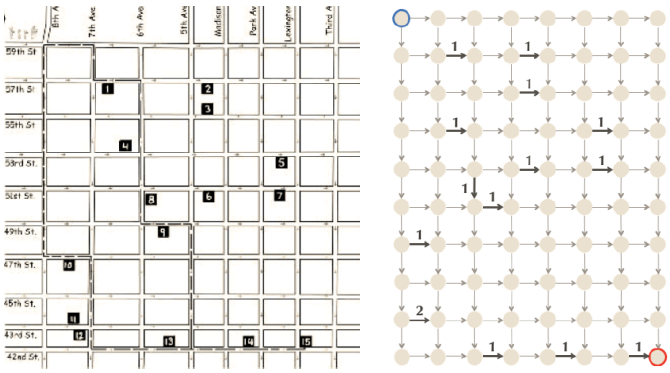
Manhattan tourist problem

We have a map of a city in which the roads are squares. Avenues are from north to south. Streets from west to east. Some roads are associated with numbers, they represent the number of attractions that can be seen if we drive on that road.

We have to start from the top left and finish to the bottom right. We want to maximise the number of attractions to see. Each road can be driven only in one direction.

We can generalise this problem with a graph. We have a directed graph, edges have direction, we can traverse edges only left to right and up to bottom. Through this graph we can model the problem. We associate with edges some numbers, called weights. These are the number of attractions that can be seen if we traverse that road. We want to find a path such that the total weight of the path is maximised.

In this example edges without number have weight zero.



Input of the problem is a graph that is called a grid. We are not taking in input a generic graph, just this shape.

We want to find the best path from a source vertex to a sink vertex s.t. the path maximises the weight on the edges that we traverse. We are going to see various approaches to solve this problem: exhaustive search, greedy approach and dynamic approach.

Exhaustive search approach

We have to generate all possible paths. We can use exhaustive search. It is just a counting problem, 0 is down, 1 is right. The output is a sequence of edges. The search space of the candidate solution is all the possible sequences of edges.
A possible way is to get the combinations of edges, then see if they make sense. Are we sure we will compute the best path? Yes, we are considering all possible combinations.

Greedy approach

We make a local choice, we pick one of the two possible directions outgoing from a node based on the weight of their edges (we chose the highest weight).
The exhaustive search approach has an exponential complexity, the greedy approach has a polynomial complexity. It doesn't guarantee a good solution. If the grid has a particular structure we will end up missing a lot, but this algorithm is much faster.

Dynamic programming approach

This problem can be solved in polynomial time, in an exact way via a dynamic programming approach. With this greedy approach a local optimal is transformed to a global optimal. Local optimal criteria allows us to build the best solution possible. Idea is similar to the change problem, we build the solution by looking at smaller portions of the input graph.

We know which are the best paths on the first column and row because we have just one solution. This allows us to say what is the best path.

Let's focus on the vertex 2,2. There are only two ways to reach this vertex, either we reach it from the top or from the left. No other option.

If we focus on further ones, there might be various ways to go from the source to the one we are considering, but all of the candidates can be divided into two families, whether they arrive from the top or from the left. The best way to reach a node is either coming from top or left.

To decide the best path, we could take the best path arriving at the top node and adding the top edge, or the best path arriving to the left node adding the left edge.

If we want to focus on this problem via dynamic programming, we need to compute the best values for all vertices on the first column and row (the number inside the vertex is the best score so far). Once we have these values available, we can proceed row by row or column by column. The maximum weight of a path reaching a vertex is either the max of the path from the top + top weight or the max of the path from the left + left weight.

We can make the choices for each vertex. In order to compute the best value of the last vertex we needed to compute the maximum weight of all paths from the previous vertexes.

In order to design an algorithm that is able to solve this problem we need a data structure capable of representing this grid: we just have to memorise the data in **two matrices** that store the edges.

Each vertex can be identified by coordinates (i, j), where 0 ≤ i ≤ n is the row index and 0 ≤ j ≤ m is the column index. Edge weights can be represented via a pair of matrices only storing the weights of edges between adjacent pairs of vertices:
A matrix →w, where →wi,j gives the number of attractions that can be seen by travelling from vertex (i, j − 1) to vertex (i, j).
A matrix ↓w , where ↓wi,j gives the number of attractions that can be seen by travelling from vertex (i − 1, j) to vertex (i, j).

There is no explicit representation of the vertices.

$\textsc{ManhattanTourist}(\downarrow\mathbf{w}, \rightarrow\mathbf{w}, n, m)$
1 $s_{0,0} \leftarrow 0$
2 **for** $i \leftarrow 1$ **to** n
3 $\quad s_{i,0} \leftarrow s_{i-1,0} + \downarrow w_{i,0}$
4 **for** $j \leftarrow 1$ **to** m
5 $\quad s_{0,j} \leftarrow s_{0,j-1} + \rightarrow w_{0,j}$
6 **for** $i \leftarrow 1$ **to** n
7 $\quad$ **for** $j \leftarrow 1$ **to** m
8 $\quad\quad s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} + \downarrow w_{i,j} \\ s_{i,j-1} + \rightarrow w_{i,j} \end{cases}$
9 **return** $s_{n,m}$

The complexity of the algorithm is proportional to the vertices of the matrix. It is polynomial, O(N^2).

The algorithm is correct because we are implementing the recursive definition. The procedure simply computes the recurrence relation for all the optimal Si,j.

What is the difference from exhaustive search? Why are we saving so much time? In order to compute the best value in destination we needed to compute the best value from source to all the other vertices. In exhaustive search each time we try a solution we throw away all of what we have done. In this case the problem has a structure that allows to reuse work. We are extending the previous solutions, this is the intuition behind dynamic programming.

Can we apply dynamic programming to a NP hard problem? From what we have said up to now, since they generate an optimal solution, we cannot apply them to NP hard, and find the optimal solution in polynomial time (NP hard problems are problems that cannot be solved optimally quickly).

But is there any reason why we cannot do this? We can use a dynamic programming approach in a tough problem, but our hope to obtain a quick answer vanishes. Dynamic programming approaches work on problems with a peculiar structure, clearly NP hard problems do not have a nice structure.

Is it true that in exhaustive search we cannot reuse any information from previous combinations? Can we reuse work?

Example knapsack:

We can compute the weights of some partial combinations, then reuse this value each time that subcombination occurs. Is it useful or not?

In knapsack and other combinatorial problems there is one main issue: these problems do not have a nice structure. When we store partial solutions in the case of the grid, we are storing partial solutions that for sure will be useful to build the total solution. They are for sure needed to get the final best solution.
With knapsack the structure is so intricate that we don't know which path to follow to find the solution, we would need to store in memory exponentially many partial solutions. This will not actually speed up the algorithm, we do not have a nice structure that allows us to store all the partial results in a matrix that can be traversed easily.

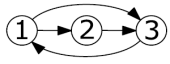
We can reuse work in knapsack, but it is not effective to speed up the work. In general, for all problems we can use whatever technique we want, but we need to see if it works and if it is practical.

Variation to manhattan tourist problem

Imagine a city with more possible roads. It can be modelled as a graph, a set of vertices and edges. We don't have a predictable structure of vertices and edges.

A directed graph is a pair G=(V,E), where V is the finite set of vertices and $E \subseteq VxV$ is the set of edges. An edge is an ordered pair of vertices.

Consider this graph:



V={1,2,3}, E={(1,3),(1,2),(2,3)}

A path in a graph is a sequence of consecutive edges (edges at which endpoint starts another edge) which joins a sequence of vertices. A path is characterised by a starting and ending node.
A path is a cycle if the first and last vertices in the path are the same.
A directed graph is acyclic (or DAG) when it has no cycles.

We can associate with each edge a number, we have a weight function or a labelling function.
A path is weighted if every edge comes equipped with a non negative number w.

Suppose we want to find the longest path between node 1 and 3. If we have a cycle the definition of maximum path is tricky, we could cycle through the cycle infinite time, that's the reason why we are interested in directed acyclic graphs.

Problem of the longest path in a directed acyclic graph:

Longest Path in a DAG Problem:
Find a longest path between two vertices in a weighted DAG.

Input: A weighted DAG G with *source* and *sink* vertices.

Output: A longest path in G from *source* to *sink*.

We can extend the approach of the previous problem to a more general case. In the Manhattan problem, by extending a previous optimal solution, we could obtain an optimal solution. We have to generalise this to generic graphs.We want to achieve a generic algorithm that works with generic graphs.
In order to find the value of the best path to a vertex, it is enough to look at the two predecessors. We have a graph with many vectors and we need to compute the best part from the start to the end.
Once we consider the end vertex, in order to know the value of the best path arriving at it we need to consider the predecessors.
The predecessors of v are the vertices w for which (w,v) ∈ E.

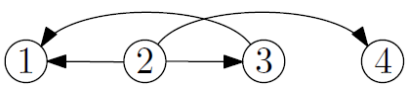
We may have n predecessors: V-1_1, V-1_2, …, V-1_n that generate v. L_1, L_2, …, L_n are the weights of the edges.

We look at the predecessors, we take the best value of the path to reach each predecessor, we add the weight of the edge L, and we select the biggest one. The intuition is that all the vertices of the graph need to be explored before the final node.

$$s_v = \max_{v' \in Predecessors(v)} (s_{v'} + w_{(v',v)})$$

This is done by topological ordering:

A topological ordering of a graph is a linear sequence of vertices of the graph, we put vertices in a sequence. In a topological ordering of a graph we want to place the nodes s.t. if we put them on a line, all edges go to the right.
A topological ordering is just a reshuffling of the vertices in a sequence, such that if we put them on a line, all the edges have the same direction. A cycle cannot be ordered topologically. Also for this reason we need acyclic graphs.



The graph represented above s is not a topological order, we would have to place the nodes as: (2, 3, 1, 4).

When we have ordered the vertices in a topological order, if we execute the algorithm presented before, then we are sure that whenever we are visiting a vertex, we have already explored all its predecessors.

To give the best value to a vertex we need to know its predecessors, we have to explore them all, for this reason we use topological ordering.

How to compute topological ordering of a graph? If graphs are not cyclic. We can see that the graph has layers. Since there are no cycles that bring us back, we can imagine a graph as a series of layers. All edges go from top to bottom.

Input: A DAG $G = (V, E)$
Output: A reordered list of the vertices E that is topological ordering

```
1  TopologicalOrdering(G)
2      verticesLeft ← V
3      edgesLeft ← E
4      result ← ∅
5      while verticesLeft ≠ ∅ do
6          Top ← the vertices of verticesLeft not having entering edges
7          result.append(Top)
8          verticesLeft.remove(Top)
9          edgesLeft.remove(all edges having an endpoint in Top)
10     return result
```

First of all we need to find vertices in the first layer of the graph, the vertices without entering edges, since they do not have entering edges we know that we can put them at the beginning. In this way we create the first layer of the topological ordering.
Then we delete the edges getting out of those vertices from the graph. Now we will have a new set of vertices without entering edges, these are the nodes belonging to the second layer. We build the topological ordering proceeding like this.

This is not a deterministic approach, we do not always generate the same topological ordering, the vertices of the same level may be interchanged.

Also in the classical Manhattan problem we are exploring the graph according to a topological order, the graph is just a grid so the topological order is really easy to find. The interesting point is that **whatever order we are following is a topological order**.

If the structure of the graph gives us a simple way to obtain the order we are lucky. In the other cases we have to build a meaningful order by considering layers (more irregular order, it needs more space because we have to store nodes and edges).

DNA alignment

We want to align DNA sequences, compare the nucleotide sequence of a gene with others already known. If they are similar, they share some function.
When we dealt with the motif finding problem we needed to invent a way to compare strings. Also in this case it is up to us to find a way to measure the distance.

We previously introduced the hamming distance, in which we count the differences. It is an easy way of defining distance, but this is not something which is the truth.

When we are considering nucleotide sequences it is not great to look at the hamming distance. Hamming distance is bad because we are bound to measure similarity between strings of the same length, it doesn't take into account regions of similarity and it doesn't consider indels. If we have two sequences that are just frame shifted they differ in every position, but if we align them by adding gaps we can have a very good alignment.

A more biologically meaningful distance is the **edit distance**.

The edit distance is the minimum amount of edit operations: insertions, deletions and substitutions in order to transform a string into another. The important part of this definition is the minimum amount of edit operations, there are many series of edit operations that can transform a sequence into another, we have to choose the shortest one.

The edit distance can be reconducted to estimate how well the two strings can be aligned. It can be seen as finding the best alignment. We need to a technique to compute the best alignment

```
-TGC-ATAT          -TGCATAT
ATCCGAT--          ATCCG-AT
```

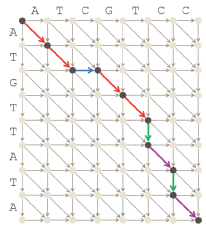
An alignment of strings v and w is a two row matrix such that: the first row contains the symbols of v (in order) and the second row contains w.

- Red symbols are conserved
- Violet symbols are substituted
- Blue symbols in the second string (aligned to dashes in the first row) are insertions to the first string.
  Greed symbols and dashes are deletions from the first string.

One alignment is better than the other: in the second alignment we have fewer edit operations. The edit operations are all symbols not coloured red.

We can enrich an alignment with a little more information: we can add two sequences of numbers to each sequence of the alignment. With the numbers we keep track of how many symbols we are seeing in the sequence (we are just making some information explicit).
The amazing thing is that we can interpret these numbers as coordinates in a graph.

We have an alignment grid or edit graph. On the rows we have the first string, on the columns the second string. We interpret the pairs of numbers as the coordinates of the vertices that are traversed in the alignment that we are considering.



- The edges traversed by this path tell us the edit operations:
  - Horizontal edges (→) are insertions
  - Vertical edges (↓) are deletions
  - Oblique edges can be either:
    - (↘) are substitutions
    - (↘) are matches (which are actually not an edit operation)

For whatever alignment we are considering there is a specific path in the edit graph.

In order to find the best alignment between two strings, given the tight relationship between alignments and path in a graph, we can solve the problem of alignment as the problem of finding a path in a graph. We want to find the best path in the graph that follows some criteria that we have to decide. This makes the problem of aligning two sequences similar to computing the path with maximum weight in a weight directed graph.

What differentiates a graph to another is the weight of the edges. We have to assign weights in order to select the paths that are more useful to us. If we want to spot paths that for us are good in an edit graph we need to invent weights for the edit graph, different weights give rise to different optimal paths.

If we invent a **scoring function** that associates a number to each edge, we can then find the best path. Different scoring functions on the edges of the graph result in different optimal paths.

Longest common subsequence problem

Among many different scoring functions we will first focus on a simple one. It is useful to solve a variant of edit distance problem, it is called longest common subsequence (LCS) problem.

A subsequence of a given sequence is an ordered sequence of characters taken from the initial sequence which are not necessarily consecutive (AGCA is a subsequence of ATTGCTA).

Given two sequences v and w, we want to compute the longest subsequence between the two.

In this case we are allowed to match, insert and delete symbols. The difference between LCS and the alignment problem is that in LCS substitutions are not allowed.

```
ATG--T--TATA
AT-CGTCC----
```

This alignment between v and w shows a common subsequence of length 3, however it is not the longest common subsequence. Since there is an alignment with a longer subsequence the one before is not the optimal.

```
AT-GT--TATA
ATCGTCC--
```

We need to show that all possible alignments are not better. We need an algorithm that shows that it is the best one.

Longest Common Subsequence Problem:
Find the longest subsequence common to two strings.

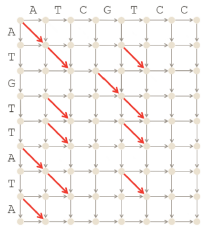
Input: Two strings, **v** and **w**.

Output: The longest common subsequence of **v** and **w**.

We need to assign weight such that the path corresponds to the longest common subsequence.

How can we do this? We can give score +1 to matches, and score 0 to gaps, we are interested in having the maximum number of matched symbols.
We can give negative weights to mismatches, but we need a low number as -∞, we cannot tolerate mismatches in the path we will choose. Another way to avoid mismatches is using a penalty that is bigger than the number of edges of the graph. Or we could also cut away the

edges that correspond to mismatches, in this way the algorithm cannot traverse them.



We will remove all edges corresponding to mismatches, in this way all remaining edges have weight one or zero. The maximum weight path is the path containing the maximum number of red edges (maximum weight).

Now we need to be careful: when considering a vertex, if the two symbols are equal we can also check the diagonal movement, 3 predecessors, otherwise we only have two possible movements.

We need in order to find the solution, we can adapt the manhattan tourist problem. Like in the Manhattan tourist problem, the optimal values for the first row and column of the edit graph are easily computed; they are all zeros as the gap penalty is zero. Then the recurrence relation for the dynamic programming iteration is:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + 0 & \text{(in case of deletion)} \\ s_{i,j-1} + 0 & \text{(in case of insertion)} \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \text{ (in case of match)} \end{cases}$$

The algorithm we used to solve that problem computes the value of the best path. In order to retrieve the actual alignment we need to add a data structure: **matrix of backtracking pointers**.
While we explore the edit graph we store information: whenever we compute the value of the best path going from a previous vertex to the current vertex we memorise the direction we were coming from in another matrix. In this matrix we store the information to retrieve the best path.

This has the same word of backtracking exhaustive search, in this case we are just going back through the path. It has nothing to do with backtracking technique. The backtracking technique is a way of building combinations of solutions by trying something, and, if it doesn't work, we go back and change our choice.

The complexity is polynomial, we fill the dynamic programming matrix and the backtracking matrix, then we traverse the backtracking matrix and we build the best path.

LCS($\mathbf{v}, \mathbf{w}$)
1 **for** $i \leftarrow 0$ **to** n
2 $\quad s_{i,0} \leftarrow 0$
3 **for** $j \leftarrow 1$ **to** m
4 $\quad s_{0,j} \leftarrow 0$
5 **for** $i \leftarrow 1$ **to** n
6 $\quad$ **for** $j \leftarrow 1$ **to** m
7 $\quad\quad s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \end{cases}$
8 $\quad\quad b_{i,j} \leftarrow \begin{cases} \text{“}\uparrow\text{”} & \text{if } s_{i,j} = s_{i-1,j} \\ \text{“}\leftarrow\text{”} & \text{if } s_{i,j} = s_{i,j-1} \\ \text{“}\nwarrow\text{”}, & \text{if } s_{i,j} = s_{i-1,j-1} + 1 \end{cases}$
9 **return** $(s_{n,m}, \mathbf{b})$

Procedure to rebuild the subsequence from the backtracking matrix, recursive approach:

```
PRINTLCS(b, v, i, j)
1  if i = 0 or j = 0
2      return
3  if b_{i,j} = "↖"
4      PRINTLCS(b, v, i − 1, j − 1)
5      print v_i
6  else
7      if b_{i,j} = "↑"
8          PRINTLCS(b, v, i − 1, j)
9      else
10         PRINTLCS(b, v, i, j − 1)
```

After we call the function we don't do anything, we print only when we have a match. After a recursive call we print the symbol associated with the edge.

Edit distance problem

We can solve the edit distance problem by changing the LCS problem a bit. We need to change the scoring function, we assign a cost of 1 to insertion and deletion, a cost of 1 to mismatches and a cost of 0 to matches. Then we need to search for a path that minimises the score (the weight).

$$d_{i,j} = \min \begin{cases} d_{i-1,j} + 1 & \text{(in case of deletion)} \\ d_{i,j-1} + 1 & \text{(in case of insertion)} \\ d_{i-1,j-1} + 1, & \text{if } v_i \neq w_j \text{ (in case of substitution)} \\ d_{i-1,j-1} + 0, & \text{if } v_i = w_j \text{ (in case of match/no edit)} \end{cases}$$

Global alignment problem

Global sequence alignment is a problem very related to the edit distance problem. In this case insertions, deletions, and mismatches have different costs. Instead of assigning a fixed cost to each edit operation, we have to assign a cost related to the single character (residue) we are deleting or inserting. These values can be changed by statistical analysis of real biological sequences).

Global Alignment Problem:
Find the best alignment between two strings under a given scoring matrix.

Input: Strings **v**, **w** and a scoring matrix δ.

Output: An alignment of **v** and **w** whose score (as defined by the matrix δ) is maximal among all possible alignments of **v** and **w**.

The approach is the same: a scoring function is giving us weights for the edges, we just need to change the computation of the weights and adapt the algorithm for the new weights.

$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma & \text{(in case of deletion)} \\ s_{i,j-1} - \sigma & \text{(in case of insertion)} \\ s_{i-1,j-1} - \mu, & \text{if } v_i \neq w_j \text{ (in case of substitution)} \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \text{ (in case of match/no edit)} \end{cases}$$

Each value might be defined by the user or retrieved by fetching a substitution matrix that has statistically relevant scores. Difference is that instead of having hard coded within the algorithm we have a scoring function in input.

The global alignment problem, as well as the edit distance problem and the longest common substring problem has quadratic complexity: we have to fill a dynamic programming matrix.

Local alignment problem

The principle behind the problem is that sometimes biological sequences have common domains along the sequence. We are interested in the portions that share the most similarity.

Local Alignment Problem:
Find the best local alignment between two strings.

Input: Strings $\mathbf{v}$ and $\mathbf{w}$ and a scoring matrix δ.

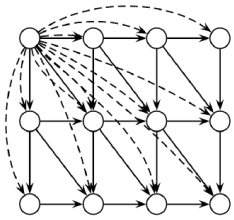
Output: Substrings of $\mathbf{v}$ and $\mathbf{w}$ whose global alignment, as defined by δ, is maximal among all global alignments of all substrings of $\mathbf{v}$ and $\mathbf{w}$.

The straightforward way to solve the local alignment problem is:
For each pair of vertices (i, j) and (i', j') find the longest path connecting them. Then select the path having maximum weight over these longest paths computed.
The issue with this approach is that its complexity is O(n^6): there are O(n^4) pairs of vertices to consider (O(n^2) to try all starting and ending points in a sequence and O(n^2) for the second sequence), and for each pair of vertices it costs O(n^2) to compute the longest path.

In this approach we want to individuate two portions inside the sequences that maximise their global alignment, we want to individuate smaller portions having higher scores. We have to try all possible substrings of the first sequence against all possible substrings of the second one.

Even if we have defined a polynomial algorithm as efficient, a polynomial of 6 is very slow.

This problem can be solved with another approach, we can search for the longest path from the source to every vertex by adding vertices of weight zero that connect the source with every node of the graph. The additional edges make the source vertex a predecessor of every other vertex in the edit graph.
We give the algorithm the possibility to jump the initial portion of the alignment. In order to do this we change the structure of the edit graph, by adding additional edges, the dashed ones.



The scoring function becomes:

$$s_{i,j} = \max \begin{cases} 0 & \text{(in case of direct jump from } (0,0)\text{)} \\ s_{i-1,j} + \delta(v_i, -) & \text{(in case of deletion)} \\ s_{i,j-1} + \delta(-, w_j) & \text{(in case of insertion)} \\ s_{i-1,j-1} + \delta(v_i, w_j), & \text{(in case of substitution or match)} \end{cases}$$

In the traceback phase, we need to start, not from the bottom right cell, but from the cell with the best value.

The complexity is polynomial as the other dynamic programming approaches: filling a matrix like this costs n*m, which is better than n^6

We can reapply all these approaches to situations like alignment with advanced gap penalties and multiple sequence alignment.

Shortest common supersequence problem

We saw the longest common subsequence. Now we have to look for the shortest common supersequence.

Input: strings V, U.
Output: shortest string T, such that V and U are subsequences of T.

Example:
V: TTATG, U: TAGCT
A possible supersequence is TTATGCT. We have that TTATG represents v, and TAGCT represents u. The symbols of a subsequence do not have to be consecutive.

(Would it make sense to look for the longest common superstring? No, because it just consists of an infinite sequence that contains the subsequences.)

We want to design a dynamic programming approach. Focus on the problem of computing the length of the shortest common supersequence.

Imagine we have V and U. We don't want to focus on the details of the sequences, we reason at a high level. We can think of two possible cases:

1. The string V and U end with different symbols.
2. V and U end with the same symbol.

Now we can reason about these two cases:

1. Assume **T** is the supersequence of U and V. We can say something about it: **it will end either with the last symbol of v or u**.
   Within this first case there are two subcases:
   a. The shortest common superstring finishes with the last simbool of V
   b. The shortest common superstring finishes with the last simbool of U

   The $S_{i,j}$, will be the minimum between $S_{i-1,j}$+1 and $S_{i,j-1}$+1 with V[i]!=U[j]. We just take the minimum one.
2. In order to evaluate the length of shortest common superstring, we get the best common superstring for U and V without considering the last symbol and we add 1 (the last symbol that is common to both strings).
   In this case $S_{i,j}$ is equal to: $S_{i-1,j-1}$+1 with V[i]==U[j]

We basically have three predecessors to each superstring we consider. If the last symbols of the strings are equal we consider all three predecessors, in the other case we just consider two of them.

The traceback procedure is the same as in the other dynamic programming approaches. We are just filling a matrix with the values of the shortest common superstrings for all possible

subsequences of V and U, in each cell we consider the subsequences up to the ith and jth character.

We need to initialise the dynamic programming matrix:
What is the length of the shortest common supersequence between a string V of length i and a string U of length 0? i.
What is the length of the shortest common supersequence between a string V of length 0 and a string U of length j? j.
$S_{i,0}$ = i for all i
$S_{0,j}$ = j for all j

We consider all possible lengths of V and U before arriving at the length of the shortest common supersequence between them.

What is the value we need to look at to know what is the length of the shortest common supersequence between the two? If the length of V is n and the length of U is m, the solution to the problem of the shortest common superstring is in the cell $S_{n,m}$ of the dynamic programming matrix.

Although we are doing a lot of work, we are reusing a lot of work. This reuse work is possible only thanks to the structure of the problem, NP hard problems do not allow to reuse work in an efficient way because their structure is very intricate (nevertheless, we can still use dynamic programming approaches on NP hard problems, the thing is that they remain exponential). We cannot build up the solution of the problem on top of sub-istances of the problem.

Satisfiability problem

In the satisfiability problem we have a boolean expression given in input and we want to find a boolean assignment for the boolean variables, such that the formula is true.

Example:
$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_3 \vee x_4 \vee x_5)$
If we assign $x_1 = x_2 = true,\ x_3 = x_4 = x_5 = false$ we get a positive solution, but there is no path or logic that we can follow to find other solutions, we just have to try the possibilities.

The interesting point is that we don't have a way to explore the candidate solution. If we increase the number of variables assigned as true, we are not sure that we will reach a true solution. This problem is NP hard, there is no smart way of solving it. In this case we cannot assemble partial solutions.

0/1 knapsack with integer weights problem

Now we will see an application of dynamic programming to knapsack. It is called 0/1 because either we choose an object or not (in fractional knapsack we can split objects) and it is very important that the weights are integers. Without integer weights we cannot use dynamic programming. (Plus this problem will be useful to explain a concept that is widely used in bioinformatics).

To solve knapsack we will use 2 greedy approaches with unbounded approximation ratio, but if these two approaches are combined, then they have a bounded approximation ratio of 2. Two separate approaches that seem to not work well, together become good.

Input: n objects {1,.., n}; w1, ..., wn integer weights; v1, …, vn values; t threshold
Output: subset s of {1, …, n}, s.t. $sum_{i\ in\ s}$(wi) <= t and $sum_{i\ in\ s}$(vi) is maximum.

In order to solve this problem via dynamic programming, we need to find the usual recursive optimality criteria, so that we can compute the total solution by solving a simple small solution.

(In the shortest common supersequence we have sij, which is a minimisation criteria. We need to define something similar to sij to evaluate the best solution of the knapsack with integer weights.)

We define the quantity: V(w,i). V stands for value, it corresponds to the best value that we can obtain by choosing objects only from 1 to i (sub instance of the problem), while the total weight is exactly w. We will fill the usual matrix with this quantity and then look for the best one among some possibilities. The quantity V(w,i) can be reused in a dynamic programming recurrence, s.t. we can fill a dynamic programming matrix.

The intuition is this: in order to compute the best value only considering objects between 1 and i, for each object j between 1 and i we have to decide whether to take it or not.

If we focus on the last object (i) there are 2 cases:

1. The object i does not contribute to the best value: In this case it means that the value we have equals the value without object i. v(w,i)=V(w,i-1)
2. The object i contributes to the best value: this means that the exact weight of w is w-wi for i-1 objects and the value will be V(w,i)-vi. v(w,i) = v(w-wi, i-1)-vi

Via this we can compute the best combination in order to optimise the global instance that we have in input.

The dynamic programming relation is:

v(w,i)= max{v(w, i-1) v(w-wi, i-1) + vi}
v(w,0) = 0 for all w
v(0,i)= 0 for all i

In order to solve 0/1 knapsack with integer weights with a dynamic programming approach we need to build a matrix. On the rows we have the objects, on the columns we have the numbers from zero to the threshold, in the cells we have the value of the combination, the column of a cell represents the weight of the combination and the row represents the number of object considered so far.

	0	1	2	3	…	Values of weight from 1 to t	t
0	0	0	0	0	…	…	0
1	0	0	4	0			
2	0	2	4	6			
3	0						
…	…						
Labels representing the objects	…						
n	0						

The best solution will be somewhere in the last row, we don't know where, we don't know what the best weight will be. But we have to give ourselves the possibility to choose all of the objects, therefore we will be in the last row.

The complexity is proportional to the time we need to fill the matrix. We have n rows and t columns so the complexity is O(n*t). Apparently we have a polynomial algorithm to solve an NP hard problem!!! How is this possible? Remember that the complexity of an algorithm must be measured with respect to the size of the input.

t is a value of the input, not the size. n is the number of objects, it's the dimension of a list, while t is just a number, it does not represent a quantity of memory associated with the weight. The size to represent t is $log_2 t$.

$|t| = log_2 t$, therefore the value of t is $2^{|t|}$.

In input we have a number represented in binary notation, which means that we represent a number in logarithmic scale. If we want to write the complexity with respect to the actual size we need to substitute t with $2^{|t|}$.

The true complexity is $O(n * 2^{|t|})$, this algorithm is exponential. Often we will find bioinformatics papers with algorithms with complexity like this, they seem polynomial but they have a value in the complexity, they are called pseudo polynomial algorithms.

A pseudo polynomial algorithm is an algorithm whose time complexity is polynomial in the value of an integer that is in the input.

In the lecture of may 3rd the professor did a general recap, look at the recording to revise.

Divide-and-conquer algorithms

Now we will start talking about a new approach to design algorithms: the divide-and-conquer approach. We already saw an algorithm that exploits this idea: merge sort.
Merge sort: divide the array in two halves that have to be sorted, once sorted, they are merged in the original array.
The intuition behind this approach is that instead of tackling the problem of sorting the initial array, we split the array in two and we solve two smaller problems. Then we aggregate the results that we obtained to solve the total problem.

As you may have noticed, this approach is somewhat related to dynamic programming. The similarity between the two approaches is that we deal with sub instances of the problem, starting from these we reach the solution.
The difference is that in dynamic programming we start essentially from the small pieces of the input instance and we build step by step the solution for bigger and bigger instances (we start from the bottom), the successive values are computed from the values that we already have. In divide and conquer we split the instance in two pieces and we solve them independently. We start from the input instance and split it, the two pieces can be in principle solved independently. In dynamic programming each layer depends on the previous layer. In divide and conquer the two parts are independent. Once we have the result for the two halves we recombine them.

Merge sort algorithm:

```
MERGESORT(c)
1  n ← size of c
2  if n = 1
3      return c
4  left ← list of first n/2 elements of c
5  right ← list of last n − n/2 elements of c
6  sortedLeft ← MERGESORT(left)
7  sortedRight ← MERGESORT(right)
8  sortedList ← MERGE(sortedLeft, sortedRight)
9  return sortedList
```

This algorithm is fast because the merging of the two sorted arrays happens in linear time.
Merge algorithm:

```
MERGE(a, b)
 1  n1 ← size of a
 2  n2 ← size of b
 3  a_{n1+1} ← ∞
 4  b_{n2+1} ← ∞
 5  i ← 1
 6  j ← 1
 7  for k ← 1 to n1 + n2
 8      if a_i < b_j
 9          c_k ← a_i
10          i ← i + 1
11      else
12          c_k ← b_j
13          j ← j + 1
14  return c
```

We add +∞ to both arrays, the array that will reach the end will compare +∞ to all the remaining numbers of the other array, in this way all remaining elements will be added to the sorted array.

Time complexity: the time complexity of sorting an array of length n is:

$$T(n) = 2T(\frac{n}{2}) + c * n$$

T(1) = 1

T(2) = 2T(1)+2 = 4

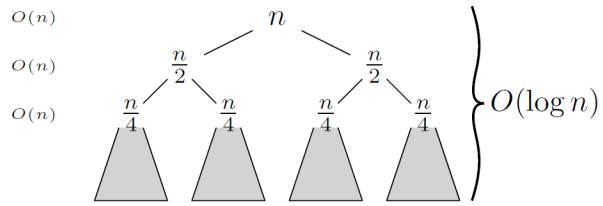
T(4) = 2T(2)+4= 12

To understand the complexity we can look at the execution of the algorithm with in an execution tree, each vertex is the complexity of performing the merge operation of the two halves.

The formula of the complexity can be interpreted like this: we have to consider the complexity of each half, and add the complexity of the merge operation needed to compute the total array. Essentially we are just measuring the merge operations

We can imagine a tree, each level we split each array in its two halves.

$$T(n) = 2T(n/2) + cn$$
$$T(1) = 1$$



At each level we have to merge the two halves of each array in the level. In the case of the second level, we have to spend O(n/2) to merge the two halves that make up the first half of n and then O(n/2) to merge the other two halves of the second half, this means that the complexity is O(n/2+n/2)=O(n).

Each level costs linear time to merge the values. The depth of the tree is log(n). Therefore the overall complexity of merge sort is O(n*log(n)).

DNA alignment problem:

How to use a divide-and-conquer approach in DNA alignment

Recap on DNA alignment: we have two sequences, we put them as the axes of a matrix. The best alignment corresponds to the best path in the matrix. Each cell stores the score of the best path considering the score of the best alignment of the neighbour cells. This algorithm has complexity O(n^2).

Now we will introduce a new concept, the space complexity of an algorithm.

Space complexity of an algorithm

The space complexity is the amount of memory that is used by the algorithm during its execution. The concept is similar to time complexity (instead of the number of steps, we count the memory we need). Space complexity can be expressed in big O notation.

For the global alignment we need to store a matrix of dimension n*m: the space complexity is O(n*m). For backtracking another matrix n*m, the total space complexity is: O(2*n*m). As in time complexity, constants are not important: O(n*m)

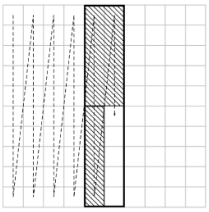
We would like to improve space complexity because n and m could be really big in dna sequences. The matrix can be so big that we cannot store it in memory. When we need to work with a data struct that is bigger than the memory available, we could focus on portions of the matrix, we put a portion of the matrix in memory and we store the rest on the disk, but this is a waste of time. If we can, we don't want to work on data structures bigger than the memory.

We might try to minimise the amount of memory.

When trying to solve the global alignment problem, we have been focusing on two problems: computing the score of the best path, and computing the alignment.

Can we invent a trick such that we can compute the value of the score using less memory? Once we have used a line of the matrix we don’t need it any longer. We might store only the part of the matrix we need.

Each time we are computing the value of a cell, we just need the 3 previous values. All of the rest is useless. When we are computing the scores of a column, the values needed for the computation are just the ones in the preceding column.



Therefore, **in memory we can keep only 2 columns**, the preceding one and the one we are computing. Each time we go to the next column we delete the previous one, in this way we save a lot of space.

This doesn't apply to all dynamic programming problems, we can generalise it to all cases in which we know that the result of a cell is given by a limited number of contiguous cells.

For example, in the Fibonacci problem we can apply this, because to compute the ith fibonacci number we have to sum the i-1 and i-2 Fibonacci numbers. We can solve this problem by keeping in memory all of the array of fibonacci numbers, or we can keep only the last two.

Getting back to the global alignment problem, we demonstrated that we can use this trick for the computation of the score, but what about the computation of the alignment? We want less than quadratic space.

Intuition: we will exploit the linear computation of the score to linearly compute the alignment.

We know that the best alignment is related to the best path in the grid. We don't know the path, we know the starting vertex (0,0) and arrival vertex (n,m).
The first thing we can observe is: since we start from column 0 and we reach column m, the path must pass through the middle column, we will need to traverse the middle column. The tricky point is that we don't know at which row we will traverse this column. Our aim is computing the value of the row in linear space (i).

We can define an i-path, which is the best path from the origin vertex to the destination vertex which is forced to traverse the middle column at row i.

There can be a 1-path, 2-path up to the n-path. Each of them consists in the best global alignment that passes through the coordinate (1,m/2), (2,m/2) and (n,m/2). They will have different scores and one of them is for sure the best global alignment between the two sequences.

Among all of the i-paths, there is one which is the overall best path, the best path from the origin to the finish. We don't know which it is but we know it is there.

We call the overall best path: $\bar{i}$-path, the best path from the beginning to the end node that passes from (i,m/2).

The idea is that we want to compute the value of $\bar{i}$, that is to say the row at which the path passes the middle column. This is the only mechanism by which we will obtain information about the alignment of the two sequences, with this approach we will not do any backtracking, we will just discover a little information about the alignment at a time, in particular the middle cell of the alignment.

If we manage to compute the $\bar{i}$ in linear space, when we have advanced toward the solution, we will know the middle point of our path (this doesn't solve the problem completely but we will know 3 points of the best path). The interesting point is that we can actually compute the value of $\bar{i}$ because it is just the value of the row index of the middle column at which the score of the path is maximum.

$\bar{i} = \ arg\,max_{0 \le i \le n} score(i - path)$

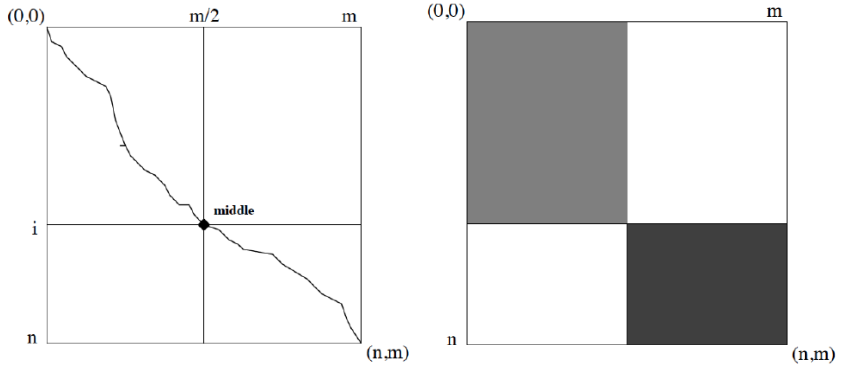
Arg max is the value of the argument at which the function reaches the maximum.

If somebody gives us the score of all paths we can find the i path by looking at the maximum. The interesting thing is that the score of the i-paths can be computed by using the linear space score matrix.

Let's focus on a generic i-path:
We can divide the matrix in two: the first part of the matrix contains the path from (0,0) to (i,m/2) and the second part contains the path from (i,m/2) to the end. The global score of the alignment is the sum of the score of the best score of the two halves.
We have to maximise the global score of the path, i is the row traversed by the best overall path between the two sequences so we have to keep into account the first half and the second one. We cannot just look for the first half of the path and select the best one. To keep into account the overall alignment we sum the scores of the first and second half both in the central column.

We call each half of the matrix respectively suffix and prefix of the i-path (with length we mean score). Therefore we can say that: length(i) = prefix(i)+suffix(i).

This means that we will have to start from (0,0) and compute the score of the best alignment to (i,m/2) in linear space, and then sum this value with an alignment computed from (n,m) to (i,m/2), we have to go basically reverse. If we do this for each i, the maximum value in the column m/2 is $\bar{i}$.



We can essentially divide each i-path in two pieces, compute each half in linear space and obtain the value score of the i-path considered, we will store the score of the i-path in the middle column together with all the other scores of all the other i-paths

If we take the index of the middle column with the maximum score we know which is the $\bar{i}$-path and most importantly we know what is $\bar{i}$, therefore we already have some little information about

the global alignment (which btw is the $\bar{i}$-path but we cannot simply retrieve it because we want to save space and we cannot use the traceback).

We can compute the score of the 1,2,3,4 path, once we have this list we choose the higher one, this is a lot of work, just to find out $\bar{i}$, but the interesting part is that we managed to do this, just in linear space. The amount of memory we are using is always the same because we are reusing the memory: each time we compute a path, we use only two columns, and the result of the algorithm is just an array of length n (the column m/2).

We are still not at the solution but we can divide each half in half, then consider all of the i-paths that bring to the first half, and select the best one. We essentially are computing the row coordinate of the half of the path. And this happens in linear space.

The problem is that we are redoing a lot of work. We need to estimate how much work we are redoing, if the time complexity is much higher maybe it is not convenient to save time

```
PATH(source, sink)
1  if source and sink are in consecutive columns
2      output longest path from source to sink
3  else
4      mid ← middle vertex (i, m/2) with largest score length(i)
5      PATH(source, mid)
6      PATH(mid, sink)
```

When the source and sink vertex are in consecutive columns we need to return the connection between the two cells in the alignment. If they are far apart, we can compute their middle point with the trick we explained above, then we call recursively the procedure on the two portions that we have obtained. We need to know the middle point to be able to do the recursive call.

Time complexity: (we want to be sure that we are not wasting too much time, the complexity of original algorithm is O(n*m)) to estimate time needed by this algorithm assume we want to compute a generic $\bar{i}$. In order to compute it we need the score of all the i paths (each i-path is divided in two, the second half is computed from the end to the centre and sum of the score of each half is stored in an array, but still, the time required to find the alignment between two sequences is the same that we described in the dynamic programming approach), then we have to pick the biggest in the list. The computation of an i-path requires O(n^2), so we need to compute the n i-paths, and then all the i-paths of the recursive calls (each level of recursion, the number of paths is always the same but the area of the dynamic programming matrix is halved).

Once we found the point in the middle we completely forget all the other scores, we take the middle point and we give it in input to a recursive call to find the middle point of the first half (as sink) and to a recursive call to find the middle point of the second half (as starting point).

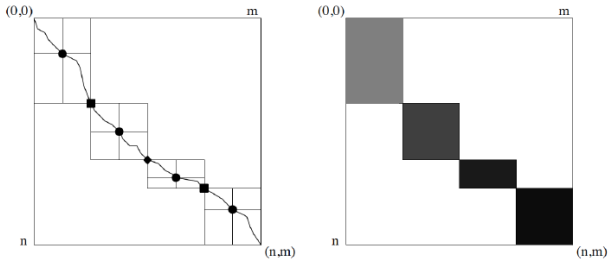
The time complexity is:

$$a + \frac{a}{2} + \frac{a}{4} + \cdots + \frac{a}{2^k} = a(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^k})$$
$$\leq a \underbrace{\left(\sum_{i=0}^{\infty} \frac{1}{2^i}\right)}_{\text{geometric series}} = a\frac{1}{1-\frac{1}{2}} = 2a$$

The total summation results in just twice the area, we are just taking double the time. Not so much more. The computation time is still O(n*m) (it's just 2*n*m so it's neglectable). We are super happy, we save space and the complexity is basically the same, we are just taking double the time. (Just remember that there are no free meals, if we want to save space we need time, if we want to improve time we will waste space. It depends on our objective. Just if an algorithm is badly designed we can improve it by getting only advantages.)

Example:

In the second recursion we have something like this:



The number of i-paths we are considering in the entire step is the same, but they are much shorter and 4 of them will give us a middle point (as opposed to the first step in which we discover only a middle point and we compute the whole dynamic programming matrix), in this case we are computing only a fourth of the dynamic programming matrix

Can we improve the computation time?

Block alignment problem

Nobody knows whether this is possible to go under quadratic time in the general context of global alignment. We will look now at a specific setting in which we will improve time. This is not general, it is just a specific setting, it is a tricky way to go below quadratic which requires some assumptions on the problem.

We will focus on the problem of block alignment: we have two strings U and W. We do not want to align the string symbol by symbol, we want to align them block by block. We refer to a portion of symbols as a t-block. We have that the strings are divided into blocks of length t. If the length of U and W is n, we assume that n is a multiple of t. We do not have shorter blocks. All blocks have the same length. The number of blocks is n/t.

Block Alignment Problem:
Find the longest block path through an edit graph.

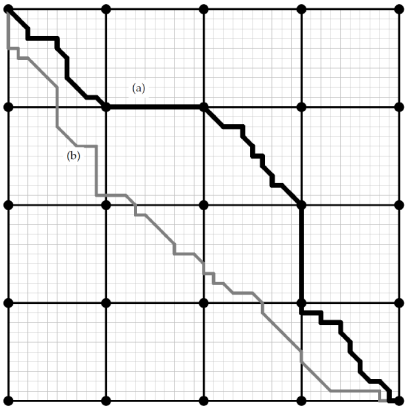
Input: Two sequences, **u** and **v** partitioned into blocks of size *t*.

Output: The block alignment of **u** and **v** with the maximum score (i.e., the longest block path through the edit graph).

What we are going to do is align two sequences U and W with a different principle.
The difference is the following: when we align sequences block by block, for each block of the sequence we decide if we delete the entire block, if we add the entire block or if we align the blocks of the two sequences.

In the previous approach, we were deciding if deleting or adding one by one. Here we consider portions of the sequences. For each pair of portions we decide what to do, we consider them as single characters.

We want to compute the best block alignment between two sequences.



The light grey is the best path. When we want to compute the block alignment of two sequences we are forced to work with blocks. For each block, we perform the alignment, with the starting and finishing vertices fixed by the boundaries of the block, then we can only decide whether to use the block in the global alignment or to skip it (indel).

In this context we can save time, we can go below quadratic complexity. This problem can be solved by dynamic programming, in a similar way to global alignment. In that case the dynamic programming matrix had a specific meaning, Sij, was the score of best alignment between the i prefix of the first sequence and the j prefix of the second sequence (score of the partial alignment). Now we have to update the meaning of this quantity.

We build a different dynamic programming matrix, the meaning of the cells in the matrix is different. It is focused on blocks rather than on symbols. Sij is the score of the best block alignment, between the first i blocks of the first sequence and the first j blocks of the second sequence. The meaning is similar, it is like zooming out, here the units we consider are the blocks.

We need to design a dynamic programming recursion in order to fill the dynamic programming matrix.

If we want to compute Sij, we have to evaluate: Si,j-1, Si-1,j, Si-1,j-1 and choose the best movement from one of the predecessors to Sij. We call βi,j the score of the best alignment (standard alignment) between the ith block of the first sequence and the jth block of the second sequence.

Therefore the maximisation problem is solved for this scoring function:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma_{block} & \text{(in case of whole block deletion)} \\ s_{i,j-1} - \sigma_{block} & \text{(in case of whole block insertion)} \\ s_{i-1,j-1} + \beta_{i,j} & \text{(in case of block alignment)} \end{cases}$$

In order to continue we need to know what is the value of βi,j. There 2 approaches that we can use:

1. **Evaluate all the betas of the block pairs**: we focus on the two input sequences, we compute the score of best alignment of each block, we call them β, for all pairs of blocks. How much does it cost? We need to carry out a standard global alignment between each block. We have (n/t)^2 pairs of blocks, and each alignment (t characters) costs t^2, so the overall cost is (n/t)^2 * t^2, which corresponds to O(n^2) ($O(\frac{n^2*t^2}{t^2})$).

   To compute the dynamic programming matrix of the block alignment and give the solution we have complexity O(n/t^2). (we have to compute the dynamic programming matrix of the blocks, we already have the betas so it is just like a normal global alignment for the number of blocks (n/t)).
   The overall complexity of this approach Is O(n^2)+O(n^2), we have done all of this work to still obtain a quadratic algorithm.
2. **Compute the betas for all possible strings of length t**: we will compute the score of the best alignment between the pairs of all possible sequences of length t. Before we were looking at the blocks in the input and computing the score of those. Now the procedure is more time demanding, we don't look at the input, we compute the score of the best alignment between all possible sequences of length t. We will align potential blocks that are not in the input. This seems strange but it will help us.
   Since we are dealing with biological data, we will consider blocks made of nucleotides.
   The number of all possible sequences of length t nucleotides is 4^t.
   We will have to evaluate the pairs of these sequences, which means 4^t * 4^t.
   Then we will need to align them, this costs O(t^2).
   We have to compute all of this stuff, and save it in memory. We want to save the data in a "dictionary", where the key is the pair of strings and the value is the value of the alignment between the two strings. If we consider blocks of length 5, we store all of the alignment scores between the pairs of all possible strings of length 5. The very important thing to do is that they need to be sorted.

AAAAA, AAAAA	$value_1$
AAAAA, AAAAT	$value_2$
AAAAA, AAAAC	$value_3$
⋮	⋮
GGGGG, GGGGC	$value_{r-1}$
GGGGG, GGGGG	$value_r$

   We store them in memory in a sorted fashion. The data structure we obtain is called a lookup table.
   At this point we can compute Sij, when we need the value of a βij for a pair of blocks that we have in our instance of the problem, we will look at the lookup table, if the blocks we are using have the same length of the strings we used to build the lookup table, we will for sure find our pair in the table.
   The score function for the dynamic programming recurrence will be the following:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma_{block} \\ s_{i,j-1} - \sigma_{block} \\ s_{i-1,j-1} + LT(u[i^{\text{th}} \text{ block}], v[j^{\text{th}} \text{ block}]) \end{cases}$$

This doesn't make sense apparently, but for particular values of t, this can make a lot of sense.

Suppose $t = \frac{log_2 n}{4}$, then:

- The *first step* of the algorithm would take time

$$\begin{aligned} 4^t \cdot 4^t \cdot O(t^2) &= 4^{\frac{\log_2 n}{4}} \cdot 4^{\frac{\log_2 n}{4}} \cdot O(\log^2 n) = (2^2)^{\frac{\log_2 n}{4}} \cdot (2^2)^{\frac{\log_2 n}{4}} \cdot O(\log^2 n) \\ &= 2^{2 \cdot \frac{\log_2 n}{4}} \cdot 2^{2 \cdot \frac{\log_2 n}{4}} \cdot O(\log^2 n) = 2^{\frac{\log_2 n}{2}} \cdot 2^{\frac{\log_2 n}{2}} \cdot O(\log^2 n) \\ &= (2^{\log_2 n})^{\frac{1}{2}} \cdot (2^{\log_2 n})^{\frac{1}{2}} \cdot O(\log^2 n) = n^{\frac{1}{2}} \cdot n^{\frac{1}{2}} \cdot O(\log^2 n) \\ &= O(n \log^2 n) \end{aligned}$$

- The *second step* of the algorithm would instead take time

$$O\left(\frac{n^2}{t^2}\right) \cdot \underbrace{O(\log n)}_{\substack{\text{time needed} \\ \text{to access the} \\ \text{lookup table} \\ \text{(binary search)}}} = O\left(\frac{n^2}{\log n}\right)$$

- The complexity is dominated by the second step, which is $O\left(\frac{n^2}{\log n}\right)$

To compute the complexity of filling the lookup table we need to substitute the value of $t = \frac{log_2 n}{4}$ into the formula of the complexity we discovered before ($4^t * 4^t * O(t^2)$), in this way we obtain: $n(log\ n)^2$

In the second step we need to compute the global alignment of the blocks, we will fill the dynamic programming matrix for each Si,j. We have $(\frac{n}{t})^2$ pairs of blocks. For each cell we need to fetch the βij in the lookup table, the cost of this operation depends on how much it takes to retrieve the value from the lookup table.

If we take advantage of the fact that the array is sorted, we can perform binary search. We proceed by bisection: we compare the key (pair of sequences of the blocks) with the middle key of the lookup table, in this way we are able to exclude half of the positions of the table as possible solutions. We can find our element with a logarithmic complexity.

So we have: O(n^2/t^2)*O(logn)

By substituting t^2 with the value we are considering as an example we get $O(\frac{n^2}{log\ n})$.

Therefore the second step dominates the complexity of the algorithm.

(We could do this with a hash table, we would access it in constant time. The problem with hash tables is that if the hash function is not well designed, collisions occur (two different keys return the same place in memory). If the function is shit we might end up with all values in a single place in memory. For this reason in the worst case scenario we would have linear complexity (O(n)) which is worse than logarithmic (O(logn)).

Recursion master theorem

We will present a theorem that is useful to get information about the complexity of recursive procedures:

$T(n) = a * T(\frac{n}{b}) + f(n)$

This is a general case of the time complexity of a recursive procedure (actually this is a simplified version, it is not universally valid) in which, in order to solve the instance we are given, we split the instance in b pieces (in merge sort b is 2), and a is the number of times that we need to recursively solve the b subpieces (it can be the case that we split an array in two and we have 3 recursive calls to manipulate the two halves, a=2, b=2). f(n) is the time complexity of merging the results.

The first part of the complexity (a*t(n/b)) contributes to the complexity of the algorithm for a factor of $O(n^{log_b a})$. We are not going to see how, but the big o notation of the first part is that one. The second part of the complexity contributes to the big o notation by a factor of $O(n^c)$.

The intuition is that the complexity of a divide and conquer procedure is built by 2 factors, the first comes from the recursion, the second comes from the combinations of two results produced by the recursion. If the first part is bigger than the second one, then the big o complexity is dominated by the first part, if the second part is bigger, the complexity is given by the second part. If the two parts are comparable the complexity is a combination.

If we call $log_b a = d$, we have: $T(n) = O(n^d) + O(n^c)$

If f(n) is $O(n^c)$ and d>c, then the complexity of the algorithm is $O(n^d)$

If f(n) is $\Omega(n^c)$, d<c $\rightarrow O(n^c)$

If f(n) is $\Theta(n^c)$(n^c), d==c $\rightarrow O(n^d * log\ n)$

This is what we will use to estimate the complexity of recursive algorithms, there are proofs that prove all of these results but we will not see them. We will use this result to analyse the last problem that we will encounter in this course.

Multiplication problem

We have to multiply two numbers of unlimited precision. The registers in a computer have limited size: 32 or 64 bits, if we want to multiply two numbers with bigger representation we need to invent something.

Definition of the problem:
Input: two numbers a and b of length n.
Output: a*b

We can implement the pen and paper algorithm: take the first element and multiply for all the others. Then shift and multiply the second for all the others, we do this for each number and we sum up the results. For each digit the cost is linear, we have n digits, so n*n operations. The complexity is O(n^2).

Is it possible to do better?

We can divide a and b in the upper and lower part. Example: if a is 4757, we can divide this number into two halves: 47 and 57. The total number can be retrieved by 47*100 + 57.

We can obtain the multiplication of a and b by multiplying:

$a = a1 * 10^{\frac{n}{2}} + a2, b = b1 * 10^{\frac{n}{2}} + b2$

The multiplication can be obtained by multiplying these two polynomials. Resulting in:

$a * b = a1 * b1 * 10^{n} + a1 * b2 * 10^{\frac{n}{2}} + a2 * b1 * 10^{\frac{n}{2}} + a2 * b2$

What is the complexity of proceeding in this way? We are in front of a recursive procedure: to compute the value of a*b we have to compute the value of other 4 smaller multiplications.

If we consider the formula of the recursion master theorem we have that a=4, because to multiply a and b we are carrying out 4 recursive calls, b=2 because we are splitting the instance into two parts, and f(n), the cost of recombining things, is linear: O(n).

Now we can compute the complexity with the bigO notation:

$T(n) = O(n^{log_b a}) + O(n^c) = O(n^{log_2 4}) + O(n^1) = O(n^2) + O(n)$

This is shit again. We need to improve. We will focus on the middle portion of the recursive calls: $a1 * b2 * 10^{\frac{n}{2}} + a2 * b1 * 10^{\frac{n}{2}}$. We will rewrite it to perform less multiplications.

Consider $(a1 + a2) * (b1 + b2) = a1 * b1 + a1 * b2 + a2 * b1 + a2 * b2$, in this formula we have the two that we have to compute, the two at the centre, while the first and the fourth are extra.

$a1 * b2 * 10^{\frac{n}{2}} + a2 * b1 * 10^{\frac{n}{2}} = [(a1 + a2) * (b1 + b2) - a1 * b1 - a2 * b2] * 10^{\frac{n}{2}}$

The quantity (a1*b2+a2*b1) can be obtained by multiplying (a1+a2)*(b1+b2) and subtracting (a1*b1+a2*b2).

Let's substitute:

$a * b = a1 * b1 * 10^{n} + a1 * b2 * 10^{\frac{n}{2}} + a2 * b1 * 10^{\frac{n}{2}} + a2 * b2$

$a * b = a1 * b1 * 10^{n} + [(a1 + a2) * (b1 + b2) - a1 * b1 - a2 * b2] * 10^{\frac{n}{2}} + a2 * b2$

We will use 3 multiplications instead of 4. We keep in memories the result of the multiplications

$a = 3, b = 2, f(n) = O(n), d = log_2 3$

$T(n) = O(n^{log_2 3}) + O(n)$

$1 < log_2 3 < 2$

This complexity dominates over the linear but it is smaller than two so the algorithm doesn't have quadratic complexity.