2. Asymptotic Notation

Motivation: For a given algorithm, we want to quantify how the algorithm's running time grows as the input of size n grows.

Normally, we are interested in knowing the worst-case running time as function of n, but sometimes we may also be interested in knowing the average (expected) running time or the best-case running time.

Not unimportantly, we want to come up with a notion of running time which is independent of features such as processor speed etc.

Definition: In the following, a step in the algorithm will refer to assigning a value to a variable. An example of a step is looking up one entry in an array.

Goal: Given an algorithm in pseudo-code such as the Gale-Shapley algorithm, specify the running time (in steps) as function of the input size n.

As our pseudo-code provides a high-level description of the algorithm, a particular step in the pseudo-code may correspond to 25 low-level machine instructions when a particular implementation of the algorithm is compiled on a computer with a particular architecture.

- $g(n) = 1.62n^2 + 3.5n + 8$ is the number of steps required on pseudo-code level for an input of size n
- $\tilde{g}(n) = 40.5n^2 + 87.5n + 200$ may be the number of steps required by the algorithm on a particular piece of hardware for an input of size n

The goal is to measure the running time of an algorithm in a way that is independent of the particular hardware and a good reflection of the features of the algorithm on pseudo-code level, i.e. we want to capture the running time in a way that is

- insensitive to constant factors and
- lower order terms

For the above example $g(n)$ we would want to state that the running time grows like n^2, up to constant factors.

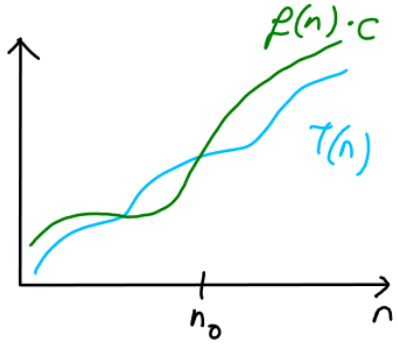
2.1 Asymptotic upper bounds O

Motivation: Given an algorithm in pseudp-code which requires at most $T(n)$ steps to complete for an input of size $n \in N, T : N \to \mathbb{R}_+$, find a function $f : N \to \mathbb{R}_+$ which, if multiplied by a constant, positive factor $c \in \mathbb{R}, c > 0$, provides an upper bound to

$T(n)$ for sufficiently large n, i.e. for all $n \in N, n \geq n_0 \in N$.
In other words, find a function $f : N \to \mathbb{R}_+$ such that

$$T(n) \leq c \cdot f(n), \forall n \in N, n \geq n_0, c \in \mathbb{R}_+$$

Note: c is a constant, i.e. does not depend on n.



Reminder def.

N := set of natural numbers = {1,2,3,...}

$N_0 := N \cup \{0\}$

$\mathbb{R}$:= set of real numbers

$\mathbb{R}_+$:= set of all non-negative real numbers $= \{x \in \Re, x \geq 0\}$

$\forall$ is short hand for "all"

$\exists$ is short hand for "exists"

Example: $T(n) = 7n^2 + 5$

Would $f(n) = n^3$ work, i.e. $T(n) = O(n^3)$?

Check: $T(n) = 7n^2 + 5 \leq 7n^2 + 5n^2 = 12n^2$ for $\forall n \in N, n \geq 1$
$\leq 12n^3$
$\Rightarrow T(n) \leq c \cdot f(n)$ for $\forall n \in N, n \geq n_0 = 1$ and $c = 12 \in \mathbb{R}_+$

Definition: Let $T(n), T : N \to \mathbb{R}_+$, be the function that describes the worst-case running time of a given algorithm in terms of steps to be completed per sitesize of input n. We say that $T(n)$ is $O(f(n))$ (read "$T(n)$ is of order $f(n)$") if T is asymptotically upper bounded by $f(n), f : N \to \mathbb{R}_+$, i.e. if there is a constant $c \in \mathbb{R}_+$ and $n_0 \in N$ such that $T(n) \leq c \cdot f(n)$ for $\forall n \in N, n \geq n_0$. If $T(n)$ is $O(f(n))$, we also write $T(n) = I(f(n))$.

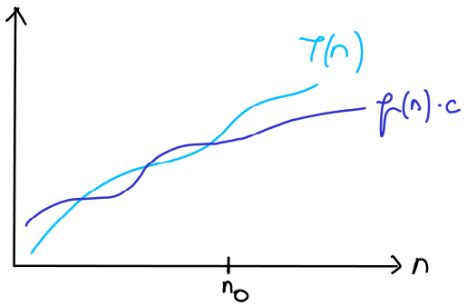
Note: $O(\cdot)$ only expresses an upper bound, budt does not necessarily provide a precise description of the worst-case running time, i.e. a tight upper bound.

example: $T(n) = 7n^2 + 5$ is $O(n^3)$, but also $O(n^4)$.

2.2 Asymptotic lower bounds Ω (greek "Omega")

Motivation: Given a function $T(n), T : N \to \mathbb{R}_+$, which describes the worst-case running time of a given algorithm, find a function $f(n), f : N \to \mathbb{R}_+$, which provides a lower bound to T.

Definition Let $T(n), T : N \to \mathbb{R}_+$, be a function that describes the worst-case running time of a given algorithm in terms of steps to be completed per size of input n.
We say that $T(n)$ is $\Omega(f(n))$ i.e. we say that ($T(n)$ is asymptotically lower bounded by $f(n), f : N \to \mathbb{R}_+$, i.e. if there is a constant $c \in \mathbb{R}_+$ and $n_0 \in N$ such that $T(n) \geq c \cdot f(n)$ for $\forall n \in N$, $n \geq n_0$. If $T(n)$ is $\Omega(f(n))$ we also write $T(n) = \Omega(f(n))$



Example: $T(n) = 7n^2 + 5$

Group work: Find a function $f(n)$ so $T(n) = \Omega(f(n))$.

Would $f(n) = n^2$ work, i.e. $T(n) = \Omega(f(n))$?
Check: $T(n) = 7n^2 + 5 \geq 7n^2$ for $\forall n \in N$
$\Rightarrow T(n) \geq c \cdot f(n), \forall n \in N, n \geq n_0 = 0, c = 7 \in \mathbb{R}_+$.

Notes:

- similar to $O(f(n))$, $\Omega(f(n))$ only express a lower bound to $f(n)$, which is not necessarily a tight bound.

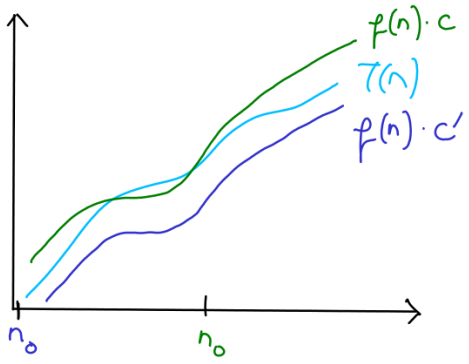
Example: $T(n) = \Omega(n^2)$, but also $T(n) = \Omega(n)$.

- Ω is most useful is used in conjunction with O, e.g. in order to give a tight bound on the running time of an algorithm.
- Ω is also useful for providing a lower bound on the worst-case running-time of all algorithms that address a specific problem as we can then, for example, show that the problem is difficult.

2.3 Asymptotically tight bounds Θ (greek "Theta")

Motivation: Often, for a given algorithm, we do not only want to know an upper and a lower bound to the worst-case running time of the algorithm, but we would like a tight bound which gives us a precise description of the algorithm's worst-case running time.

Definition: We say that $f(n)$ is an asymptotically tight bound for $T(n)$ or that $T(n)$ is $\Theta(f(n))$ if $T(n) = O(f(n)) = \Theta(f(n))$, where $T, f : N \to \mathbb{R}_+$.



Group work: Given $T : N \to \mathbb{R}_+, T(n) = 7n^2 + 5$, find a tight bound.

Answer: $T(n) = \Theta(n^2)$ because

(a) $T(n) = O(n^2)$:

Proof: $T(n) = 7n^2 + 5 \leq 7n^2 + 5n^2, \forall n \in N, n > 1$
$= 12n^2$

i.e. $T(n) \leq c \cdot f(n)$, where $f(n) = n^2$ and $c = 12 > 0$ and all $n \in N, n \geq n_0 = 1$.

(b) $T(n) = \Omega(n^2)$:

Proof: $T(n) = 7n^2 + 5 \geq n^2, \forall n \in N$

i.e. $T(n) \geq c \cdot f(n)$, where $f(n) = n^2$ and $c = 1 > 0$ for all $n \in N$.

□

Note: Asymptotically tight bounds on worst-case running times are very useful as they characterize the worst-case performance of an algorithm in a precise way up to constant factors.

2.4 Properties of asymptotic growth rates

Motivation: one strategy for deriving an asymptotically tight bound is to compute the limit of n to infinity for the function $f(n)$ which describes the worst-case running time of a given algorithm.

Theorem: Let $f, g : N \to \mathbb{R}_+$ be two functions for which

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}_+$$

Then $f(n) = \Theta(g(n))$.

Interpretation: If the ration of two functions f and g converges to a positive constant for infinitely large values of n, then f has an asymptotically tight bound g.

Group work: How do you go about proving this theorem?

Proof: Use $f(n) = \Theta(g(n)) \Leftrightarrow (f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0 \Rightarrow \exists n_0 \in N$ so that $\frac{1}{2} \cdot c \leq \frac{f(n)}{g(n)} \leq 2c$.

This implies (a) $f(n) \leq 2cg(n), \forall n \in N, n \geq n_0$, i.e. $f(n) = O(g(n))$,

and (b) $\frac{c}{2} g(n) \leq f(n), \forall n \in N, n \geq n_0$, i.e. $f(n) = \Omega(g(n))$.

Note:

(a) $f(n) \leq 2cg(n) \Leftrightarrow \frac{1}{2c} f(n) \leq g(n), \forall n \in N, n \geq n_0$, i.e. $g(n) = \Omega(f(n))$

(b) $\frac{c}{2} g(n) \leq f(n), \Leftrightarrow g(n) \leq \frac{2}{c} f(n), \forall n \in N, n \geq n_0$, i.e. $g(n) = O(f(n))$

$\Rightarrow f(n) = \Theta(g(n))$ and $g(n) = \Theta(f(n))$.

Conclusion: f not only has as asymptotically tight bound g, but g also has f as asymptotically tight bound.

Assumption: For the rest of this section, assume that f, g and h are functions $N \to \mathbb{R}_+$.

Lemma: Transitivity of O, Ω and Θ If a function f is asymptotically upper-bounded by a function g and g is asymptotically upper-bounded by a function h, then f is asymptotically upper-bounded by h.
Likewise for lower bounds and tight bounds.
In other words:

(a) $f = O(g)$ and $g = O(h) \Rightarrow f = O(h)$

(b) $f = \Omega(g)$ and $g = \Omega(h) \Rightarrow f = \Omega(h)$

(c) $f = \Theta(g)$ and $g = \Theta(h) \Rightarrow f = \Theta(h)$

Proof:

(a) Group work

$$f = O(g) \text{ i.e. } f(n) \leq^{(*)} c_g \cdot g(n),\ c_g \in \mathbb{R}_+, \forall n \geq n_g \text{ and}$$
$$g = O(h) \text{ i.e. } g(n) \leq^{(\Delta)} c_h \cdot h(n),\ c_h \in \mathbb{R}_+, \forall n \geq n_h.$$

We can combine this as follows

$$f(n) \leq^{(*)} c_g \cdot g(n) \leq^{(\Delta)} c_g \cdot c_h \cdot h(n), \forall n \geq \max\{n_g, n_h\}$$

Which implies $f = O(h)$. □

(b) similar proof to (a)

(c) From $f = O(g)$ and $g = \Theta(h)$ follows with the definition of Θ and (a) that $f = O(h)$. It also follows with (b) that $f = \Omega(h)$. Again using the definition of Θ, we can summarize both as $g = \Theta(g)$. □

Lemma: (Additivity of O) $f = O(h)$ and $g = O(h) \Rightarrow f + g = O(h)$

Proof: (Group work) We have $f = O(h)$, i.e. $f(n) \leq c_f \cdot h(n), \forall n \geq n_f, c_f \in \mathbb{R}_+$

and likewise we have $g(n) \leq c_g h(n), \forall n \geq n_g, c_g \in \mathbb{R}_+$.

We can combine this into

$$f(n) + g(n) \leq (c_f + c_g)h(n), \forall n \geq \max\{n_f, n_g\}, \widetilde{c} = (c_f + c_g) \in \mathbb{R}_+$$

□

Lemma: (Additivity of O) Given $k \in N$ and k functions $f_i : N \to \mathbb{R}_+, i \in \{1, 2, ..., k\}$ with $f_i = O(n)$, then

$$\sum_{i=1}^{k} f_i = O(n).$$

Reminder: Σ denotes the summation sign. Example: $\sum_{i=1}^{k} f_i = f_1 + f_2 + ... + f_k$.

Proof: Extension of proof for previous lemma. Omitted here.

Lemma: Given $g = O(f)$, then $f + g = O(f)$.

Group work: Prove the lemma.

Proof: $g = O(f)$ implies that $g(n) \leq c \cdot g(n), \forall n \geq n_0, c \in \mathbb{R}_+$.

This means that $f(n) + g(n) \leq f(n) + c \cdot f(n) = (1+c)f(n), \forall n \geq n_0, \widetilde{c} = (c+1) \in \mathbb{R}_+$.

This implies $f + g = O(f)$.

We also have $f + g = \Omega(f)$, because for $\forall n \in N$, we have $f(n) + g(n) \geq f(n)$.

As $f + g = O(f)$, and $f + g = \Omega(f)$, we thus obtain $f + g = \Theta(f)$.

2.5 Asymptotic bounds for some common functions

Reminder:

- A polynomial function F is a function $F : \mathbb{R} \to \mathbb{R}$, where: $F(x) := \sum_{i=0}^{n} a_i * x^i$, where all $a_i \in R$ and a_n does not equal to 0 and $n \in$ degree of the polynomial.

Examples:

- F(x) $= 3x^2 + 5x^3, F(x) = x/2$

Lemma:

- Let F be a polynomial of degree d, then F $= O(n^d)$

Proof of lemma:

- $F(n) = \sum_{i=0}^{d} a_i * n^i$, for each term $a_i * n^i$ can be viewed as a separate function F_i. Then $a_i * n^i \leq |a_i| * n^i, \forall n \in \mathbb{N}$, c $= |a_i| \in R+$, i.e. $F_i = O(n^1)$. For every function F_i we can also find a $n_i \in \mathbb{N}$ such that $F_i = a_i * n^i \leq |a_i| * n^i \leq |a_d| * n^d, \forall n \geq n_i$; $c_i = |a_d| \in R+$, i.e. $F_i \in O(n^d)$. Based on the additivity of O, we can thus conclude that F $= O(n^d)$.

Conclusion:

- The asymptotic growth rate of polynomial is determined by their highest-order term.

Lemma:

- Let F be a polynomial of degree d with $a_d > 0$, then F*$\Omega(n^d) = \Theta(n^d)$.

Proof:

- omitted here (idea: apply first theorem in 2.4 to g=$a_d * n^d$ and f as above.)

Lemma:

- For every $b, r \in \mathbb{R}, b > 1$ and $r > 0$, we have $\log_b(n) = O(n^r)$

Reminder:

- $\log_b(n) = x \in R$ i.e. x is the solution to $n = b^x$.
- $\log_b(n) = (\log_k(n))/(\log_k(b))$, for k>1, i.e. we can switch from the base b to another base k simply by a multiplying by a constant factor that dose not depend on n
- $\log_e(n) \leq n$ for all $n \in \mathbb{N}$, $n \geq 1$.

Proof:

- omitted.

Conclusion:

- The logarithm (for any base b > 1) grows much slower than any power function n^r with $r > 0$, $r \in \mathbb{R}$.

Reminder:

- An exponential or exponential function is a function of the form F(x) = r^x, $x \in \mathbb{R}$, $r \in \mathbb{R}$, $r > 1$.

Lemma:

- $n^d = O(r^n)$ for every $n \in \mathbb{N}$, $r, d \in \mathbb{R}$, $r > 1$ and $d > 0$.

Proof:

- omitted.

Conclusion:

- The exponential function r^n, $r > 1$, grows faster than any polynomial function.

Warning:

- Two logarithm functions $log_b(n)$ and $log_k(n)$ differs only by a constant factor, but for different bases $b > k > 1$, the two corresponding exponential functions b^n and k^n don't! We thus have $b^n \neq \Theta(k^n)$.

Conclusion:

- When dealing with an exponential function, we have to explicitly specify its level. It is inaccurate to say "the running time of this algorithm grows exponentially."

2.6 A survey of common running times

2.6.1 Algorithm that require O(n) i.e. linear time

Example: Merging two already sorted lists into one sorted list.

given: two sorted lists A = $(a_1, a_2, \ldots, a_n)$ and B = $(b_1, b_2, \ldots, b_n)$

goal: derive sorted list C = $(c_1, c_2, \ldots, c_{2n})$ so that the entries are arranged in the same (say ascending) order

Group Work: idea for an efficient algorithm

Algorithm: 1: for each list A and B, have a pointer which points to the next element in the list.

loop starts: while we haven't reached the end of any list

let a_i and b_j be the next element pointed to

append min$\{a_i,b_j\}$ to list C and advance corresponding pointers in list from which the element derives.

loop ends: append rest of list whose end has not yet been reached to list C.

Proof that the algorithm is of order O(n): group work - show the above algorithm

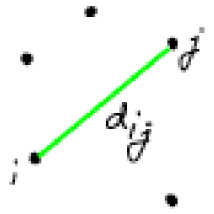
- It is clear that the algorithm is O(n^2) as every element in one list is compared to at most all elements in the other list. It is possible to come up with an even tighter asymptotic upper bound as follows:
- There are 2n elements in the merged list. Once each of these elements is put into list C, it is never again considered in the algorithm. The algorithm therefore requires 2n iterations at most to complete and every iteration results in exactly one element (either from A or B) to be inserted into list C. On average, each of 2 elements is considered twice before it ends up in list C. ⇒ The worst-case running time of the algorithm is of order O(n), i.e. requires linear time.

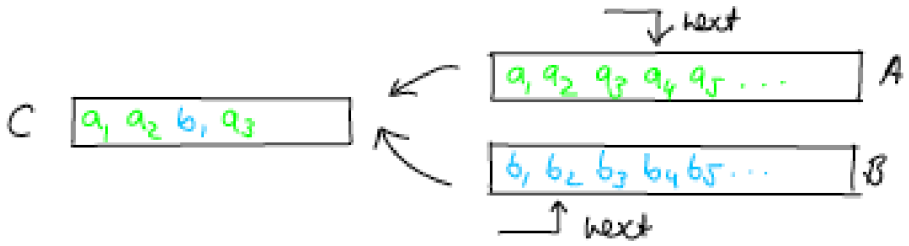
2.6.2 Algorithm that require $O(n^2)$ i.e. quadratic time

Example: Given a set of n points $(x_i, y_i) \in R^2$, i $\in$ 1, ... ,n find the closest euclidean distance d between two points (x_i, y_i) and (x_j, y_j) and $i \neq j$.

d_{ij} = d((x_i, y_i),(x_j, y_j)) = $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \geq 0$

Observation: for n points, there are $\frac{n*(n-1)}{2}$ unsorted pairs to consider as $d_{ij} = d_{ji}$





Algorithm :

for each input point (x_i, y_i) {
  for each input point (x_j, y_j) with $i \neq j$ {
    compute $d_{i,j}$
    if $d_{i,j}$ us smaller than current minimum distance d, update d
  }
}

The worst-case running time of this algorithm is $O(n^2)$ as we have two nested loops which each require $O(n)$ time.

remark : the task can also be computed more efficiently in $O(n \log(n))$ time

2.6.3 Algorithms that require $O(n^3)$ i.e. cubic time

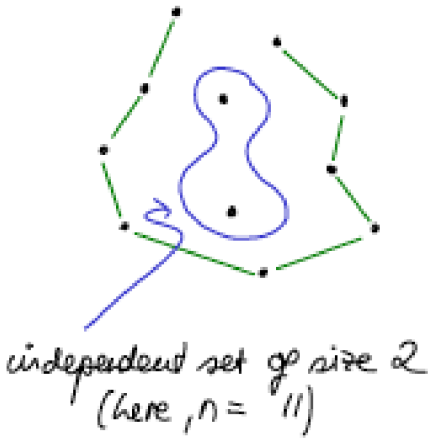
Example: Given a subset of $N_n = \{1, 2, 3, \ldots, n\}$, i $\in \mathbb{N}$, find out if there is a pair of subsets S_i and S_j, $i \neq j$, that is disjoint, i.e $S_i \cap S_j = \emptyset$

Algorithm :

for each set S_i {
  for each other set S_j with $i \neq j$ {
    for each element f $\in S_i$ {
      determine if f $\in S_j$ [assume we can do this in constant time]
    }
    if no element of S_i finds in S_j then they are disjoint
  }
}

The worst case running time of this algorithm is O(n^3) as we have three nested loops which require O(n) time each.

2.6.4 Algorithm that require $O(n^k)$ time



Example: Given a graph with n nodes, check if the graph contains an independent set of size k $\leq$ n, i.e. a set of k nodes where there are no edges between any pair of nodes.

Algorithm :

for each subset S of k nodes {
    check if S is an independent set
    if S is an independent set {
        stop and declare success
        }
    }
    if no k-node independent set was found {
        declare failure
    }
}

for $k \in \mathbb{N}$, $k \leq n$, there are the following number of subsets of S of size k to consider in the for-loop:

$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n*(n-1)*...*1}{(k*(k-1)*...*1)((n-k)*(n-k-1)*...*1)} = \frac{n*(n-1)*...*(n-k+1)}{(k!)} \leq \frac{n^k}{k!}$

As we regard k as a fixed parameter, time is is $O(n^k)$

inside the for - loop, each k-node subset is tested for independence. This requires looking at $\binom{k}{2}$ pairs which is O(k^2). As we regard k as a constant, the overall worst case running time can be written as O(n^k).

remark : This problem is believed to be computationally hard as it is thought that there is no algorithm to find k-node independent sets in arbitrary graphs with a running time which does not have a dependence on k in the exponent.

2.6.5 Algorithm that require beyond polynomial time

Example: Similar to 2.6.4, we are given a graph of n nodes. The task now, however, is to identify an independent k-node set of maximize k ∈ N, k≤n.

Algorithm :

for each subset S of nodes {
decide if S constitutes an independent set
if S is independent and larger than any independent set found so far {
record the size of S as the current maximum
}
}

On the contrast to the previous algorithm, this algorithm requires considering all subsets of S. This number is:

$$\sum_{k=0}^{n} \binom{n}{k} = \sum_{k=0}^{n} \binom{n}{k} 1^k * 1^{n-k} = 2^n$$

reminder : $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k}\ a^k * b^{n-k}$, $a, b \in \mathbb{R}$, $n \in \mathbb{N}$

The for-loop is thus of order $O(2^n)$. Inside the for-loop, we need to check if the current subset S is independent. As each subset has at most n elements, this requires at most $O(n^2)$ time.

The overall worst-case running time of this algorithm is therefore $O(2^n * n^2)$

Group work: Can you think of ways of making the above algorithm more efficient? Remember the definition of independent set. For example, if you have found an independent k-node sets, then all of its subsets are independent as well.

2.6.6 Algorithm that require less than linear time. e.g. O(log(n))

Example : Suppose we are given an array of $n \in \mathbb{N}$ already sorted numbers. The task is to find out if a number $p \in \mathbb{R}$ is stored in the array or not.

Group work: Can you think about an efficient algorithm for doing this?

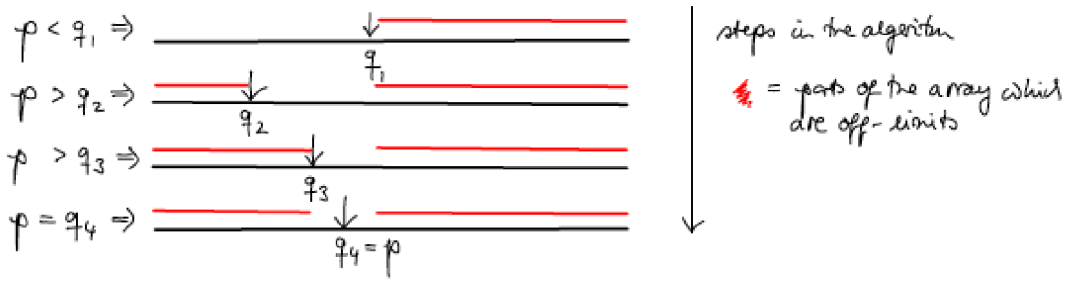
Algorithm : If p is smaller than the first array entry or larger than the last entry, it cannot be part of the array. Else, look at the middle entry q of the array. If $p = q$, we are done. If $p < q$, probe the remaining array to the left of q. If $p > q$, probe the remaining array to the right of q.

At every step in the algorithm, we are halving the remaining interval to be investigated. After k steps, we are left with an interval of size $(1/2)^k * n$

How many steps k do we need for this interval length to be constant $C \in \mathbb{R}$, $C > 0$?

i.e. want: $(1/2)^k * n \Leftrightarrow$ n = c*$2^k \Leftrightarrow \log_2(n)$ =$\log_2 c$+k $\Leftrightarrow$ k = $\log_2(n)$ - $\log_2 c$ choose c=1, then k = $\log_2(n)$.

We therefore require at most O(log(n)) steps until we can finish the task in constant time.

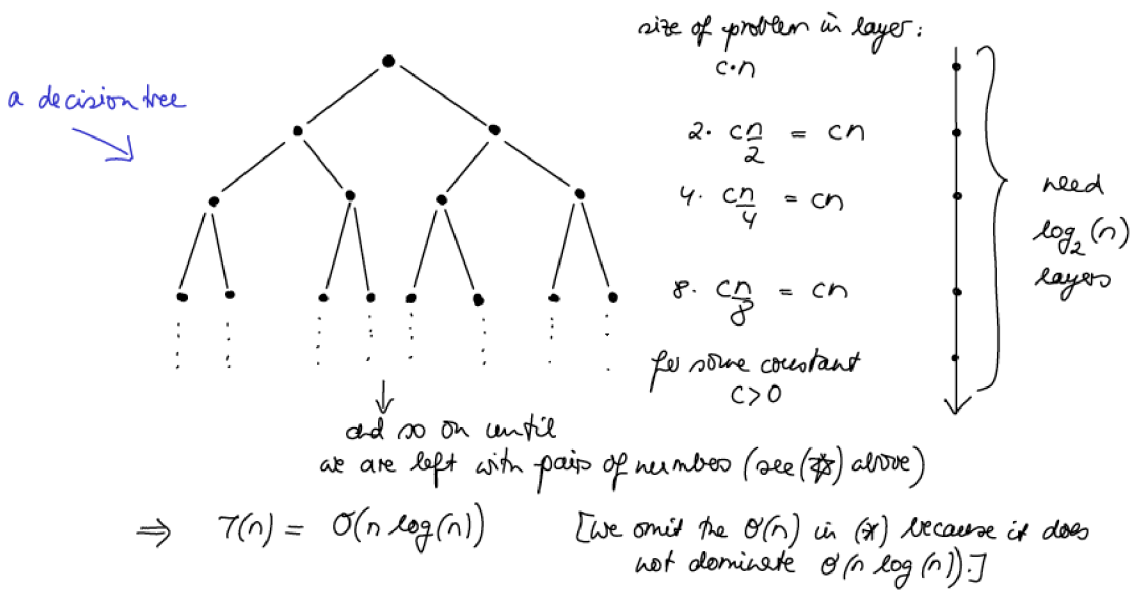


2.6.7 Algorithms that require $O(n \log(n))$ time

Example: Given a set S which contains an even number of numbers, divide the set into two disjoint subset S_1 and S_2 of same size, sort the numbers in each subset, do this recursively and finally combine them in the form of two sorted lists into a single sorted list (see 2.6.1 for how to do this efficiently)

- assume in the following that we stop the recursion when we are left with pairs of numbers
- some upfront consideration on the running time of the recursive algorithm
  - let T(n) here denote the worst-case running time of the algorithm for an input of size n
  - because the above procedure is recursive, we have
    (*) $T(n) \leq 2T(n/2) + O(n)$
    because the algorithm requires $O(n)$ time to split the initial set into two sets, $2T(n/2)$ time to run for the two resulting sets (S_1 and S_2) and $O(n)$ time to finally combine the final results of the algorithm S_1 and S_2 into a single list (see 2.6.1).
  - Inequelity (*) constitutes a **recurrence relation** because it describes $T(n)$ as function of itself. Note that for n=2, we have $T(2) \leq c, c > 0$
  - How do we solve (*)? In other words, how do we find an expression for $T(n)$ that does not express it as function of itself?

Idea: We can visualize the recursive algorithm as follows (figure 1):



$\Rightarrow \quad T(n) = O(n \log(n))$ [We omit the $O(n)$ in (*) because it does not dominate $O(n \log(n))$.]