

Algorithms and Datastructures

Runtime analysis Minsort / Heapsort, Induction

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science
Algorithms and Datastructures, October 2017

Runtime Example

Minsort

Basic Operations

Runtime analysis

Minsort

Heapsort

Introduction to Induction

Logarithms



How long does the program run?

- In the last lecture we had a schematic
- **Observation:** It is going to be “disproportionally” slower the more numbers are being sorted
- How can we say more precisely what is happening?

How can we analyze the runtime?

- Ideally we have a formula which provides the runtime of the program for an specific input
- **Problem:** The runtime is also depending on many other influences, especially:
 - Which kind of computer is the code executed on
 - What is running in the background
 - Which compiler is used to compile the code
- **Abstraction 1:** Analyze the number of basic operations, rather than analyzing the runtime

Incomplete list of basic operations:

- Arithmetic operation, for example: $a + b$
- Assignment of variables, for example: $x = y$
- Function call, for example: *minsort(lst)*

Intuitive:

lines of code

Better:

lines of machine
code

Best:

process cycles

Important:

The actual runtime has to be roughly proportional to the number of operations.

How many operations does *Minsort* need?

- **Abstraction 2:** We calculate the upper (lower) bound, rather than counting the operations exactly

Reason: Runtime is approximated by number of basic operations, but we can still infer:

- Upper bound
 - Lower bound
- **Basic Assumption:**
 - n is size of the input data (i.e. array)
 - $T(n)$ number of operations for input n

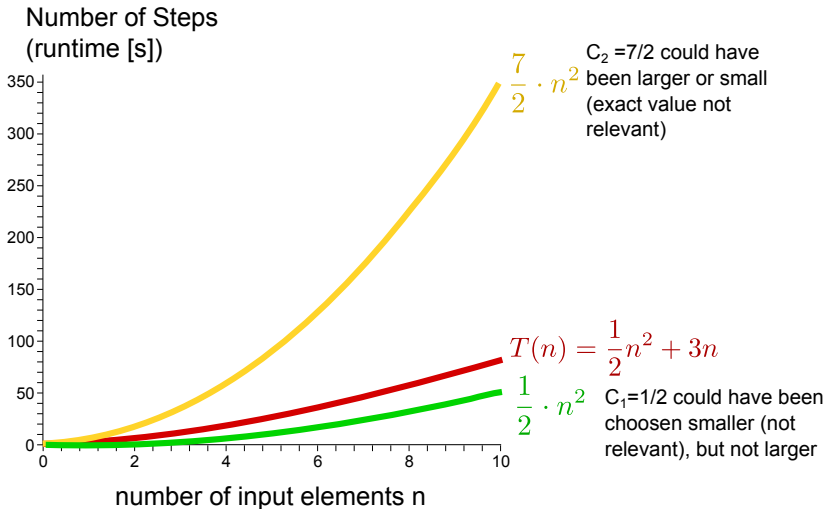
How many operations does *Minsort* need?

- **Observation:** The number of operations depends only on the size n of the array and not on the content!
- **Claim:** There are constants C_1 and C_2 such that:

$$C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$$

- This is called “quadratic runtime” (due to n^2)

Runtime Example



We declare:

- Runtime of operations: $T(n)$
- Number of Elements: n
- Constants: C_1 (lower bound), C_2 (upper bound)

$$C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$$

- Number of operations in round i : T_i

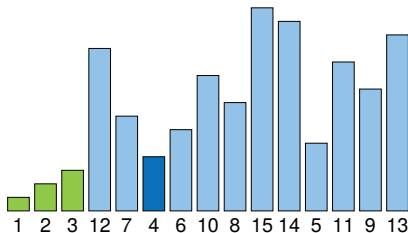


Figure: *Minsort* at the iteration $i = 4$. We have to check $n - 3$ elements

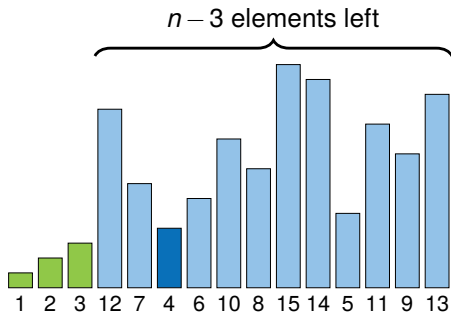


Figure: Minsort at iteration $i = 4$

Compares at each iteration:

$$T_1 \leq C'_2 \cdot (n - 0)$$

$$T_2 \leq C'_2 \cdot (n - 1)$$

$$T_3 \leq C'_2 \cdot (n - 2)$$

$$T_4 \leq C'_2 \cdot (n - 3)$$

\vdots

$$T_{n-1} \leq C'_2 \cdot 2$$

$$T_n \leq C'_2 \cdot 1$$

$$T(n) = C'_2 \cdot (T_1 + \dots + T_n) \leq \sum_{i=1}^n (C'_2 \cdot i)$$

Alternative: Analyse the Code:

```
def minsort(elements):  
    for i in range(0, len(elements)-1):  
        minimum = i  
  
        for j in range(i+1, len(elements)):  
            if elements[j] < elements[minimum]:  
                minimum = j  
  
        if minimum != i:  
            elements[i], elements[minimum] = \  
                elements[minimum], elements[i]  
  
    return elements
```

const. runtime } n-i-1 times } n-1 times

$$T(n) \leq \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} C'_2 = \sum_{i=0}^{n-2} (n-i-1) \cdot C'_2 = \sum_{i=1}^{n-1} (n-i) \cdot C'_2 \leq \sum_{i=1}^n i \cdot C'_2$$

Remark: C'_2 is cost of comparison \Rightarrow assumed constant

Proof of upper bound: $T(n) \leq C_2 \cdot n^2$

$$\begin{aligned} T(n) &\leq \sum_{i=1}^n C'_2 \cdot i \\ &= C'_2 \cdot \sum_{i=1}^n i \\ &\quad \downarrow \text{Small Gauss sum} \\ &= C'_2 \cdot \frac{n(n+1)}{2} \\ &\leq C'_2 \cdot \frac{n(n+n)}{2}, \quad 1 \leq n \\ &= C'_2 \cdot \frac{2 \cdot n^2}{2} = C'_2 \cdot n^2 \end{aligned}$$

Proof of lower bound: $C_1 \cdot n^2 \leq T(n)$

Like for the upper boundary there exists a C_1 . Summation analysis is the same, **only final approximation differs**

$$\begin{aligned} T(n) &\geq \sum_{i=1}^{n-1} C'_1 \cdot (n-i) = C'_1 \sum_{i=1}^{n-1} i \\ &\geq C'_1 \cdot \frac{(n-1) \cdot n}{2} \quad \text{How do we get to } n^2? \\ &\quad \Downarrow \quad n-1 \geq \frac{n}{2} \text{ for } n \geq 2 \\ &\geq C'_1 \cdot \frac{n \cdot n}{2 \cdot 2} = \frac{C'_1}{4} \cdot n^2 \end{aligned}$$

Runtime Analysis:

- Upper bound: $T(n) \leq C'_2 \cdot n^2$
- Lower bound: $\frac{C'_1}{4} \cdot n^2 \leq T(n)$

Summarized:

$$\frac{C'_1}{4} \cdot n^2 \leq T(n) \leq C'_2 \cdot n^2$$

Quadratic runtime proven:

$$C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$$

- The runtime is growing quadratic with the number of elements n in the list
- Let constants C_1 and C_2 for which $C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$
- $2 \times$ elements $\Rightarrow 4 \times$ runtime
 - $C = 1 \text{ ns}$ (1 simple instruction $\approx 1 \text{ ns}$)
 - $n = 10^6$ (1 million numbers = 4 MB with 4 B/number)
 - $C \cdot n^2 = 10^{-9} \text{ s} \cdot 10^{12} = 10^3 \text{ s} = 16.7 \text{ min}$
 - $n = 10^9$ (1 billion numbers = 4 GB)
 - $C \cdot n^2 = 10^{-9} \text{ s} \cdot 10^{18} = 10^9 \text{ s} = 31.7 \text{ years}$
- **Quadratic runtime = “big” problems unsolvable**

Intuitive to extract minimum:

- **Minsort:** To determine the minimum value we have to iterate through all the unsorted elements.
- **Heapsort:** The root node is always the smallest (minheap). We only need to repair a part of the full tree after delete operation.

Formal:

- Let $T(n)$ be the runtime for the *Heapsort* algorithm with n elements
- On the next pages we will proof $T(n) \leq C \cdot n \log_2 n$

Depth of a binary tree:

- **Depth d :** longest path through the tree
- Complete binary tree has $n = 2^d - 1$ nodes
- Example: $d = 4$
 $\Rightarrow n = 2^4 - 1 = 15$

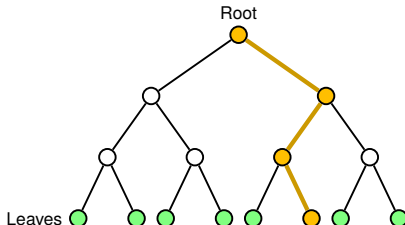


Figure: Binary tree with 15 nodes

Basics:

- You want to show assumption $A(n)$ is valid $\forall n \in \mathbb{N}$
- We show induction in two steps:
 - 1 **Induction basis:** we show that our assumption is valid at one point (for example: $n = 1, A(1)$).
 - 2 **Induction step:** we show that the assumption is valid for all n (normally one step forward: $n = n + 1, A(1), \dots, A(n)$).
- If both has been proven, then $A(n)$ holds for all natural numbers n by **induction**

Claim:

A **complete** binary tree of depth d has $n(d) = 2^d - 1$ nodes

- **Induction basis:** Assumption holds for $d = 1$

Root



$$n(1) = 2^1 - 1 = 1$$

\Rightarrow correct ✓

Figure: Tree of depth 1 has 1 node

Induction - Example 1

Number of nodes $n(d)$ in a binary tree with depth d :

- **Induction assumption:** $n(d) = 2^d - 1$
- **Induction basis:** $n(1) = 2^1 - 1 = 2^1 - 1 = 1$ ✓
- **Induction step:** to show for $d := d + 1$



Figure: Binary tree with subtrees

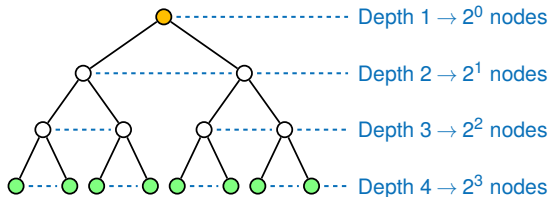
$$\begin{aligned} n(d+1) &= 2 \cdot n(d) + 1 \\ &= 2 \cdot (2^d - 1) + 1 \\ &= 2^{d+1} - 2 + 1 \\ &= 2^{d+1} - 1 \quad \checkmark \end{aligned}$$

⇒ **By induction:** $n(d) = 2^d - 1 \quad \forall n \in \mathbb{N} \quad \square$

Heapsort has the following steps:

- **Initially:** heapify list of n elements
- **Then:** until all n elements are sorted
 - Remove root as minimal element
 - Move last leaf to root position
 - Repair heap by sifting

Runtime of heapify depends on depth d :



Runtime of heapify with depth of d :

- No costs at depth d with 2^{d-1} (or less) nodes
- The cost for sifting with depth 1 is at most $1C$ per node
- In general: Sifting costs are linear with path length **and** number of nodes

Heapify total runtime:



Depth	Nodes	Path length	Costs per node	Upper bound
d	2^{d-1}	0	$\leq C \cdot 0$	$\leq C \cdot 1$
$d-1$	2^{d-2}	1	$\leq C \cdot 1$	Standard $\leq C \cdot 2$
$d-2$	2^{d-3}	2	$\leq C \cdot 2$	Equation $\leq C \cdot 3$
$d-3$	2^{d-4}	3	$\leq C \cdot 3$	$\leq C \cdot 4$

In total:
$$T(d) \leq \sum_{i=1}^d \left(C \cdot (i-1) \cdot 2^{d-i} \right) \leq \sum_{i=1}^d \left(C \cdot i \cdot 2^{d-i} \right)$$

Heapify total runtime:

$$T(d) \leq \underbrace{C \cdot \sum_{i=1}^d (i \cdot 2^{d-i})}_{\text{See next slides}} \leq C \cdot 2^{d+1}$$

- **Hence:** Resulting costs for heapify:

$$T(d) \leq C \cdot 2^{d+1}$$

- **However:** We want costs in relation to n

Heapify total runtime:

$$T(d) \leq C \cdot 2^{d+1}$$

■ A binary tree of depth d has $2^{d+1} - 1 \leq n$ nodes

■ $2^{d-1} - 1$ nodes in full tree
till layer $d - 1$

■ At least 1 node in layer d

■ Equation multiplied by 2^2
 $\Rightarrow 2^{d-1} \cdot 2^2 \leq 2^2 \cdot n$

■ Cost for heapify:
 $\Rightarrow T(n) \leq C \cdot 4 \cdot n$



Figure: Partial binary tree

- We want to proof (induction assumption):

$$\underbrace{\sum_{i=1}^d (i \cdot 2^{d-i})}_{A(d)} \leq \underbrace{2^{d+1}}_{B(d)}$$

- We denote the left side with A , the right side with B

■ **Induction basis:** $d := 1$:

$$A(d) \leq B(d)$$

$$\sum_{i=1}^d (i \cdot 2^{d-i}) \leq 2^{d+1}$$

$$\sum_{i=1}^1 (i \cdot 2^{1-i}) \leq 2^{1+1}$$

$$2^0 \leq 2^2 \quad \checkmark$$

Induction step: ($d := d + 1$):

- **Idea:** Write down right hand formula and try to get $A(d)$ and $B(d)$ out of it

$$A(d) \leq B(d) \quad \Rightarrow \quad A(d+1) \leq B(d+1)$$

$$\sum_{i=1}^{d+1} (i \cdot 2^{d+1-i}) \leq 2^{d+1+1}$$

$$2 \cdot \sum_{i=1}^{d+1} (i \cdot 2^{d-i}) \leq 2 \cdot 2^{d+1}$$

\vdots

Induction step: ($d := d + 1$):

\vdots

$$2 \cdot \sum_{i=1}^{d+1} (i \cdot 2^{d-i}) \leq 2 \cdot 2^{d+1}$$

$$2 \cdot \sum_{i=1}^{d+1} (i \cdot 2^{d-i}) \leq 2 \cdot B(d)$$

$$2 \cdot \sum_{i=1}^d (i \cdot 2^{d-i}) + 2 \cdot (d+1) \cdot 2^{d-(d+1)} \leq 2 \cdot B(d)$$

$$2 \cdot A(d) + (d+1) \leq 2 \cdot B(d)$$

■ **Problem:** Does not work but claim still holds

Working proof:

- Show a **little bit stronger** claim

$$\sum_{i=1}^d (i \cdot 2^{d-i}) \leq 2^{d+1} - d - 2 \leq 2^{d+1}$$

- **Advantage:** Results in a stronger induction assumption
 \Rightarrow **exercise**

Runtime of the other operations:

- Constant costs for taking out $n \times$ maximum
- Maximum of d steps repairing the heap n times
- Depth of heap at the start is $d \leq 1 + \log_2 n$

$$2^{d-1} \leq n \Rightarrow d-1 \leq \log_2 n \Rightarrow d \leq 1 + \log_2 n$$

- **Recall:** The depth and number of elements is decreasing
 - **Hence:** $T(n) \leq n \cdot (1 + \log_2 n) \cdot C$
 - We can reduce this to:

$$T(n) \leq 2 \cdot n \log_2 n \cdot C \quad (\text{holds for } n > 2)$$

Runtime costs:

- Heapify: $T(n) \leq 4 \cdot n \cdot C$
- Remove: $T(n) \leq 2 \cdot n \log_2 n \cdot C$
- Total runtime: $T(n) \leq 6 \cdot n \log_2 n \cdot C$
- Constraints:
 - **Upper bound:** $C_2 \cdot n \log_2 n \geq T(n)$ (for $n \geq 2$)
 - **Lower bound:** $C_1 \cdot n \log_2 n \leq T(n)$ (for $n \geq 2$)
 - $\Rightarrow C_1$ and C_2 are constant

Logarithm to different bases:

$$\log_a n = \frac{\log_b n}{\log_b a} = \log_b n \cdot \frac{1}{\log_b a}$$

The only difference is a constant coefficient $\frac{1}{\log_b a}$

Examples:

- $\log_2 4 = \log_{10} 4 \cdot \frac{1}{\log_2 10} = 0.602 \dots \cdot 3.322 \dots = 2 \checkmark$
- $\log_{10} 1000 = \log_e 1000 \cdot \frac{1}{\log_e 10} = \ln 1000 \cdot \frac{1}{\ln 10} = 3 \checkmark$

Runtime of $n \log_2 n$:

- Assume we have constants C_1 and C_2 with

$$C_1 \cdot n \cdot \log_2 n \leq T(n) \leq C_2 \cdot n \cdot \log_2 n \quad \text{for } n \geq 2$$

- $2 \times$ elements \Rightarrow only slightly larger than $2 \times$ runtime
 - $C = 1$ ns (1 simple instruction ≈ 1 ns)
 - $n = 2^{20}$ (1 million numbers = 4 MB with 4 B/number)
 - $C \cdot n \cdot \log_2 n = 10^{-9} \text{ s} \cdot 2^{20} \cdot 20 = 21.0 \text{ ms}$
 - $n = 2^{30}$ (1 billion numbers = 4 GB)
 - $C \cdot n \cdot \log_2 n = 10^{-9} \text{ s} \cdot 2^{30} \cdot 30 = 32 \text{ s}$
- **Runtime $n \log_2 n$ is nearly as good as linear!**

■ General for this Lecture

[CRL01] Thomas H. Cormen, Ronald L. Rivest, and Charles E. Leiserson.

Introduction to Algorithms.

MIT Press, Cambridge, Mass, 2001.

[MS08] Kurt Mehlhorn and Peter Sanders.

Algorithms and data structures, 2008.

<https://people.mpi-inf.mpg.de/~mehlhorn/ftp/Mehlhorn-Sanders-Toolbox.pdf>.

■ Mathematical Induction

[Wik] [Mathematical induction](https://en.wikipedia.org/wiki/Mathematical_induction)

`https://en.wikipedia.org/wiki/Mathematical_induction`