INTRODUZIONE

Insertion sort: sorting an unordered list. Worst case $O(n^2)$

An **algorithm** is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. It must terminate in a finite number of steps

Running time is useful to compare two algorithms one with another. We could also have different running time for the same algorithm, because it depends on the input: an already sorted sequence is easier to sort for example.

We need to parametrize it according to the size of the data. We would like to **seek an upper bound** on the running time, to have a sort of guarantee.

In general we use only one parameter, with 3 possible different ways of analyzation:

- **Worst case**, in this case we can understand the complexity of the algorithm. For example if we are designing a critical application, time matters and so the correctness of the application depends on the completion before the deadline. So we need an analysis in worst case terms, to be sure to always not reach the deadline.
  It is the maximum time of the algorithm on any input size.
- **average case**, is the most used in general, because we analyze the complexity of the problem by considering a statistical distribution such that there are some situations more frequent than others
- **best case**, it is theoretically important, because sometimes is important to know the best case because we can manipulate data to reach this situation

Try to do this analysis without relying on the characteristics of the machine running the algorithm (**machine independent**). It is important because in most cases we need to understand how algorithms are working and in order to compare them. In order to do this it is easier to try to understand the complexity of the solution provided not relying on the specificity of the machine.

We need also to understand the complexity not based on the input size of the data, thinking that the input data is tending to infinity. Because for example a complexity of n^2 or n^3 if we look at the beginning n^3 is better, but by looking at n to infinity we can conclude that the best is always n^2.

Theta is not a function Θ (tight bounds) , but is a classification, it is a set of function such that

$$\mathbf{\Theta(g(n))} = \{f(n) : \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n_0 \leq n\}$$

so it is a set of function that has the same property.

La Theta notation rappresenta il "caso medio" o la complessità esatta di un algoritmo.

All functions in the same set are considered equivalent. Engineering speaking we need to drop low order terms and ignore leading constants

Example $3n^3 + 90n^2 – 5n + 6046 = \Theta(n^3)$

The **O notation** (upper bounds) is instead

$$O(g(n)) = \{f(n) : \text{there exist constants } c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

So in this case $2n^2 \in O(n^3)$

La O notation rappresenta il "caso peggiore" o la complessità massima di un algoritmo

The **Ω omega notation** (lower bounds)

$$\Omega(g(n)) = \{f(n) : \text{there exist constants } c > 0, \ n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

La Omega notation rappresenta il "caso migliore" o la complessità minima di un algoritmo,

For example $\sqrt{n} = \Omega(\lg n)$

In particular, since the Θ is the average case, while O and Ω are the worst and best, we can define Θ as

$$\mathbf{\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))}$$

we can have also lower o notation or omega, such that in this case we restrict the definition

Merge sort

Example of a problem with recursion with subproblems

1. If n = 1, done.
2. Recursively sort A[1 . . n/2] and A[n/2+1 . . n] .
3. "Merge" the 2 sorted lists.

If we have just one element **Θ(1)**

If we have more than two elements then by applying the master theorem we have 2*T(n/2)+O(n) so final complexity is **Θ(n ln n)**

Substitution method

1. by some preliminary analysis you try to guess the form of the solution
2. verify your hypothesis by induction
3. solve for constants

if everything went fine, you have found a possible complexity of the algorithm

Master Method

E' un teorema ampiamente utilizzato nell'analisi della complessità degli algoritmi ricorsivi. È utilizzato principalmente per risolvere **equazioni di ricorrenza**, che descrivono il tempo di esecuzione di algoritmi ricorsivi in termini del loro input. Il teorema è particolarmente utile quando si analizzano algoritmi di tipo "divide et impera", in cui un problema viene suddiviso in sottoproblemi più piccoli, risolti separatamente, e poi le loro soluzioni vengono combinate per ottenere la soluzione del problema principale

$$\mathbf{T(n) = a * T(n/b) + f(n)}$$

- T(n) è il tempo di esecuzione dell'algoritmo per un input di dimensione n
- "a" rappresenta il numero di sottoproblemi generati dall'algoritmo
- "b" rappresenta il fattore di suddivisione dell'input.
- "f(n)" rappresenta il tempo di lavoro svolto dall'algoritmo per dividere il problema in sottoproblemi e combinare le loro soluzioni (no recursion here, because those two phases of splitting the problem and combining the solutions can be done using the data and without any type of recursion, so it is a constant time)

3 common cases, how the **f(n) is comparing respect to $n^{\log_b(a)}$**

1. if $f(n) = O(n^{\log_b(a) - \varepsilon})$ for some constant $\varepsilon > 0$, f(n) grows slower than the $n^{\log_b(a)}$, so the final solution is that $T(n) = \Theta(n^{\log_b(a)})$
2. if f(n) has a similar rate of $n^{\log_b(a)}$, then the $T(n) = \Theta(n^{\log_b(a)} \lg(n))$
3. if f(n) grows faster than the term of comparison, then T(n) = Θ(f(n))

Il master method è **usato per capire la recurrence**, quindi può essere anche applicato per poter capire la space complexity. T(n) è solo una recurrence formula, è solo un tool per risolvere la recurrence, non è studiato solo per capire le performance (un esempio è nel VLSI dove usiamo il master method per capire lo spazio occupato)

DIVIDE AND CONQUER

It is a paradigm:

1. problem splitted into subproblems without recursion (**divide**), so using the data that we have
2. **conquer**, since the size of the problem is smaller than the previous, maybe the solution is easier to compute

Those two parts are repeated recursively

3. **combine** is taking all the solutions done by the recursive part, and combine them to create a solution. In general it is constant, but it could also not be constant.

MERGE SORT

1. split in 2 parts
2. sort 2 sub arrays
3. linear time merge

$$\mathbf{T(n) = 2\ T(n/2) + \Theta\ (n)}$$

master theorem, a = b = 2, so $n^{\log_b(a)} = n$, so the function f(n) is in the second case because we have Θ (n) and $\Theta(n)$ which grow at the same rate, so at the end we get that $T(n) = \Theta(n^{\log_b(a)} \lg(n)) = \mathbf{\Theta(n \log n)}$

BINARY SEARCH is a sort of divide and conquer problem

we start from a sorted array, and we are looking for a certain item

1. check the middle element, and decide if we need to go on the left or right part or the array (if it is equal we have finished)
2. recursively search one subarray (so the recurrence it will happen only once)
3. Nothing, because we do not have to combine something, we need only to return the index that we have found

$$\mathbf{T(n) = 1\ T(n/2) + \Theta(1)}$$

(the recurred part is called only one, because we decide to go left or right respect to the middle element, in fact we have 1 as coefficient of T(n/2))

So in this case a = 1, b = 2 so $n^{\log_b(a)} = n^0 = 1$ so same rate of $\Theta(1)$, so we are in the second case and then the $T(n) = \Theta(n^{\log_b(a)} \lg(n)) = \Theta(n^0 \log n) = \mathbf{\Theta(\log n)}$

POWERING A NUMBER

Since it depends on the value of the power, we can divide the situation based on the nature of the power.

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

The naive algorithm is $\mathbf{\Theta(n)}$

The recurrence formula is:

$$\mathbf{T(n) = T(n/2) + \Theta(1)}$$

So as the case of binary search, we get $\Theta(\log n)$, this because of course when we need to do for example in the case of n even, $a^{n/2}$, we need to compute it only once, and not twice even if the formula is $a^{n/2}\, a^{n/2}$

L'idea è quella quindi di riutilizzare parte della computation già svolta, andando quindi a ridurre la complessità da un $\Theta(n)$ ad appunto $\mathbf{\Theta(\log n)}$.

MATRIX MULTIPLICATION

The value in the matrix C is computed by summing all the elements of A of a certain row multiplied by all the values of B of a certain column.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

It is very common, in AI deep learning we have lots of matrix multiplication, so we need of course to find out the best algorithm in order to reduce the time to compute this.
The standard algorithm needs **3 nested for loops**

- one for all the rows
- one for all the columns
- one to iterate all the elements of the column-jth and row i-th (only one because they have the same size, if not we couldn't do the matrix multiplication)

So complexity is $\Theta(n^3)$

In order to reach a better complexity, we apply the divide and conquer.
We consider 2x2 matrices in which each element like a,b,c… is a submatrix that will have half of the columns and rows.

$n \times n$ matrix $= 2 \times 2$ matrix of $(n/2) \times (n/2)$ submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}\right\} \begin{aligned} &8 \text{ mults of } (n/2)\times(n/2) \text{ submatrices} \\ &4 \text{ adds of } (n/2)\times(n/2) \text{ submatrices} \end{aligned}$$

If we consider also the way in which we get a value, for example

r = ae + bg

So we have

- 2 multiplications
- 1 addition

These are repeated for all the 4 elements of the matrix so at the end we get 8 total multiplications and 4 additions.

Of course since we said that all elements are submatrices of n/2 x n/2 size in conclusion we have

- 8 mults of(n/2 x n/2) submatrices
- 4 adds of(n/2 x n/2) submatrices

The **recursive part** of the problem is the one that does the **matrix multiplication**, the addition instead combines the results of the conquer part.
So the complexity of the matrix addition depends on the elements that we have in the matrix.

$$\mathbf{T(n) = 8*T(n/2) + \Theta(n^2)}$$

la parte di combine è così perchè bisogna andare a sommare il risultato di due matrici (n/2 x n/2) (cioè devo sommare membro a membro, quindi per forza mi servono due loop uno dentro l'altro per andare a ciclare colonne e righe)
essendo $n^{\log b(a)} = n^{\log 2(8)} = n^3$ siamo nel caso 1, per cui $\mathbf{T(n) = \Theta(n^3)}$
So in this case **the version proposed by the divide and conquer is worse than the standard algorithm**

A better way to solve matrix multiplication is
Strassen's idea
Solve the multiplication between two 2x2 matrices with only **7 recursive multiplications** (thanks to some formulas on the left)

$$\begin{aligned} P_1 &= a \cdot (f - h) \\ P_2 &= (a + b) \cdot h \\ P_3 &= (c + d) \cdot e \\ P_4 &= d \cdot (g - e) \\ P_5 &= (a + d) \cdot (e + h) \\ P_6 &= (b - d) \cdot (g + h) \\ P_7 &= (a - c) \cdot (e + f) \end{aligned}$$

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

He found out some strange formula in order to compute all the elements of the matrix result, by doing sum and difference, and by doing one less multiplication.

- 7 multiplication
- 18 add and diff

$$\mathbf{T(n) = 7\ T(n/2) + \Theta(n^2)}$$

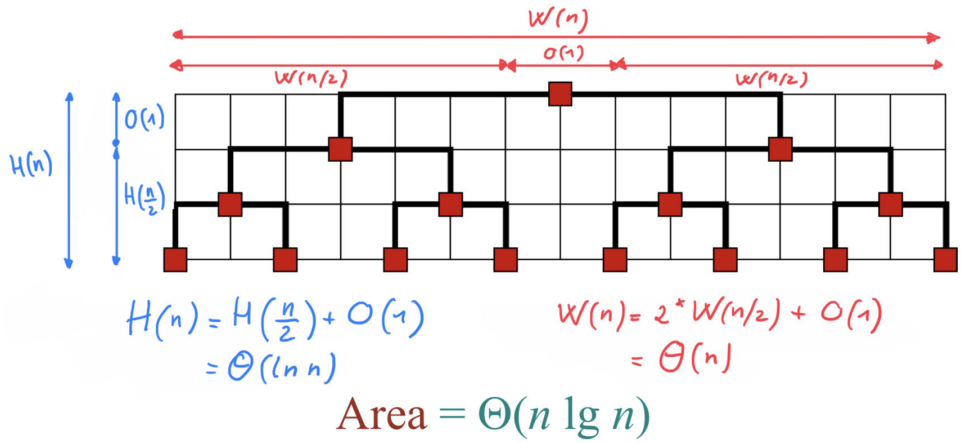
we are in case 1, because $n^{\log b(a)} = n^{\log 2(7)} = \mathbf{n^{2.81}}$ so $\mathbf{\Theta(n^{lg7})}$

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, **the impact on running time is significant**.
In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for n >=32 or so.

VLSI LAYOUT

Embed a complete binary tree with n leaves in a grid using minimal area (smaller chip used)
We have 2 parameters, H(n) and W(n) height and width of the grid.
We can describe those two variables with this recursive formula

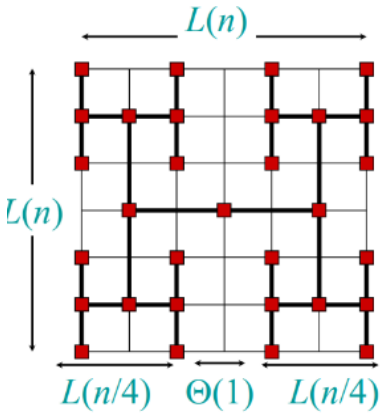
- **H(n)** = H(n/2) + Θ(1) = **Θ(lg n)** (qua ci serve sapere la posizione del nostra nuova root + l'effettiva altezza del nuovo tree con n/2 nodi, per cui non serve nessun 2*)
- **W(n)** = 2*W(n/2) + Θ(1) = **Θ(n)** (perché la grandezza totale del nostro albero è data da parte destra + parte sinistra che essendo uguali possiamo scrivere come 2*x)
- **Area = H(n) * W(n) = Θ(n*lg n)**



There is another possible way, which is not the binary tree, but the **h-tree**. The idea is that we split the problem in 4 parts

- upper right
- upper left
- down right
- down left

L(n) = 2*L(n/4) + Θ(1) and then with the master method we get that L(n) is **Θ(√n)** and then as final **Area = Θ(n)**



PRAM

Concept introduced in 1981, but parallel systems were also there at that time so this concept is still valid today. **PRAM is used to understand how fast a certain algorithm could go**.

Models are used to solve problems, so we use models to describe a parallel system. We need to describe how the machine is in order to get some conclusions by looking at how the applications are running on that machine, and maybe we can get some conclusions on how fast a certain algorithm is.

Pram is based on a machine called **Random Access Machine (RAM)**, which is a very abstract machine

- unbounded number of local memory cells
- each memory cell can hold an integer of unbounded size
- instruction set includes simple operations, data operations, comparator, branches
- all operations take unit time
- time complexity = number of instructions executed
- space complexity = number of memory cells used

Parallel Random Access Machine (PRAM), replicate a Ram m times.

In general we would need to define who is the producer, who is the consumer, where to retrieve the data, and coordination between all processors to not permit reading data when another is writing it. Instead in PRAM everything is much easier.

PRAM is an abstract machine for designing the algorithms applicable to parallel computers. m' is a system of infinitely many

- ram's m_1, m_1, … each m_i is called a processor of m'. All the processors are assumed to be identical. each has ability to recognize its own index i
- input cells x(1), x(2),…,
- output cells y(1), y(2),…,
- shared memory cells a(1), a(2),…, where data is exchanged between processors

So what are the characteristics of a PRAM

- unbounded collection of ram processors p_0 , p_1 …
- processors don't have tape
- each processor has unbounded registers
- unbounded collection of share memory cells
- all processors can access all memory cells in unit time
- all communication via shared memory

How does the **execution happen**? We have a program that is the same for all the processor (**single program, multiple data**) Each processing unit is doing the same thing but on different data, and in order to get a better communication between processes we follow those steps

1. reads a value from one of the cells x(1),…, x(n)
2. reads one of the shared memory cells a(1), a(2),…
3. performs some internal computation
4. may write into one of the output cells y(1), y(2),…
5. may write into one of the shared memory cells a(1), a(2),…

All the processors are doing the same phase, and there is **no synchronization** to be done, because all the process units are doing the same phase simultaneously. The **algorithm is sequential but is using multiple processing units**.

It could also happen that some processor could not be used, and maybe turned off. Some processor units can be used and others not in a certain cycle of the computation.

Even PRAM is easy, there might be **some problem** in the execution

- Two or more processors may read simultaneously from the same cell (not a real problem since the consistency is still valid)

- A **write conflict** occurs when two or more processors try to write simultaneously into the same cell (and this is a problem of course this happens in the model, so we need to be careful of this situation in the model and when we pass to the real machine we need to add some code to deal with the common write to know how to manage this situation)

Pram are classified based on their read/write abilities (realistic and useful)

- **exclusive read**(er) : all processors can simultaneously read from distinct memory locations
- **exclusive write**(ew) : all processors can simultaneously write to distinct memory locations
- **concurrent read**(cr) : all processors can simultaneously read from any memory location
- **concurrent write**(cw) : all processors can write to any memory location
- erew, crew, crcw

How to deal with concurrent writes?

- **priority CW**, the highest priority (of the processing unit) is allowed to complete write, not easy to implement
- **common CW**, all the processing units that are writing in the same place, if they are writing the same value we are ok. If we are not in this situation, then the algorithm is illegal and the machine state will be undefined.
- **random CW**, we chose randomly which processor wins

Advantages of PRAM

- it is natural: the number of operations executed per one cycle on p processors is at most p
- it is strong: any processor can read/write any shared memory cell in unit time
- it is simple: it abstracts from any communication or synchronization overhead, which makes the complexity and correctness of pram algorithm easier
- it can be used as a benchmark: if a problem has **no feasible/efficient solution on pram**, it has **no feasible/efficient solution for any parallel machine**

Model a is **computationally stronger** than model b (a>=b) iff any algorithm written for b will run unchanged on a in the same parallel time and same basic properties.

$T^*(n)$	Time to solve problem of input size *n* on <u>one</u> processor, using best <u>sequential</u> algorithm
$T_p(n)$	Time to solve on *p* processors
$SU_p(n)=\frac{T^*(n)}{T_p(n)}$	Speedup on *p* processors
$E_p(n)=\frac{T_1(n)}{pT_p(n)}$	Efficiency (work on 1 / work that could be done on p)
$T_\infty(n)$	Shortest run time on any p
$C(n)=P(n)\cdot T(n)$	Cost (processors and time)
W(n)	Work = total number of operations

- $T^* \neq T_1$
- $SU_p \leq P$
- $SU_p \leq \frac{T_1}{T_\infty}$
- $E_p \leq 1$
- $T_1 \geq T^* \geq T_p \geq T_\infty$
- IF $T^* \approx T_1$, $E_p \approx \frac{T^*}{pT_p} = \frac{SU_p}{p}$
- $E_p = \frac{T_1}{pT_p} \leq \frac{T_1}{pT_\infty}$
  - NO USE MAKING *P* LARGER THAN MAX SU:
    - E→0, EXECUTION NOT FASTER
- $T_1 \in O(C)$, $T_p \in O(C/p)$
- $W \leq C$
- $p \approx$ AREA, $W \approx$ ENERGY, $\frac{W}{T_p} \approx$ POWER

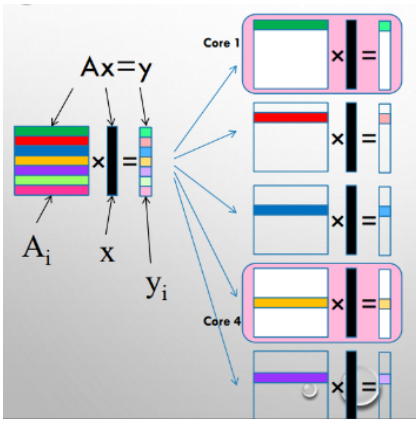
These are some definitions for some parameters of the PRAM.

Speedup: comparing your time of execution with the best case of the sequential algorithm
Cost: is the total time that you spend on each processor

If only 80% of the algorithm can be parallelized, we will not reach a speedup of 5.
It is not a good result. The **best possible situation is when all the code can be parallelized and no sequential part**. In this case we will obtain a **linear speedup**, if we increase the number of processors, we also increase the speedup.

examples **DA SAPERE (chiesto l'anno scorso)**

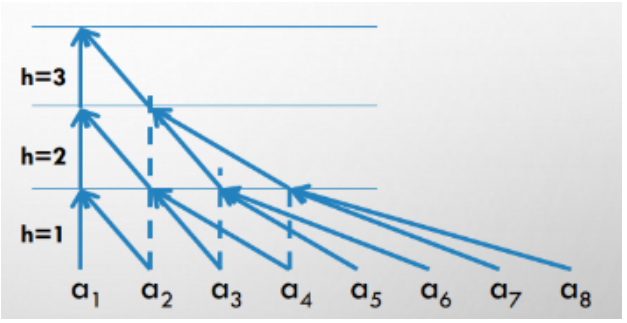
1. To understand how PRAM works, we look at **a matrix multiplied by a vector**. So we need to take a row of the matrix and multiply it for the value in the vector and we repeat this for all the rows of the matrices.
   Best possible parallelization is to have a number of processors equal to the number of rows.
   We will also have exclusive writes, since all the calculations are independent.
   T*(N) è n^2 perché stiamo lavorando con una matrice e il numero di moltiplicazioni da effettuare è nell'ordine di n^2.
   There is no conflict, so linear speedup (no sequential, all processes are active)



2. Here we look at how we cannot always use all the processors available. We would like to **make an addition of a series of elements** in order to get the final result
   - sequential problem: T*(N) repeat N times the addition for the
   - parallel algorithm: in the first phase we are reading all the data with all the processor, but then we start summations until we have
     - first phase when all the processors are reading the data from memory and store them in intermediate values O(1)
     - store back the final result to the memory O(1)
     - then because **each time we divide by 2 the number of processor used** O(log n)

     T(N) = 2+log n
   - SU = T(N) sequential / T(N) parallel
   - by increasing p we have an efficiency that decreases (Ep)

   we can change the way of analyzing the problem, so that the **number of data that we have is very large in comparison to the available processors** (N>>P)
   - for the parallel algorithm, we have a different formula (on webeep there is a document that explains it)
   - so more or less the **SU depends on P (linear speedup)**
   - and the efficiency is something very similar to 1ù



Nei due casi precedenti lo pseudo code è semplicemente due global read, computazione e poi global write

3. Now we look at **matrix multiplication**. Each cell of the final matrix can be parallelized. We need n^3 processors because there are 2 levels of parallelization

- one processor for each cell in the matrix,
- but then we can further parallelize the work that has to be done for each cell

For each cell in fact we need to perform

- n multiplications
- log n summations to reduce everything to a single value

So at least we need n^3 processors (n*n*log n)

(EACH PROCESSOR KNOWS ITS I,J,L INDICES)

BEGIN

1. $T_{i,j,l} = A_{i,l}B_{l,j}$
2. FOR H=1:K
   IF $l \leq n/2^h$ THEN
   $T_{i,j,l} = T_{i,j,2l-1} + T_{i,j,2l}$
3. IF $l = 1$ THEN $C_{i,j} = T_{i,j,1}$

END

- STEP 1: COMPUTE $A_{i,l}B_{l,j}$
  - CONCURRENT READ
- STEP 2: SUM
- STEP 3: STORE
  - EXCLUSIVE WRITE

1. compute all the possible combinations for the multiplication (A elements with B elements). We store the result in a matrix (it takes 1)
2. then we need to perform the reduction on the matrix of before, summing up the pairs of multiplication (this part is gonna take 1 unit of time repeated the number of iteration for the loops, which are log n)
3. we need to store the values in a final matrix, only some of the values of the temporary matrix (of course this is done with exclusive writes) (takes 1)

Final complexity is $2+\log_2 n$, of course this is true only if the number of processors is n^3.

There are also some other **variants of the PRAM**:

- bounded number of shared memory cells. small memory pram (input data set exceeds capacity of the share memory i/o values can be distributed evenly among the processors)
- bounded number of processor small pram. if # of threads of execution is higher, processors may interleave several threads.
- bounded size of a machine word. word size of pram
- handling access conflicts. constraints on simultaneous access to share memory cells

Lemma

Assume p'<p, **any problem that can be solved for a p processor pram in t steps can be solved in a p' processor pram in T' = O(tp/p') steps** (assuming same size of shared memory)
(10 times less processors, we will have 10 times slower results)
This lemma can be generalized also in the case of memory, if we have less memory than the one required by the best unbounded algorithm

With those two lemmas we are able to understand the performance in respect to real machines, or at least the bounds.

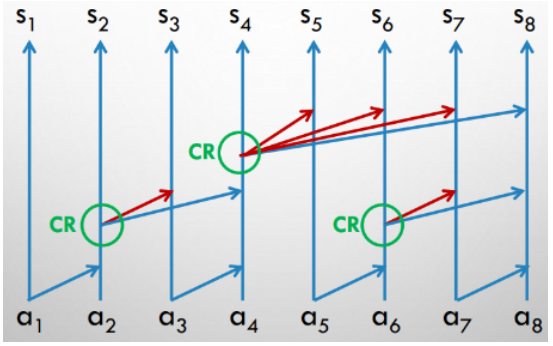
example the **prefix sum** (also applicable not only to the sum, but also to other things, we need only that the operation must be associative)

The reduction parallel pattern creates an efficiency which is not 1, because some of the processors are not used. **Prefix sum tries to use those idle processors to do something more**.
We have an item, like n numbers, and we would like to compute the summation of all those numbers, but also the summation of all the prefixes (like considering only first and second, only first second and third, ecc.) to obtain a final vector like this

$$a_1,\ a_1+a_2\ ,\ a_1+a_2+a_3\ \ldots$$

In order to compute this in a sequential way you need to take in consideration the accumulated value and in each iteration you need to create a new value.
In a parallel way we can parallelize the reduction, the summation. Is there any way to exploit the middle values? Yes
Vertical lines are the outputs, so with 8 values, we need to have 8 results. We need to create a parallel tree, starting from the lower part and parallelizing the additions 1+2, 3+4, 5+6… and then we pass to another iteration and parallelize the other operations and so on.

The total number of processors that we are using in the reduction are 4 in the first stage then 2 and finally 1. So, since the total number of processors are 4, then respectively in each stage 0,2,3 processes are doing nothing. For example the 2 processors which are not used in the reduction during the second stage, they are used to do another thing like combining (1+2)+3 (all the red lines). So meanwhile we create the binary tree, we are using the processors not used to combine some values.
CR sono le concurrent reads



In this case we are performing more work than the sequential case. Here we have an algorithm that is spending more energy than the sequential one. Of course we are doing more operations, but the time taken in order to get the results is in this case log n, which is better than the n of the sequential case. **So less time, but more work to do**.
If we have a number of processors equal to N/2

Is PRAM implementable?
Today we have architectures that can exploit the same concepts of PRAM (like vector parallelizations, GPU word, generalization of the parallel architecture)

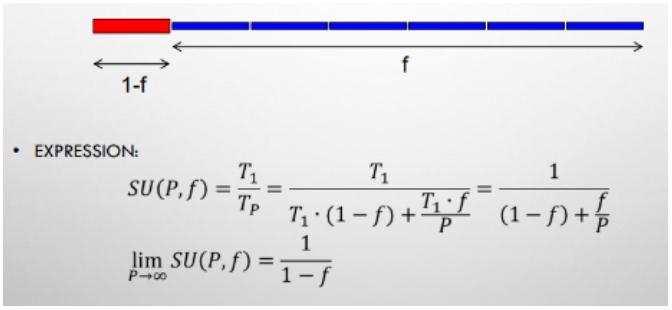
Why PRAM?

- large body of algorithms
- easy to think about
- sync version of shared memory → eliminates sync and comm issues, allows focus on algorithms,cbut allows adding these issues and allows conversion to async versions
- exist architectures for both sync (pram) model and async (sm) model
- pram algorithms can be mapped to other models

Amdahl's law
It is also named strong scaling. Gene Amdhal was one of the three architects of IBM mainframes.
In order to do the computation I have a certain model and an algorithm that has 2 parts

- one related to something non parallelizable
- one potentially parallelizable

The sequential time is T_1 and the parallelizable is T_p, so the time taken by T_p is the time for the sequential algorithm times the fraction of the total algorithm + (the parallel part) the time of parallel multiplied by the fraction and divided by the number of total processors, so at the end by simplifying we get the final formula



This formula **is used to know how much the parallelization impacts on the total time of the algorithms**.

If we have an infinite number of processors, we couldn't have a speedup greater than the part of the algorithm that is not parallelizable.

se una parte significativa di un programma deve essere eseguita in modo sequenziale, non importa quanto si aggiungano processori o risorse parallele, il miglioramento delle prestazioni sarà limitato dalla parte sequenziale.

Gustafson's law

With the hypercube application they were able to get a speedup in the same order of the numbers of processors used.

What changes are the assumptions. **Gustafson understood that the problem is not only to parallelize the algorithm, but also to understand the speedup in terms of the amount of the data that we have**, and in general the sequential part is always the same, even if the multiple data is more than before. So if we want **to achieve better speedup, the solution is to try to work with the most possible data on the parallel phase**.

Weak scaling because amdahl is considering only a fixed portion of data, instead **here we are considering a fixed amount of time**. In that fixed time we can increase the number of processors to increase the amount of data that we can process.

We get a linear speedup by changing the amount of data (increasing it)

The examples are the training models, where the more data we have the better it is.

RANDOMIZED BASED ALGORITHMS

Algorithms are deterministic. Here instead we are trying to understand how to analyze algorithms with a sort of randomization. So the time taken to execute the algorithm could have different values thanks to this randomization.

Randomization is sometimes the best solution to find a solution for a problem. If it is done correctly, it can find a solution in a better way than other complex solutions.

The problem is that the time taken is very variable. So the analysis technique

- Assume a probability distribution for your inputs
- Analyze item of interest over the distribution

We need to distinguish between the average case and the worst case. The average case is the one with more interest.

Moreover:

- Specific inputs may have much worse performance
- If the distribution is wrong, analysis may give misleading picture

"Randomize" the algorithm; **for a fixed input, it will run differently depending on the result of random "coin tosses".**

Key points

- Works well with high probability on every input (try to understand for which cases the algorithm works well)
- May fail on every input with low probability

Indicator variables. Suppose we want to study random variable X that represents a composite of many random events Define a collection of "indicator" (not a single event, but it is used to aggregates more events) variables X_i that focus on individual events; typically $X = \Sigma X_i$

Linearity of expectations, $E[X] = E[Y+Z] = E[Y] + E[Z]$, because we address in general we work with the average case (the expected value).

HIRING PROBLEM AND GENERATING RANDOM PERMUTATIONS

Explanation of the problem: *You need to hire a new employee. The headhunter sends you a different applicant every day for n days. If the applicant is better than the current employee, then fire the current employee and hire the applicant. Firing and hiring is expensive. How expensive is the whole process?*

- **Worst case** is when the headhunter sends you the n applicants in increasing order of goodness. Then you hire (and fire) each one in turn: n hires
- **Best case** is when the headhunter sends you the best applicant on the first day. Total cost is just 1 (fire and hire once).
- **Average cost?** An input to the hiring problem is an ordering of the n applicants. There are n! different inputs, any of them can happen. The average cost is the expected value.
  All the possible sequences of input are permutations, all of which have the same probability of happening, which is 1/n!. Random variable X(s) is the number of applicants that are hired, given the input sequence s. What is E[X]?

Change viewpoint: instead of one random variable that counts how many applicants are hired, consider n random variables, each one keeping track of whether or not a particular applicant is hired.

Indicator random variable X_i for applicant i (the random variable is focused on the applicant and not on the sequence of them):

- 1 if applicant i is hired,
- 0 otherwise

In order to know how many applicants are considered in a sequence it is just a matter of summing up all the values of X_i

$E[X_i]$ = Pr["applicant i is hired"] and probability of hiring i is probability that i is better than the previous i-1 applicants…

1234	2134	3124	4123
1243	2143	3142	4132
1324	2314	3214	4213
1342	2341	3241	4231
1423	2413	3412	4312
1432	2431	3421	4321

8/24 = 1/3

So the idea is that for example if we chose 4 applicants, we have 4! = 24 possible permutations. If we consider i = 3, then we need to check in how many permutations, the third applicant is better than the two values before him (only the red cases are correct).

Recall that X = the sum of the X_i 's (each X_i is the random variable that tells whether or not the i-th applicant is hired)

$E[X] = E[\sum X_i] = \sum E[X_i] = \sum Pr[X_i = 1] = \sum 1/i \leq \ln n + 1$ (ultima disequazione vale per harmonic number formula)

In conclusion the **average hire is ln n which is much better than the worst case n**.

Two types of randomized based algorithms

randomized sorting algorithm and the min-cut algorithm exemplify two different types of randomized algorithms

- **Las Vegas algorithm**: the sorting algorithm always gives the correct solution the only variation from one run to another is its running time
- **Monte Carlo algorithm**: the min-cut algorithm may sometimes produce a solution that is incorrect. We are able to bound the probability of such an incorrect solution (sometimes in fact like in the optimization problems we do not need a perfect correct result, but something in the same order).

  T**here are two possible ways to be wrong when finding a solution:**
  - those with **one sided error**, if the probability that it errs is zero for at least one of the possible outputs (YES/NO) that it produces
  - those with **two sided error**, if there is a nonzero probability that it errs when it outputs either YES or NO

Which is better, Monte Carlo or Las Vegas? The answer depends on the application. In some applications an incorrect solution may be catastrophic

A **Las Vegas algorithm is by definition a Monte Carlo algorithm with error probability 0**.

- A Las Vegas algorithm is an efficient Las Vegas algorithm if on any input its expected running time is bounded by a polynomial function of the input size (if the average is polynomial)
- A Monte Carlo algorithm is an efficient Monte Carlo algorithm if on any input its worst-case running time is bounded by a polynomial function of the input size

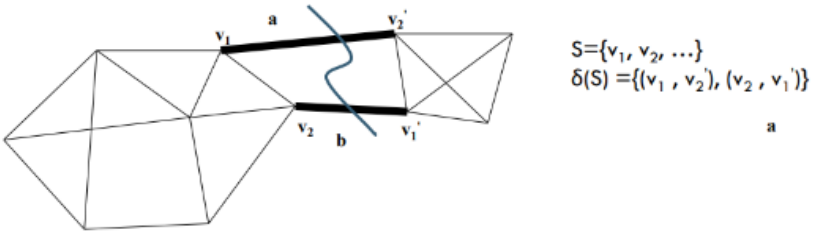
For example if we need to find out an element in an array of elements, where half of the items is a and half is b

- Las Vegas, each time you select randomly an element of the array. This algorithm succeeds with probability 1. The running time is random (and arbitrarily large) but its expectation is upper-bounded by O(1) (worst case could be infinite, since it could run forever without finding a)
- Monte Carlo, in this case we perform a number of interactions maximum equal to k, so if after k iterations we are still searching, the algorithm fails. The probability of finding an a after k steps is $1 - 1/2^k$

The selection is executed exactly k times, therefore the runtime is O(k)

MIN CUT ALGORITHM

It is a monte carlo algorithm. The problem is **finding out the minimum cut of a graph**.



Let G = (V, E) be a connected undirected graph. Let n = |V|, m = |E|. For S ⊂ V , the set δ(S) = {(u, v) ∈ E : u ∈ S, v ∈ S' } is a cut since **their removal from G disconnects G into more than one component**.

Goal: Find the cut of minimum size. So the size of S must be minimum in order to split the graph in two different parts.

This problem is really linked to the st-cut problem, in particular the min cut problem was solved by solving n-1 min st-cut problems. Then t**he size of the min-st-cut is equal to the value of the max-st-flow** (equivalent by linear programming duality). In general if the graph is dense, then $m = n^2$, and the total complexity of this problem is $O(nm \log(n^2/m))$

KARGER APPROACH

To solve the min-cut problem (without the st-condition) without using any max-flow computations we use **Karger which is based on a randomized approach**.

The algorithm will start initially with a simple graph as input, and then it will need to consider **multigraphs**, which are graphs where there are possibly multiple edges between a pair of vertices. It also removes all self edges.

Then we perform a **contraction**, we select an edge at random, for example the one that connects 1 and 2, and we build a new graph with the same edges, but for any edges connecting to 1, or 2 we connect to a new node 1,2. We combine those 2 vertices. Moreover it deletes also the self loops

The number of edges is still the same between the nodes, but of course what changes are the edges between the nodes considered (elimina l'edge che connette i due nodi)

The algorithm continues the contraction process until only two vertices remain (n − 2 edges contracted). These two vertices correspond to a partition (S, S') of the original graph, and the edges remaining in the two vertex graph correspond to δ(S) in the original input graph.

Lemma 1: The probability that the Karger's algorithms ends up with the min cut of δ(S), is ≥ 1/(n 2)
No proof

If we repeat the process and each time we keep the best solution, it will become better and better until we obtain the minimum. If we repeat the algorithm a number of times which is l(n 2). The probability that at least one run succeeds is at least $(1- 1/(n\ 2))^{l(n\ 2)} \geq 1-e^{l}$

If we put l = c ln n, then the error probability is $\leq 1/n^c$ and then it is easy to implement Karger so that one run takes $O(n^2)$ time (n edges, do the contractions and then repeat).

By if we want a polynomial error, we need to repeat this process a number of times that depends on the size problem, so the final complexity of this algorithm is **$O(\log n\ n^4)$**

KARGER AND STEIN

At the beginning the probability of picking the wrong edge is pretty low, but this probability increases by the end of the algorithm (perchè se piglio il wrong edge all'inizio posso comunque probabilmente

arrivare al minimum cut, ma se lo piglio verso la fine, probabilmente questo mi porta a non riuscire a raggiungere il minimum cut)
Proprio perché la probabilità di errore è molto più alta verso la fine dell'algoritmo, l**'idea è quella di fermarsi prima di un certo threshold t.**
L'azione di contract viene fermata quando si raggiunge un numero di nodi che mi da 50% di possibilità di scegliere edge corretti e 50% di edge sbagliati, ovvero un numero di nodi pari a $n/\sqrt{2} +1$.
Questa ricorsione viene effettuata due volte e si sceglie il minimo risultato tra le due, di modo tale che contraendo in questo modo abbiamo una alta probabilità di non contrarre la cosa sbagliata (contraggo 2 volte fino a t nodi e scelgo il minimo)
We can choose l to have the probability of success is about ½. Then, the probability of success of running from n vertices down to l is = (l 2)/(n 2).
So if we put (l 2)/(n 2) >= ½, then we can say that $l = n/\sqrt{2}$, and that's why the threshold t is equal to this value + 1, in order to not go down the ½ probability.
Seguendo il master theorem la complessità dell'algoritmo è la seguente

$$\mathbf{T(n) = 2^* T (n/\sqrt{2}) + O(n^2)}$$

- 2 è perché richiamo 2 volte il problema di contraction
- the contraction procedure creates a graph of size $n/\sqrt{2}$
- n^2 perchè in genere il processo di contrazione potrebbe richiedere molte iterazioni per trovare il minimum cut con alta probabilità (al quadrato perchè i grafi sono rappresentati con matrici e quindi devi iterare colonne e righe)

Since f(n) has a similar rate of $n^{\log_b(a)}$, then the $T(n) = \Theta(n^{\log_b(a)} \lg(n)) = \mathbf{\Theta (n^2 lg(n))}$
If we want a polynomial error then the final complexity is $\mathbf{O(n^2 lg^3(n)}$

La soluzione è quindi fermarsi prima nella contraction almeno sono statisticamente sicuro di non raggiungere la parte di grafo dove è più facile sbagliare, e ripeto lo stesso processo due volte andando a selezionare il min (così miglioro anche la stima).
Poi da un certo numero di nodi in poi (tipo 6 assolutamente arbitrario) farò la contraction normalmente, dato che poi se faccio $n/\sqrt{2} + 1$ ottengo 5,... e quindi non ho più la fascia garantita quindi posso solo applicare normalmente l'algoritmo

SORTING ALGORITHM

QUICKSORT

Proposed by C.A.R. Hoare in 1962. It is a **divide-and-conquer algorithm**. Sorts "in place" (like insertion sort, but not like merge sort). Very practical (with tuning).

Divide and conquer approach, because there are a lot of ways to implement quicksort. **Now there is no randomization**, after we will introduce some randomization.

1. Divide: Partition the array into two subarrays around a pivot x such that elements in lower subarray x elements in upper subarray.
2. Conquer: Recursively sort the two subarrays.
3. Combine: Trivial

This divide part must be implemented with the lowest complexity possible, because if we do so, we can have very good implementation.

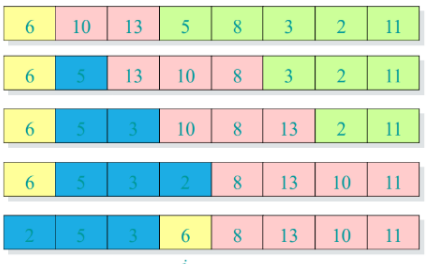
Partition subroutine

p, q are the extremes of the subarrays that we are considering.

- pivot element is the element on which the ordering is based (the first one), **all the elements greater than the pivot are put after, instead the lower elements are put before**.
- we start selecting the first element as pivot = A(p)
- then we analyze all the array until we get to the final element, then if the j-th element is less then the pivot, then we increase i and we store in the A(i) the element, and the element in the position A(i) is stored in the A(j) (i is the index to indicate the elements stored before the pivot) (the idea is that the first element greater than the pivot and the new element which is less than the pivot are exchanged, see the example on the slides)
- at the end we switch the pivot with the last element smaller than the pivot (A(i)) so that we have the pivot exactly in the middle of the array

```
PARTITION(A, p, q)  ▷ A[p . . q]
  x ← A[p]          ▷ pivot = A[p]
  i ← p
  for j ← p + 1 to q
      do if A[j] ≤ x
            then  i ← i + 1
                  exchange A[i] ↔ A[j]
  exchange A[p] ↔ A[i]
  return i
```

Running time = $O(n)$ for n elements.



The worst case complexity is Θ(n)

Then we need to **re order another time the two subarrays**, so we need to introduce some **recursion**

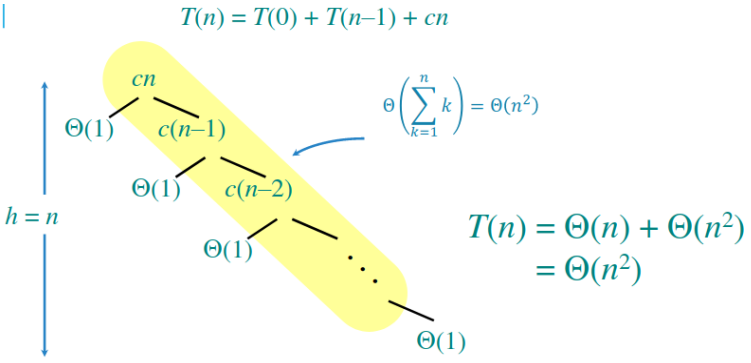
- if p < r then there is something to sort and we need to create the partition first, and then recall this function for the two different subarrays
- otherwise there is nothing to sort, because the subarrays has nothing between the two extremes so we are done

In the **worst case** we can have two different situations

- all the items sorted
- all the items are reverse sorted

Those are bad situations, because we cannot perform any partitioning operation. Because if the pivot is the first one, which could be the lowest one or the highest one, we can

- perform no operation T(0)



- change all the elements T(n-1)

$T(n) = T(0) + T(n-1) + \Theta(n) = \Theta(1) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n) = \mathbf{\Theta(n^2)}$

This could be seen also as an arithmetic series, because we need to perform that summation.

This complexity is very bad, but fortunately is only for worst cases, we need to understand now how many bad cases do we have

Best case is the one in which when we select a pivot we have exactly half elements in a half and half elements in the other, so the recursion formula is this one

$T(n) = 2T(n/2) + \Theta(n) = \mathbf{\Theta(n \lg n)}$

Even if we are not splitting the array correctly, like having for example one set with one element and the other set with 9 elements, then if we look at the tree of possibilities, then the longest path is $\log_{10/9} n$ in this case, which is still in the order of n. So luckily also in this case we still get **Θ(n lg n)**

If one of the two partitions is not empty we always get this value, otherwise we are not lucky and we get the worst case complexity.

RANDOMIZED QUICKSORT

What can we do to always be in the lucky part?

Not always the first element as a pivot, but chose it in a random way. Whatever is the initial order, we do not have problems because if we chose it in a random way we are pretty sure that we will get no recursion tree where we always have one side with 0 element, it is very improbable. **Randomization removes the worst possible case, so the n^2 factor (perchè è solo 1 caso su tutti i possibili, quindi è statisticamente molto improbabile)**, without making assumptions on the initial order (since it will be destroyed by the randomization), we need only a random number generator.

Let T(n) = the random variable for the running time of a randomized quicksort on an input of size n, assuming random numbers are independent.

For k = 0, 1, ..., n–1, define the **indicator random variable,** which describes which kind of partition we may have (we analyze a class of events), we look at the effects and not to the input data

X_k = 1 if PARTITION generates a k : n–k–1 split, 0 otherwise.

Since we have n possible partitions as outcome to the pivot selection, then the probability that X_k happens is 1/n, all equal probable.

So we can define **T(n) as different possible recursions based on the different possible selections of the pivot** (image on the left), and since all are equally probable

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split}, \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split}, \\ \vdots \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split}, \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))$$

$$\begin{aligned} E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))\right] \\ &= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \\ &= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \\ &= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n) \\ &= \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n) \end{aligned}$$

we can compute T(n) as summations of all those recurrences (image above).

Then by applying the expected value to this summation (image on the right), and by then saying that the expected value of X_k is 1/n, because as we said before all X_k are equally probable, so $E[X_k] = Pr\{X_k = 1\} = 1/n$ we arrive to that final formula.

Then we can upper bound this E[T(n)] to a a*n*lg n for constant a > 0.
Of course we need to choose a value of a large enough in order to have a*n*ln n >= E[t(n)].
Then by using the substitution method, we arrive at the final solution (image left), which proves our initial idea of bounding E[T(n)] with a*n*ln n.
So at the end we get that in the average case the quicksort works in **Θ(n*ln n)**.

$$
\begin{aligned}
E[T(n)] &\le \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n) \\
&= \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n) \\
&= an\lg n - \left(\frac{an}{4} - \Theta(n)\right) \\
&\le an\lg n
\end{aligned}
$$

Some properties

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort (since merge sort uses auxiliary memory)
- Quicksort can benefit substantially from code tuning.
- Quicksort behaves well even with caching and virtual memory.

Is there any better algorithm with complexity lower than n logn? Theoretically no
The idea is to look at sorting algorithms, and we always have to pick two elements in the array and according to the sorting comparison function we may do different things. By doing so we can create a **decision tree**, which can be created always with any algorithm.
The **running time of the algorithm is the length of the path taken**, so the worst case is the longest path possible in the decision tree. So the next theorem will show how the complexity cannot be less then n*ln n.
In conclusion, the best worst-case running time for comparison sorting is O(n lg n) .

Theorem. Any decision tree that can sort n elements must have height Ω(n lg n) .
Proof.

- The tree must contain >= n! leaves, since there are n! possible permutations.
- A height-h binary tree has <= 2^h leaves.
- So we have x >= n! and 2^h >= x, so 2^h >= n!

$$
\begin{aligned}
\therefore\ h &\ge \lg(n!) && \text{(lg is mono. increasing)} \\
&\ge \lg\left((n/e)^n\right) && \text{(Stirling's formula)} \\
&= n\lg n - n\lg e \\
&= \Omega(n\lg n)\,. \ \square
\end{aligned}
$$

Apply lg to both sides of the equation, Then by reforming the second term and using stirling formula we get that the h = Ω(n lg n)

This **lower bound on the complexity of sorting is only for comparison sorting algorithms**. If we do not compare values, we can reach better complexities, like in the next algorithms.

COUNTING SORT

No comparison between elements, we need only an auxiliary storage to store some temporary values. The **only assumption is that the elements that are stored in the array are always between 1 and k**.

1. initialize the auxiliary array with 0, with a number of elements equal to k
2. in the C we will add 1 to the i-th cell that when we encounter the value i in the A(j)

$\Theta(k)$ for $i \leftarrow 1$ to k do $C[i] \leftarrow 0$

$\Theta(n)$ for $j \leftarrow 1$ to n do $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ for $i \leftarrow 2$ to k do $C[i] \leftarrow C[i] + C[i-1]$

$\Theta(n)$ for $j \leftarrow n$ downto 1 do $B[C[A[j]]] \leftarrow A[j]$, $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$

3. then you C(i) = C(i) + C(i-1)
4. then we add the elements in the B array (the final one). In the example of the slides there is also the C' which is the C updated with the last row of code on the left
   We start by looking at the last item in the initial array, this element will be put inside B at the C(A(j)).
   Then we subtract 1 at the C(A(j)) element, and we repeat this process for all the elements

If k = Θ(n), **then counting sort takes Θ(n) time**. But, sorting takes Θ(n lg n) time! Where's the fallacy?
This is because here we are not comparing, there are no if statements in the code. Counting sort is not a comparison sort, it is a count sort.
The **problem of this algorithm is that it is going to use a lot of space** even if the time complexity is better.
Moreover, Counting sort is a stable sort: **it preserves the input order among equal elements**.

RADIX SORT

Origin: Herman Hollerith's card-sorting machine for the 1890 U.S. Census.
Digit-by-digit sort.

- Hollerith's original (bad) idea: sort on the most significant digit first.
- Good idea: **Sort on least-significant digit first with auxiliary stable sort**

It was just a matter of wrong order, but the algorithm was correct.
Apply each time the counting sort when you move from a type of digit to another
This is because for example **if we use the normal counting sort we would need an additional array equal to the largest value in the array**, but if this is too big, then we would lose a lot of space. **By looking at the digits then we need only an array of size 10**.
Of course this is if we use the decimal digits.

Now we look at how much memory and passes do we need to perform if we use as stable sort the counting sort and we encode the numbers in binary (lo facciamo per non fare il confronto bit a bit ma lavorando su gruppi un po' più grandi per diminuire il numero di volte da ripetere il counting sort)
If we consider n words each of b bits, then each word can be viewed as having b/r base (posso suddividere una parola da b bits in gruppi da r bits).
So if we choose 32 bit words, and then r = 8, then

- the counting sort will be repeated b/r = 4 times.
- the total auxiliary memory needed is $2^r = 256$ bits = k since it is the max value

Recall: Counting sort takes Θ(n + k) time to sort n numbers in the range from 0 to k – 1.
If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time (so here the $k = 2^r$). Since there are b/r passes, we have

$$\mathbf{T(n,b) = \Theta(b/r\ (n + 2^r))}$$

Choose r to minimize T(n, b): **increasing r means fewer passes, but as r > lg n, the time grows exponentially**, since k grows too much
Choosing **r = lg n** implies **T(n, b) = Θ(b*n/lg n)**
Then in general b = d*ln n so in the end we get **Θ(d*n)** which is linear

Radix sort is fast for large input. Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies

SELECT I-TH

Input: A set of n (distinct) numbers and an integer i, with $1 \leq i \leq n$

Output: The element with rank i x in A that is larger than exactly i-1 other elements of A.
In order to do so we should sort the array, and then find out the i-th element, so the total complexity would be **n*ln n**.
Can we spend less instead of n*ln n , like only n to find the i-th element?
We can say that if we want for example the minimum or the maximum value we can have a lower complexity than the one proposed, so there can be something better that we can do.

Select the i th smallest of n elements (the element with rank i).
Two versions:
RANDOMIZED DIVIDE-AND-CONQUER ALGORITHM (linear in average)
Which is in some way very similar to the quicksort, if the two extremes overlap then there is no value in position i, instead we take the pivot at random and we partition in two parts in the same way as the quicksort. The only difference is that here **we do not have to perform the recursion in both the two subpartitions**, because we need to find and element, and if it is less than the pivot, we need only to perform the recursion on that subarray, and not also for the part of the elements greater than the pivot (in the case it is in the upper part, then you do not need to search the element i, but the element in position in relative position in respect to the pivot element, so i-k).

```
RAND-SELECT(A, p, q, i)   ▷ ith smallest of A[p..q]
  if p = q then return A[p]
  r ← RAND-PARTITION(A, p, q)
  k ← r − p + 1           ▷ k = rank(A[r])
  if i = k then return A[r]
  if i < k
     then return RAND-SELECT(A, p, r − 1, i)
     else return RAND-SELECT(A, r + 1, q, i − k)
```

The recurrence formula is exactly the same as the quicksort, but without the 2 before the T(n) part, as we just explained before. As for quicksort we can have a lucky situation and an unlucky one

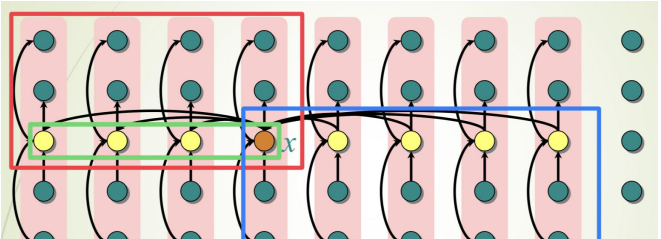
- **T(n) = T(9n/10) + Θ(n)** (if the array is splitted evenly, but it can be generalized if none of the two parts is empty)
- **T(n) = T(n-1) + Θ(n)** (one of the two is empty)

Randomized order statistic selection is quite good since it has a linear expected time (because as in quicksort the randomization deletes the worst case scenario, it is less probable), but the **worst case** is very bad since it is **$\Theta(n^2)$**.
Is there an algorithm that runs in linear time in the worst case?
Yes, due to Blum, Floyd, Pratt, Rivest, and Tarjan [1973] without any randomization
IDEA: Generate a good pivot recursively. So we look now at a new algorithm

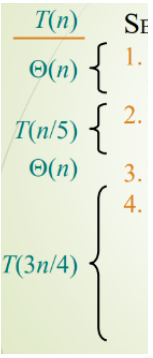
WORST-CASE LINEAR-TIME ORDER STATISTICS
Very similar structure of before, the recursion is the same (point 3 and 4), the pivot is a little bit different than before.
SELECT(i, n)

1. Divide the n elements into groups of 5. Find the median of each 5-element group.
2. Recursively SELECT the median x of the floor(n/5) group medians to be the pivot.
3. Partition around the pivot x. Let k = rank(x).
4. if i = k then return x
   else if i < k
   a. then recursively SELECT the ith smallest element in the lower part
   b. else recursively SELECT the (i–k)th smallest element in the upper part

Drawing example in the slides



By looking at the drawing, we can see that for sure x is bigger than **floor(floor(n/5)/2) = floor(n/10)** medians so the **green part** of the drawing.
If we also consider the fact that each of those medians is bigger than half of each of its vector, than in this case with vectors of 5 elements, the x will be bigger than **3*floor(n/10)**.
By symmetry we can also say that at least **3*floor(n/10)** are bigger than x.
And with the minor simplification we can easily see that if n >= 50 than 3*floor(n/10) >= n/4 for very slightly. So in this case the recursive call to SELECT in step 4 is performed on a number of elements <= ¾*n
So in the worst case Step 4 takes T(3n/4).
And so for n < 50 th worst time is T(n) = Θ(1).



Analyzing the complexity

1. First part is mostly linear
2. Then for the recursive parts we are working with the median and we are computing them, so that's why it depends on T (because we need to know how much this algorithm takes and then we can divide it by 5)
3. linear again
4. same seen in the example, and with the minor simplification we saw that the worst case will have T(3n/4)

So the recurrence formula in the end is this one, and since we do not have a good formula for the master theorem method (abbiamo due termini con T(n)), we use the substitution method to prove that this formula is actually less than cn

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) + \Theta(n)$$

Substitution: $T(n) \le cn$

$$\begin{aligned} T(n) &\le \frac{1}{5}cn + \frac{3}{4}cn + \Theta(n) \\ &= \frac{19}{20}cn + \Theta(n) \\ &= cn - \left(\frac{1}{20}cn - \Theta(n)\right) \\ &\le cn \end{aligned}$$

when we perform the operation of cn -... we can do this because we need to chose c large enough to handle both the Θ(n) and the initial conditions, so that the Θ(n)>= 0 and then we can conclude that **T(n) <= cn**
Even **if it has a worst case complexity equal to n, it is only used theoretically, since it is much more complicated than the randomized select-ith**.
The groups are by 5 because otherwise we are not able to upperbound and lowerbound the complexities, and we are not able to prove with the substitution method the final complexity.

Deterministic version (derandomization – linear in the worst case)

PRIMALITY TEST

We would like to solve primality tests with a randomized approach.
What is a **prime number**? An integer p >= 2 is prime iff (a | p → a = 1 or a = p).
One naive way to check this property is to make some guesses, and for all the numbers not even, the approach is to try and check all the elements from 2 up to √n and we control if the number has a divisor between all those values. We need only to check up to √n, the others do not add any information and can be seen as multiple of others value less than √n
Of course if the number is quite large this could take a lot of time, since the total complexity is **Θ(√n)**.

composite numbers: number with more than 2 factors

RANDOMIZED APPROACH

New algorithm that in some cases is going to answer yes it is a prime, or not it is not a prime (sort of Monte Carlo approach). But not always those sentences are true.
In order to improve the number of numbers that we are declaring prime are really prime, the idea is to **run the algorithms a lot of iterations (k)**
So the **probability that n is not prime is at most p^k**.

Now we define some definitions or observations or theorems which are important to understand how effectively we can create an algorithm to check if a certain number is prime or not.

- **Observation from Fermat**: each odd prime number p divides 2^{p-1}-1. Of course this is true if we know in advance that p is prime, then if it is not or we do not know, it could still happen but we have no certainty. **Necessary condition but not sufficient**
  *For example p = 17, then 2^{16}-1 = 17*2855.*
  Simple primality test:
  1. Calculate z = 2^{n-1} mod n (like the divider and conquer powering number)
  2. if z = 1
     a. **then n is possibly prime** (since we are going backwards and the observation is only necessary not sufficient)
     b. else n is composite

  The advantage is that this only takes polynomial time
  So from 2^{n-1} we can say if a number is possibly prime but we are not still sure
- A natural number n>=2 is a base-2 **pseudoprime** if n is composite and 2^{n-1} mod n = 1.

(for our study now the value 341 seems to be a prime, because 2^{340} mod 341 = 1, but we are not sure because as before Fermat observation is only necessary)

- **Theorem: (Fermat's little theorem)**: If p prime and 0 < a < p, then a^{p-1} mod p = 1
  The idea is that we are **enlarging the first observation of Fermat by saying that the base can be not only 2, but any number a**.
  So the definition of pseudoprime can change, to n is pseudoprime to base a, if n not prime and a^{n-1} mod n = 1.

So now by doing so we can prove that 341 is not prime because with base 3 we obtain something different than 1 (3^{340} mod 341 = 56, so 340 is not prime)

Now with this last theorem we are ready to create an **algorithm** but not by looking at all possible values of a, but butter randomizing the value of a

1. Randomly choose a [2, n-1]
2. Calculate a^{n-1} mod n
3. if a^{n-1} mod n = 1

a. then n is possibly prime
b. else n is composite

Everything depends on the probability of not being a prime and classified as a prime, so we need to run a lot of times the algorithm so that we can increase the probability that a real prime is classified a prime by our algorithm

- **Carmichael numbers**
  A number n>=2 is a Carmichael number if n is composite and for any a with GCD(a, n) = 1 (greatest common divisors) we have $a^{n-1} \bmod n = 1$
  Only a smaller subpart of numbers is going to satisfy this additional condition

The idea is that by only looking at mod n = 1 we cannot be sure that the n is prime, so we need to add some constraints with this necessary theorem.

- **Theorem** (always necessary condition)
  if p prime and $0 < a < p$, then the only solutions to the equation $a^2 \bmod p = 1$ are a = 1 and a = p – 1.
  So a is called **non-trivial square root** of 1 mod n, if $a^2 \bmod n = 1$ and a is not either 1, n – 1.

For our algorithm, we can perform the **fast exponentiation**, and so **during the computation of a^{n-1}** (0 < a < n randomly chosen), **test whether there is a non-trivial square root mod n**. (using the powering of the number that takes lg n). If we get a non trivial square root during the fast exponentiation, then we are sure that this number is not a prime even if the final result of mod n = 1.

The final algorithm takes in input a value n for which we would like to check if it is prime, then we need to repeat k times this procedure to increase the probability of being right in what we are saying. Then we take a value "a" at random and we perform the fast exponentiation. If we get mod n == 1 and we are sure that there are no trivial square roots, then the number is prime.

- **Theorem** if n is composite, there are at most n-9/4 integers $0 < a < n$, for which the algorithm primalityTest fails.
  The probability of failing is pretty high, because there are a lot of bases that will fail the algorithm.

Where was this used?
For example in **public key cryptosystems** like the RSA. They are related to exchanging messages between parties. So with this traditional encryption of messages with secret keys the
Disadvantages:

- The key k has to be exchanged between A and B before the transmission of the message.
- For messages between n parties n(n-1)/2 keys are required, one for each pair

Advantage: Encryption and decryption can be computed very efficiently

Diffie and Hellman (1976)

Each participant A has two keys:

- a public key P_A accessible to every other participant
- a private key S_A only known to A

We use those two keys in order to code and decode messages that we want to share.

Prima encrypt con la chiave pubblica così poi quello con la privata può decriptarlo.

The idea is that S_A is not computable from P_A

RSA cryptosystem

Remarks the idea of Diffie and Hellman using the prime generator number

1. Randomly select **two primes p and q** of similar size, each with l+1 bits ($l \geq 500$)
2. Let $\mathbf{n = p \cdot q}$
3. Let **e be an integer that does not divide $(p - 1) \cdot (q - 1)$** so e relatively prime to $(p - 1) \cdot (q - 1)$ so $GCD(e, (p - 1) \cdot (q - 1)) \equiv 1$
4. Calculate $\mathbf{d = e^{-1} \bmod (p - 1)(q - 1)}$ which means $d \cdot e \equiv 1 \bmod (p - 1)(q - 1)$
5. **Publish P = (e, n)** as public key
6. **Keep S = (d, n)** as private key

Quindi chiunque voglia comunicare con me, che ho pubblicato questi valori deve andare a suddividere il messaggio in interi tra 1 e n-1 che chiamiamo ai, e quindi poi andrà a pubblicare a^e. Dato che poi io sono l'unico ad avere l'esponente segreto d, sono in grado di poter decifrare il messaggio. Quindi un ipotetico hacker che vede il messaggio criptato e sa le informazioni pubbliche fa fatica a decifrare il messaggio, in quanto per decifrare ha bisogno di d. d si calcola veloce se hai p e q, cosa che però devo ottenere. Ma trovare due numeri p e q partendo da n e che siano primi, ovvero fattorizzare n è un problema computazionalmente impegnativo.
In this computation we use a lot of compositions of numbers, which is a quite hard problem to solve. It is not the most secure way, but it is a nice idea

DATA STRUCTURES

We add randomization in the way a data structure is defined, so we add randomization in another way (no more to the algorithms)

- Random Treaps
- Skip Lists

Both the data structures related to this topic are very linked to **dictionaries.**

A dictionary is a collection of elements each of which has a **unique search key** (Uniqueness criteria may be relaxed (multiset)(i.e. do not force uniqueness)) Keep track of current members, with periodic insertions and deletions into the set

It is similar to a database.

We would like to understand how complex it is to work with dictionaries, and we would like to have a data structure that is also dynamic so that it can change with some primitives like insert or delete.

Some general primitives

- Search(x,S): Is $x \in S$?
- Insert(x,S): Insert x into S if not already in S.
- Delete(x,S): Delete x from S.
- Minimum(S): Return smallest key
- Maximum(S): Return largest key
- List(S): Output elements of S in increasing order by key
- Union(S_1,S_2): Merge S_1 and S_2 Condition: $\forall\ x_1 \in S_1$, $x2 \in S_2 : x_1 < x_2$
- Split(S,x,S_1 ,S_2): Split S into S_1 and S_1 . $\forall\ x_1 \in S_1$, $x2 \in S_2 : x_1 \leq x$ and $x < x_2$

So we would like to know how much times does it take to perform those operations

UNORDERED ARRAY

- searching and removing takes O(n)
- inserting takes O(1)

useful for **frequent insertions, rare searches and removals**

ORDERED ARRAY

- searching takes O(log n) time (binary search)
- inserting and removing takes O(n) time (perchè devi modificare anche tutto l'array)

useful for **frequent searches and rare insertions and removals**

BINARY SEARCH TREES

A binary search tree is a binary tree T such that keys stored at nodes in the left subtree of v are less than or equal to k. Keys stored at nodes in the right subtree of v are greater than or equal to k.

Of course **operations on the binary tree depends on how much balanced is the structure**

There are some ways to increase the performance

- **AVL trees**, trees that along the way are rotated partly in order to create a more balanced tree. For example during insertion or deletions we can check the depth of the right and of the left part of the tree to know exactly if the tree is well balanced. We need then some operations to rotate O(log n)
- **Splay trees**, self adjust the binary tree. Any time we perform the insertion of something, we reorganize the tree such that the most searched items are moved on top of the tree (so that the most frequent have a lower cost).

  Drawback is that maybe performing this operation will need some auxiliary data so it has a cost, moreover they restructure the entire tree not only during updates but also while performing simple search and finally it could perform a logarithmic number of rotations.

Now we look at the data structures with randomization.

RANDOM TREAPS

They are randomized binary trees. treaps achieve essentially the same time bounds. **they do not require any explicit balance information**.

The **expected number of rotations performed is small for each operation**.

Simple to implement. Skip lists similar benefits but alternative randomized data structure

When we build a binary search tree, if **n elements are inserted in random order into a binary search tree, the expected depth is 1.39 log n** (this only because it is done in a random way). So **insertion is log n** in random binary search trees built at random (mostly very well balanced).

But since insertions cannot be controlled by the programmer then this property could not hold. This is not true, because the idea is to find a way to insert an element in the binary tree such that that insertion will build a new binary tree as if the element was inserted in a randomized way.

The problem is finding a way to insert elements as if it is completely random.

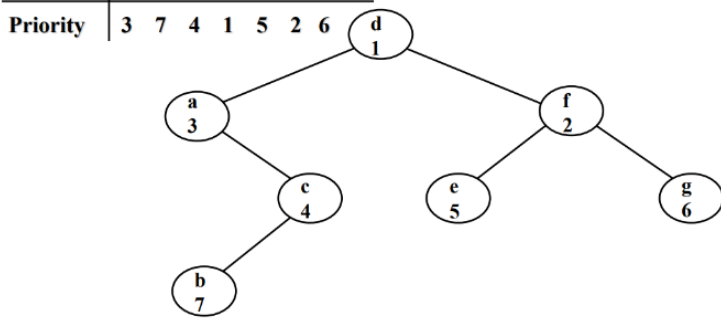
So we assign a priority to each value to have an order with which we decide who has to be associated with the binary search tree.

If this **priority is chosen at random, then we are creating a randomized binary tree, with the well balanced property shown before**.

So a treap is a binary tree, and each node has not only the key value, but also the priority of before. All the vertex are going to have those two values

- **Search tree property**
  For each element x:
  - elements y in the left subtree of x satisfy: key(y) < key(x)
  - elements y in the right subtree of x satisfy : key(y) > key(x)
- **Heap property**
  For all elements x,y:
  - If y is a child of x, then prio(y) > prio(x) child priority is greater than parent priority
  - All priorities are pairwise distinct.

key	a	b	c	d	e	f	g
Priority	3	7	4	1	5	2	6



Once defined a priority and a relative order, we can say that

Lemma: For elements x1 , ... , xn with key(xi) and prio(xi), there exists a **unique treap**

Priority and key values define the structure of the tree

Proof by induction on n:

- n = 1: ok
- n > 1:
  - The root has the smallest priority (k1 ,p1) (the root will be always this one)
  - All the elements y with key(y) < k1 go on the left subtree ≤ n-1 (since this holds for n-1, if we add another unit it still holds, and so it works for every possible treap)
  - All the elements y with key(y) > k1 go on the right subtree ≤ n-1

We could also say that the **search tree has the structure that would result if elements were inserted in the order of their priorities**

Proof by induction on n:

- n = 1: ok
- n > 1:
  - The element with the smallest priority is in the root whatever it is the insertion order (k1 ,p1)

- All y such that key(y) < k1 go in the left subtree
- All y such that key(y) > k1 go in the right subtree

Then recursively we could prove it also for the right subtree and the left one

For **searching a certain node in the tree with a key K**, the **Running time: O(# elements on the search path)**
To prove this we start by looking at some characteristics of the search path.

Let's consider a set of xi elements such that they are ordered based on the key (so i = 2 means it is the 2nd element with smallest key)
Let's M be a subset of those elements and $P_{min}(M)$ = element in M with lowest priority. Let's find out x_m
Lemma (to try to understand who is parent of who):

- If **i < m** then

  x_i is ancestor of x_m iff $P_{min}(\{xi,\ldots,xm\}) = xi$ (**xi is traversed in the search before xm iif it is the element with lowest priority in the subset M**)
- If m < i

  then x_i is ancestor of x_m iff $P_{min}(\{xm,\ldots,xi\}) = xi$

Proof (only for the first part, since the second one is pretty similar only reversed)
Since it is a iif we need to prove both ways so

- **"⇐"**, so we start from the fact that the element having with minimum priority is xi. So it will be inserted before all the other elements in P (priority property). The path that xi has followed is the same also for others because the tree contains keys k that can be k < key(x_i) or k > key(x_m). Let's see key(x_{i+1}) > key(x_i)
  - right path, so we have **key(x_i) > k**, but key(x_{i+1}) > key(x_i) > k, so **key(x_{i+1})> k**
  - left part, so we have **key(x_i) < k,** but key(x_{i+1}) < key(x_i) < k, so **key(x_{i+1})< k** (because for sure k is greater than all the keys of set P)
- **"⇒"**, we know that xi is an ancestor of xm and we need to see that the Pmin is the one associated to x_i. Let's assume by contradiction that this element is not x_i but x_j.
  Using the property 1, we consider the search path when x_j is inserted, which is the first one to be inserted (the one with lowest priority). Then all the items x_l have the same path of x_j, so they are all childs of x_j. So x_j ancestor of x_l. Two cases
  - $x_j = x_m$, but by saying this than x_m is an ancestor of all x_l it contradicts the hypothesis
  - $x_j \neq x_m$, but also in this case the structure does not hold, because we would have xm on the right and all the others elements on the left

  The only possible structure possible is to have i = j

Now we are ready to look at the complexity of a successful search.
Let's be the n-th harmonic number the image of the right.

$$H_n = \sum_{k=1}^{n} 1/k$$
$$H_n = \ln n + O(1)$$

Lemma:

1. Successful search: The expected number of nodes on the path to xm is $H_m + H_{n-m+1} - 1$.
2. Unsuccessful search : Let m be the number of keys that are smaller than the search key k. The expected number of nodes on the search path is $H_m + H_{n-m}$.

And then since the H_n (image) is bounded up to log n then we can conclude that the complexity is ln n
Proof (only for part 1 since 2 is similar)
Since we have some randomization, we define an indicator random variable
$X_{m,i}$ = 1 if xi is an ancestor of xm, 0 otherwise

So we can see from the picture that Xm is equal to 1 (because we assume that at the end we find out the element), plus the right part, the left one

La foto di destra riprende esattamente il lemma che abbiamo osservato prima, ovvero considerare il subset P che va da i a m. Quindi tutti gli elementi da i+1 fino a m sono ancestor di m perchè hanno key minore di i, quindi un range di (m-i+1) All those elements have the same probability of happening so the final value is the one shown here.

Similarly for i > m we have 1/(i-m+1)

X_m = **# nodes on the path from the root to x_m (incl. x_m)**

$$X_m = 1 + \sum_{i<m} X_{m,i} + \sum_{i>m} X_{m,i}$$

$$E[X_m] = 1 + E\left[\sum_{i<m} X_{m,i}\right] + E\left[\sum_{i>m} X_{m,i}\right]$$

i < m **:**

$$E[X_{m,i}] = \text{Prob}[x_i \text{ is ancestor of } x_m] = 1/(m-i+1)$$

$$E[X_m] = 1 + \sum_{i<m} \frac{1}{m-i+1} + \sum_{i>m} \frac{1}{i-m+1}$$

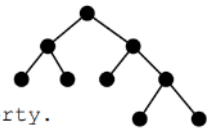
$$= 1 + \frac{1}{m} + \ldots + \frac{1}{2} + \frac{1}{2} + \ldots + \frac{1}{n-m+1}$$

$$= H_m + H_{n-m+1} - 1$$

So **searching the element is in the order of log(n)**

How the **insertion** works

```
1. Choose prio(x).
2. Search for the position of x in the tree.
3. Insert x as a leaf.
4. Restore the heap property.
   while prio(parent(x)) > prio(x) do
     if x is left child then RotateRight(parent(x))
     else RotateLeft(parent(x));
     endif
   endwhile;
```



search takes log n and then we expect that the number of rotation will be maximum 2.

Deletion of an element

```
1.Find x in the tree.
2. while x is not a leaf do
       u := child with smaller priority;
       if u is left child then RotateRight(x))
       else RotateLeft(x);
       endif;
   endwhile;
3. Delete x;
```

Another Lemma (non spiegato molto bene, non so se gli interessi)

Lemma: The expected running time of insert and delete operations is O(log *n*). The expected number of rotations is 2.

Proof: Analysis of insert (delete is the inverse operation)
rotations = depth of *x* after being inserted as a leaf (1)
- depth of *x* after the rotations (2)
Let $x = x_m$.
(2) Expected depth is $H_m + H_{n-m+1} - 1$.
(1) Expected depth is $H_{m-1} + H_{n-m} + 1$.
The tree contains *n-1* elements, *m-1* of them being smaller.

rotations = $H_{m-1} + H_{n-m} + 1 - (H_m + H_{n-m+1} - 1) < 2$

n = number of elements in treap T. Complexities for others operations for the dictionaries

- Minimum(T): Return the smallest key. O(log n) (going left left left all the time)
- Maximum(T): Return the largest key. O(log n) (going right right all the time
- List(T): Output elements of S in increasing order. O(n)
- Union(T1 ,T2):
- Split(T,k,T1 ,T2):

We consider now the split and the union

- **Split** T into T1 and T2 . $\forall$ x1 $\in$ T1 , x2 $\in$ T2 : key(x1) $\leq$ k and k < key(x2), and we would like to perform this as soon as possible.
  Two situation
  - if K is in T then we need to delete the element with key k and re-insert it into T1 after the split operation (we need only to add another log n to the total cost)
  - otherwise we can
    - Generate a new element x with key(x)=k and prio(x)=-∞.
    - Insert x into T, and it will be putted in the root, because it has the lowest possible priority (so it is changed accordingly to the insertion algorithm such that all the lowest elements are on the left and all the highest are on the right)
    - Delete the new root. The left subtree is T1 , the right subtree is T2

    In terms of cost constant time + log n (**since insertion cost log n**) + constant time

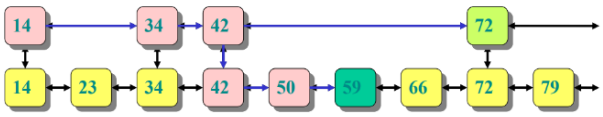
  In the end **log n complexity** for split
- **Union**, Merge T1 and T2 . Condition: $\forall$ x1 $\in$ T1 , x2 $\in$ T2 : key(x1) < key(x2)
  - Determine key k with key(x1) < k < key(x2) for all x1 $\in$ T1and x2 $\in$ T2 (so **the max for T1 and the minimum of T2** which cost both log n)
  - Generate element x with key(x)=k and prio(x) = -∞.
  - Generate treap T with root x, left subtree T1 and right subtree T2 .
  - Delete x from T (**perché x non è un valore reale, l'abbiamo scelto noi**)

  Total cost is log on for finding k, then constant and constant and finally again log n.
  In the end **log n complexity** for union

How to generate the priority is now the most important problem (using a random generator)
Priorities from [0,1). In case of equality, extend both priorities by bits chosen uniformly at random until two corresponding bits differ.

SKIP LISTS

Another data structure invented in 1989. It is easy to implement.
All the operations will have O(lg n) time per operation in expectation and with high probability (in average). In addition we can also control how probable this complexity will be.
Start from the simplest data structure: (sorted) **linked list**. Searches take Θ(n) time in worst case
How can we speed up searches?
Suppose we had two sorted linked lists (on subsets of the elements of the first list). Each element can appear in one or both lists. How can we speed up searches?

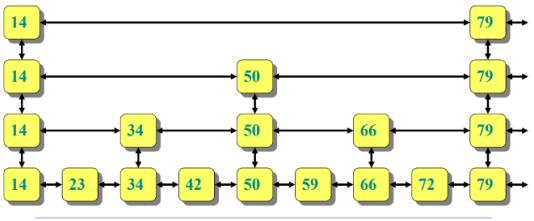


We start from the list having the smallest number of vertex, because by doing so you can know the between elements in which you

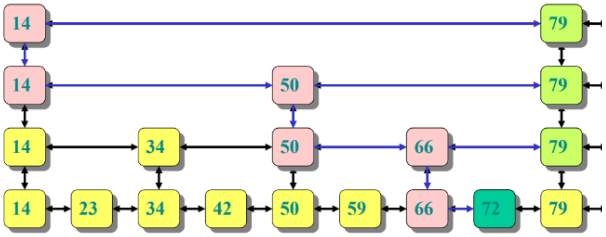
have to search (very similar to the binary search, first you take the fastest lane skipping a lot of stations and then you move to the local one to do all the stations).
Which nodes should be in L1 in order to have better performance?
Best approach: **Evenly space the nodes in L1**. But how many nodes should be in L1?
The **cost is minimized when the cost taken for the fast lane and the cost to move between the station in between is equa**l, so when |L1| = |L2|/|L1|.
So everything is fine when $|L1|^2 = |L2| = n$ so $|L1| = \sqrt{n}$.
So finally the total cost is $\sqrt{n} + n / \sqrt{n} = 2\sqrt{n}$. And so the number of stations in between the stations of the fast line is in the order of $\sqrt{n}$.

To go towards the case of binary search, we consider more linked lists

- 2 sorted lists $\Rightarrow 2 \cdot \sqrt{n}$
- 3 sorted lists $\Rightarrow 3 \cdot \sqrt[3]{n}$
- k sorted lists $\Rightarrow k \cdot \sqrt[k]{n}$
- lg n sorted lists $\Rightarrow \lg n \cdot \sqrt[\lg n]{n} = 2 \lg n$



On the right we have the example of log n sorted list.
The number of items in between is going to be 2^x-1, in fact we have 1,3,7…
So now we really have binary search.
Ok so now how can we do this in a dynamic way, so that i can also change the data structure?



Insertion
To insert an element x into a skip list:

- SEARCH(x) to see where x fits in bottom list (of course starting from the upper list).
- Always insert into bottom list

It will take log n
But of course sometimes we also need to add this element to upper lists. So **to which lists should we promote our value**?
This can be done by randomization, so flip a (fair) coin; if HEADS, promote to the next level up and flip again. Probability of promotion to the next level = ½.
On average:

- 1/2 of the elements promoted 0 levels
- 1/4 of the elements promoted 1 level
- 1/8 of the elements promoted 2 levels
- etc.

Simple approach but very balanced and it will create the target that we want.

Delete
parti dal basso ed elimini l'elemento e riarrangia i puntatori per mantenere la lista. Poi sali e continui ad eliminare se trovi l'elemento. Eventualmente se la lista è vuota la elimini

A **skip list** is the result of insertions (and deletions) from an initially empty structure (containing just $-\infty$)

- INSERT(x) uses random coin flips to decide promotion level

- DELETE(x) removes x from all lists containing it

The big complexity is to prove that the structure of the skip list goes with the distribution that we have in mind. **How good are skip lists?** (speed/balance)

- INTUITIVELY: Pretty good on average
- CLAIM: Really, really good, almost always because of the next theorem

Theorem: With high probability, every search in an n-element skip list costs O(lg n)

This is by using c log n linked lists, and if we increase the value of c then this probability is more likely to be O(lg n) and the error that we will have is $1/n^{c}-1$, so by increasing c we can see that the error becomes lower.

DYNAMIC PROGRAMMING

The term dynamic programming was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another.
The word dynamic was chosen by Bellman to capture the time-varying aspect of the problems, and because it sounded impressive.
The word programming referred to the use of the method to find an optimal program, in the sense of a military schedule for training or logistics. This usage is the same as that in the phrases linear programming and mathematical programming, a synonym for mathematical optimization.

Longest Common Subsequence (LCS)
Given two sequences x[1 . . m] and y[1 . . n], find a longest subsequence common to them both
(the sequences do not have to be with characters all near one to the other)
There are many possible solutions for this problem.
Brute force solution: Check every subsequence of x to see if it is also a subsequence of y.

- Checking = O(n) time per subsequence.
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time = $O(n2^m)$ = exponential time.

Simplification: at the beginning do not compute the subsequence itself, but first compute the length

1. Look at the length of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Consider prefixes of x and y, subsets of the two sequences, then we define

- c[i, j] = |LCS(x[1 . . i], y[1 . . j])|
- c[m, n] = |LCS(x, y)|.

Theorem
c[i, j] =

- c[i–1, j–1] + 1 if x[i] = y[j] (because we know for sure that at least one is in common since we have x[i] = y[j] so that's the +1, and then we need to repeat the idea for the before part)
- max{c[i–1, j], c[i, j–1]} otherwise.

With this idea we are saying that the solution of the problem is created by looking at the solutions of subproblems.

```
LCS(x, y, i, j)    // ignoring base cases
  if x[i] = y[ j]
     then c[i, j] ← LCS(x, y, i–1, j–1) + 1
     else c[i, j] ← max{LCS(x, y, i–1, j),
                        LCS(x, y, i, j–1)}
  return c[i, j]
```

#1 Key idea of DP: an optimal solution to a problem (instance) contains optimal solutions to subproblems.
This is the recursive algorithm in order to find out the length of the LCS.
#2 : A recursive solution contains a "small" number of distinct subproblems repeated many times. We may reuse past solutions
In the case of the LCS, the number of distinct LCS subproblems for two strings of lengths m and n is only mn. Because those are all the possible combinations that we can have.
Memoization: So the idea now is that after computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work
And the solution is to use a 2D matrix
And thanks to how we have created the algorithm from the beginning, the value c[i,j]

```
LCS(x, y, i, j)
  if c[i, j] = NIL
     then if x[i] = y[j]
        then c[i, j] ← LCS(x, y, i–1, j–1) + 1
        else c[i, j] ← max{LCS(x, y, i–1, j),
                           LCS(x, y, i, j–1)}
```

same as before

can be easily stored in a matrix. So now the idea is that we do not have to call recursion, but instead we only need to look at other cells in the matrix.
Finally the **time complexity is Θ(mn**) because we need to compute all the possible situation, but of course we need to use an additional **space which is Θ(mn)**

So the table can be constructed normally, then we can also use this in reverse to obtain the effective string.

DP APPLIED TO BOOLEAN FUNCTION

DP applied to data structure. It is another possible way on how DP can be applied.
General idea of this data structure is to **describe the boolean function**.
With this data structure we can solve this problem in a polynomial time in general (not always).
The idea is related to how logic functions are described, with a true table with all the possible combinations of the input. In order to this logic function we can use

- the **true table** 2^n rows (exponential to the number of variables)
- **first canonical form**
- **second canonical form** even in those two cases the problem is exponential

The problem is that in all those solutions the number of entries is exponential. So the idea is to try and represent the function with a graph (DAG).
Key idea is to **exploit an hashing mechanism or a memoization table in order to avoid redoing some computations**. And this representation is the DAG. We would like to achieve the canonical form, such that each graph represents exactly one boolean function.
If the size of the graph is small, then the number of operations on this graph will be polynomial (because the number of operations that we need to perform do not depend on the number of entries, but on the size of the graph)

DAG is built recursively with the channon decomposition: given a function I may describe this function as the variable i am considering multiplied by the function with value = 1, plus the complemented variable multiplied by the function with value = 0.
By doing so we get a binary tree, and the leafs of this tree are the mid terms of the function.
Then we need to apply **two rules**

- every time you have any **node with two identical children, the node is eliminated**
- **two nodes with isomorphic graphs BDD, we keep only one of them**

These two rules are important, because it make that each node represents a distinct logic function

- Then we need to maintain the same relative order of the variables from the root following any possible path

If we maintain this thing, then we may say that by following the rules of before we always create a canonical representation of the boolean function (**Reduced Ordered BDD is canonical**). Since it has the same order of the Shannon.

Comparing **if two functions are the same is to compare if the two DAGs are the same** (which is a constant time problem when we already have the two DAGs).
The idea is to always keep track of each vertex in the memoization table which is built from the leaves to the root.

All the possible paths from the root to the value 1 describe possible assignments to the function that makes the function equal to 1

Two types of memoization table (l'idea è di stoppare la recursion e perdere meno tempo come nel DP di LCS)

- **Unique table**, which avoids duplication of existing nodes, Hash table: hashfunction(key) = value
- **Computed Table**, which avoids recomputation of existing results (we store not everything but only the most frequent ones so no collision, because if the value is there we can use it, otherwise it is not there because it is not a frequent case)

Dynamic tables that grow dynamically based on the variables and the number of operations that we perform.

So it is possible to represent each possible logic function like the classical AND, OR… with the ITE operator (ite(f,g,h)), in which all the parameters are boolean functions represented as DAG. The idea is to solve this problem in a recursive way.
There is one to one mapping between this logic functionality and the recursive algorithm that goes along the graph to create all the possible logic functions.
Penso che l'idea sia quella di applicare la recursion per scendere nel grafo e quindi devi andare poi a gestire tutte le varie casistiche con degli if else (se f == 1, f == 0, g == h, se ho il valore nella unique table, se ho valore nella computed table…)

AMORTIZED ANALYSIS

It is related to **dynamic tables**, tables that change size according to the data that we have to store.
The cost of insertions is not always the same in dynamic tables (very high in some cases and very low in others). We cannot simply use the worst case, because it is very far from the real value.
An amortized analysis is any strategy for analyzing a sequence of operations to show that the **average cost per operation is small, even though a single operation within the sequence might be expensive**.
An amortized analysis guarantees the average performance of each operation in the worst case.

Hash tables are similar to arrays, but here we do not store all the information, we store only the data that we would like to insert to this data structure and not more.
The real problem is that we do not know the proper size of the table in advance
We would like to have **a table as small as possible, but large enough so that it won't overflow**.
The idea is to use a table that stores information

- if the table is not able to store something
- we create a new data structure and we save the new item
- we store all the previous items to the new data structure
- we deallocate the old array

The policy is to double the size from the previous table.
How about complexity?
Consider a sequence of n insertions. The worst case time to execute one insertion is **Θ(n)**, so if we have n insertions we have n***Θ(n) = Θ(n²)**
The worst case is happening very frequently at the beginning, but then is not happening so frequently, in fact the worst case cost for n insertion is only **Θ(n)**. Let's see why
Let ci = the cost of the i-th insertion.

- if i-1 is an exact power of 2 we need to reallocate for a new table with double size, and so the cost is equal to i
- otherwise it takes 1

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

If we recompute the cost of n insertions we can use this summation, so for n insertions we get as final total cost **Θ(n)**. This means that in average each dynamic table operation is **Θ(n)/n = Θ(1)**

$$
\begin{aligned}
&= \sum_{i=1}^{n} c_i \\
&\leq n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \\
&\leq 3n \\
&= \Theta(n)
\end{aligned}
$$

So we are not computing the worst case by considering the avg cost and multiplying it by the number of operations that we are performing

Three types of amortization

- aggregate, **cost of each insertion, sum and then divide by n**
- accounting
- potential

Accounting method
Using a cost for doing the thing is very similar to the bank account

- spend > average cost => we take some money from the bank account.
- spend < average cost => we are increasing the bank account with some money.

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$	2	3	3	3	3	3	3	3	3	3
$bank_i$	1 *	2	2	4	2	4	6	8	2	4

Copying the data cost only 1, the idea is to store more than only one, so that **every time that we need to spend more for reallocation we always have enough money in the bank account so we never go red**.
Il vantaggio del metodo del conto bancario è che offre un modo semplice per calcolare il costo ammortizzato delle operazioni, considerando sia i costi che i risparmi associati a operazioni precedenti. Questo fornisce una stima più accurata delle prestazioni complessive di una struttura dati o di un algoritmo rispetto all'analisi dei casi peggiori o migliori
Any amount not immediately consumed is stored in the bank for use by subsequent operations.
L'idea è che devo andare ad accumulare nel mio saldo tanto denaro quanto me ne serve poi per potermi creare una nuova data struttura che sia il doppio più grande di quella precedente. Quindi per tutte le operazioni di insertion che non mi portano overflow io vado a risparmiare del denaro, che poi andrò a spendere quando devo passare a una struttura dati più grande.
Nel nostro caso la prima insertion costa sempre solo 2, poi tutte le altre 3 (di modo tale che 1 lo spendo per insert e 2 li risparmio e ogni volta che vado a una potenza di due spendo quello che ho risparmiato, di fatti al valore successivo della potenza ho un +2 perchè avevo 0 e ho risparmiato +2 dalla nuova insertion)
Same results of the previous solution, but in a different way (here you need to keep the amortized cost minimum using the bank account)

Potential method
View the bank account as potential energy.

1. Start with an initial data structure D0 .
2. Operation i transforms Di–1 to Di .
3. The cost of operation i is ci .
4. Define a potential function F : {Di } → R, such that F(D0) = 0 and F(Di) >= 0 for all i.
5. The amortized cost ĉi with respect to F is defined to be ĉi = ci + F(Di) – F(Di–1)
   Differential of Fi = F(Di) – F(Di–1)
   - If Differential of Fi > 0 Storing energy for later use
   - If Differential of Fi < 0 Using Energy stored

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$
$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$
$$\geq \sum_{i=1}^{n} c_i$$

since $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$.

The formula describes how much it is gonna change
Define the potential of the table after the ith insertion by F(Di) = 2i – 2 $^{[\lg i]}$. (Assume that 2 $^{[\lg 0]}$ = 0)

The amortized cost of the i th insertion is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise;} \end{cases}$$
$$+ \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right)$$
$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise;} \end{cases}$$
$$+ 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} .$$

First part is related to the ci, the second one instead is related to the difference F(Di) – F(Di–1).

Amortized costs can provide a clean abstraction of data-structure performance.

Any of the analysis methods can be used when an amortized analysis is called for, but each method has some situations where it is arguably the simplest or most precise (depends on if we are considering a situation that is more linked to money, energy or whatever).

Case 1: $i-1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1) \\ &= i + 2 - 2i + 2 + i - 1 \\ &= 3\end{aligned}$$

Case 2: $i-1$ is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= 3\end{aligned}$$

Therefore, n insertions cost $\Theta(n)$ in the worst case.

Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.

COMPETITIVE ANALYSIS

Here we are considering a data structure that is storing data and **it changes its structure such that the next search is going to cost the least possible**.

List L of n elements

- The operation **ACCESS(x) costs** rankL (x) = **distance of x from the head of L**.
- L can be reordered by transposing adjacent elements at a cost of 1.

Self organizing is related to the fact that we do not know the access pattern (searching depends on something that is not in the control of the programmer). We would like to keep the most frequent element at the beginning of the list

A sequence S of operations is provided one at a time. For each operation,

- an **on-line algorithm** A must execute the operation immediately without any knowledge of future operations (e.g Tetris, Falling objects and according to the windows that you have to try to position them in the best possible way, but you are not doing it optimally, because you do not know the future moves. You do the analysis only by looking at the elements falling now).
- an **off-line algorithm** may see the whole sequence S in advance. (all algorithms made by now can be considered off line algorithms, were you know in advance everything)

The problem is how to analyze complexity of online algorithms?

Worst case analysis is not working very well. We are searching always the last item in all the operations, so the complexity is the number of operations times the number of the list (this works for any possible algorithm that we are considering, there is not too much difference between an algo and another)

Heuristic: Keep a count of the number of times each element is accessed, and maintain L in order of decreasing count.

The move to front heuristics works well, but in order to use it we need to compare it with the best possible heuristics.

We need to compare the complexity of the online algorithm with the offline version.(OPT is the optimal offline algorithm). An on-line algorithm A is **alpha-competitive** if there exists a constant k such that for any sequence S of operations, $C_A(S) <= \text{alpha } C_{OPT}(S) + k$.

This is the competitive analysis, in order to understand how competitive is the proposed algorithm with the best possible algorithm that could have been built.

Move to front is O(1) competitive (move the items that are most searched on top of the array)

Theorem: **MTF costs 4 times the optimal one**.

Proof: First we define the cost of the MTF heuristics. Let Li be MTF's list after the ith access, and let Li * be OPT's list after the ith access.

- ci = MTF's cost for the ith operation = 2 $\text{rank}_{Li-1}(x)$ if it accesses x (one for the access and one for the swap to the front position)
- ci * = OPT's cost for the ith operation = $\text{rank}_{Li-1*}(x) + t_i$, (position the element we are searching for inside the optimal list + t_i which is the number of transposes that OPT performs)

Now we look at the amortized analysis to know the average cost with a sequence of operations. This is the case because we would like to know the cost of the sequence of those search operations in self organized lists.

We would like to compare the amortized cost of the offline algorithm with the amortized cost with the god algorithm (competitive analysis on amortized costs).

We use in order to compute the amortized costs one of the methods: the potential method.

Define the potential function F:{Li } → R by F(Li) = 2 |{(x, y) : x $<_{Li}$ y and y $<_{Li}$ * x}| = 2 # inversions. We look of the two lists, the Li* the one optimal with the Li with the move to front heuristics, we define the potential function to know how much we are near between the two lists. If x stays before y in the Li and it is in reversed order in the optimal one we are going to have an inversion. In summary the potential function is 2*inversions between the two lists after i-th searches on the list.

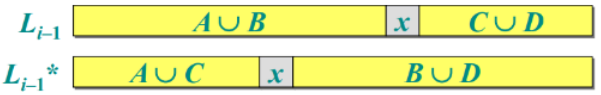
Note that

- F(Li) >= 0 always positive with any possible value of i
- F(L0) = 0 if MTF and OPT start with the same list.

How much does F change from 1 transpose? A transposition (take an item and swap with the previous one) creates/destroys 1 inversion. DeltaF = 2

Suppose that operation i accesses element x, and define 4 sets over the two lists for all the possible combinations

- A describes all the item in list L before the transformation that stays **before x in both lists**
- B describes all the items that stay **before in the heuristics x** and **after x in the optimal**
- C describes all items that stay **after x in the heuristics** and **before x in the optimal**
- D describes all items that stay **after x in the heuristics** and **after x in the optimal**



When we perform the transformation we need to pay the value of r which is r = rankLi–1 (x) and r* = rankLi–1 * (x).

We have r = |A| + |B| + 1 and r* = |A| + |C| + 1.

When MTF moves x to the front, it creates |A| inversions (because A is before x in both lists, so we need to perform such inversions) and destroys |B| inversions (because it is only in Li-1 and not in the optimal one). Each transpose by OPT creates 1 inversion (we think it's like this since we do not know what happens in the OPT). Thus, we have

$$F(L_i) - F(L_{i-1}) <= 2(|A| - |B| + t_i)$$

The amortized cost for the ith operation of MTF with respect to F is

ĉi = ci + F(Li) – F(Li–1) (actual cost + the differential of the potential function).

Then we perform those operations on the left, because

- r = |A| + |B| + 1
- r* = |A| + |C| + 1 >= |A| + 1

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(L_i) - \Phi(L_{i-1}) \\
&\le 2r + 2(|A| - |B| + t_i) \\
&= 2r + 2(|A| - (r - 1 - |A|) + t_i) \\
&= 2r + 4|A| - 2r + 2 + 2t_i \\
&= 4|A| + 2 + 2t_i \\
&\le 4(r^* + t_i) \\
&= 4c_i^*.
\end{aligned}$$

If moving x toward the front is free (like taking x from the list and putting it in front of the list directly), then the MRF is **2-competitive**.

What if the two lists are not equal? MTF is still less the 4 times of the OPT + something