Introduction to algorithms

September 8, 2017

Overview

Asymptotic notations - motivation

- As we study different algorithms we would like to understand their efficiency in terms of complexity and sometimes their memory requirements.
- Exact computation of the complexity of an algorithms is usually difficult and not very insightful.
- It is more important to understand the behaviour of the function that represent the complexity as the input becomes very large.

Asymptotic notations - motivation

- As we study different algorithms we would like to understand their efficiency in terms of complexity and sometimes their memory requirements.
- Exact computation of the complexity of an algorithms is usually difficult and not very insightful.
- It is more important to understand the behaviour of the function that represent the complexity as the input becomes very large.

Asymptotic notations - motivation

- As we study different algorithms we would like to understand their efficiency in terms of complexity and sometimes their memory requirements.
- Exact computation of the complexity of an algorithms is usually difficult and not very insightful.
- It is more important to understand the behaviour of the function that represent the complexity as the input becomes very large.

A few examples

- Assume A_1, A_2, A_3, A_4 are 4 algorithms that solves the same problem.
- For an input of size n the complexities of the algorithms A_1, A_2, A_3, A_4 are given by $100 \times \lg(n), n + 30, n^2, \frac{1}{10}e^n$ respectively.
  Which algorithm should we use?
- The following table demonstrates the behaviour of these functions:

  What do we mean by too big?

A few examples

- Assume A_1, A_2, A_3, A_4 are 4 algorithms that solves the same problem.
- For an input of size n the complexities of the algorithms A_1, A_2, A_3, A_4 are given by $100 \times \lg(n)$, $n + 30$, n^2, $\frac{1}{10}e^n$ respectively.
  Which algorithm should we use?
- The following table demonstrates the behaviour of these functions:

  What do we mean by too big?

A few examples

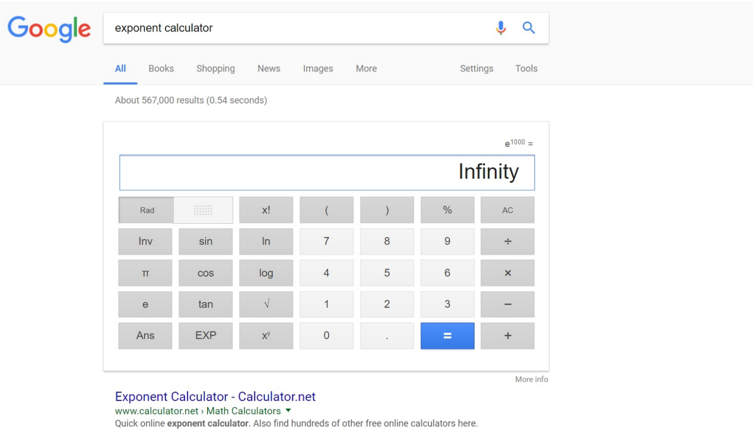
- Assume A_1, A_2, A_3, A_4 are 4 algorithms that solves the same problem.
- For an input of size n the complexities of the algorithms A_1, A_2, A_3, A_4 are given by $100 \times \lg(n)$, $n + 30$, n^2, $\frac{1}{10}e^n$ respectively.
  Which algorithm should we use?
- The following table demonstrates the behaviour of these functions:

Function / n	n= 10	n= 100	n= 1000	n= 10000
lg(n)	3	7	10	13
n	10	100	1000	10000
n²	100	10000	1000000	100000000
exp(n)	22026	2.68812E+43	TOO BIG	TOO BIG

What do we mean by too big?

A few examples

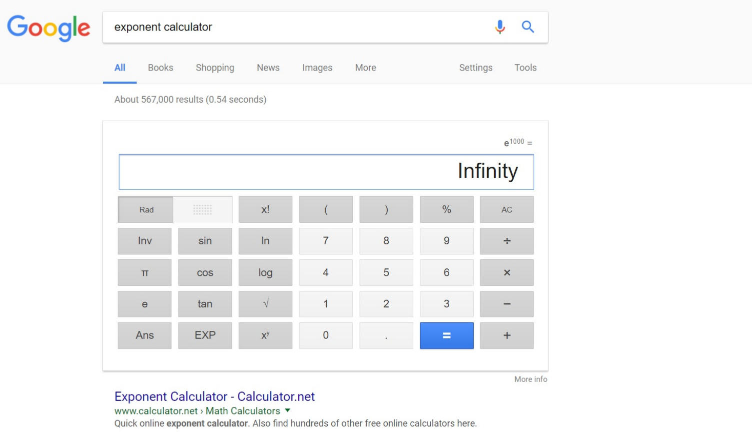
- What do we mean by too big?



- It is easy to see that for large n, the fastest algorithm is A_1 followed by A_2, A_3 and finally A_4.
- We want a way to describe the complexity of an algorithm without the trouble of finding the exact formula for the complexity.

A few examples

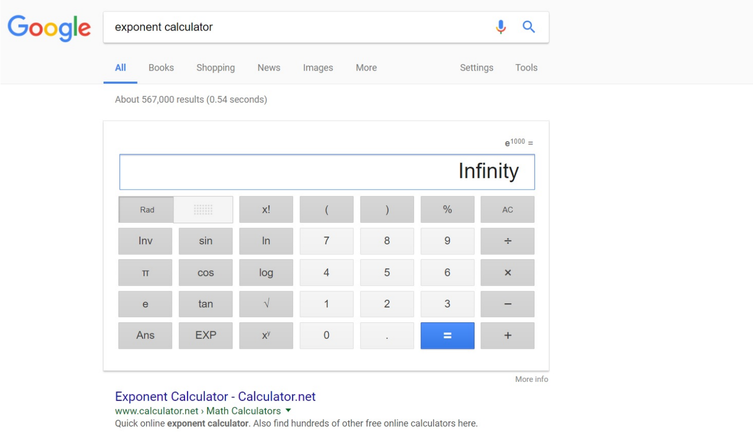
- What do we mean by too big?



- It is easy to see that for large n, the fastest algorithm is A_1 followed by A_2, A_3 and finally A_4.
- We want a way to describe the complexity of an algorithm without the trouble of finding the exact formula for the complexity.

A few examples

- What do we mean by too big?



- It is easy to see that for large n, the fastest algorithm is A_1 followed by A_2, A_3 and finally A_4.
- We want a way to describe the complexity of an algorithm without the trouble of finding the exact formula for the complexity.

$\Theta(f(n))$ - definition

- To describe the notion that a function behaves like the function $f(n)$ for large n we use the notion $\Theta(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $\Theta(f(n))$ to be:

$$\Theta(f(n)) = \{g(n) \mid \text{there exist } \textbf{positive} \text{ constants } c_1, c_2$$

$$\text{and } n_0 \text{ such that for any } n > n_0 \text{ we have}$$

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

- Note that the definition implies that the functions are non-negative for large enough n.

$\Theta(f(n))$ - definition

- To describe the notion that a function behaves like the function $f(n)$ for large n we use the notion $\Theta(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $\Theta(f(n))$ to be:

$$\Theta(f(n)) = \{g(n) \mid \text{there exist } \textbf{positive} \text{ constants } c_1, c_2$$

$$\text{and } n_0 \text{ such that for any } n > n_0 \text{ we have}$$

$$0 \le c_1 \cdot f(n) \le g(n) \le c_2 \cdot f(n)\}$$

- Note that the definition implies that the functions are non-negative for large enough n.

$\Theta(f(n))$ - definition

- To describe the notion that a function behaves like the function $f(n)$ for large n we use the notion $\Theta(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $\Theta(f(n))$ to be:

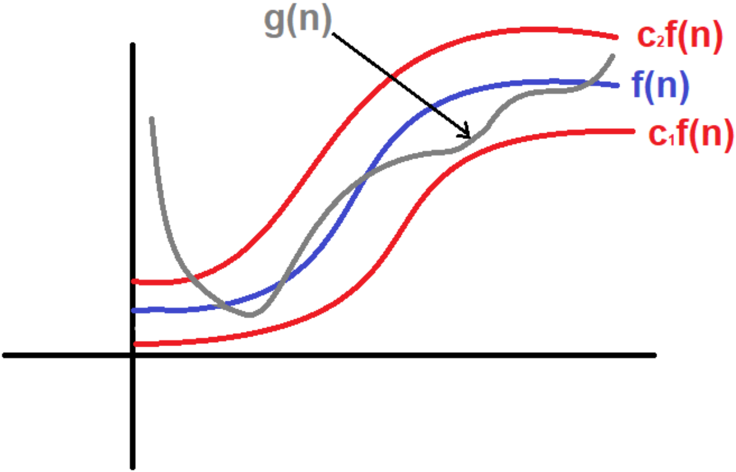
$$\Theta(f(n)) = \{g(n) \mid \text{there exist } \textbf{positive} \text{ constants } c_1, c_2$$

$$\text{and } n_0 \text{ such that for any } n > n_0 \text{ we have}$$

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

- Note that the definition implies that the functions are non-negative for large enough n.

$\Theta(f(n))$ - definition



$\Theta(f(n))$ - example

- It is easy to see that $2n^2 + 5 \in \Theta(n^2)$.
- Indeed, since $\lim_{n\to\infty} \frac{2n^2+5}{n^2} = 2$ which implies that there exists n_0 such that for any $n > n_0$ we have:

$$1 \le \frac{2n^2+5}{n^2} \le 3 \text{ which implies: } n^2 \le 2n^2 + 5 \le 3n^2$$

- **Notation:** We denote this by $2n^2 + 5 = \Theta(n^2)$.

$\Theta(f(n))$ - example

- It is easy to see that $2n^2 + 5 \in \Theta(n^2)$.
- Indeed, since $\lim_{n \to \infty} \frac{2n^2+5}{n^2} = 2$ which implies that there exists n_0 such that for any $n > n_0$ we have:

$$1 \leq \frac{2n^2 + 5}{n^2} \leq 3 \text{ which implies: } n^2 \leq 2n^2 + 5 \leq 3n^2$$

- **Notation:** We denote this by $2n^2 + 5 = \Theta(n^2)$.

$\Theta(f(n))$ - example

- It is easy to see that $2n^2 + 5 \in \Theta(n^2)$.
- Indeed, since $\lim_{n\to\infty} \frac{2n^2+5}{n^2} = 2$ which implies that there exists n_0 such that for any $n > n_0$ we have:

$$1 \leq \frac{2n^2+5}{n^2} \leq 3 \text{ which implies: } n^2 \leq 2n^2 + 5 \leq 3n^2$$

- **Notation:** We denote this by $2n^2 + 5 = \Theta(n^2)$.

$O(f(n))$ - definition

- Many times it is enough to know that a function is not (significantly) larger than some function $f(n)$ (for large n). We use the notion $O(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $O(f(n))$ to be:

$$O(f(n)) = \{g(n) \mid \text{there exist } \textbf{positive} \text{ constants } c$$

$$\text{and } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have}$$

$$0 \le g(n) \le c \cdot f(n))\}$$

- Note that the definition implies that the functions are non-negative for large enough n.

$O(f(n))$ - definition

- Many times it is enough to know that a function is not (significantly) larger than some function $f(n)$ (for large n). We use the notion $O(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $O(f(n))$ to be:

$$O(f(n)) = \{g(n) \mid \text{there exist } \textbf{positive} \text{ constants } c$$

$$\text{and } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have}$$

$$0 \leq g(n) \leq c \cdot f(n))\}$$

- Note that the definition implies that the functions are non-negative for large enough n.

$O(f(n))$ - definition

- Many times it is enough to know that a function is not (significantly) larger than some function $f(n)$ (for large n). We use the notion $O(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $O(f(n))$ to be:

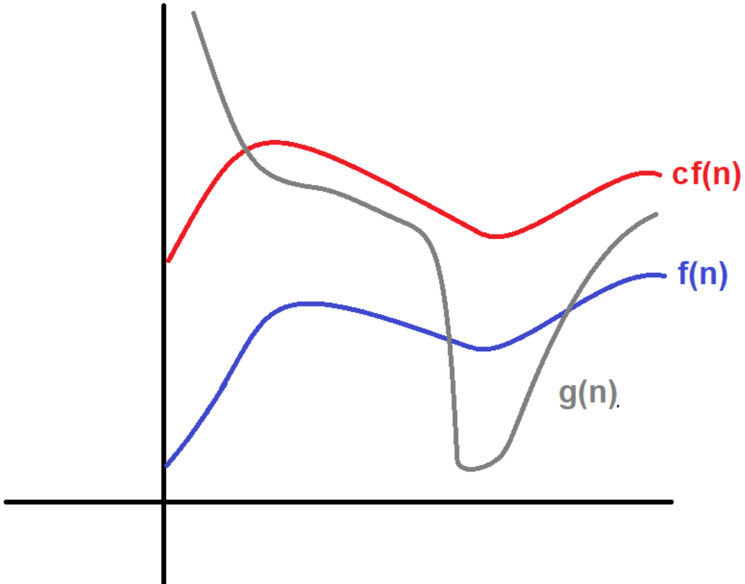
$$O(f(n)) = \{g(n) \mid \text{there exist } \textbf{positive} \text{ constants } c$$

$$\text{and } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have}$$

$$0 \leq g(n) \leq c \cdot f(n))\}$$

- Note that the definition implies that the functions are non-negative for large enough n.

$O(f(n))$ - definition



$o(f(n))$ - definition

- We may also want to describe the fact that a function is *significantly* smaller than some function $f(n)$ (for large n). We use the notion $o(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $o(f(n))$ to be:

$$o(f(n)) = \{g(n) \mid \text{for } \textbf{any} \text{ positive constant } c$$

$$\text{there exists } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have}$$

$$0 \leq g(n) \leq c \cdot f(n))\}$$

- Note that the definition implies that the functions are non-negative for large enough n.

$o(f(n))$ - definition

- We may also want to describe the fact that a function is *significantly* smaller than some function $f(n)$ (for large n). We use the notion $o(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $o(f(n))$ to be:

$$o(f(n)) = \{g(n) \mid \text{for } \underline{\textbf{any}} \text{ positive constant } c$$

$$\text{there exists } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have}$$

$$0 \leq g(n) \leq c \cdot f(n))\}$$

- Note that the definition implies that the functions are non-negative for large enough n.

$o(f(n))$ - definition

- We may also want to describe the fact that a function is *significantly* smaller than some function $f(n)$ (for large n). We use the notion $o(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $o(f(n))$ to be:

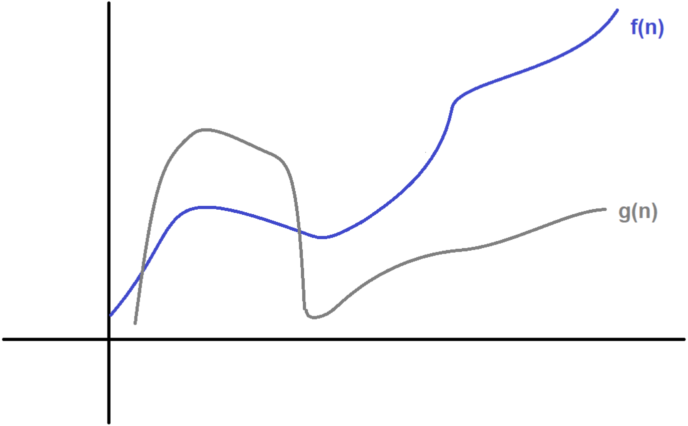
$$o(f(n)) = \{g(n) \mid \text{for } \underline{\textbf{any}} \text{ positive constant } c$$

$$\text{there exists } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have}$$

$$0 \leq g(n) \leq c \cdot f(n))\}$$

- Note that the definition implies that the functions are non-negative for large enough n.

$o(f(n))$ - definition



$\Omega(f(n))$ and $\omega(f(n))$ - definitions

- Similarly for the case of "bigger than some function $f(n)$" (for large n). We use the notions $\Omega(f(n))$ and $\omega(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $\Omega(f(n))$ to be:

$$\Omega(f(n)) = \{g(n) \mid \text{there exist a positive constant } c \text{ and } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have } 0 \leq c \cdot f(n) \leq g(n))\}$$

- Definition: For a function $f(n)$ we define the set of functions $\omega(f(n))$ to be:

$$\omega(f(n)) = \{g(n) \mid \text{for any positive constant } c \text{ there exists } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have } 0 \leq c \cdot f(n) \leq g(n))\}$$

$\Omega(f(n))$ and $\omega(f(n))$ - definitions

- Similarly for the case of "bigger than some function $f(n)$" (for large n). We use the notions $\Omega(f(n))$ and $\omega(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $\Omega(f(n))$ to be:

$$\Omega(f(n)) = \{g(n) \mid \text{there exist a positive constant } c$$
$$\text{and } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have}$$
$$0 \leq c \cdot f(n) \leq g(n))\}$$

- Definition: For a function $f(n)$ we define the set of functions $\omega(f(n))$ to be:

$$\omega(f(n)) = \{g(n) \mid \text{for any positive constant } c$$
$$\text{there exists } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have}$$
$$0 \leq c \cdot f(n) \leq g(n))\}$$

$\Omega(f(n))$ and $\omega(f(n))$ - definitions

- Similarly for the case of "bigger than some function $f(n)$" (for large n). We use the notions $\Omega(f(n))$ and $\omega(f(n))$.
- Definition: For a function $f(n)$ we define the set of functions $\Omega(f(n))$ to be:

$$\Omega(f(n)) = \{g(n) \mid \text{there exist a positive constant } c \text{ and } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have } 0 \leq c \cdot f(n) \leq g(n))\}$$

- Definition: For a function $f(n)$ we define the set of functions $\omega(f(n))$ to be:

$$\omega(f(n)) = \{g(n) \mid \text{for any positive constant } c \text{ there exists } n_0 > 0 \text{ such that for any } n > n_0 \text{ we have } 0 \leq c \cdot f(n) \leq g(n))\}$$

Some useful facts

The following facts (and many many others) can be easily verified:

- For $\alpha, \beta > 0$ we have:
  - $n^\alpha = O(n^\beta) \;\Leftrightarrow\; \alpha \leq \beta$
  - $n^\alpha = o(n^\beta) \;\Leftrightarrow \alpha < \beta$
- For any $\alpha > 0$, $c > 1$ we have $n^\alpha = o(c^n)$.
- For any $\alpha > 0$, $b > 1$ we have $\log_b(n) = o(n^\alpha)$.
- For any $a, b > 1$ we have $\log_a(n) = \Theta(\log_b(n))$.
- For any $a, b > 0$ with $a \leq b$ we have $a^n = O(b^n)$.

Some useful facts

The following facts (and many many others) can be easily verified:

- For $\alpha, \beta > 0$ we have:
  - $n^\alpha = O(n^\beta) \;\Leftrightarrow\; \alpha \leq \beta$
  - $n^\alpha = o(n^\beta) \;\Leftrightarrow \alpha < \beta$
- For any $\alpha > 0$, $c > 1$ we have $n^\alpha = o(c^n)$.
- For any $\alpha > 0$, $b > 1$ we have $\log_b(n) = o(n^\alpha)$.
- For any $a, b > 1$ we have $\log_a(n) = \Theta(\log_b(n))$.
- For any $a, b > 0$ with $a \leq b$ we have $a^n = O(b^n)$.

Some useful facts

The following facts (and many many others) can be easily verified:

- For $\alpha, \beta > 0$ we have:
  - $n^\alpha = O(n^\beta) \;\Leftrightarrow\; \alpha \leq \beta$
  - $n^\alpha = o(n^\beta) \;\Leftrightarrow \alpha < \beta$
- For any $\alpha > 0$, $c > 1$ we have $n^\alpha = o(c^n)$.
- For any $\alpha > 0$, $b > 1$ we have $\log_b(n) = o(n^\alpha)$.
- For any $a, b > 1$ we have $\log_a(n) = \Theta(\log_b(n))$.
- For any $a, b > 0$ with $a \leq b$ we have $a^n = O(b^n)$.

Some useful facts

The following facts (and many many others) can be easily verified:

- For $\alpha, \beta > 0$ we have:
  - $n^\alpha = O(n^\beta) \iff \alpha \le \beta$
  - $n^\alpha = o(n^\beta) \iff \alpha < \beta$
- For any $\alpha > 0$, $c > 1$ we have $n^\alpha = o(c^n)$.
- For any $\alpha > 0$, $b > 1$ we have $\log_b(n) = o(n^\alpha)$.
- For any $a, b > 1$ we have $\log_a(n) = \Theta(\log_b(n))$.
- For any $a, b > 0$ with $a \le b$ we have $a^n = O(b^n)$.

Some useful facts

The following facts (and many many others) can be easily verified:

- For $\alpha, \beta > 0$ we have:
  - $n^\alpha = O(n^\beta) \;\Leftrightarrow\; \alpha \leq \beta$
  - $n^\alpha = o(n^\beta) \;\Leftrightarrow \alpha < \beta$
- For any $\alpha > 0$, $c > 1$ we have $n^\alpha = o(c^n)$.
- For any $\alpha > 0$, $b > 1$ we have $\log_b(n) = o(n^\alpha)$.
- For any $a, b > 1$ we have $\log_a(n) = \Theta(\log_b(n))$.
- For any $a, b > 0$ with $a \leq b$ we have $a^n = O(b^n)$.

Some useful facts

The following facts (and many many others) can be easily verified:

- For $\alpha, \beta > 0$ we have:
  - $n^\alpha = O(n^\beta) \;\Leftrightarrow\; \alpha \le \beta$
  - $n^\alpha = o(n^\beta) \;\Leftrightarrow \alpha < \beta$
- For any $\alpha > 0$, $c > 1$ we have $n^\alpha = o(c^n)$.
- For any $\alpha > 0$, $b > 1$ we have $\log_b(n) = o(n^\alpha)$.
- For any $a, b > 1$ we have $\log_a(n) = \Theta(\log_b(n))$.
- For any $a, b > 0$ with $a \le b$ we have $a^n = O(b^n)$.

Some useful facts

The following facts (and many many others) can be easily verified:

- For $\alpha, \beta > 0$ we have:
  - $n^{\alpha} = O(n^{\beta}) \;\Leftrightarrow\; \alpha \leq \beta$
  - $n^{\alpha} = o(n^{\beta}) \;\Leftrightarrow \alpha < \beta$
- For any $\alpha > 0$, $c > 1$ we have $n^{\alpha} = o(c^{n})$.
- For any $\alpha > 0$, $b > 1$ we have $\log_b(n) = o(n^{\alpha})$.
- For any $a, b > 1$ we have $\log_a(n) = \Theta(\log_b(n))$.
- For any $a, b > 0$ with $a \leq b$ we have $a^n = O(b^n)$.

The sorting problem

- Notations:
  We denote the set $\{1, 2, 3, \ldots, n\}$ by $[n]$.
  We denote the set of all permutations of $[n]$ by S_n (this is actually a group).
- Given a sequence of real numbers $a = (a_1, a_2, \ldots, a_n)$ we would like to reorder them from the smallest to the largest.

$$4, 6, 1, 7, 11, 5 \;\rightarrow\; 1, 4, 5, 6, 7, 11$$

- Now try to sort the following sequence:

92, 13, 60, 10, 39, 80, 91, 52, 58, 61, 79, 94, 29, 82, 7, 59, 37, 41, 38, 12, 15, 85, 3, 87, 20, 83, 68, 27, 73

The sorting problem

- Notations:
  We denote the set $\{1, 2, 3, \ldots, n\}$ by $[n]$.
  We denote the set of all permutations of $[n]$ by S_n (this is actually a group).
- Given a sequence of real numbers $a = (a_1, a_2, \ldots, a_n)$ we would like to reorder them from the smallest to the largest.

$$4, 6, 1, 7, 11, 5 \;\rightarrow\; 1, 4, 5, 6, 7, 11$$

- Now try to sort the following sequence:

92, 13, 60, 10, 39, 80, 91, 52, 58, 61, 79, 94, 29, 82, 7, 59, 37, 41, 38, 12, 15, 85, 3, 87, 20, 83, 68, 27, 73

The sorting problem

- Notations:
  We denote the set $\{1, 2, 3, ..., n\}$ by $[n]$.
  We denote the set of all permutations of $[n]$ by S_n (this is actually a group).
- Given a sequence of real numbers $a = (a_1, a_2, ..., a_n)$ we would like to reorder them from the smallest to the largest.

$$4, 6, 1, 7, 11, 5 \;\rightarrow\; 1, 4, 5, 6, 7, 11$$

- Now try to sort the following sequence:

$92, 13, 60, 10, 39, 80, 91, 52, 58, 61, 79, 94, 29, 82, 7, 59, 37, 41, 38, 12, 15, 85, 3, 87, 20, 83, 68, 27, 73$

The sorting problem

- **The Sorting problem:** Given a sequence of (different) real numbers $a = (a_1, a_2, ..., a_n)$, find the (unique) permutation $\pi \in S_n$ such that:

$$a_{\pi(1)} \leq a_{\pi(2)} \leq ... \leq a_{\pi(n)}$$

- We will consider sorting algorithms that uses only comparisons (and not algebraic properties of numbers for example).
- Such algorithms do not assume anything other then the existence of an order relation so they can be applied for objects other than numbers.
- We will take a brief look at algorithms that use other properties and see the differences.

The sorting problem

- **The Sorting problem:** Given a sequence of (different) real numbers $a = (a_1, a_2, \ldots, a_n)$, find the (unique) permutation $\pi \in S_n$ such that:

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$$

- We will consider sorting algorithms that uses only comparisons (and not algebraic properties of numbers for example).
- Such algorithms do not assume anything other then the existence of an order relation so they can be applied for objects other than numbers.
- We will take a brief look at algorithms that use other properties and see the differences.

The sorting problem

- **The Sorting problem:** Given a sequence of (different) real numbers $a = (a_1, a_2, ..., a_n)$, find the (unique) permutation $\pi \in S_n$ such that:

$$a_{\pi(1)} \leq a_{\pi(2)} \leq ... \leq a_{\pi(n)}$$

- We will consider sorting algorithms that uses only comparisons (and not algebraic properties of numbers for example).
- Such algorithms do not assume anything other then the existence of an order relation so they can be applied for objects other than numbers.
- We will take a brief look at algorithms that use other properties and see the differences.

The sorting problem

- **The Sorting problem:** Given a sequence of (different) real numbers $a = (a_1, a_2, \ldots, a_n)$, find the (unique) permutation $\pi \in S_n$ such that:

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$$

- We will consider sorting algorithms that uses only comparisons (and not algebraic properties of numbers for example).
- Such algorithms do not assume anything other then the existence of an order relation so they can be applied for objects other than numbers.
- We will take a brief look at algorithms that use other properties and see the differences.

Insertion sort

- Input: An array of real numbers $A = (a_1, a_2, ..., a_n)$.
  Output: The sorted array $(a_{\pi(1)}, a_{\pi(2)}, ..., a_{\pi(n)})$ such that $a_{\pi(1)} \leq a_{\pi(2)} \leq ... \leq a_{\pi(n)}$.
- The basic idea of the algorithm is to break the algorithm to n steps. At each step we "take" an element and put it in the "correct" position.
  In the i-th step the first $i-1$ elements are sorted, and we put the i-th element of the array in the "correct" position among the sorted numbers.

Insertion sort

- Input: An array of real numbers $A = (a_1, a_2, \ldots, a_n)$.
  Output: The sorted array $(a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)})$ such that $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$.
- The basic idea of the algorithm is to break the algorithm to n steps. At each step we "take" an element and put it in the "correct" position.
  In the i-th step the first $i-1$ elements are sorted, and we put the i-th element of the array in the "correct" position among the sorted numbers.

Insertion sort

- Let's apply this idea for the sequence we saw before:

92, 13, 60, 10, 39, 80, 91, 52, 58, 61, 79, 94, 29, 82, 7, 59, 37, 41, 38, 12, 15, 85, 3, 87, 20, 83, 68, 27, 73

We begin with:

92,13,60,10,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

At this point, the first integer (which is 92) is sorted:

92|13,60,10,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Insertion sort

We set: $k = 13$.

We start by comparing $13 <^{?} 92$. Since the answer is yes, we get:

$k = 13$

92,13,60,10,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Insertion sort

We get:

13,92,60,10,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Now the first 2 numbers are sorted.

Insertion sort

We set: $k = 60$.

We compare $60 <^? 92$. Since the answer is yes, we get:

13,92,60,10,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Insertion sort

Next we compare $60 <^{?} 13$. Since the answer is no, we get:

$k = 60$

13,92|92,10,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

So we have:

13,60,92|10,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Now the first 3 numbers are sorted.

Insertion sort

We set: $k = 10$.

We compare $10 <^? 92$. Since the answer is yes, we get:

13,60,92,10,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

And we get:

13,60,92,92,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Insertion sort

We compare $10 <^{?} 60$. Since the answer is yes, we get:

13,60,92|92,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

And we get:

13,60,60|92,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Insertion sort

We compare $10 <^{?} 13$. Since the answer is yes, we get:

13,60,60|92,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

And we get:

13,13,60|92,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Insertion sort

Since there are no more comparisons to perform we put $k = 10$ in the first place:

$k = 10$

13,13,60,92,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

And we get:

10,13,60,92,39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Now the first 4 numbers are sorted:

10,13,60,92 39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Insertion sort

We set $k = 39$.

We compare $39 <^{?} 92$. Since the answer is yes, we get:

10,13,60,92|39,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

We get:

10,13,60,92|92,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Insertion sort

We compare $39 <^? 60$. Since the answer is yes, we get:

10,13,60,92,92,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

We get:

10,13,60,60,92,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Insertion sort

We compare $39 <^? 13$. Since the answer is no, we insert k after 13:

$k = 39$

10,13,60,60|92,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

We get:

10,13,39,60|92,80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

Now the first 5 numbers are sorted:

10,13,39,60,92|80,91,52,58,61,79,94,29,82,7,59,37,41,38,12,15,85 ,3,87,20,83 ,68,27,73

We continue in this manner until we finish...

Pseudo code for Insertion sort

- Insertion-Sort(A,n)
  1. For $i = 2$ to n
  2. $\quad k = A[i]$
  3. $\quad j = i - 1$
  4. $\quad$ while $(k < A[j]$ && $1 \leq j)$
  5. $\quad\quad A[j+1] = A[j]$
  6. $\quad\quad j = j - 1$
  7. $\quad A[j+1] = k$
- Note: insertion sort preforms the sorting algorithm "in place". This means that at any given time, at most a constant number (with respect to n) of elements are stored outside the array being sorted.

Pseudo code for Insertion sort

- Insertion-Sort(A,n)
  1. For $i = 2$ to n
  2. $\quad k = A[i]$
  3. $\quad j = i - 1$
  4. $\quad$ while $(k < A[j]$ && $1 \leq j)$
  5. $\qquad A[j+1] = A[j]$
  6. $\qquad j = j - 1$
  7. $\quad A[j+1] = k$
- Note: insertion sort preforms the sorting algorithm "in place". This means that at any given time, at most a constant number (with respect to n) of elements are stored outside the array being sorted.

Correctness of the algorithm

- Proposition: At the end of the iteration (of the for loop) in which $i = m$, the elements in positions 1 to m of the array are the same elements in the original sub-array A[1,...,m] in a sorted order.
- Corollary: The correctness of the algorithm follows immediately form the proposition.

Correctness of the algorithm

- Proposition: At the end of the iteration (of the for loop) in which $i = m$, the elements in positions 1 to m of the array are the same elements in the original sub-array A[1,...,m] in a sorted order.
- Corollary: The correctness of the algorithm follows immediately form the proposition.

Proof of correctness

Proof: We will prove this by induction on n.
If $n = 1$ there is nothing to prove.
For $n = 2$ at the beginning of the first loop, the relevant sub-array has only one element $A[1]$. If A[1] is bigger than $A[2]$ than $A[1]$ is copied to the second position and $A[2]$ is copied to the first position of the array. Otherwise, the array is not changed. In any case, once the iteration is finished the Array A is sorted.

Proof of correctness

Induction step: Now the algorithm is applied on $(a_1, \ldots, a_n, a_{n+1})$. For $i = 1, 2, \ldots, n$ the algorithm is identical to the algorithm applied on $(a_1, \ldots, a_n)$ and hence the proposition follows from the induction hypothesis. It remains to show that at the end of the last iteration, the array is sorted.
This follows form the fact that the algorithm copies all the elements bigger than $A[n+1]$ (in the sorted array in places 1 to n) one place to the right, and puts $A[n+1]$ in the first position such that it is bigger or equal to the element on its left.

Complexity of insertion sort

- An exact analysis of the complexity of an algorithm is complicated and depends on many details. It may be very "machine dependant".
  We therefore, try to analyse the complexity of the algorithm in an abstract way.
  Since We are considering comparison algorithms, we will count the number of comparisons.
- Claim: In the worst case, the Insertion sort algorithm performs $C_{IS}(n) = \binom{n}{2} = \frac{n(n+1)}{2}$ comparisons.

Complexity of insertion sort

- An exact analysis of the complexity of an algorithm is complicated and depends on many details. It may be very "machine dependant".
  We therefore, try to analyse the complexity of the algorithm in an abstract way.
  Since We are considering comparison algorithms, we will count the number of comparisons.
- Claim: In the worst case, the Insertion sort algorithm performs $C_{IS}(n) = \binom{n}{2} = \frac{n(n+1)}{2}$ comparisons.

Complexity of insertion sort

- Proof: On the i-th step, the algorithm performs at most $i-1$ comparisons. Therefore:

$$C_{IS}(n) \leq \sum_{i=2}^{n} i-1 = \sum_{i=1}^{n} i-1 = \frac{n(n-1)}{2}$$

On the other hand for the input $(n, n-1, n-2, ..., 1)$ the algorithm will perform exactly $i-1$ comparisons ont the i-th step and hence:

$$C_{IS}(n) \geq \sum_{i=1}^{n} i-1 = \frac{n(n-1)}{2}$$

- It follows that the complexity of Insertion sort is $\Theta(n^2)$. We now define this notation.

Complexity of insertion sort

- Proof: On the i-th step, the algorithm performs at most $i-1$ comparisons. Therefore:

$$C_{IS}(n) \leq \sum_{i=2}^{n} i-1 = \sum_{i=1}^{n} i-1 = \frac{n(n-1)}{2}$$

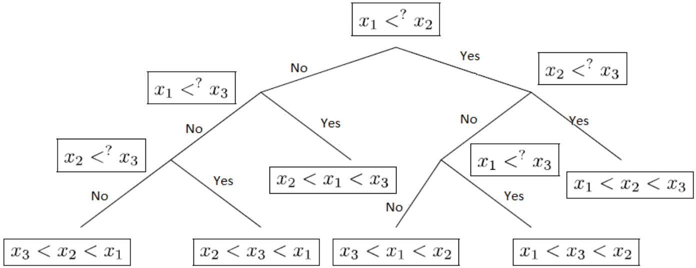
On the other hand for the input $(n, n-1, n-2, ..., 1)$ the algorithm will perform exactly $i-1$ comparisons ont the i-th step and hence:

$$C_{IS}(n) \geq \sum_{i=1}^{n} i-1 = \frac{n(n-1)}{2}$$

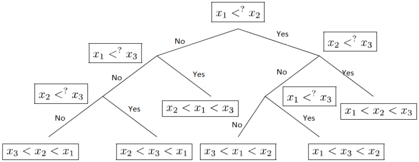
- It follows that the complexity of Insertion sort is $\Theta(n^2)$. We now define this notation.

Decision-tree - an example

- We can describe a comparison algorithm using a comparison tree.
  Here is an example for an array (x_1, x_2, x_3):

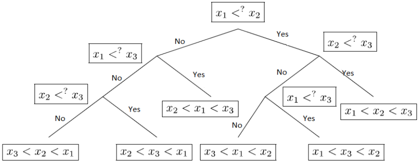


Binary trees



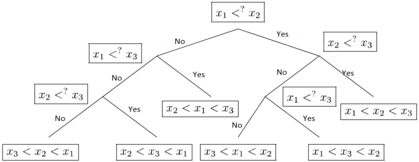
- A binary tree T is a (directed) graph such that the outdegree of each node (vertex) is smaller or equal 2.
- Leafs are nodes with outdegree $= 0$.
- For a node v we denote its left and right children (if they exist) by $left(v)$, $right(v)$.
- The hight of the tree $h(T)$ is the maximal number of edges in a directed simple path from the root to a leaf.

Binary trees



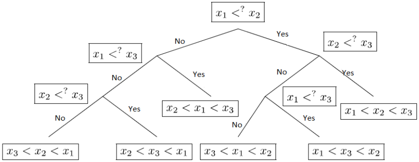
- A binary tree T is a (directed) graph such that the outdegree of each node (vertex) is smaller or equal 2.
- Leafs are nodes with outdegree $= 0$.
- For a node v we denote its left and right children (if they exist) by $left(v)$, $right(v)$.
- The hight of the tree $h(T)$ is the maximal number of edges in a directed simple path from the root to a leaf.

Binary trees



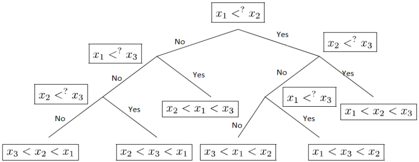
- A binary tree T is a (directed) graph such that the outdegree of each node (vertex) is smaller or equal 2.
- Leafs are nodes with outdegree $= 0$.
- For a node v we denote its left and right children (if they exist) by $left(v)$, $right(v)$.
- The hight of the tree $h(T)$ is the maximal number of edges in a directed simple path from the root to a leaf.

Binary trees



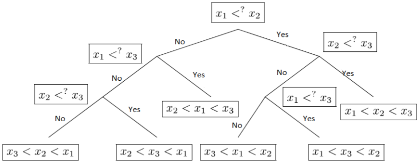
- A binary tree T is a (directed) graph such that the outdegree of each node (vertex) is smaller or equal 2.
- Leafs are nodes with outdegree $= 0$.
- For a node v we denote its left and right children (if they exist) by $left(v)$, $right(v)$.
- The hight of the tree $h(T)$ is the maximal number of edges in a directed simple path from the root to a leaf.

The decision-tree model



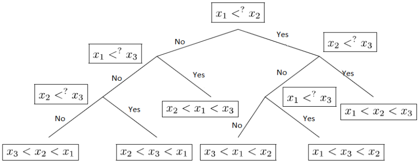
- A decision-tree model is a full binary tree that represents the comparisons preformed by a sorting algorithm (other aspects are ignored).
- Each node corresponds to a comparison $x_i \leq^? x_j$.
- Each leaf corresponds to a permutation (The result of the sorting algorithm).
- The execution of the sorting algorithm corresponds to a path from the root to a leaf.

The decision-tree model



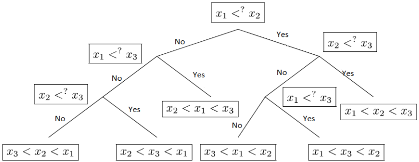
- A decision-tree model is a full binary tree that represents the comparisons preformed by a sorting algorithm (other aspects are ignored).
- Each node corresponds to a comparison $x_i \leq^? x_j$.
- Each leaf corresponds to a permutation (The result of the sorting algorithm).
- The execution of the sorting algorithm corresponds to a path from the root to a leaf.

The decision-tree model



- A decision-tree model is a full binary tree that represents the comparisons preformed by a sorting algorithm (other aspects are ignored).
- Each node corresponds to a comparison $x_i \leq^? x_j$.
- Each leaf corresponds to a permutation (The result of the sorting algorithm).
- The execution of the sorting algorithm corresponds to a path from the root to a leaf.

The decision-tree model



- A decision-tree model is a full binary tree that represents the comparisons preformed by a sorting algorithm (other aspects are ignored).
- Each node corresponds to a comparison $x_i \leq^? x_j$.
- Each leaf corresponds to a permutation (The result of the sorting algorithm).
- The execution of the sorting algorithm corresponds to a path from the root to a leaf.

Complexity of sorting algorithms

- Any (comparison based) sorting algorithm can be represented by a decision-tree.
- The complexity of a sorting algorithm T is the maximal number of comparisons the algorithms perform on an input of length n (that is: sorting $(x_1, x_2, \dots, x_n)$). We denote it by $C_T(n)$.
- The worst case for the number of comparisons an algorithm performs, is the longest path from the root to a leaf in the corresponding decision-tree. **This is the hight of the tree.**
- Since any permutation is possible, the decision making tree must have (at least) $n!$ leafs.
- **The above observations implies the following theorem.**

Complexity of sorting algorithms

- Any (comparison based) sorting algorithm can be represented by a decision-tree.
- The complexity of a sorting algorithm T is the maximal number of comparisons the algorithms perform on an input of length n (that is: sorting $(x_1, x_2, \ldots, x_n)$). We denote it by $C_T(n)$.
- The worst case for the number of comparisons an algorithm performs, is the longest path from the root to a leaf in the corresponding decision-tree. **This is the hight of the tree.**
- Since any permutation is possible, the decision making tree must have (at least) $n!$ leafs.
- **The above observations implies the following theorem.**

Complexity of sorting algorithms

- Any (comparison based) sorting algorithm can be represented by a decision-tree.
- The complexity of a sorting algorithm T is the maximal number of comparisons the algorithms perform on an input of length n (that is: sorting $(x_1, x_2, \ldots, x_n)$). We denote it by $C_T(n)$.
- The worst case for the number of comparisons an algorithm performs, is the longest path from the root to a leaf in the corresponding decision-tree. **This is the hight of the tree.**
- Since any permutation is possible, the decision making tree must have (at least) $n!$ leafs.
- **The above observations implies the following theorem.**

Complexity of sorting algorithms

- Any (comparison based) sorting algorithm can be represented by a decision-tree.
- The complexity of a sorting algorithm T is the maximal number of comparisons the algorithms perform on an input of length n (that is: sorting $(x_1, x_2, ..., x_n)$). We denote it by $C_T(n)$.
- The worst case for the number of comparisons an algorithm performs, is the longest path from the root to a leaf in the corresponding decision-tree. **This is the hight of the tree.**
- Since any permutation is possible, the decision making tree must have (at least) $n!$ leafs.
- **The above observations implies the following theorem.**

Complexity of sorting algorithms

- Any (comparison based) sorting algorithm can be represented by a decision-tree.
- The complexity of a sorting algorithm T is the maximal number of comparisons the algorithms perform on an input of length n (that is: sorting $(x_1, x_2, \ldots, x_n)$). We denote it by $C_T(n)$.
- The worst case for the number of comparisons an algorithm performs, is the longest path from the root to a leaf in the corresponding decision-tree. **This is the hight of the tree.**
- Since any permutation is possible, the decision making tree must have (at least) $n!$ leafs.
- **The above observations implies the following theorem.**

Complexity of sorting algorithms

- **Theorem:** Any comparison sorting algorithm performs $\Omega(n \lg(n))$ comparisons in the worst case.
- proof: Since the worst case number of comparisons that an algorithm performs, corresponds to the longest path from the root to a leaf in its decision-tree, it is enough to give a lower bound for the hight of a decision tree in which any permutation corresponds to some leaf.
  Since a binary tree of hight h has at most 2^h leafs, and there are $n!$ permutations, we must have:

$$n! \leq 2^h$$

  This immediately implies that $h \geq \log_2(n!) = \Omega(n \lg n)$.

Complexity of sorting algorithms

- **Theorem:** Any comparison sorting algorithm performs $\Omega(n\lg(n))$ comparisons in the worst case.
- proof: Since the worst case number of comparisons that an algorithm performs, corresponds to the longest path from the root to a leaf in its decision-tree, it is enough to give a lower bound for the hight of a decision tree in which any permutation corresponds to some leaf.
  Since a binary tree of hight h has at most 2^h leafs, and there are $n!$ permutations, we must have:

  $$n! \leq 2^h$$

  This immediately implies that $h \geq \log_2(n!) = \Omega(n\lg n)$.

A technical remark

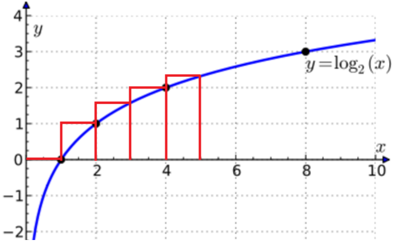
Note: We have

$$\lg(n!) = \lg(1 \cdot 2 \cdot 3 \cdot \ldots \cdot n) = \sum_{k=1}^{n} \lg(k)$$

and:

$$\sum_{k=1}^{n} \lg(k) \geq \int_1^n \log_2(x)dx = \Big[\text{const}(x \log_2(x) - x)\Big]_1^n = \Omega(n \lg n)$$

(Which the following graph explains)



Merge sort

- We will now see another sorting algorithm called merge sort.
- This is a first example of an approach called divide and conquer. Such algorithms typically breaks the problem into smaller similar sub problems, solve the subproblems recursively and then combine the solutions to a solution of the original problem.
- The idea of the merge-sort algorithm is the following:
  1. Divide the sequence to 2 sub-sequences with $\approx \frac{n}{2}$ elements.
  2. Sort each sub-sequence.
  3. Merge the 2 sorted sequences into a sorted sequence.

Merge sort

- We will now see another sorting algorithm called merge sort.
- This is a first example of an approach called divide and conquer. Such algorithms typically breaks the problem into smaller similar sub problems, solve the subproblems recursively and then combine the solutions to a solution of the original problem.
- The idea of the merge-sort algorithm is the following:
  1. Divide the sequence to 2 sub-sequences with $\approx \frac{n}{2}$ elements.
  2. Sort each sub-sequence.
  3. Merge the 2 sorted sequences into a sorted sequence.

Merge sort

- We will now see another sorting algorithm called merge sort.
- This is a first example of an approach called divide and conquer. Such algorithms typically breaks the problem into smaller similar sub problems, solve the subproblems recursively and then combine the solutions to a solution of the original problem.
- The idea of the merge-sort algorithm is the following:
  1. Divide the sequence to 2 sub-sequences with $\approx \frac{n}{2}$ elements.
  2. Sort each sub-sequence.
  3. Merge the 2 sorted sequences into a sorted sequence.

Merge sort

- We will now see another sorting algorithm called merge sort.
- This is a first example of an approach called divide and conquer. Such algorithms typically breaks the problem into smaller similar sub problems, solve the subproblems recursively and then combine the solutions to a solution of the original problem.
- The idea of the merge-sort algorithm is the following:
  1. Divide the sequence to 2 sub-sequences with $\approx \frac{n}{2}$ elements.
  2. Sort each sub-sequence.
  3. Merge the 2 sorted sequences into a sorted sequence.

Merge sort

- We will now see another sorting algorithm called merge sort.
- This is a first example of an approach called divide and conquer. Such algorithms typically breaks the problem into smaller similar sub problems, solve the subproblems recursively and then combine the solutions to a solution of the original problem.
- The idea of the merge-sort algorithm is the following:
    1. Divide the sequence to 2 sub-sequences with $\approx \frac{n}{2}$ elements.
    2. Sort each sub-sequence.
    3. Merge the 2 sorted sequences into a sorted sequence.

Merge sort

- We will now see another sorting algorithm called merge sort.
- This is a first example of an approach called divide and conquer. Such algorithms typically breaks the problem into smaller similar sub problems, solve the subproblems recursively and then combine the solutions to a solution of the original problem.
- The idea of the merge-sort algorithm is the following:
  1. Divide the sequence to 2 sub-sequences with $\approx \frac{n}{2}$ elements.
  2. Sort each sub-sequence.
  3. Merge the 2 sorted sequences into a sorted sequence.

The merging problem

- We begin by considering the problem of merging 2 sorted sequences into a sorted sequence.
- Let $a = (a_1 < a_2 < \dots < a_k)$ and $b = (b_1 < b_2 < \dots < b_l)$ be (disjoint) sorted sequences. The following algorithm merges a (of length k) and b (of length l) into a new sorted sequence c:
- $merge(a, b, k, l, c)$
  1. if $k == l == 0$ stop
  2. if $k == 0$ do $c[j] = b[j]$ for $j = 1, 2, \dots, l$ and stop
  3. if $l == 0$ do $c[j] = a[j]$ for $j = 1, 2, \dots, k$ and stop
  4. if $a[k] < b[l]$
  5. $\quad c[k+l] = b[l]$
  6. $\quad$ merge$(a, b, k, l-1, c)$
  7. if $a[k] > b[l]$
  8. $\quad c[k+l] = a[k]$
  9. $\quad$ merge$(a, b, k-1, l, c)$

The merging problem

- We begin by considering the problem of merging 2 sorted sequences into a sorted sequence.
- Let $a = (a_1 < a_2 < \ldots < a_k)$ and $b = (b_1 < b_2 < \ldots < b_l)$ be (disjoint) sorted sequences. The following algorithm merges a (of length k) and b (of length l) into a new sorted sequence c:
- $merge(a, b, k, l, c)$
  1. if $k == l == 0$ stop
  2. if $k == 0$ do $c[j] = b[j]$ for $j = 1, 2, \ldots, l$ and stop
  3. if $l == 0$ do $c[j] = a[j]$ for $j = 1, 2, \ldots, k$ and stop
  4. if $a[k] < b[l]$
  5. $\quad c[k + l] = b[l]$
  6. $\quad$ merge$(a, b, k, l - 1, c)$
  7. if $a[k] > b[l]$
  8. $\quad c[k + l] = a[k]$
  9. $\quad$ merge$(a, b, k - 1, l, c)$

The merging problem

- We begin by considering the problem of merging 2 sorted sequences into a sorted sequence.
- Let $a = (a_1 < a_2 < ... < a_k)$ and $b = (b_1 < b_2 < ... < b_l)$ be (disjoint) sorted sequences. The following algorithm merges a (of length k) and b (of length l) into a new sorted sequence c:
- *merge*(a, b, k, l, c)
  1. if $k == l == 0$ stop
  2. if $k == 0$ do $c[j] = b[j]$ for $j = 1, 2, ..., l$ and stop
  3. if $l == 0$ do $c[j] = a[j]$ for $j = 1, 2, ..., k$ and stop
  4. if $a[k] < b[l]$
  5. $\quad c[k + l] = b[l]$
  6. $\quad$ merge$(a, b, k, l - 1, c)$
  7. if $a[k] > b[l]$
  8. $\quad c[k + l] = a[k]$
  9. $\quad$ merge$(a, b, k - 1, l, c)$

The merging problem

- We begin by considering the problem of merging 2 sorted sequences into a sorted sequence.
- Let $a = (a_1 < a_2 < ... < a_k)$ and $b = (b_1 < b_2 < ... < b_l)$ be (disjoint) sorted sequences. The following algorithm merges a (of length k) and b (of length l) into a new sorted sequence c:
- $merge(a, b, k, l, c)$
  1. if $k == l == 0$ stop
  2. if $k == 0$ do $c[j] = b[j]$ for $j = 1, 2, ..., l$ and stop
  3. if $l == 0$ do $c[j] = a[j]$ for $j = 1, 2, ..., k$ and stop
  4. if $a[k] < b[l]$
  5. $\quad c[k + l] = b[l]$
  6. $\quad$ merge$(a, b, k, l - 1, c)$
  7. if $a[k] > b[l]$
  8. $\quad c[k + l] = a[k]$
  9. $\quad$ merge$(a, b, k - 1, l, c)$

The complexity of the merging algorithm

- We denote the (worst case) complexity of *merge*(a, b, k, l, c) by $C_M(k, l)$.
- Proposition: If $(k, l) \neq (0, 0)$ then $C_M(k, l) \leq k + l - 1$.
- Proof: We prove by induction on $k + l$.
  If $k = 0$ or $l = 0$ there is nothing to prove.
  Assume $k, l \geq 1$. By the recursive nature of the algorithm we have:

$$C_M(k, l) \leq 1 + max\{C_M(k-1, l), C_M(k, l-1)\}$$

and by the induction hypothesis we know that:

$$\leq 1 + (k + l - 2) = k + l - 1$$

The complexity of the merging algorithm

- We denote the (worst case) complexity of $merge(a, b, k, l, c)$ by $C_M(k, l)$.
- Proposition: If $(k, l) \neq (0, 0)$ then $C_M(k, l) \leq k + l - 1$.
- Proof: We prove by induction on $k + l$.
  If $k = 0$ or $l = 0$ there is nothing to prove.
  Assume $k, l \geq 1$. By the recursive nature of the algorithm we have:

$$C_M(k, l) \leq 1 + max\{C_M(k-1, l), C_M(k, l-1)\}$$

and by the induction hypothesis we know that:

$$\leq 1 + (k + l - 2) = k + l - 1$$

The complexity of the merging algorithm

- We denote the (worst case) complexity of $merge(a, b, k, l, c)$ by $C_M(k, l)$.
- Proposition: If $(k, l) \neq (0, 0)$ then $C_M(k, l) \leq k + l - 1$.
- Proof: We prove by induction on $k + l$.
  If $k = 0$ or $l = 0$ there is nothing to prove.
  Assume $k, l \geq 1$. By the recursive nature of the algorithm we have:

  $$C_M(k, l) \leq 1 + max\{C_M(k - 1, l), C_M(k, l - 1)\}$$

  and by the induction hypothesis we know that:

  $$\leq 1 + (k + l - 2) = k + l - 1$$

  $\blacksquare$

Merge sort

- We use the merge algorithm to construct another sorting algorithm - **Merge sort**.
- $mergeSort(a, n)$
  1. if $n == 1$ stop
  2. $b = (a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor})$
  3. $c = (a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n)$
  4. $mergeSort(b, \lfloor \frac{n}{2} \rfloor)$
  5. $mergeSort(c, \lceil \frac{n}{2} \rceil)$
  6. $merge(b, c, \lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil, a)$

Merge sort

- We use the merge algorithm to construct another sorting algorithm - **Merge sort**.
- $mergeSort(a, n)$
  1. if $n == 1$ stop
  2. $b = (a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor})$
  3. $c = (a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n)$
  4. $mergeSort(b, \lfloor \frac{n}{2} \rfloor)$
  5. $mergeSort(c, \lceil \frac{n}{2} \rceil)$
  6. $merge(b, c, \lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil, a)$

Complexity of merge sort

- Denote the complexity of merge sort by $C_{MS}(n)$.
- Clearly, for $n \geq 2$ we have:

$$C_{MS}(n) \leq C_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil - 1\right)$$

$$= C_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + (n-1)$$

- Corollary: $C_{MS}(2^k) \leq k \cdot 2^k$
  Proof: by induction on k:

$$C_{MS}(2^k) = 2 \cdot C_{MS}(2^{k-1}) + 2^k - 1 \leq$$

$$2 \cdot (k-1)2^{k-1} + 2^k - 1 = k2^k - 1 < k2^k$$

Complexity of merge sort

- Denote the complexity of merge sort by $C_{MS}(n)$.
- Clearly, for $n \geq 2$ we have:

$$C_{MS}(n) \leq C_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil - 1\right)$$

$$= C_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + (n-1)$$

- Corollary: $C_{MS}(2^k) \leq k \cdot 2^k$

  Proof: by induction on k:

$$C_{MS}(2^k) = 2 \cdot C_{MS}(2^{k-1}) + 2^k - 1 \leq$$

$$2 \cdot (k-1)2^{k-1} + 2^k - 1 = k2^k - 1 < k2^k$$

Complexity of merge sort

- Denote the complexity of merge sort by $C_{MS}(n)$.
- Clearly, for $n \geq 2$ we have:

$$C_{MS}(n) \leq C_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil - 1\right)$$

$$= C_{MS}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C_{MS}\left(\left\lceil \frac{n}{2} \right\rceil\right) + (n-1)$$

- Corollary: $C_{MS}(2^k) \leq k \cdot 2^k$
Proof: by induction on k:

$$C_{MS}(2^k) = 2 \cdot C_{MS}(2^{k-1}) + 2^k - 1 \leq$$

$$2 \cdot (k-1)2^{k-1} + 2^k - 1 = k2^k - 1 < k2^k$$

Complexity of merge sort

- Corollary: $C_{MS}(n) \leq 2n\log_2(n) + 2n$
  Proof: let k be such that $2^{k-1} < n \leq 2^k$:

$$C_{MS}(n) \leq C_{MS}(2^k) \leq k2^k \leq \underbrace{(\log_2(2n))}_{k\leq} \underbrace{\cdot 2n}_{2^k\leq}$$

$$= 2n\log_2(n) + 2n$$

- Actually we have:

$$C_{MS}(n) \leq \lceil n\log_2 n \rceil \leq n\log_2 n + n$$

Complexity of merge sort

- Corollary: $C_{MS}(n) \leq 2n\log_2(n) + 2n$
  Proof: let k be such that $2^{k-1} < n \leq 2^k$:

$$C_{MS}(n) \leq C_{MS}(2^k) \leq k2^k \leq \underbrace{(\log_2(2n))}_{k\leq} \underbrace{\cdot 2n}_{2^k\leq}$$

$$= 2n\log_2(n) + 2n$$

- Actually we have:

$$C_{MS}(n) \leq \lceil n log_2 n \rceil \leq n log_2 n + n$$

Recurrence relations

- The divide and conquer approach makes recursive formulas similar to those we saw in the analysis of merge sort appear frequently.
- It is convenient to have a systematic way to estimate such recursive formulas.
- The following example demonstrate the essence of the relevant theorem.

Recurrence relations

- The divide and conquer approach makes recursive formulas similar to those we saw in the analysis of merge sort appear frequently.
- It is convenient to have a systematic way to estimate such recursive formulas.
- The following example demonstrate the essence of the relevant theorem.

Recurrence relations

- The divide and conquer approach makes recursive formulas similar to those we saw in the analysis of merge sort appear frequently.
- It is convenient to have a systematic way to estimate such recursive formulas.
- The following example demonstrate the essence of the relevant theorem.

Recurrence relations - an example

- In the following calculations, $\frac{n}{2}$ means $\left[\frac{n}{2}\right]$
- Example: Let $a \geq 1$. Assume the function $T(n)$ satisfies:

$$T(n) = \begin{cases} n + aT\left(\frac{n}{2}\right), & n \geq 2; \\ 1, & n < 2. \end{cases}$$

Evaluate $T(n)$, for $n = 2^k$.

- Solution: By the recurrence relation we have (for some large n):

$$T(n) = n + aT\left(\frac{n}{2}\right) = n + a\left(\frac{n}{2} + aT\left(\frac{n}{2^2}\right)\right) =$$

$$n + \frac{an}{2} + a^2\left(\frac{n}{2^2} + aT\left(\frac{n}{2^3}\right)\right) = \ldots = \sum_{t=0}^{k-1} n \cdot \left(\frac{a}{2}\right)^t + a^k \underbrace{T\left(\frac{n}{2^k}\right)}_{=1}$$

Recurrence relations - an example

- In the following calculations, $\frac{n}{2}$ means $\left\lceil \frac{n}{2} \right\rceil$
- Example: Let $a \geq 1$. Assume the function $T(n)$ satisfies:

$$T(n) = \begin{cases} n + aT\left(\frac{n}{2}\right), & n \geq 2; \\ 1, & n < 2. \end{cases}$$

Evaluate $T(n)$, for $n = 2^k$.

- Solution: By the recurrence relation we have (for some large n):

$$T(n) = n + aT\left(\frac{n}{2}\right) = n + a\left(\frac{n}{2} + aT\left(\frac{n}{2^2}\right)\right) =$$

$$n + \frac{an}{2} + a^2\left(\frac{n}{2^2} + aT\left(\frac{n}{2^3}\right)\right) = \ldots = \sum_{t=0}^{k-1} n \cdot \left(\frac{a}{2}\right)^t + a^k \underbrace{T\left(\frac{n}{2^k}\right)}_{=1}$$

Recurrence relations - an example

- In the following calculations, $\frac{n}{2}$ means $\left\lceil \frac{n}{2} \right\rceil$
- Example: Let $a \geq 1$. Assume the function $T(n)$ satisfies:

$$T(n) = \begin{cases} n + aT\left(\frac{n}{2}\right), & n \geq 2; \\ 1, & n < 2. \end{cases}$$

Evaluate $T(n)$, for $n = 2^k$.

- Solution: By the recurrence relation we have (for some large n):

$$T(n) = n + aT\left(\frac{n}{2}\right) = n + a\left(\frac{n}{2} + aT\left(\frac{n}{2^2}\right)\right) =$$

$$n + \frac{an}{2} + a^2\left(\frac{n}{2^2} + aT\left(\frac{n}{2^3}\right)\right) = \ldots = \sum_{t=0}^{k-1} n \cdot \left(\frac{a}{2}\right)^t + a^k \underbrace{T\left(\frac{n}{2^k}\right)}_{=1}$$

Recurrence relations - an example

- Case 1: $a = 2$.
  We have:

$$T(n) = \sum_{t=0}^{k-1} \left(\frac{a}{2}\right)^t n + a^k = n \cdot \sum_{t=0}^{k-1} 1 + 2^k =$$

$$n \cdot k + n = n \log_2(n) + n$$

- We see that $T(n) = \Theta(n \log_2 n)$.

Recurrence relations - an example

- Case 1: $a = 2$.
  We have:

$$T(n) = \sum_{t=0}^{k-1} \left(\frac{a}{2}\right)^t n + a^k = n \cdot \sum_{t=0}^{k-1} 1 + 2^k =$$

$$n \cdot k + n = n \log_2(n) + n$$

- We see that $T(n) = \Theta(n \log_2 n)$.

Recurrence relations - an example

- Case 2: $1 \leq a < 2$.
  We have:

$$T(n) = \sum_{t=0}^{k-1} \left(\frac{a}{2}\right)^t n + a^k = n \cdot \underbrace{\frac{1 - \left(\frac{a}{2}\right)^k}{1 - \frac{a}{2}}}_{< \frac{1}{1-\frac{a}{2}}} + \underbrace{a^k}_{<n} \leq$$

$$\underbrace{n \left(\frac{1}{1 - \frac{a}{2}} + 1\right)}_{\text{const}}$$

- Clearly $T(n) \geq n$ (Since $T(n) = n + \ldots$).
- We see that $T(n) = \Theta(n)$.

- Case 2: $1 \leq a < 2$.
  We have:

$$T(n) = \sum_{t=0}^{k-1} \left(\frac{a}{2}\right)^t n + a^k = n \cdot \underbrace{\frac{1-\left(\frac{a}{2}\right)^k}{1-\frac{a}{2}}}_{<\frac{1}{1-\frac{a}{2}}} + \underbrace{a^k}_{<n} \leq$$

$$\underbrace{n\left(\frac{1}{1-\frac{a}{2}} + 1\right)}_{\text{const}}$$

- Clearly $T(n) \geq n$ (Since $T(n) = n + \ldots$).
- We see that $T(n) = \Theta(n)$.

Recurrence relations - an example

- Case 2: $1 \le a < 2$.
  We have:

$$T(n) = \sum_{t=0}^{k-1} \left(\frac{a}{2}\right)^t n + a^k = n \cdot \underbrace{\frac{1-\left(\frac{a}{2}\right)^k}{1-\frac{a}{2}}}_{<\frac{1}{1-\frac{a}{2}}} + \underbrace{a^k}_{<n} \le$$

$$\underbrace{n\left(\frac{1}{1-\frac{a}{2}} + 1\right)}_{\text{const}}$$

- Clearly $T(n) \ge n$ (Since $T(n) = n + \ldots$).
- We see that $T(n) = \Theta(n)$.

Recurrence relations - an example

- Case 3: $a > 2$.
  We have:

$$T(n) = \sum_{t=0}^{k-1} \left(\frac{a}{2}\right)^t n + a^k = n \cdot \frac{\left(\frac{a}{2}\right)^k - 1}{\frac{a}{2} - 1} + a^k \leq$$

$$n \cdot \frac{\left(\frac{a}{2}\right)^k}{\frac{a}{2} - 1} + a^k \leq na^k \left(\frac{\left(\frac{1}{2}\right)^k}{\frac{a}{2} - 1}\right) + a^k = n2^{(\log_2 a) \cdot k} \left(\frac{\frac{1}{n}}{\frac{a}{2} - 1}\right) + a^k$$

$$= \frac{n^{\log_2(a)}}{\frac{a}{2} - 1} + \underbrace{2^{\log_2(a) \cdot k}}_{= n^{\log_2(a)}}$$

- We see that $T(n) = \Theta(n^{\log_2 a})$.

Recurrence relations - an example

- Case 3: $a > 2$.
  We have:

$$T(n) = \sum_{t=0}^{k-1} \left(\frac{a}{2}\right)^t n + a^k = n \cdot \frac{\left(\frac{a}{2}\right)^k - 1}{\frac{a}{2} - 1} + a^k \leq$$

$$n \cdot \frac{\left(\frac{a}{2}\right)^k}{\frac{a}{2} - 1} + a^k \leq na^k \left(\frac{\left(\frac{1}{2}\right)^k}{\frac{a}{2} - 1}\right) + a^k = n2^{(\log_2 a)\cdot k} \left(\frac{\frac{1}{n}}{\frac{a}{2} - 1}\right) + a^k$$

$$= \frac{n^{\log_2(a)}}{\frac{a}{2} - 1} + \underbrace{2^{\log_2(a)\cdot k}}_{= n^{\log_2(a)}}$$

- We see that $T(n) = \Theta(n^{\log_2 a})$.

The Master Theorem

- We now state the Master theorem that generalize the above example.
  Notation: any $\frac{n}{b}$ is either $\lceil \frac{n}{b} \rceil$ or $\lfloor \frac{n}{b} \rfloor$.
- Master theorem: Let $a \geq 1$ and $b > 1$ be constants. Let $f(n)$ be a function and let $T(n)$ be defined by the recurrence relation:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

  Then:
  1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$.
  2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log_2 n)$.
  3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ and if $a \cdot f(\frac{n}{b}) \leq cf(n)$ for some constant c and n sufficiently large, then $T(n) = \Theta(f(n))$.
- Remark: Check that this agrees with the previous example.

The Master Theorem

- We now state the Master theorem that generalize the above example.
  Notation: any $\frac{n}{b}$ is either $\lceil \frac{n}{b} \rceil$ or $\lfloor \frac{n}{b} \rfloor$.
- Master theorem: Let $a \geq 1$ and $b > 1$ be constants. Let $f(n)$ be a function and let $T(n)$ be defined by the recurrence relation:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

  Then:

  1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$.
  2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log_2 n)$.
  3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ and if $a \cdot f(\frac{n}{b}) \leq cf(n)$ for some constant c and n sufficiently large, then $T(n) = \Theta(f(n))$.

- Remark: Check that this agrees with the previous example.

The Master Theorem

- We now state the Master theorem that generalize the above example.
  Notation: any $\frac{n}{b}$ is either $\lceil \frac{n}{b} \rceil$ or $\lfloor \frac{n}{b} \rfloor$.
- Master theorem: Let $a \geq 1$ and $b > 1$ be constants. Let $f(n)$ be a function and let $T(n)$ be defined by the recurrence relation:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

  Then:
  1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$.
  2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log_2 n)$.
  3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ and if $a \cdot f(\frac{n}{b}) \leq cf(n)$ for some constant c and n sufficiently large, then $T(n) = \Theta(f(n))$.
- Remark: Check that this agrees with the previous example.