Informatics

Computational complexity

Claudio Sartori
Department of Computer Science and Engineering
claudio.sartori@unibo.it
https://www.unibo.it/sitoweb/claudio.sartori/

Learning Objectives

- Why should we study computational complexity?
- The principles
- Some general problems
- Comparison between algorithms
- Examples

Searching in a dictionary

- How much time does it take to search a word in a dictionary?
- How would you search if you wouldn't know that the words are organized in alphabetic order?
  - how much time would it take?

Searching in a dictionary

- The task can be done using two different algorithms
  - one based relying on alphabetic order and one *sequential*
- The two algorithms can be compared based on the *effort* they require

Effort?

- Amount of *computing resources* used to complete a given task
- Computing resources
  - time
    - time necessary for a single operation * number of operations
  - memory
    - amount of RAM
    - amount of memory on disk
  - input/output
    - reading from hard disk into memory
    - writing from memory to hard disk
  - transmission
    - number of bytes to transmit over a network
      - significant for problems involving data distributed over a network

Time complexity

- The shorter, the better
  - provided that the result is the same
- Number of operations – it depends on
  - amount of data
    - we will always refer to it with the symbol *n*
  - characteristics of data
    - sorting a vector where only two elements are in the wrong order (say, swapped) with bubble sort has a cost very low, if all the elements are in the wrong order the costs is much higher
  - algorithm
- Time for each operation
  - it depends on the hardware/software environment of execution

Time complexity (ii)

- Worst case
  - when the characteristics of data imply the maximum effort
- Best case
  - when the characteristics of data imply the minimum effort
- Average case
  - in principle should consider the effort for every possible case and weight it with the probability of the case
- The number of operation can be evaluated by considering the algorithm

Time complexity
Empirical approach

- system.time function

```
> system.time(Sys.sleep(10))
   user  system elapsed
  0.035   0.036  10.000
```

The result is a vector with three components

- time spent in user mode and system mode
- total elapsed time

Empirical measure of Time Complexity (TC)

- decide the range of sizes you want to test and the step of sizes
- prepare a data structure to collect the results
- for each size of interest n
  - prepare a random, artificial, set of data of size n
    - as an alternative, you can use real data
  - execute the program inside system.time and collect the resulting elapsed time
- show the results
- see program **`timing_execution.R`**

Time complexity - theory

- count the statements in a sequence
- a two way test should be evaluated considering the length of each branch and the branch chosen in the best or worst case, or the respective lenghts weighted with the probability of branches for the average case
- repetition
  - if the number of repetitions is proportional to *n*, the number of operations in the block must be multiplied by n (and by the proportionality coefficient)
- nested repetition
  - multiply the contribution of each repetition
- function call
  - consider the number of operations of the entire function

d x d Matrix multiplication

```
for (i in 1:d){
  for (j in 1:d){
     c(i,j) <- 0
     for (k in 1:d){
         c(i,j) <- c(i,j) + a(i,k) * b(k,j)
     }
  }
}
```

complexity (approximated): $d^3 + d^2 + d^3 + d^2 + d$

Computational complexity examples

Consider the examples shown in class

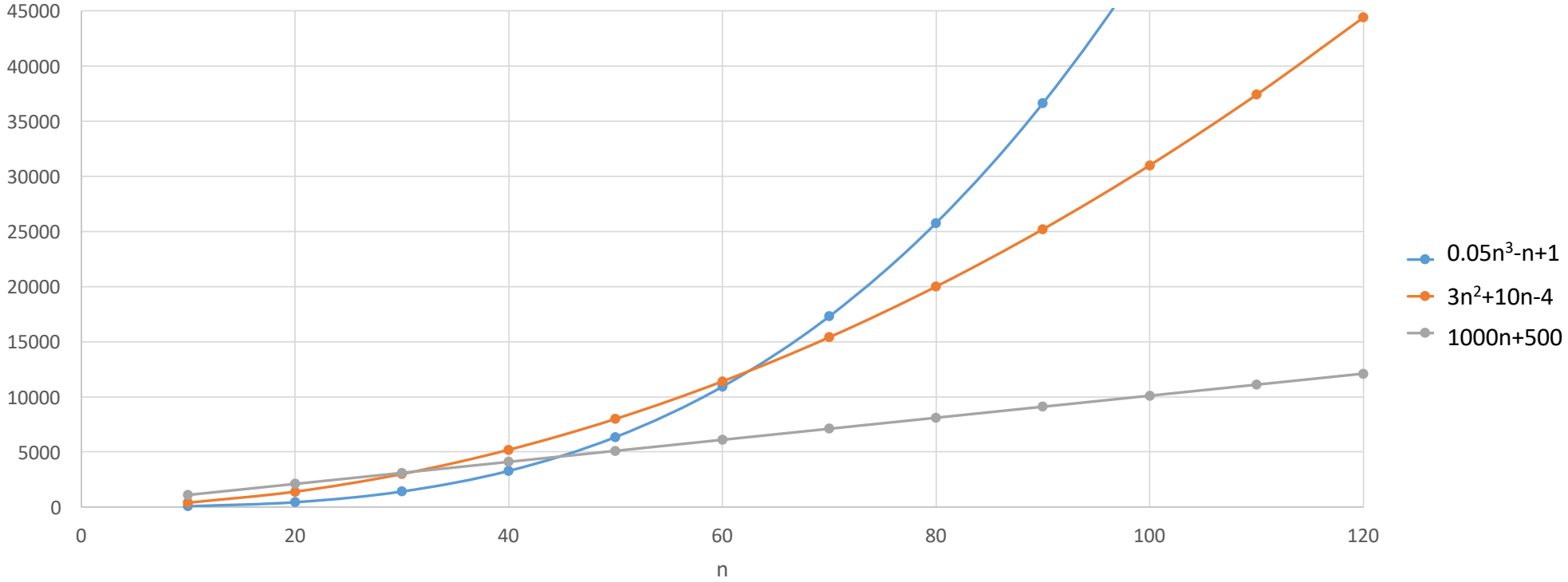
- looking_in_vector_fun
- selectionSort_fun
- determinant_laplace
- echelon reduction

- can you guess the complexity?

Evaluate time complexity

- Suppose that for the solution of a given problem we found three different algorithms, with the following numbers of operations
  - $f1(n)=0.05n^3 + 2n -1$
  - $f2(n)=3n^2+10n-4$
  - $f3(n)=1000n+500$
- which is the best?

Evaluate time complexity



Asymptotic time complexity

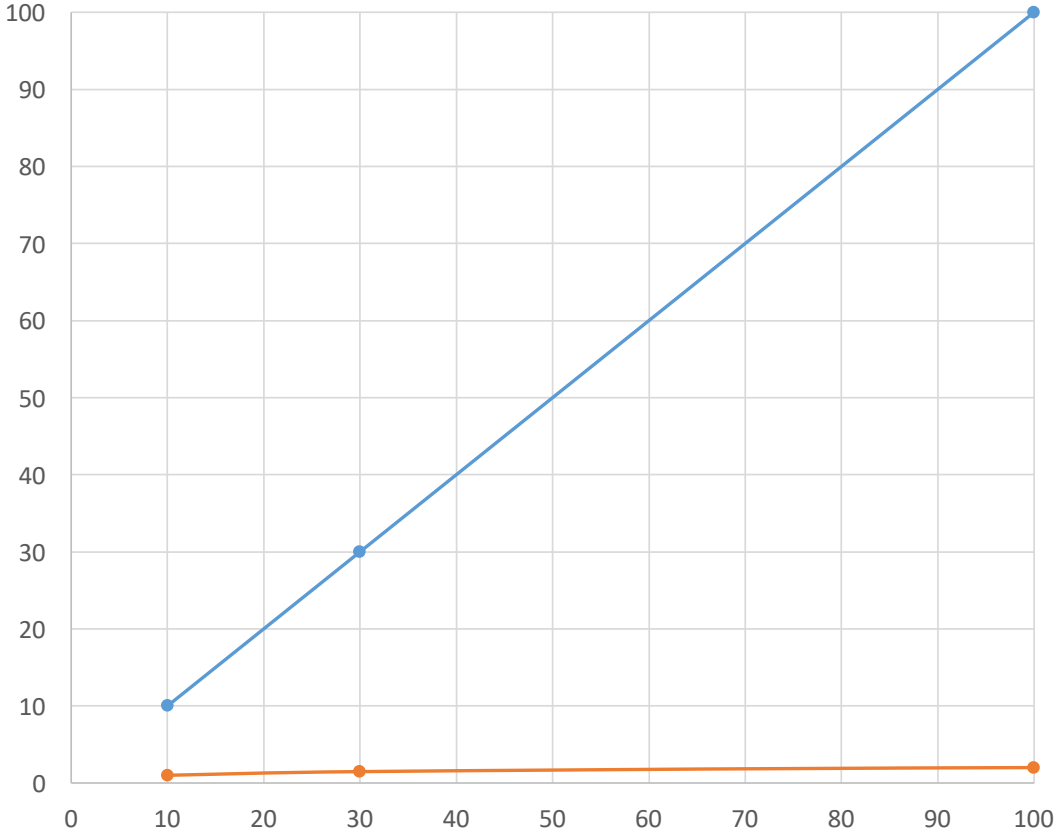
- For a first comparison we can disregard the details
  - additive and multiplicative constants
  - the terms with smaller increase rate
- Big O notation
  - $f1(n)=0.05n^3 + 2n - 1$ $O(n^3)$
  - $f2(n)=3n^2+10n-4$ $O(n^2)$
  - $f3(n)=1000n+500$ $O(n)$

Example: binary search

- It is a divide-and-conquer algorithm
- The scope of the search is halved at each iteration
- In the worst case, the iteration stops when the scope of the search has size 1
- how many times can I halve *n* to obtain 1?
  - $\log_2 n$

Search algorithms comparison

- problem:
  looking for an element in an ordered list
- solution: sequential search
  - O(n)
- solution: binary search
  - O(log n)



Example: selection sort

- The external loop is repeated n-1 times
- The number of repetitions of the internal loop is decreasing, from n-2 to 1
- We need a summation

(n-2) + (n-3) + (n-4) + … + 1

- Combining the external and internal loop, the number of operations is

$$O(\frac{(n-2)(n-1)}{2}) \quad \text{which is the same as} \quad O(n^2)$$

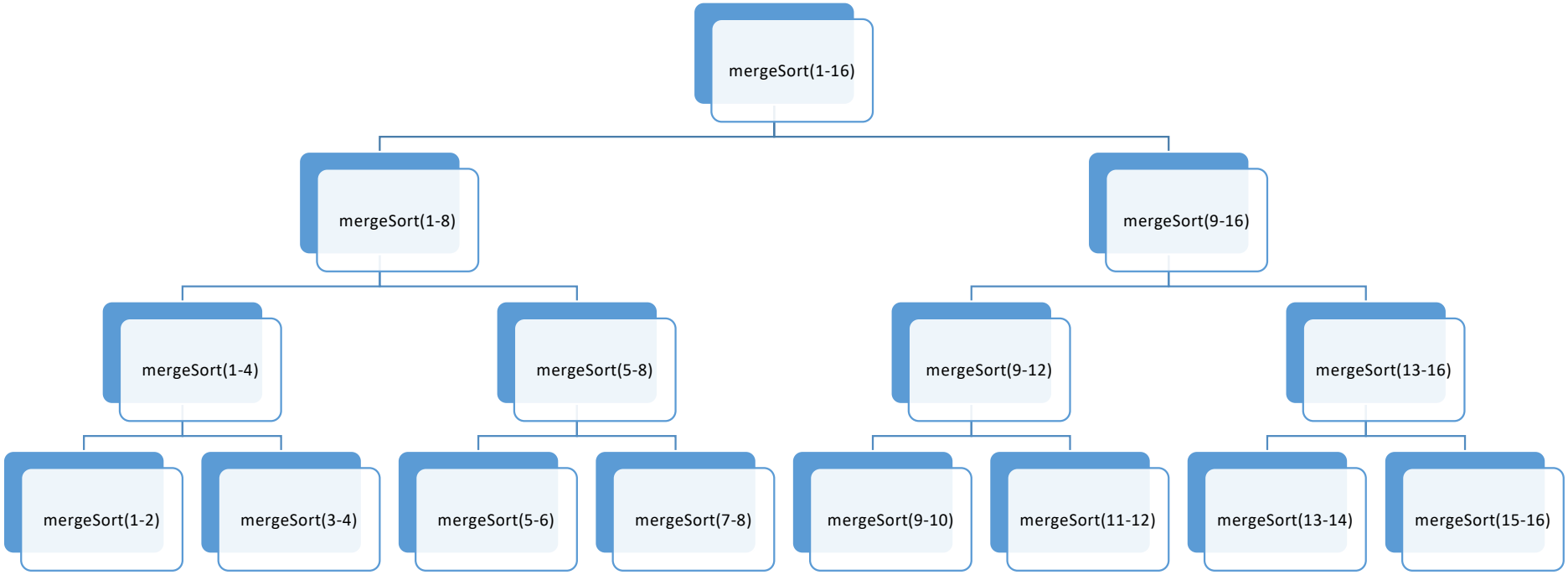
Merge Sort

- Another sorting algorithm
- Worst case O(n * log n)
- Another algorithm with *divide and conquer* strategy
  - split the data in smaller parts
  - process the parts
  - combine the partial results to obtain the general one

Merge sort – algorithm
recursive definition

- If n=size of the vector is 1 return the vector
- set center to size of the vector / 2
- Sort the portion of the vector from 1 to center
- Sort the portion of the vector from center + 1 to end
- Merge the two portions

Merge sort -
How many recursive calls? (e.g. 16 elements)



Merge sort – number of operations

- recursive calls: ceiling($\log_2(n)$)
- for each level of the tree of recursive calls
  - merge of the pairs of portions
  - merging n elements in total costs n/2
- the number of operations is, approximately

`n * ceiling(log2(n))`

Determinant Laplace: complexity

- for n=2 the complexity is constant
- for n=3 it calls 3 times a 2-sized determinant
- for n=4 it calls 4 times a 3-sized determinant
- ….
- the complexity is O(n!)
- since the amount of data in the matrix is m = n* n, the complexity in matrix size is, approximately, $\mathcal{O}(\sqrt{m}!)$ worse than any polynomial

Determinant via Row Reduced Echelon Form (rref)

- rref(A) is obtained with
  - row switches, *rsn = row swithces number*
  - row scalar multiplications, *smf = scalar multiplication factor*
  - linear combination of rows
- *scale* of rref(A) $(-1)^{rsn} * \prod 1/smf$
- From algebra we know that
  determinant of A =
  determinant of rref(A) * scale of rref(A)

Complexity of rref computation

- main loop for row scanning: n
  - internal loop for pivot search: n - current row
  - internal loop for linear combination of rows: n
    - each linear combination: n
- in summary: n^3
- or, in the size of the matrix, wich is m = n*n

$$\mathcal{O}(m^{\frac{3}{2}})$$

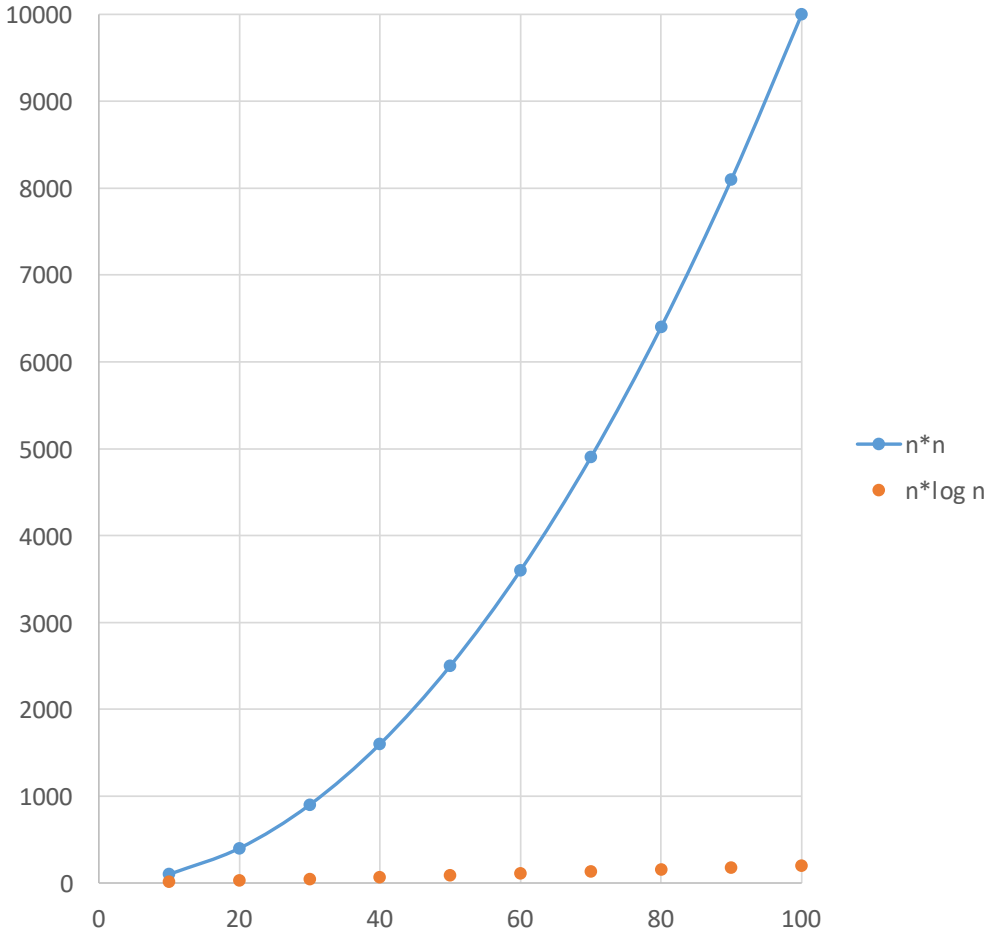
- since the scale multiplication has constant complexity, this is also the complexity of the determinant computation via rref

Approximation errors

- In principle, the determinant does not involve divisions, therefore, if the matrix elements are integers, the determinant is an integer as well
- The rref implies divisions, this *changes many things*
  - the approximation errors can lead to *wrong results,* in particular, but not only, when looking for exact comparisons
    - equality test between non-integer numbers
- It is a good idea to consider rounding when exact comparisons are critical for the decisions
- Look at the myRref_ws procedure, where is the critical point?

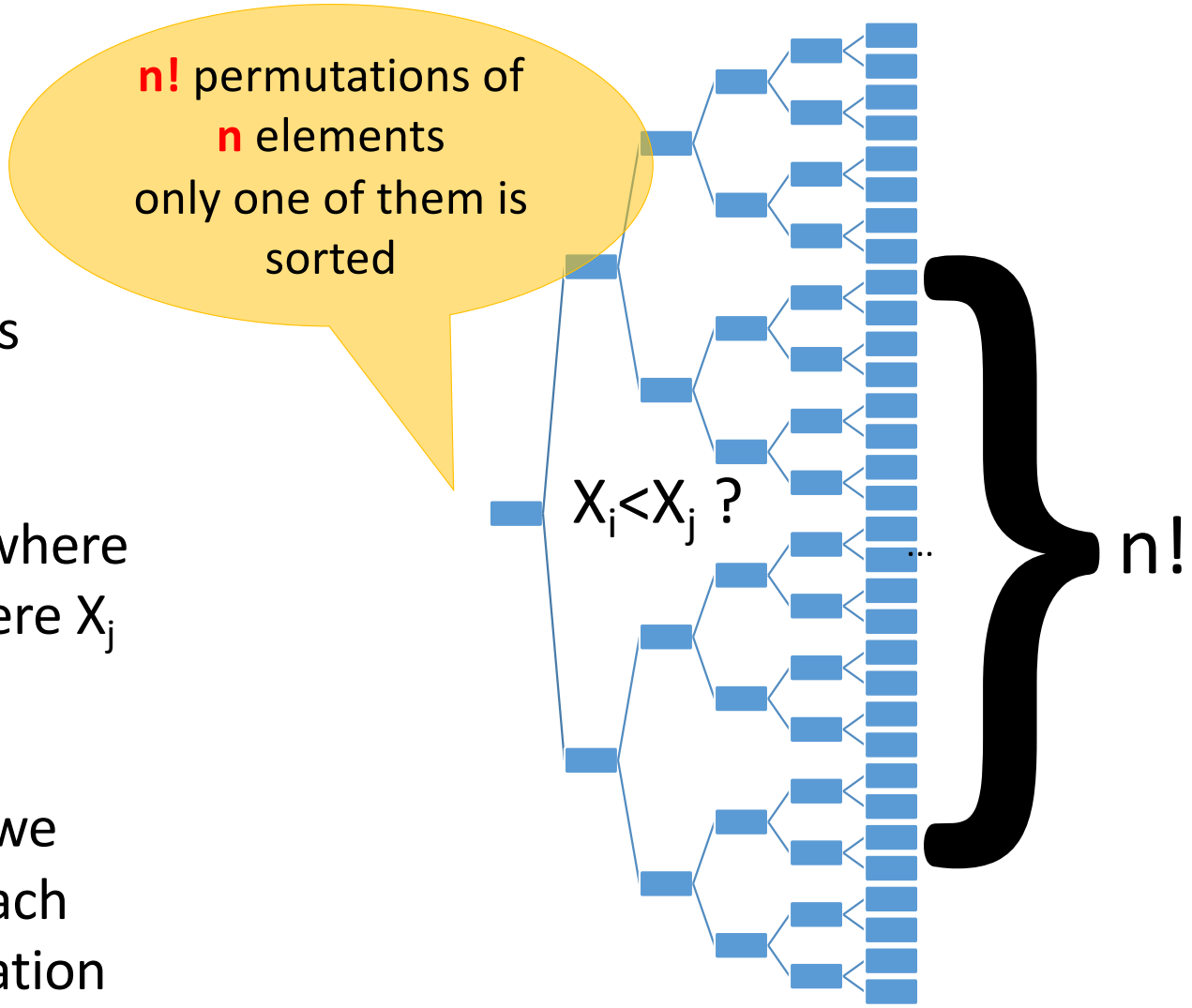
Example: sorting

- Bubble sort: $O(n^2)$
  - average case
- Merge sort: O(n*log n)
  - average case



What is the "natural" complexity of sorting?

- Unsorted vector $[X_1, X_2, \ldots, X_n]$
- A permutation of the indexes gives the sorted vector
- A test $X_i<X_j$ allows to choose between the permutations where X_i precedes X_j and those where X_j precedes X_i
- After a series of tests, and consequent binary choices, we reach one of the **n!** leaves each containing only one permutation



Which is the *natural complexity* of sorting?

- *n* distinct elements can be arranged in *n!* ways
- the best strategy should be able to halve the number of possible arrangement at each step
- eventually the solution will be reached
- the *natural complexity* of a sorting algorithm based on the comparison of two elements is the height of the tree in the previous page

$$O(log(n!)) \Leftrightarrow O(n\ log(n))$$

the base of the logarithm does not change the bigO class, since it implies only a multiplication factor

Stirling approximantion of n!

Stirling approximation of n!

$$\lim_{n \to +\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{n!} = 1$$

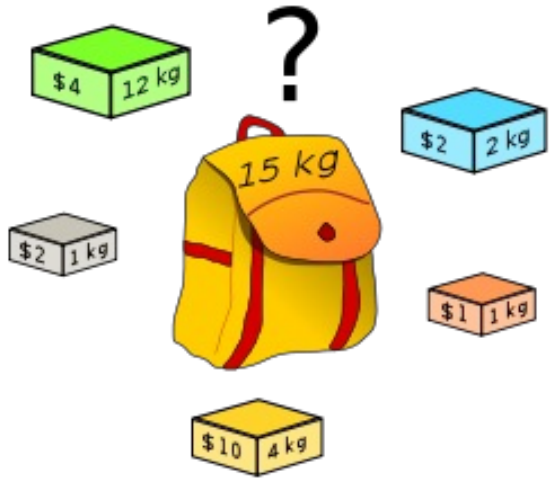
$$\lim_{n \to +\infty} \frac{n \log n}{\log n!} = 1$$

More on sorting algorithms complexity

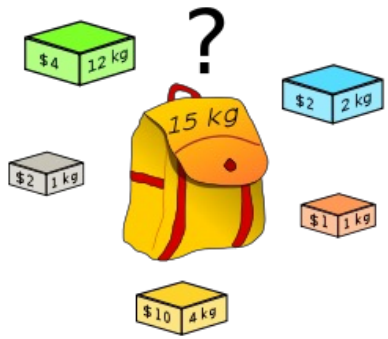
- merge sort
  - average and worst case O(n*log n)
- tree sort
  - average and worst case O(n*log n)
- quick sort
  - average case O(n*log n), worst case $O(n^2)$
  - an additional analysis shows that in the average case quick sort performs better than tree sort
- bubble sort
  - average case $O(n^2)$, best case O(n)

A tough one: the knapsack problem

- It belongs to the class of *optimisation problems*
  - among a set of options find the best one
- n objects, each one has *weight* and *value*
- the knapsack has a *capacity*
  - either in weight or volume
- choose m (< n) objects such that
  - they do not exceed the capacity
  - the total value is *maximum*



The knapsack problem

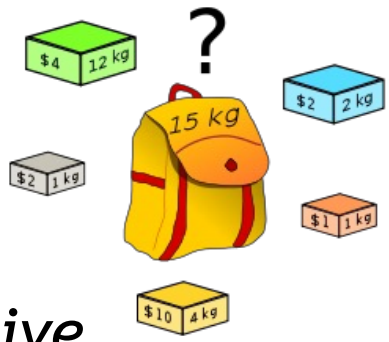


Exhaustive algorithm

- Consider all the possible configurations of objects
  - i.e. the *powerset* of the set of all the objects
- set the temporary best total value to 0
- repeat for each element s of the powerset
  - compute the value of s
  - compute the weight of s
  - if the total weight of s does not exceed the capacity
    - if the total value of s exceeds the temporary best total value
      - set the temporary best to s
      - set the temporary best total value to total value of s
- return temporary best, its value and its weight

see the R implementation

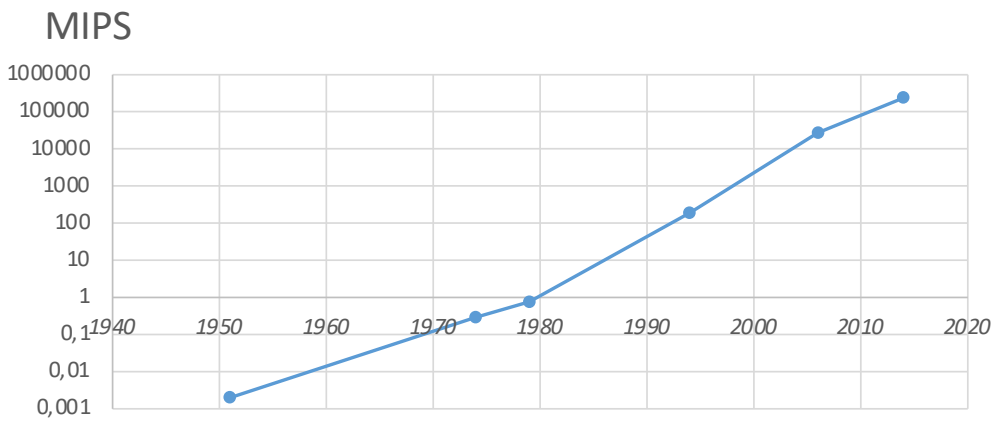
The knapsack problem



- Which is the computational complexity of the *exhaustive algorithm*? (it is also called *brute force algorithm*)
- i.e. how many times is the loop executed?
- it is the *size of the powerset*
- n objects → 2^n elements of the powerset
- $O(2^n)$
- it belongs to the category of *np-hard* problems
  - there are very powerful theories which study this kind of problems and seek for approximate, faster solutions
- See the *knapsack* example

What's the influence of computer speed?

MIPS =
millions of instructions per second



Processor	Year	MIPS
Univac I	1951	0.002
Intel 8080	1974	0.290
Intel 8088	1979	0.750
Intel Pentium	1994	188
Intel Core 2 …	2006	27K
Intel Core i7 …	2014	238K

Combining number of operations with computer speed - today

Complexity n	10	10 000	1 000 000
$O(n^2)$	0.42 msec	7 min	4863 days
O(n * log n)	0.042 msec	0.16 sec	< 5 min

The best sorting algorithm, with respect to a bad one, for large amounts of data transforms an *infeasible* problem into a *feasible* one

Reverse reasoning

- Suppose we can wait for a solution only a limited time
- We can compute the maximum amount of data that can be processed in that time
- For most of the problem we cannot expect big advances in algorithm performance
- We can expect advances in processing power
- We can expect increases in the amount of data that can be processed within our *time limit*

Maximum data size vs speed

- For the most difficult problems, also called *np-hard* problems, the increase of speed is not a big help
- For such problems we prefer *approximate solutions*
- Operation research:
  - find the best tradeoff between computational complexity and approximation

Computer Speed / Complexity	Present technology	100 times faster	1000 times faster
O(n)	N1	N1 * 100	N1 * 1000
$O(n^2)$	N2	N2 * 10	N2 * 31.6
$O(2^n)$	N3	N3 + 6.65	N3 + 9.9

Profiling computer programs

- Measure the computational resources required to execute
  - a program
  - a function
  - a single instruction
  - a piece of code

Computational Complexity vs Profiling

Computational complexity

- focused on algorithms
- based on theoretical reasoning
- produces *functions* and *asymptotic* analyses
  - big "O"
- allows to find the "best" algorithm for the *problem* and the *data* at hand

Profiling

- focused on programs or pieces of programs
- based on *measures*
- measures are made available by the *operating systems*
- allows to find the "best" implementation for the *algorithm* and the *data* at hand

Profiling R programs (1)

`system.time()`

- takes an arbitrary R expression and returns the amount of time taken to evaluate that expression
- the output is a list of three time counts
  - user
    - the time spent for executing the code
  - system
    - the time spent for operating system tasks
  - elapsed
    - the time between the beginning and the end of the execution

Profiling R programs (2)

`system.time()`

- the argument can be either a single instruction or a sequence of instructions included in {}

Profiling R programs (3)

`system.time()`

Without parallel processing

user + system <= elapsed

why “<=“ and not “=“?