4.4 Algorithm Design Technique: Randomization

Randomization is one of the design techniques of algorithms.

- A pathological occurence of the worst-case inputs can be avoided with it.
- The best-case and the worst-case running-times don't usually change, but their likelyhood in practise decreases.
- Disadvantageous inputs are exactly as likely as any other inputs regardless of the original distribution of the inputs.
- The input can be randomized either by randomizing it before running the algorithm or by embedding the randomization into the algorithm.
  - the latter approach usually gives better results
  - often it is also easier than preprocessing the input.

- Randomization is usually a good idea when
  - the algorithm can continue its execution in several ways
  - it is difficult to see which way is a good one
  - most of the ways are good
  - a few bad guesses among the good ones don't make much damage
- For example, QUICKSORT can choose any element in the array as the pivot
  - besides the almost smallest and the almost largest elements, all other elements are a good choise
  - it is difficult to guess when making the selection whether the element is almost the smallest/largest
  - a few bad guesses now and then doesn't ruin the efficiency of QUICKSORT

  $\Rightarrow$ randomization can be used with QUICKSORT

With randomization an algorithm RANDOMIZED-QUICKSORT which uses a randomized PARTITION can be written

- $A[r]$ is not always chosen as the pivot. Instead, a random element from the entire subarray is selected as the pivot
- In order to keep PARTITION correct, the pivot is still placed in the index r in the array

  $\Rightarrow$ Now the partition is quite likely even regardless of the input and how the array has earlier been processed.

```
RANDOMIZED-PARTITION(A, left, right)
1  i := RANDOM(left, right)              (choose a random element as pivot)
2  exchange A[right] ↔ A[i]              (store it as the last element)
3  return PARTITION(A, left, right)      (call the normal partition)
```

```
RANDOMIZED-QUICKSORT(A, left, right)
1  if left < right then
2       pivot := RANDOMIZED-PARTITION(A, left, right)
3       RANDOMIZED-QUICKSORT(A, left, pivot − 1)
4       RANDOMIZED-QUICKSORT(A, pivot + 1, right)
```

The running-time of RANDOMIZED-QUICKSORT is $\Theta(n \lg n)$ on average just like with normal QUICKSORT.

- However, the assumption made in analyzing the average-case running-time that the pivot-element is the smallest, the second smallest etc. element in the subarray with the same likelyhood holds for RANDOMIZED-QUICKSORT for sure.
- This holds for the normal QUICKSORT only if the data is evenly distributed.

$\Rightarrow$ RANDOMIZED-QUICKSORT is better than the normal QUICKSORT in general

QUICKSORT can be made more efficient with other methods:

- An algorithm efficient with small inputs (e.g.INSERTIONSORT) can be used to sort the subarrays.
  - they can also be left unsorted and in the end sort the entire array with INSERTIONSORT
- The median of three randomly selected elements can be used as the pivot.
- It's always possible to use the median as the pivot.

The median can be found efficiently with the so called lazy QUICKSORT.

- Divide the array into a "small elements" lower half and a "large elements" upper half like in QUICKSORT.
- Calculate which half the ith element belongs to and continue recursively from there.
- The other half does not need to be processed further.

```
RANDOMIZED-SELECT(A, left, right, goal)
1  if left = right then                          (if the subarray is of size 1...)
2      return A[left]                            (... return the only element)
3  pv := RANDOMIZED-PARTITION(A, left, right)    (divide the array into two halves)
4  k := pv − left + 1                            (calculate the number of the pivot)
5  if k = goal then                              (if the pivot is the goalth element...)
6      return A[pv]                              (...return it)
7  else if goal < k then                         (continue the search from the small ones)
8      return RANDOMIZED-SELECT(A, left, pv − 1, goal)
9  else                                          (continue on the large ones)
10     return RANDOMIZED-SELECT(A, pv + 1, right, goal − k)
```

The lower-bound for the running-time of RANDOMIZED-SELECT:

- Again everything else is constant time except the call of RANDOMIZED-PARTITION and the recursive call.
- In the best-case the pivot selected by RANDOMIZED-PARTITION is the $goal$th element and the execution ends.
- RANDOMIZED-PARTITION is run once for the entire array.

$\Rightarrow$ The algorithm's best case running-time is $\Theta(n)$.

The upper-bound for the running-time of RANDOMIZED-SELECT:

- RANDOMIZED-PARTITION always ends up choosing the smallest or the largest element and the $goal$th element is left in the larger half.
- the amount of work is decreased only by one step on each level of recursion.

$\Rightarrow$ The worst case running-time of the algorithm is $\Theta(n^2)$.

The average-case running-time is however $\Theta(n)$.

The algorithm is found in STL under the name `nth_element`.

The algorithm can also be made to always work in linear time.

5 Complexity notations

This chapter discusses the notations used to describe the asymptotic behaviour of algorithms.

Θ is defined together with two other related useful notations O and Ω.

5.1 Asymptotic notations

The equation for the running time was simplified earlier significantly:

- only the highest order term was used
- its constant coefficient was left out

$\Rightarrow$ studying the behaviour of the algorithm as the size of its input increases to infinity

- i.e. the *asymptotic* efficiency of algorithms

$\Rightarrow$ usefull information **only with inputs larger than a certain limit**

- often the limit is rather low

  $\Rightarrow$ the algorithm fastest according to Θ- and other notations is the fastest also in practice, except on very small inputs

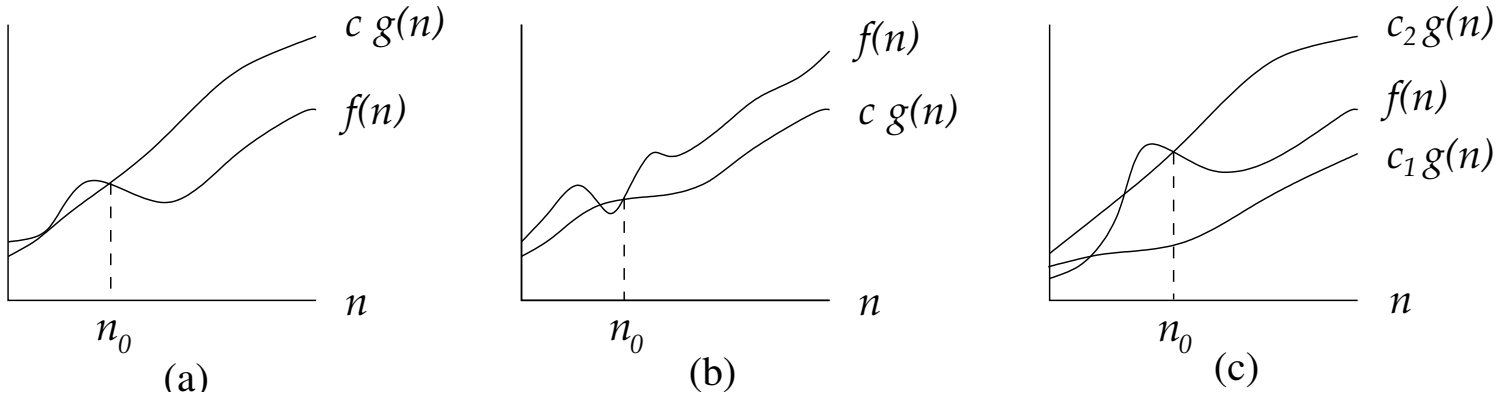
Θ-notation

- let $g(n)$ be a function from a set of numbers to a set of numbers

  $\Theta(g(n))$ is the set of those functions $f(n)$ for which there exists positive constants c_1, c_2 and n_0 so that for all $n \geq n_0$

  $$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

- the function in the picture (c) $f(x) = \Theta(g(n))$
  - $\Theta(g(n))$ is a *set* of functions
    $\Rightarrow$ we should, and earlier did, write $f(n) \in \Theta(g(n))$, but usually $=$ is used.



Whether a function $f(n)$ is $\Theta(g(n))$ can be proven by finding suitable values for the constants c_1, c_2 and n_0 and by showing that the function stays larger or equal to $c_1 g(n)$ and smaller or equal to $c_2 g(n)$ with values of n starting from n_0.

For example: $3n^2 + 5n - 20 = \Theta(n^2)$

- let's choose $c_1 = 3$, $c_2 = 4$ ja $n_0 = 4$
- $0 \leq 3n^2 \leq 3n^2 + 5n - 20 \leq 4n^2$ when $n \geq 4$, since $0 \leq 5n - 20 \leq n^2$
- just as well we could have chosen $c_1 = 2$, $c_2 = 6$ and $n_0 = 7$ or $c_1 = 0{,}000\,1$, $c_2 = 1\,000$ and $n_0 = 1\,000$
- what counts is being able to choose *some* positive c_1, c_2 and n_0 that fullfill the requirements

An important result: if $a_k > 0$, then

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0 = \Theta(n^k)$$

- in other words, if the coefficient of the highest-order term of a polynomial is positive the Θ-notation allows us to ignore all lower-order terms and the coefficient.

The following holds for constant functions $c = \Theta(n^0) = \Theta(1)$

- $\Theta(1)$ doesn't indicate which variable is used in the analysis
  $\Rightarrow$ it can only be used if the variable is clear from the context

O-notation (pronounced "big-oh")

The O-notation is otherwise like the Θ-notation, but it bounds the function only from above.
$\Rightarrow$ *asymptotic upper bound*

Definition:

> $O(g(n))$ *is the set of functions* $f(n)$ *for which there exists positive constants* c *and* n_0 *such that, for all* $n \geq n_0$
>
> $$0 \leq f(n) \leq c \cdot g(n)$$

- the function in the picture (a) $f(x) = O(g(n))$
- it holds: if $f(n) = \Theta(g(n))$, then $f(n) = O(g(n))$
- the opposite doesn't always hold: $n^2 = O(n^3)$, but $n^2 \neq \Theta(n^3)$
- an important result: if $k \leq m$, then $n^k = O(n^m)$
- if the running time of the **slowest** case is $O(g(n))$, then the running time of **every** case is $O(g(n))$

The O-notation is important in practise as, for example, the running times quaranteed by the C++-standard are often given in it.

Often some upper bound can be given to the running time of the algorithm in the slowest possible case with the O-notation (and every case at the same time).

We are often interested in the upper bound only.

For example: INSERTION-SORT

line	efficiency
for j := 2 **to** $A.length$ **do**	$O(n)$
key := $A[j]$	$\cdot \quad O(1)$
i := $j-1$	$\cdot \quad O(1)$
while $i > 0$ and $A[i] > key$ **do**	$\cdot \quad O(n)$
$A[i+1]$:= $A[i]$	$\cdot \quad \cdot \quad O(1)$
i := $i-1$	$\cdot \quad \cdot \quad O(1)$
$A[i+1]$:= key	$\cdot \quad O(1)$

The worst case running time is $O(n) \cdot O(n) \cdot O(1) = O(n^2)$

Ω-notation (pronounced "big-omega")

The Ω-notation is otherwise like the Θ-notation but is bounds the function only from below.
$\Rightarrow$ *asymptotic lower bound*

Definition:

> $\Omega(g(n))$ *is the set of functions* $f(n)$ *for which there exist positive constants* c *and* n_0 *such that, for all* $n \geq n_0$
>
> $$0 \leq c \cdot g(n) \leq f(n)$$

- the function in the picture (b) function $f(x) = \Omega(g(n))$
- the following result follows from the definitions: $f(n) = \Theta(g(n))$ *if and only if* $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$.
- if the running time of the **fastest** case is $\Omega(g(n))$, the running time of **every** case is $\Omega(g(n))$

The Ω-notation is usefull in practise mostly in situations where even the best-case efficiency of the solution is unsatisfactory and the result can be rejected straightaway

Properties of asymptotic notations

$f(n) = \Omega(g(n))$ and $f(n) = O(g(n)) \iff f(n) = \Theta(g(n))$

Many of the relational properties of real numbers apply to asymptotic notations:

$$\begin{array}{ll} f(n) = O(g(n)) & a \leq b \\ f(n) = \Theta(g(n)) & a = b \\ f(n) = \Omega(g(n)) & a \geq b \end{array}$$

i.e. if the highest-order term of $f(n)$ whose constant coefficient has been removed $\leq g(n)$'s corresponding term, $f(n) = O(g(n))$ etc.

Note the difference: for any two real numbers exactly one of the following must hold: $a < b$, $a = b$ ja $a > b$. However, this does not hold for all asymptotic notations.
$\Rightarrow$ Not all functions are asymptotically comparable to eachother (e.g. n and $n^{1+\sin n}$).

An example simplifying things a little:

- If an algorithm is $\Omega(g(n))$, its consumption of resources is at least $g(n)$.
  - cmp. a book costs at least about 10 euros.
- If an algorithm is $O(g(n))$, its consumption of resources is at most $g(n)$.
  - cmp. a book costs at most 10 euros.
- If an algorithms is $\Theta(g(n))$, its consumption of resources is always $g(n)$.
  - cmp. a book costs about 10 euros

Note that the running time of all algorithms cannot be determined with the Θ-notation.

For example Insertion-Sort:

- the best-case is $\Omega(n)$, but not $\Omega(n^2)$
- the worst-case is $O(n^2)$, but not $O(n)$
  $\Rightarrow$ a Θ-value common to all cases cannot be determined.

An example

Let's take a function $f(n) = 3n^2 + 5n + 2$.

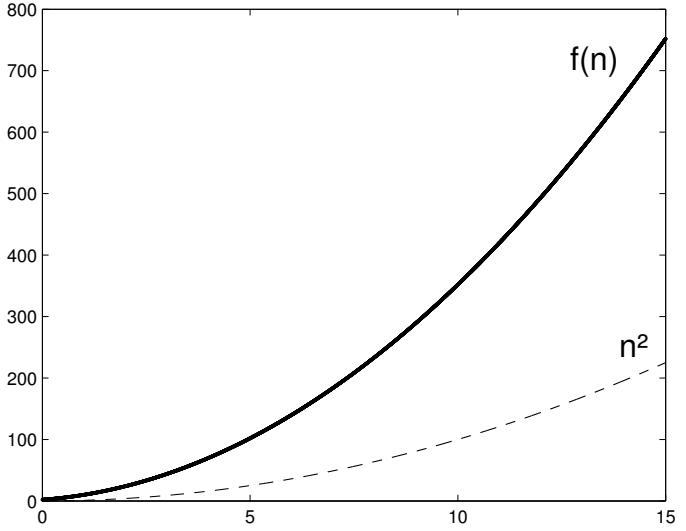
and simplify it according to the rules given earlier:

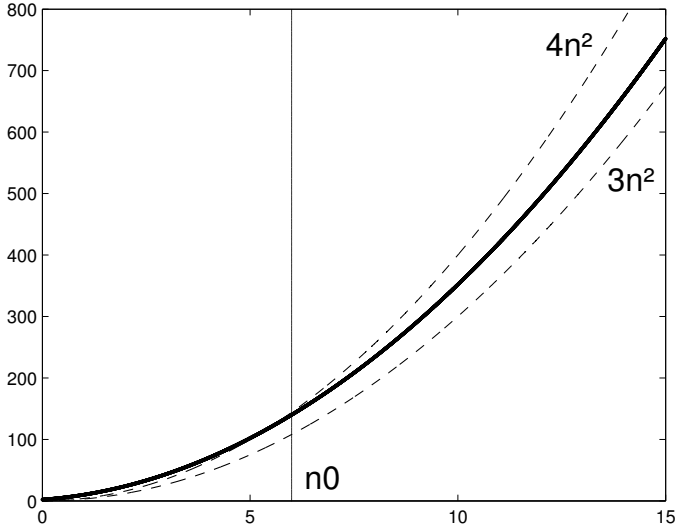
- lower-order terms ignored
- constant coefficients ignored

$\Rightarrow f(n) = \Theta(n^2)$

To be completely convinced we'll determine the constants c_1 ja c_2:

$3n^2 \leq 3n^2 + 5n + 2 \leq 4n^2$, when $n \geq 6$
$\Rightarrow c_1 = 3$, $c_2 = 4$ and $n_0 = 6$ work correctly
$\Rightarrow f(n) = O(n^2)$ and $\Omega(n^2)$
$\Rightarrow f(n) = \Theta(n^2)$



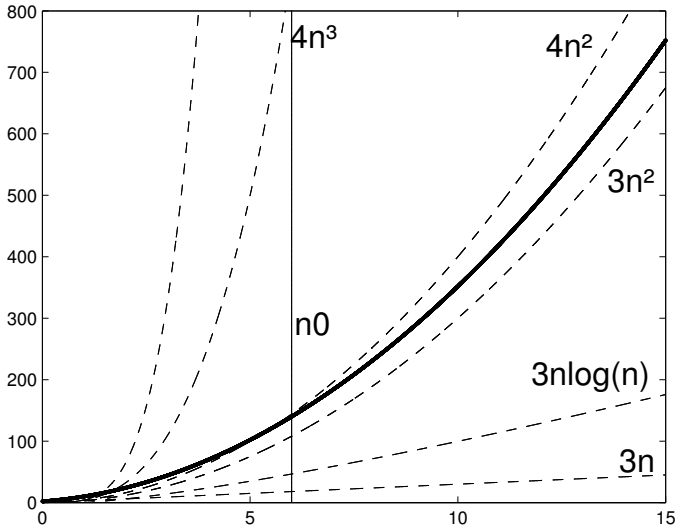


Clearly the constant $c_2 = 4$ works also when $g(n) = n^3$, since when $n \geq 6, n^3 > n^2$
$\Rightarrow f(n) = O(n^3)$

- the same holds when $g(n) = n^4$...

And below the constant $c_1 = 3$ works also when $g(n) = n \lg n$, since when $n \geq 6$, $n^2 > n \lg n$
$\Rightarrow f(n) = \Omega(n \lg n)$

- the same holds when $g(n) = n$ or $g(n) = \lg n$



5.2 Concepts of efficiency

So far the efficiency of algorithms has been studied from the point of view of the running-time. However, there are other alternatives:

- We can measure the memory consumption or bandwidth usage

In addition, in practise at least the following need to be taken into account:

- The unit used to measure the consumption of resources
- The definition of the size of the input
- Whether we're measuring the best-case, worst-case or average-case resource consumption
- What kind of input sizes come in question
- Are the asymptotic notations precise enough or do we need more detailed information

Units of the running-time

When choosing the unit of the running-time, the "step", usually a solution as independent of the machine architecture as possible is aimed for.

- Real units like a second cannot be used
- The constant coefficients become unimportant
  $\Rightarrow$ We're left with the precision of the asymptotic notations.

$\Rightarrow$ any constant time operation can be used as the step

- Any operation whose time consumption is independent of the size of the input can be seen as constant time.
- This means that the execution time of the operation never exceeds a given time, independent of the size of the input
- Assignment of a single variable, testing the condition of a **if**-statement etc. are all examples of a single step.
- There is no need to be too precise with defining the step as $\Theta(1) + \Theta(1) = \Theta(1)$.

Units of memory use

A bit, a byte (8bits) and a word (if its length is known) are almost always exact units

The memory use of different types is often known, although it varies a little between different computers and languages.

- an integer is usually 16 or 32 bits
- a character is usually 1 byte = 8 bits
- a pointer is usually 4 bytes = 32 bits
- an array $A[1 \ldots n]$ is often $n \cdot$ <element size>

$\Rightarrow$ Estimating the exact memory use is often possible, although requires precision.

Asymptotic notations are usefull when calculating the exact number of bytes is not worth the effort.

If the algorithm stores the entire input simultaneously, it makes sense to separate the amount of memory used for the input from the rest of the memory consumption.

- $\Theta(1)$ extra memory vs. $\Theta(n)$ extra memory
- It's worth pointing out that searching for a character string from an input file doesn't store the entire input but scans through it.

5.3 Binary Search

An efficient search in sorted data

Works by comparing a search key to the middle element in the data

- divide the search area into two
- choose the half where the key must be, ignore the other
- continue until only one element left
  - it is the key
  - the element is not in the data set

```
BIN-SEARCH(A, 1, n, key)        ▷ requirement: n ≥ 1, the array is sorted
1   low := 1; hi := n           (initialize the search to cover the entire array)
2   while low < hi do           (search until there are no more elements to cover)
3       mid := ⌊(low + hi)/2⌋   (divide the search area in half)
4       if key ≤ A[mid] then    (If the key is in the bottom half...)
5           hi := mid           (...choose the bottom half as the new search area)
6       else                    (else...)
7           low := mid + 1      (...choose the upper half as the search area)
8   if A[low] = key then
9       return low              (the element is found)
10  else
11      return 0                (the element is not found)
```

- The running time of BIN-SEARCH is $\Theta(\lg n)$.