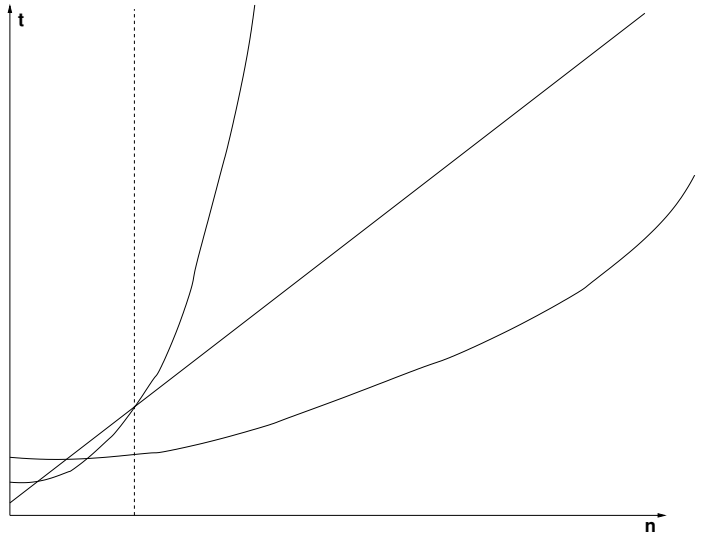
4 Measuring efficiency

This chapter dicusses the analysis of algorithms: the efficiency of algorithms and the notations used to describe the *asymptotic* behavior of an algorithm.

In addition the chapter introduces two algorithm design techniques: *decrease and conquer* and *divide and conquer*.



4.1 Asymptotic notations

It is occasionally important to know the exact time it takes to perform a certain operation (in real time systems for example).

Most of the time it is enough to know how the running time of the algorithm changes as the input gets larger.

- The advantage: the calculations are not tied to a given processor, architecture or a programming language.
- In fact, the analysis is not tied to programming at all but can be used to describe the efficiency of any behaviour that consists of successive operations.

- The time efficiency analysis is simplified by assuming that all operations that are independent of the size of the input take the same amount of time to execute.
- Furthermore, the amount of times a certain operation is done is irrelevant as long as the amount is constant.
- We investigate how many times each row is executed during the execution of the algorithm and add the results together.

- The result is further simplified by removing any constant coefficients and lower-order terms.

  $\Rightarrow$ This can be done since as the input gets large enough the lower-order terms get insigficant when compared to the leading term.

  $\Rightarrow$ The approach naturally doesn't produce reliable results with small inputs. However, when the inputs are small, programs usually are efficient enough in any case.

- The final result is the efficiency of the algorithm and is denoted it with the greek alphabet theta, Θ.

$$f(n) = 23n^2 + 2n + 15 \Rightarrow f \in \Theta(n^2)$$

$$f(n) = \tfrac{1}{2} n \lg n + n \Rightarrow f \in \Theta(n \lg n)$$

Example 1: addition of the elements in an array

```
1   for i := 1 to A.length do
2       sum := sum + A[i]
```

- if the size of the array A is n, line 1 is executed n + 1 times
- line 2 is executed n times
- the running time increases as n gets larger:

n	time = $2n$ + 1
1	3
10	21
100	201
1000	2001
10000	20001

- notice how the value of n dominates the running time

- let's simplify the result as described earlier by taking away the constant coefficients and the lower-order terms:

  $f(n) = 2n + 1 \Rightarrow n$

  $\Rightarrow$ we get $f \in \Theta(n)$ as the result

  $\Rightarrow$ the running time depends *linearly* on the size of the input.

Example 2: searching from an unsorted array

```
1   for i := 1 to A.length do
2       if A[i] = x then
3           return i
```

- the location of the searched element in the array affects the running time.
- the running time depends now both on the size of the input and on the order of the elements
  $\Rightarrow$ we must separately handle the best-case, worst-case and average-case efficiencies.

- in the best case the element we're searching for is the first element in the array.

  $\Rightarrow$ the element is found in *constant time*, i.e. the efficiency is $\Theta(1)$
- in the worst case the element is the last element in the array or there are no matching elements.
- now line 1 gets executed n + 1 times and line 2 n times

  $\Rightarrow$ efficiency is $\Theta(n)$.
- determining the average-case efficiency is not as straightforward

- first we must make some assumptions on the average, typical inputs:
  - the probability p that the element is in the array is ($0 \leq p \leq 1$)
  - the probability of finding the first match in each position in the array is the same
- we can find out the average amount of comparisons by using the probabilities
- the probability that the element is not found is 1 - p, and we must make n comparisons
- the probability for the first match occuring at the index i, is p/n, and the amount of comparisons needed is i
- the number of comparisons is:

$$[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

- if we assume that the element is found in the array, i.e. p = 1, we get (n+1)/2 which is $\Theta(n)$

  $\Rightarrow$ since also the case where the element is not found in the array has linear efficiency we can be quite confident that the average efficiency is $\Theta(n)$

- it is important to keep in mind that all inputs are usually not as probable.

  $\Rightarrow$ each case needs to be investigated separately.

Example 3: finding the common element in two arrays

```
1   for i := 1 to A.length do
2       for j := 1 to B.length do
3           if A[i] = B[j] then
4               return A[i]
```

- line 1 is executed 1 – (n + 1) times
- line 2 is executed 1 – ($n \cdot$ (n + 1)) times
- line 3 is executed 1 – ($n \cdot n$) times
- line 4 is executed at most once

- the algorithm is fastest when the first element of both arrays is the same

  $\Rightarrow$ the best case efficiency is $\Theta(1)$
- in the worst case there are no common elements in the arrays or the last elements are the same

  $\Rightarrow$ the efficiency is $2n^2 + 2n + 1 = \Theta(n^2)$
- on average we can assume that both arrays need to be investigated approximately half way through.

  $\Rightarrow$ the efficiency is $\Theta(n^2)$ (or $\Theta(nm)$ if the arrays are of different lengths)

RETURN OF INSERTION-SORT

```
INSERTION-SORT(A)                          (input in array A)
1   for j := 2 to A.length do              (move the limit of the sorted range)
2       key := A[j]                        (handle the first unsorted element)
3       i := j − 1
4       while i > 0 and A[i] > key do      (find the correct location of the new element)
5           A[i + 1] := A[i]               (make room for the new element)
6           i := i − 1
7       A[i + 1] := key                    (set the new element to it's correct location)
```

- line 1 is executed n times
- lines 2 and 3 are executed n - 1 times
- line 4 is executed at least n - 1 and at most (2 + 3 + 4 + $\cdots$ + n - 2) times
- lines 5 and 6 are executed at least 0 and at most (1 + 2 + 3 + 4 + $\cdots$ + n - 3) times

- in the best case the entire array is already sorted and the running time of the entire algorithm is at least $\Theta(n)$
- in the worst case the array is in a reversed order. $\Theta(n^2)$ time is used
- once again determining the average case is more difficult:
- let's assume that out of randomly selected element pairs half is in an incorrect order in the array

  $\Rightarrow$ the amount of comparisons needed is half the amount of the worst case where all the element pairs were in an incorrect order

  $\Rightarrow$ the average-case running time is the worst-case running time divided by two: ((n - 1)n)/ 4 = $\Theta(n^2)$

4.2 Algorithm Design Technique: Divide and Conquer

We've earlier seen the *decrease and conquer* algorithm design technique and the algorithm INSERTION-SORT as an example of it.

Now another technique called *divide and conquer* is introduced. It is often more efficient than the decrease and conquer approach.

- the problem is divided into several subproblems that are like the original but smaller in size.
- small subproblems are solved straightforwardly
- larger subproblems are further divided into smaller units
- finally the solutions of the subproblems are combined to get the solution to the original problem

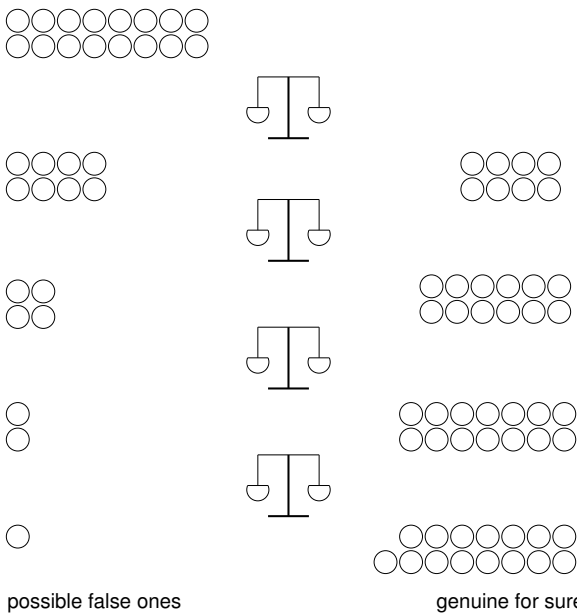
Let's get back to the claim made earlier about the complexity notation not being fixed to programs and take an everyday, concrete example

Example: finding the false goldcoin

- The problem is well-known from logic problems.
- We have n gold coins, one of which is false. The false coin looks the same as the real ones but is lighter than the others. We have a scale we can use and our task is to find the false coin.
- We can solve the problem with Decrease and conquer by choosing a random coin and by comparing it to the other coins one at a time.

  $\Rightarrow$ At least 1 and at most n - 1 weighings are needed. The best-case efficiency is $\Theta(1)$ and the worst and average case efficiencies are $\Theta(n)$.
- Alternatively we can always take two coins at random and weigh them. At most $n/2$ weighings are needed and the efficiency of the solution is still the same.

The same problem can be solved more efficiently with divide and conquer:



- Divide the coins into the two pans on the scales. The coins on the heavier side are all authentic, so they don't need to be investigated further.
- Continue the search similarly with the lighter half, i.e. the half that contains the false coin, until there is only one coin in the pan, the coin that we know is false.
- The solution is recursive and the base case is the situation where there is only one possible coin that can be false.

- The amount of coins on each weighing is 2 to the power of the amount of weighings still required: on the highest level there are $2^{weighings}$ coins, so based on the definition of the logarithm:

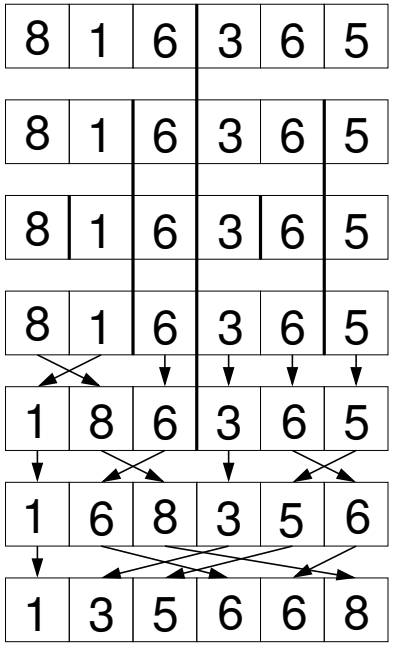
$$2^{weighings} = n \Rightarrow log_2 n = weighings$$

- Only $\log_2 n$ weighings is needed, which is significantly fewer than $n/2$ when the amount of coins is large.

  $\Rightarrow$ The complexity of the solution is $\Theta(\lg n)$ both in the best and the worst-case.

MERGE-SORT:

- divide the elements in the array into two halves.
- continue dividing the halves further in half until the subarrays contain atmost one element
- arrays of 0 or 1 length are already sorted and require no actions
- finally merge the sorted subarrays



- the MERGE-algorithm for merging the subarrays:

```
MERGE(A, left, middle, right)
1   for i := left to right do      (scan through the entire array...)
2       B[i] := A[i]               (... and copy it into a temporary array)
3   i := left                      (set i to indicate the endpoint of the sorted part)
4   j := left; k := middle + 1     (set j and k to indicate the beginning of the subarrays)
5   while j ≤ middle and k ≤ right do      (scan until either half ends)
6       if B[j] ≤ B[k] then        (if the first element in the lower half is smaller...)
7           A[i] := B[j]           (... copy it into the result array...)
8           j := j + 1             (... increment the starting point of the lower half)
9       else                       (else...)
10          A[i] := B[k]           (... copy the first element of the upper half...)
11          k := k + 1             (... and increment its starting point)
12      i := i + 1                 (increment the starting point of the finished set)
13  if j > middle then
14      k := 0
15  else
16      k := middle − right
17  for j := i to right do         (copy the remaining elements to the end of the result)
18      A[j] := B[j + k]
```

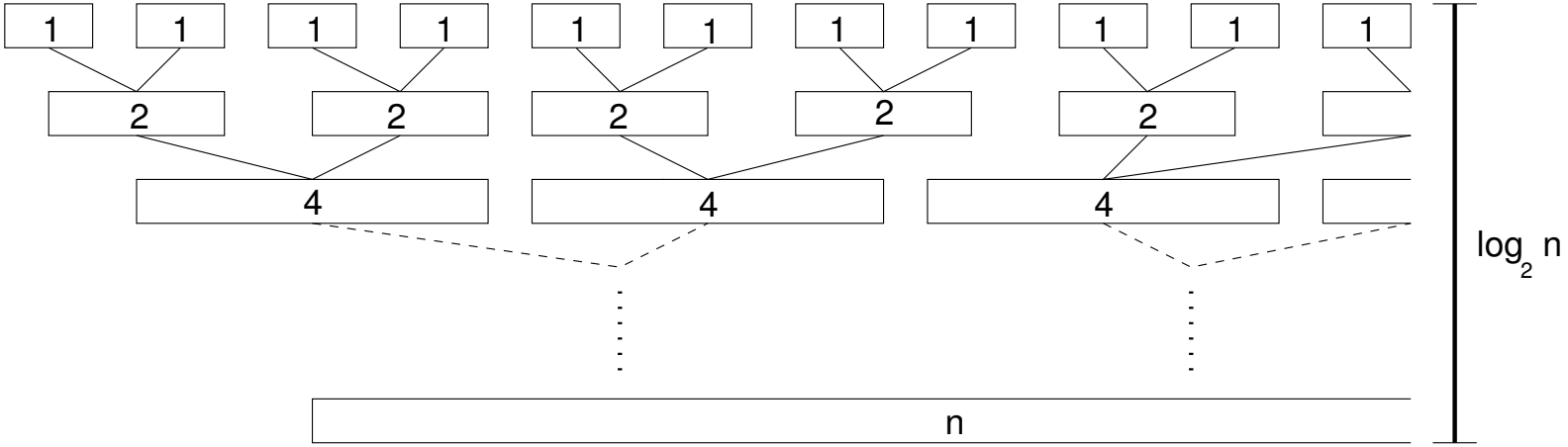
- MERGE-SORT

```
MERGE-SORT(A, left, right)
1  if left < right then                    (if there are elements in the array...)
2      middle := ⌊(left + right)/2⌋        (... divide it into half)
3      MERGE-SORT(A, left, middle)         (sort the upper half...)
4      MERGE-SORT(A, middle + 1, right)    (... and the lower)
5      MERGE(A, left, middle, right)       (merge the parts maintaining the order)
```

- MERGE merges the arrays by using the "decrease and conquer" method.
  - the first **for**-loop uses a linear amount of time $\Theta(n)$ relative to the size of the subarray
  - the **while**-loop scans through the upper and the lower halves at most once and at least one of the halves entirely $\Rightarrow \Theta(n)$
  - the second **for**-loop scans through at most half of the array and its running time is $\Theta(n)$ in the worst case.
  - other operations are constant time

$\Rightarrow$ the running time of the entire algorithm is obtained by combining the results. It's $\Theta(n)$ both in the best and the worst case.

- Like with QUICKSORT, the analysis of MERGE-SORT is not as straghtforward since it is recursive.
- MERGE-SORT calls itself and MERGE, all other operations are constant time $\Rightarrow$ we can concentrate on the time used by the intances of MERGE, everything else is constant time.



- The instances of MERGE form a treelike structure shown on the previous page.
  - the sizes of the subarrays are marked on the instances of MERGE in the picture
  - on the first level the length of each subarray is 1 (or 0)
  - the subarrays on the upper levels are always two times as large as the ones on the previous level
  - the last level handles the entire array
  - the combined size of the subarrays on each level is n
  - the amount of instances of MERGE on a given level is two times the amount on the previous level.
    $\Rightarrow$ the amount increases in powers of two, so the amount of instances on the last level is 2^h, where h is the height of the tree.
  - on the last level there are approximately n instances
    $\Rightarrow 2^h = n \Leftrightarrow \log_2 n = h$, the height of the tree is $\log_2 n$
  - since a linear amount of work is done on each level and there are $\lg n$ levels, the running time of the entire algorithm is $\Theta(n \lg n)$

MERGE-SORT is clearly more complicated than INSERTION-SORT. Is using it really worth the effort?

Yes, on large inputs the difference is clear.

- if n is 1000 000 n^2 is 1000 000 000 000, while n log n is about 19 930 000

Advantages and disadvantages of Mergesort

Advantages

- Running time $\Theta(n \lg n)$
- Stable

Disadvantages

- MERGE-SORT requires $\Theta(n)$ extra memory, INSERTION-SORT and QUICKSORT sort in place.
- Constant coefficient quite large

4.3 RETURN OF QUICKSORT

Let's next cover a very efficient sorting algorithm QUICKSORT.

QUICKSORT is a divide and conquer algorithm.

The division of the problem into smaller subproblems

- Select one of the elements in the array as a *pivot*, i.e. the element which partitions the array.
- Change the order of the elements in the array so that all elements smaller or equal to the pivot are placed before it and the larger elements after it.
- Continue dividing the upper and lower halves into smaller subarrays, until the subarrays contain 0 or 1 elements.

Smaller subproblems:

- Subarrays of the size 0 and 1 are already sorted

Combining the sorted subarrays:

- The entire (sub) array is automatically sorted when its upper and lower halves are sorted.
  - all elements in the lower half are smaller than the elements in the upper half, as they should be

QUICKSORT-algorithm

QUICKSORT$(A, left, right)$

```
1  if left < right then                          (do nothing in the trivial case)
2      pivot := PARTITION(A, left, right)        (partition in two)
3      QUICKSORT(A, left, pivot − 1)             (sort the elements smaller than the pivot)
4      QUICKSORT(A, pivot + 1, right)            (sort the elements larger than the pivot)
```

The *partition algorithm* rearranges the subarray in place

```
PARTITION(A, left, right)
1   pivot := A[right]                    (choose the last element as the pivot)
2   i := left − 1                        (use i to mark the end of the smaller elements)
4   for j := left to right − 1 do        (scan to the second to last element)
6       if A[j] ≤ pivot                  (if A[j] goes to the half with the smaller elements...)
9           i := i + 1                   (... increment the amount of the smaller elements...)
12          exchange A[i] ↔ A[j]         (... and move A[j] there)
12  exchange A[i + 1] ↔ A[right]         (place the pivot between the halves)
13  return i + 1                         (return the location of the pivot)
```

How fast is PARTITION?

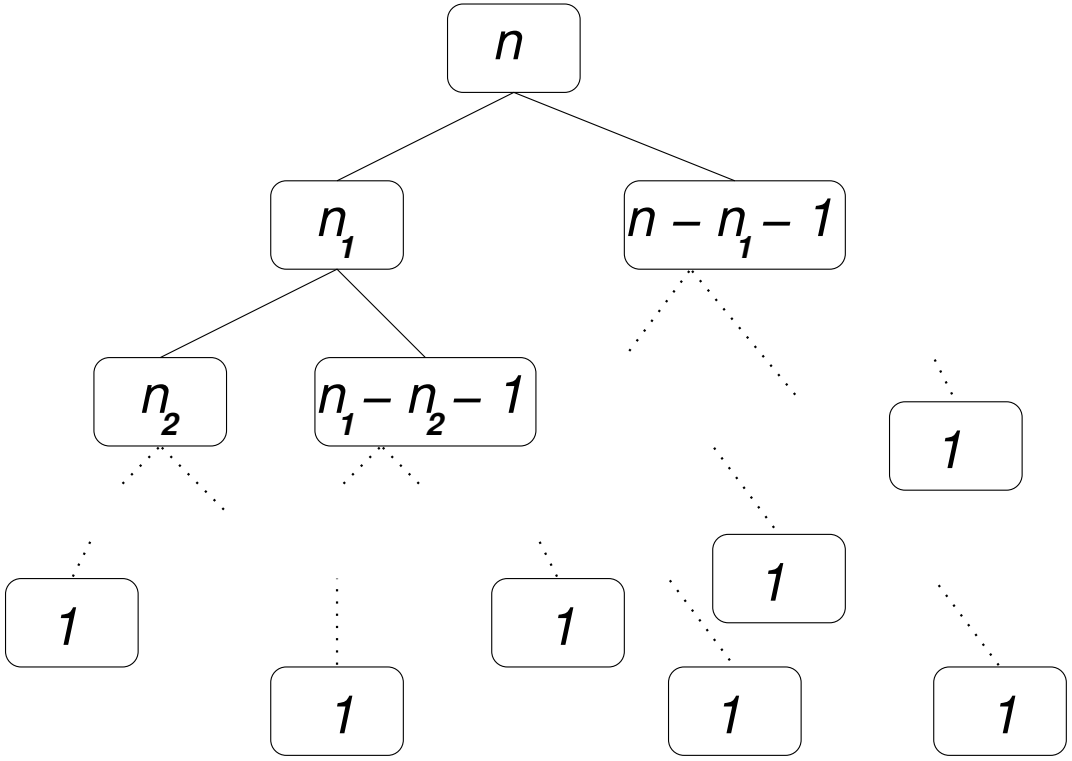
- The **for**-loop is executed n - 1 times when n is r - p
- All other operations are constant time.

$\Rightarrow$ The running-time is $\Theta(n)$.

Determining the running-time of QUICKSORT is more difficult since it is recursive. Therefore the equation for its running time would also be recursives.

Finding the recursive equation is, however, beyond the goals of this course so we'll settle for a less formal approach

- As all the operations of QUICKSORT except PARTITION and the recursive call are constant time, let's concentrate on the time used by the instances of PARTITION.



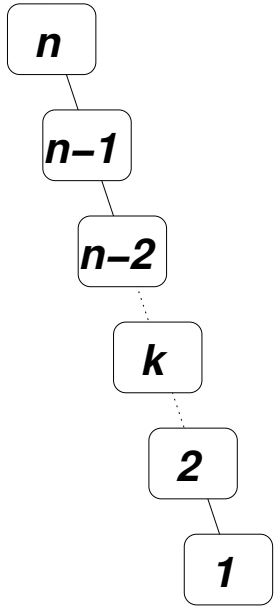
- The total time is the sum of the running times of the nodes in the picture abowe.
- The execution is constant time for an array of size 1.
- For the other the execution is linear to the size of the array.
  $\Rightarrow$ The total time is Θ(the sum of the numbers of the nodes).

Worst-case running time

- The number of a node is always smaller than the number of its parent, since the pivot is already in its correct location and doesn't go into either of the sorted subarrays

  $\Rightarrow$ there can be atmost n levels in the tree

- the worst case is realized when the smallest or the largest element is always chosen as the pivot
  - this happens, for example, with an array already sorted
- the sum of the node numbers is n + n - 1 + $\cdots$ + 2 + 1

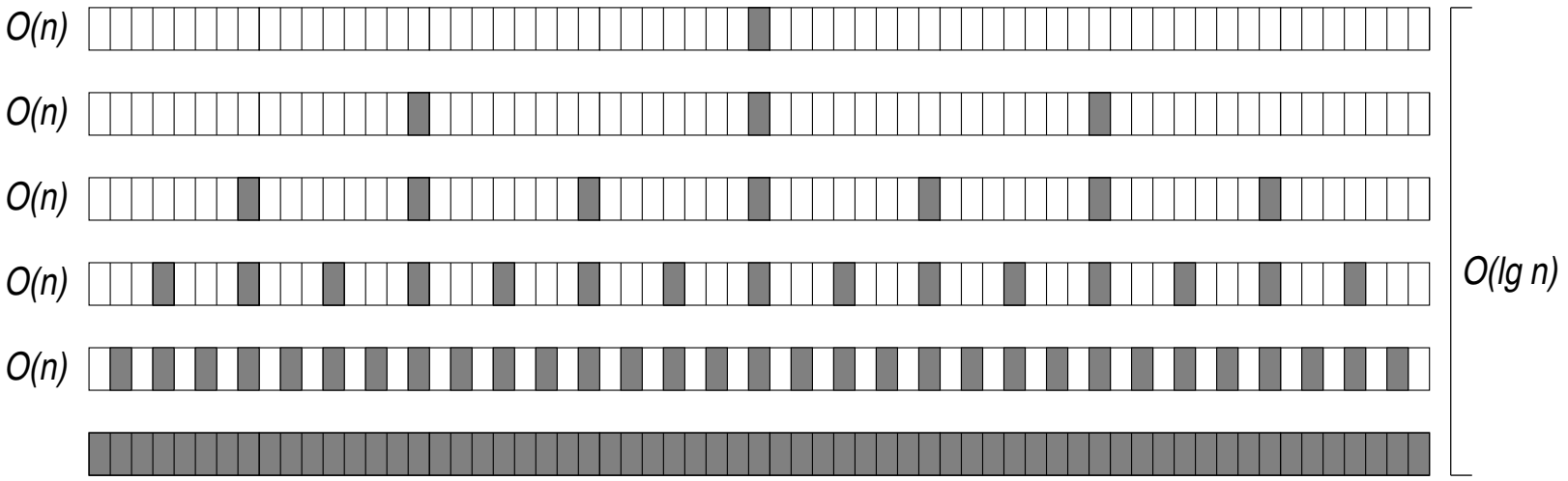


$\Rightarrow$ the worst case running time of QUICKSORT is $\Theta(n^2)$

The best-case is when the array is always divided evenly in half.

- The picture below shows how the subarrays get smaller.
  - The grey boxes mark elements already in their correct position.
- The amount of work on each level is in $\Theta(n)$.
  - a pessimistic estimate on the height of the execution tree is in the best-case $\Rightarrow \Theta(\lg n)$

$\Rightarrow$ The upper limit for the best-case efficiency is $\Theta(n \lg n)$.



The best-case and the worst-case efficiencies of QUICKSORT differ significantly.

- It would be interesting to know the average-case running-time.
- Analyzing it is beyond the goals of the course but it has been shown that if the data is evenly distributed its average running-time is $\Theta(n \lg n)$.
- Thus the average running-time is quite good.

An unfortunate fact with QUICKSORT is that its worst-case efficiency is poor and in practise the worst-case situation is quite probable.

- It is easy to see that there can be situations where the data is already sorted or almost sorted.

$\Rightarrow$ A way to decrease the risk of the systematic occurence of the worst-case situation's likelyhood is needed.

Randomization has proved to be quite efficient.

Advantages and disadvantages of QUICKSORT

Advantages:

- sorts the array very efficiently in average
  - the average-case running-time is $\Theta(n \lg n)$
  - the constant coefficient is small
- requires only a constant amount of extra memory
- if well-suited for the virtual memory environment

Disadvantages:

- the worst-case running-time is $\Theta(n^2)$
- without randomization the worst-case input is far too common
- the algorithm is recursive
  $\Rightarrow$ the stack uses extra memory
- instability