Algorithms and Data Structures

Runtime analysis Minsort / Heapsort, Induction

Prof. Dr. Rolf Backofen

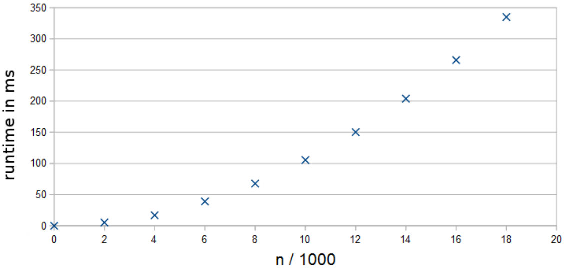
Bioinformatics Group / Department of Computer Science

Algorithms and Data Structures, October 2018

Structure

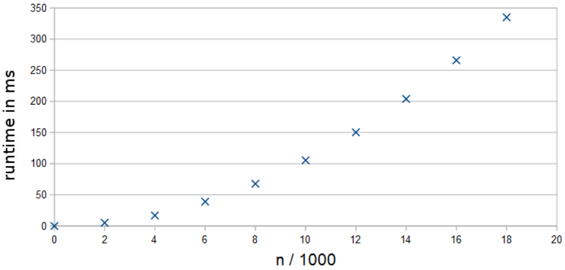
Structure

Runtime analysis - Minsort



How long does the program run?

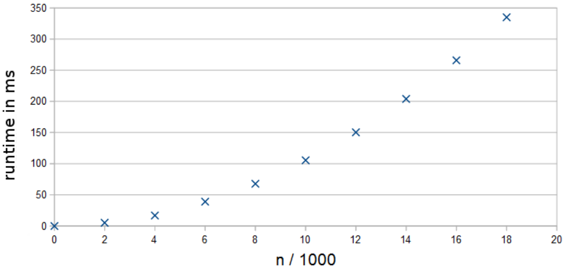
Runtime analysis - Minsort



How long does the program run?

- In the last lecture we had a schematic
- **Observation:** it is going to be "disproportionately" slower the more numbers are being sorted

Runtime analysis - Minsort



How long does the program run?

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

Runtime analysis - Minsort

How can we analyze the runtime?

- Ideally we have a formula which provides the runtime of the program for a specific input

Runtime analysis - Minsort

How can we analyze the runtime?

- Ideally we have a formula which provides the runtime of the program for a specific input
- **Problem:** the runtime is depends on many variables, especially:
  - What kind of computer the code is executed on
  - What is running in the background
  - Which compiler is used to compile the code

Runtime analysis - Minsort

How can we analyze the runtime?

- Ideally we have a formula which provides the runtime of the program for a specific input
- **Problem:** the runtime is depends on many variables, especially:
  - What kind of computer the code is 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

Structure

Basic Operations

Incomplete list of basic operations:

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

Basic Operations

Intuitive:	**Better:**	**Best:**
lines of code	lines of machine code	process cycles

Important:

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

Structure

Runtime analysis - Minsort

How many operations does *Minsort* need?

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

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

  - Upper bound
  - Lower bound

Runtime analysis - Minsort

How many operations does *Minsort* need?

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

  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

Runtime analysis - Minsort

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!

Runtime analysis - Minsort

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:

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

Runtime analysis - Minsort

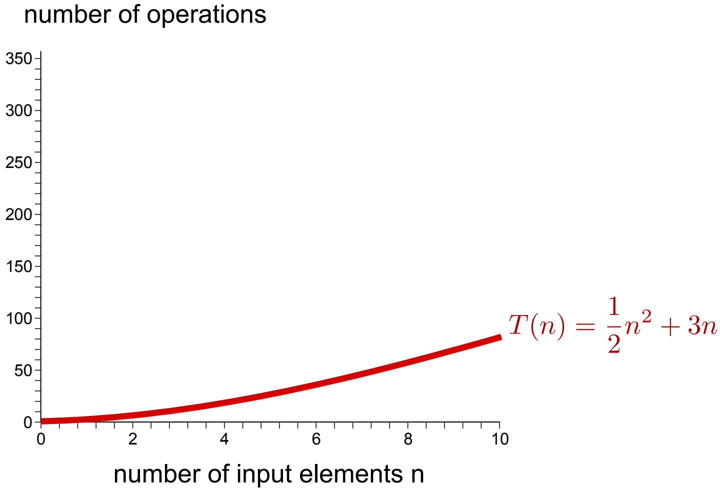
How many operations does *Minsort* need?

- **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:

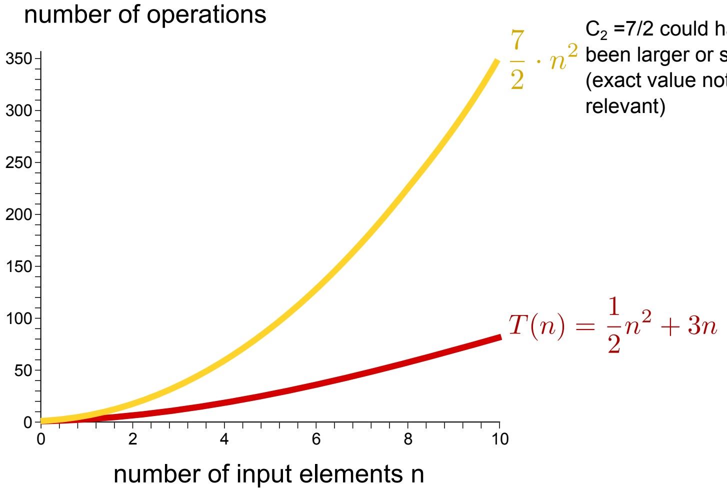
$$C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$$

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

Runtime Example

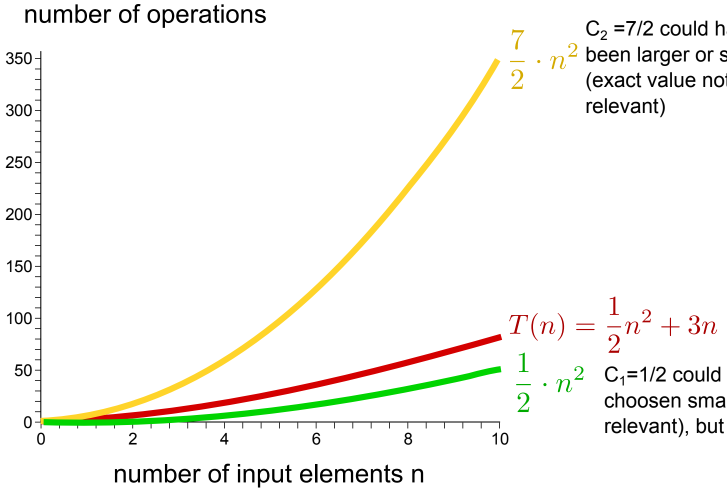


Runtime Example



C_2 =7/2 could have been larger or small (exact value not relevant)

Runtime Example



C_2 =7/2 could have been larger or small (exact value not relevant)

C_1=1/2 could have been choosen smaller (not relevant), but not larger

Runtime analysis - Minsort

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

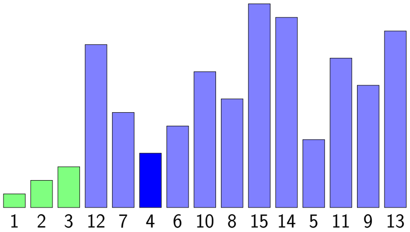


Figure: *Minsort* at iteration $i = 4$. We have to check $n - 3$ elements

Runtime analysis - Minsort

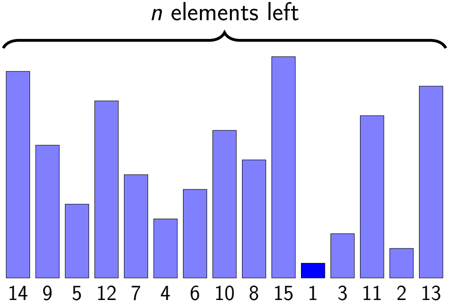


Figure: *Minsort* with start data

Runtime for each iteration:

Runtime analysis - Minsort

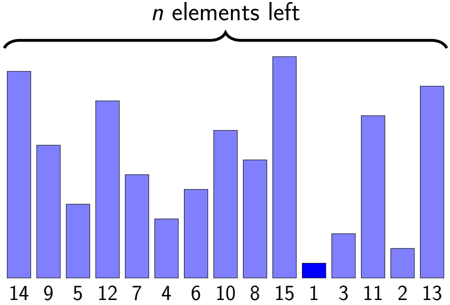


Figure: *Minsort* at iteration $i = 1$

Runtime for each iteration:

$$T_1 \leq C_2' \cdot (n - 0)$$

Runtime analysis - Minsort

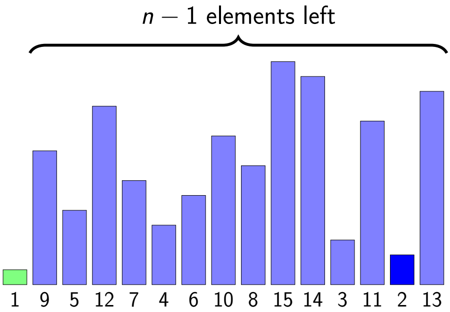


Figure: *Minsort* at iteration $i = 2$

Runtime for each iteration:

$$T_1 \leq C_2' \cdot (n - 0)$$
$$T_2 \leq C_2' \cdot (n - 1)$$

Runtime analysis - Minsort

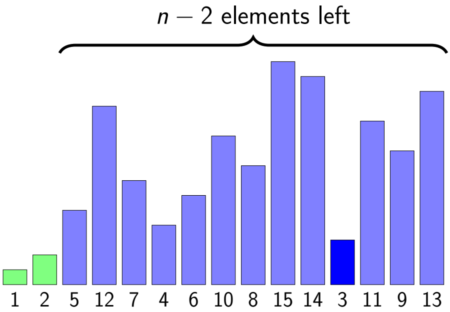


Figure: *Minsort* at iteration $i = 3$

Runtime for each iteration:

$$T_1 \leq C_2' \cdot (n - 0)$$
$$T_2 \leq C_2' \cdot (n - 1)$$
$$T_3 \leq C_2' \cdot (n - 2)$$

Runtime analysis - Minsort

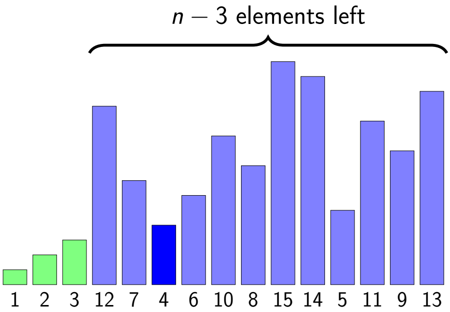


Figure: *Minsort* at iteration $i = 4$

Runtime for each iteration:

$$T_1 \leq C_2' \cdot (n-0)$$
$$T_2 \leq C_2' \cdot (n-1)$$
$$T_3 \leq C_2' \cdot (n-2)$$
$$T_4 \leq C_2' \cdot (n-3)$$

Runtime analysis - Minsort

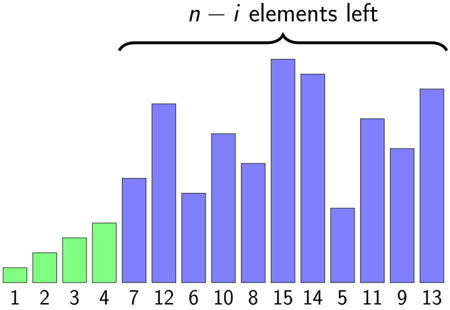


Figure: *Minsort* at iteration i

Runtime for each iteration:

$$
\begin{aligned}
T_1 &\leq C_2' \cdot (n - 0) \\
T_2 &\leq C_2' \cdot (n - 1) \\
T_3 &\leq C_2' \cdot (n - 2) \\
T_4 &\leq C_2' \cdot (n - 3) \\
&\vdots \\
T_{n-1} &\leq C_2' \cdot 2 \\
T_n &\leq C_2' \cdot 1
\end{aligned}
$$

Runtime analysis - Minsort

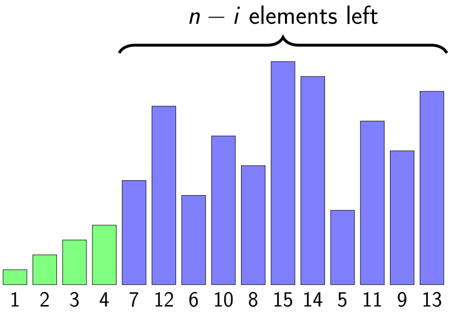


Figure: *Minsort* at iteration

Runtime for each iteration:

$$\begin{aligned} T_1 &\leq C_2' \cdot (n-0) \\ T_2 &\leq C_2' \cdot (n-1) \\ T_3 &\leq C_2' \cdot (n-2) \\ T_4 &\leq C_2' \cdot (n-3) \\ &\vdots \\ T_{n-1} &\leq C_2' \cdot 2 \\ T_n &\leq C_2' \cdot 1 \end{aligned}$$

$$T(n) = (T_1 + \cdots + T_n) \leq \sum_{i=1}^{n} \left(C_2' \cdot i\right)$$

Runtime analysis - Minsort

Alternative: Analyse the Code:

```python
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
```

Runtime analysis - Minsort

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

Runtime analysis - Minsort

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

Runtime analysis - Minsort

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

Runtime analysis - Minsort

Alternative: Analyse the Code:

```python
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

$$T(n) \leq \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} C_2'$$

Runtime analysis - Minsort

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

$$T(n) \leq \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} C_2' = \sum_{i=0}^{n-2} (n - i - 1) \cdot C_2'$$

Runtime analysis - Minsort

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

$$T(n) \leq \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} C_2' = \sum_{i=0}^{n-2} (n-i-1) \cdot C_2' = \sum_{i=1}^{n-1} (n-i) \cdot C_2'$$

Runtime analysis - Minsort

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

$$T(n) \leq \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} C_2' = \sum_{i=0}^{n-2} (n-i-1) \cdot C_2' = \sum_{i=1}^{n-1} (n-i) \cdot C_2' \leq \sum_{i=1}^{n} i \cdot C_2'$$

Runtime analysis - Minsort

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

Runtime analysis - Minsort

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

$$T(n) \;\; \le \;\; \sum_{i=1}^{n} C_2' \cdot i$$

Runtime analysis - Minsort

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

$$\begin{aligned} T(n) &\leq \sum_{i=1}^{n} C_2' \cdot i \\ &= C_2' \cdot \sum_{i=1}^{n} i \end{aligned}$$

Runtime analysis - Minsort

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

$$\begin{aligned} T(n) &\leq \sum_{i=1}^{n} C_2' \cdot i \\ &= C_2' \cdot \sum_{i=1}^{n} i \\ &\Downarrow \quad \text{Small Gauss sum} \\ &= C_2' \cdot \frac{n(n+1)}{2} \end{aligned}$$

Runtime analysis - Minsort

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

$$
\begin{aligned}
T(n) &\leq \sum_{i=1}^{n} C_2' \cdot i \\
&= C_2' \cdot \sum_{i=1}^{n} i \\
&\quad\Downarrow \quad \text{Small Gauss sum} \\
&= C_2' \cdot \frac{n(n+1)}{2} \\
&\leq C_2' \cdot \frac{n(n+n)}{2},\ 1 \leq n
\end{aligned}
$$

Runtime analysis - Minsort

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

$$\begin{aligned}
T(n) &\leq \sum_{i=1}^{n} C_2' \cdot i \\
&= C_2' \cdot \sum_{i=1}^{n} i \\
&\quad \Downarrow \quad \text{Small Gauss sum} \\
&= C_2' \cdot \frac{n(n+1)}{2} \\
&\leq C_2' \cdot \frac{n(n+n)}{2}, \; 1 \leq n \\
&= C_2' \cdot \frac{2 \cdot n^2}{2}
\end{aligned}$$

Runtime analysis - Minsort

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

$$
\begin{aligned}
T(n) &\leq \sum_{i=1}^{n} C_2' \cdot i \\
&= C_2' \cdot \sum_{i=1}^{n} i \\
&\quad\ \Downarrow \quad \text{Small Gauss sum} \\
&= C_2' \cdot \frac{n(n+1)}{2} \\
&\leq C_2' \cdot \frac{n(n+n)}{2},\ 1 \leq n \\
&= C_2' \cdot \frac{2 \cdot n^2}{2} \quad = \quad C_2' \cdot n^2
\end{aligned}
$$

Excursion - Small Gauss Formula

Runtime analysis - Minsort

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

Like for the upper bound there exists a C_1. Summation analysis is the same

$$T(n) \geq \sum_{i=1}^{n-1} C_1' \cdot (n-i)$$

Runtime analysis - Minsort

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

Like for the upper bound there exists a C_1. Summation analysis is the same

$$T(n) \;\; \geq \;\; \sum_{i=1}^{n-1} C_1' \cdot (n-i) \; = \; C_1' \sum_{i=1}^{n-1} i$$

Runtime analysis - Minsort

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

Like for the upper bound there exists a C_1. Summation analysis is the same

$$
\begin{aligned}
T(n) &\geq \sum_{i=1}^{n-1} C_1' \cdot (n-i) = C_1' \sum_{i=1}^{n-1} i \\
&\geq C_1' \cdot \frac{(n-1)\cdot n}{2}
\end{aligned}
$$

Runtime analysis - Minsort

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

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

$$
\begin{aligned}
T(n) &\geq \sum_{i=1}^{n-1} C_1' \cdot (n-i) = C_1' \sum_{i=1}^{n-1} i \\
&\geq C_1' \cdot \frac{(n-1) \cdot n}{2}
\end{aligned}
$$

How do we get to n^2?

Runtime analysis - Minsort

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

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

$$T(n) \geq \sum_{i=1}^{n-1} C_1' \cdot (n-i) = C_1' \sum_{i=1}^{n-1} i$$

$$\geq C_1' \cdot \frac{(n-1) \cdot n}{2} \qquad \text{How do we get to } n^2\text{?}$$

$$\Downarrow \qquad n-1 \geq \frac{n}{2}, \text{ if } n \geq 2$$

Runtime analysis - Minsort

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

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

$$
\begin{aligned}
T(n) \;&\geq\; \sum_{i=1}^{n-1} C_1' \cdot (n-i) \;=\; C_1' \sum_{i=1}^{n-1} i \\
&\geq\; C_1' \cdot \frac{(n-1)\cdot n}{2} \qquad \text{How do we get to } n^2\text{?} \\
&\qquad \Downarrow \qquad n-1 \geq \frac{n}{2},\ \text{if } n \geq 2 \\
&\geq\; C_1' \cdot \frac{n \cdot n}{2 \cdot 2}
\end{aligned}
$$

Runtime analysis - Minsort

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

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

$$
\begin{aligned}
T(n) &\geq \sum_{i=1}^{n-1} C_1' \cdot (n-i) = C_1' \sum_{i=1}^{n-1} i \\
&\geq C_1' \cdot \frac{(n-1) \cdot n}{2} \qquad \text{How do we get to } n^2\text{?} \\
&\Downarrow \qquad n - 1 \geq \frac{n}{2}, \text{ if } n \geq 2 \\
&\geq C_1' \cdot \frac{n \cdot n}{2 \cdot 2} = \frac{C_1'}{4} \cdot n^2
\end{aligned}
$$

Runtime analysis - Minsort

Runtime Analysis:

- Upper bound: $T(n) \leq C_2' \cdot n^2$

Runtime analysis - Minsort

Runtime Analysis:

- Upper bound: $T(n) \leq C_2' \cdot n^2$
- Lower bound: $\frac{C_1'}{4} \cdot n^2 \leq T(n)$

Runtime analysis - Minsort

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:

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

Runtime Example

- The runtime is growing quadratically with the number of elements n in the list

Runtime Example

- The runtime is growing quadratically with the number of elements n in the list
- With constants C_1 and C_2 for which $C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$

Runtime Example

- The runtime is growing quadratically with the number of elements n in the list
- With constants C_1 and C_2 for which $C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$
- $3 \times$ elements $\Rightarrow 9 \times$ runtime

Runtime Example

- The runtime is growing quadratically with the number of elements n in the list
- With constants C_1 and C_2 for which $C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$
- $3\times$ elements $\Rightarrow 9\times$ runtime
  - $C = 1\,\text{ns}$ (1 simple instruction $\approx 1\,\text{ns}$)

Runtime Example

- The runtime is growing quadratically with the number of elements n in the list
- With constants C_1 and C_2 for which $C_1 \cdot n^2 \leq T(n) \leq C_2 \cdot n^2$
- $3 \times$ elements $\Rightarrow 9 \times$ runtime
  - $C = 1\,\text{ns}$ (1 simple instruction $\approx 1\,\text{ns}$)
  - $n = 10^6$ (1 million numbers $= 4\,\text{MB}$ with $4\,\text{B}$/number)
    - $C \cdot n^2 = 10^{-9}\,\text{s} \cdot 10^{12} = 10^3\,\text{s} = 16.7\,\text{min}$

Runtime Example

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

Runtime Example

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

Structure

Runtime - Heapsort

Intuitive to extract minimum:

- **Minsort:** to determine the minimum value we have to iterate through all the unsorted elements.

Runtime - Heapsort

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 the delete operation.

Runtime - Heapsort

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 the delete operation.

Formal:

Runtime - Heapsort

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 the delete operation.

Formal:

- Let $T(n)$ be the runtime for the *Heapsort* algorithm with n elements

Runtime - Heapsort

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 the delete operation.

Formal:

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

Runtime - Heapsort

Depth of a binary tree:

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

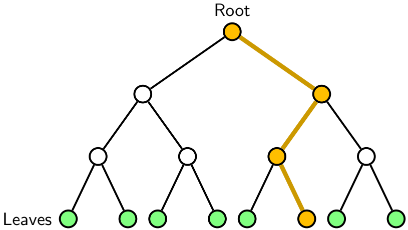


Figure: Binary tree with 15 nodes

Structure

Induction

Basics:

Induction

Basics:

- You want to show that assumption $A(n)$ is valid $\forall n \in \mathbb{N}$

Induction

Basics:

- You want to show that assumption $A(n)$ is valid $\forall n \in \mathbb{N}$
- We show induction in two steps:

Induction

Basics:

- You want to show that 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 for one value (for example: $n = 1$, $A(1)$).

Induction

Basics:

- You want to show that 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 for one value (for example: $n = 1$, $A(1)$).
    2. **Induction step:** we show that the assumption is valid for all n (normally one step forward: $n = n + 1, A(1), \ldots, A(n)$).

Induction

Basics:

- You want to show that 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 for one value (for example: $n = 1$, $A(1)$).
  2. **Induction step:** we show that the assumption is valid for all n (normally one step forward: $n = n + 1, A(1), \ldots, A(n)$).
- If both has been proven, then $A(n)$ holds for all natural numbers n by **induction**

Induction - Example 1

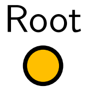
Claim:
A **complete** binary tree of depth d has $v(d) = 2^d - 1$ nodes

Induction - Example 1

Claim:

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

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



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

Figure: Tree of depth 1 has 1 node

Induction - Example 1

Claim:

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

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

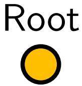


Figure: Tree of depth 1 has 1 node

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

$\Rightarrow$ correct ✓

Induction - Example 1

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

- **Induction assumption:** $v(d) = 2^d - 1$

Induction - Example 1

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

- **Induction assumption:** $v(d) = 2^d - 1$
- **Induction basis:** $v(1) = 2^d - 1 = 2^1 - 1 = 1$ ✓

Induction - Example 1

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

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

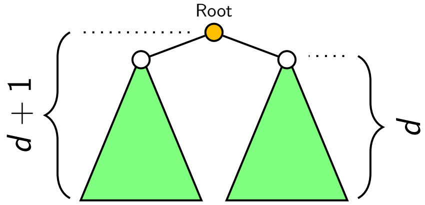


Figure: binary tree with subtrees

Induction - Example 1

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

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

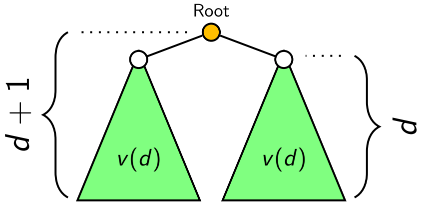


Figure: binary tree with subtrees

$$v(d+1) = 2 \cdot v(d) + 1$$

Induction - Example 1

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

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

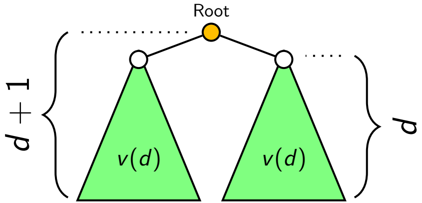


Figure: binary tree with subtrees

$$
\begin{aligned}
v(d+1) &= 2 \cdot v(d) + 1 \\
&= 2 \cdot \left(2^d - 1\right) + 1
\end{aligned}
$$

Induction - Example 1

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

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

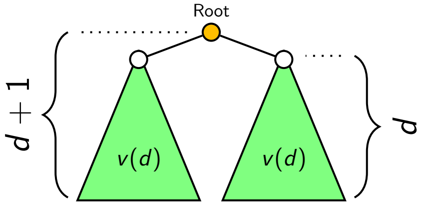


Figure: binary tree with subtrees

$$
\begin{aligned}
v(d+1) &= 2 \cdot v(d) + 1 \\
&= 2 \cdot \left(2^d - 1\right) + 1 \\
&= 2^{d+1} - 2 + 1
\end{aligned}
$$

Induction - Example 1

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

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

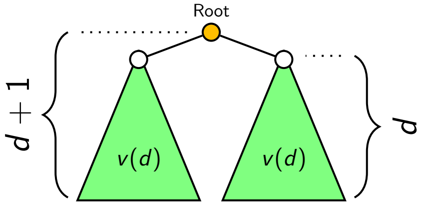


Figure: binary tree with subtrees

$$
\begin{aligned}
v(d+1) &= 2 \cdot v(d) + 1 \\
&= 2 \cdot \left(2^d - 1\right) + 1 \\
&= 2^{d+1} - 2 + 1 \\
&= 2^{d+1} - 1 \checkmark
\end{aligned}
$$

Induction - Example 1

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

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

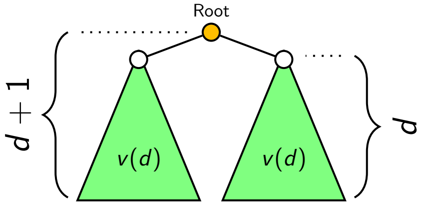


Figure: binary tree with subtrees

$$\begin{aligned} v(d+1) &= 2 \cdot v(d) + 1 \\ &= 2 \cdot \left(2^d - 1\right) + 1 \\ &= 2^{d+1} - 2 + 1 \\ &= 2^{d+1} - 1 \checkmark \end{aligned}$$

$\Rightarrow$ **By induction:**

$$v(d) = 2^d - 1 \; \forall d \in \mathbb{N} \; \square$$

Structure

Runtime - Heapsort

Heapsort has the following steps:

- **Initially:** `heapify` list of n elements

Runtime - Heapsort

Heapsort has the following steps:

- **Initially:** `heapify` list of n elements
- **Then:** until all n elements are sorted

Runtime - Heapsort

Heapsort has the following steps:

- **Initially:** `heapify` list of n elements
- **Then:** until all n elements are sorted
  - Remove root (=minimum element)

Runtime - Heapsort

Heapsort has the following steps:

- **Initially:** `heapify` list of n elements
- **Then:** until all n elements are sorted
  - Remove root (=minimum element)
  - Move last leaf to root position

Runtime - Heapsort

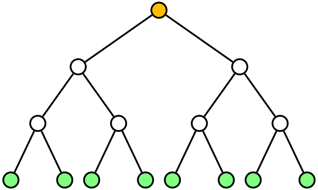
Heapsort has the following steps:

- **Initially:** `heapify` list of n elements
- **Then:** until all n elements are sorted
  - Remove root (=minimum element)
  - Move last leaf to root position
  - Repair heap by `sifting`

Runtime - Heapsort

Heapify

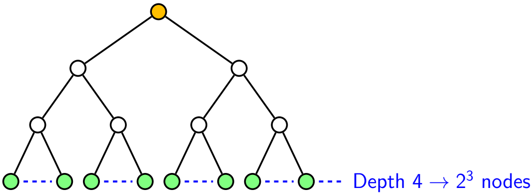
Runtime of `heapify` depends on depth d:



Runtime - Heapsort

Heapify

Runtime of `heapify` depends on depth d:



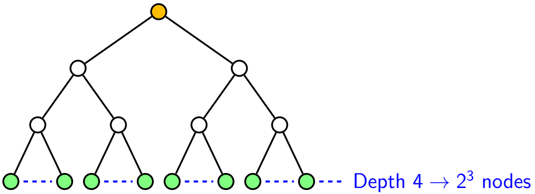
Runtime of `heapify` with depth of d:

- No costs at depth d with 2^{d-1} (or less) nodes

Runtime - Heapsort

Heapify

Runtime of `heapify` depends on depth d:



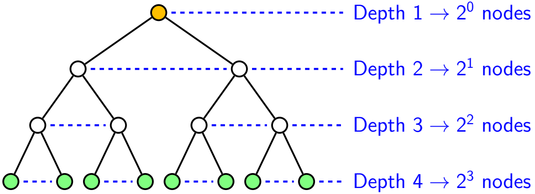
Runtime of `heapify` with depth of d:

- 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

Runtime - Heapsort

Heapify

Runtime of `heapify` depends on depth d:



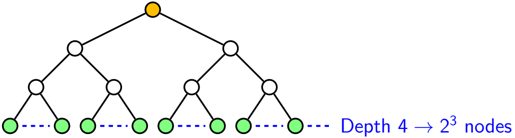
Runtime of `heapify` with depth of d:

- 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

Runtime - Heapsort

Heapify

`Heapify` total runtime:

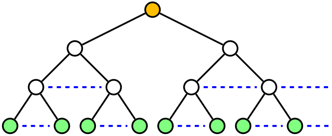


Depth	Nodes	Path length	Costs per node
d	2^{d-1}	0	$\leq C \cdot 0$

Runtime - Heapsort

Heapify

`Heapify` total runtime:

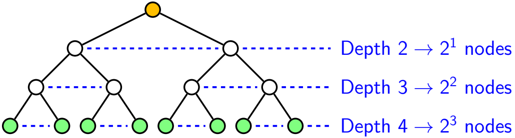


Depth	Nodes	Path length	Costs per node	
d	2^{d-1}	0	$\leq C \cdot 0$	
$d-1$	2^{d-2}	1	$\leq C \cdot 1$	

Runtime - Heapsort

Heapify

`Heapify` total runtime:

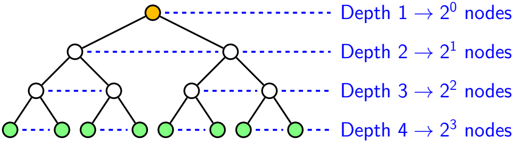


Depth	Nodes	Path length	Costs per node	
d	2^{d-1}	0	$\leq C \cdot 0$	
$d-1$	2^{d-2}	1	$\leq C \cdot 1$	
$d-2$	2^{d-3}	2	$\leq C \cdot 2$	

Runtime - Heapsort

Heapify

`Heapify` total runtime:



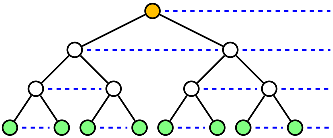
Generally: Depth $d \to 2^{d-1}$ nodes

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

Runtime - Heapsort

Heapify

`Heapify` total runtime:



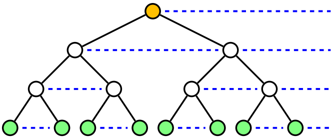
Depth	Nodes	Path length	Costs per node	
d	2^{d-1}	0	$\leq C \cdot 0$	
$d-1$	2^{d-2}	1	$\leq C \cdot 1$	
$d-2$	2^{d-3}	2	$\leq C \cdot 2$	
$d-3$	2^{d-4}	3	$\leq C \cdot 3$	

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

Runtime - Heapsort

Heapify

`Heapify` total runtime:

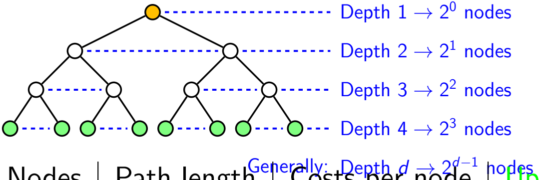


Depth	Nodes	Path length	Costs per node	Upper bound
d	2^{d-1}	0	$\leq C \cdot 0$	
$d-1$	2^{d-2}	1	$\leq C \cdot 1$	Standard
$d-2$	2^{d-3}	2	$\leq C \cdot 2$	Equation
$d-3$	2^{d-4}	3	$\leq C \cdot 3$	

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

Runtime - Heapsort
Heapify

`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	$\leq C \cdot 1$	$\leq C \cdot 2$
$d-2$	2^{d-3}	2	$\leq C \cdot 2$	$\leq C \cdot 3$
$d-3$	2^{d-4}	3	$\leq C \cdot 3$	$\leq C \cdot 4$

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

Runtime - Heapsort

Heapify

`Heapify` total runtime:

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

Runtime - Heapsort

Heapify

`Heapify` total runtime:

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

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

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

Runtime - Heapsort

Heapify

`Heapify` total runtime:

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

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

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

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

Runtime - Heapsort

Heapify

`Heapify` total runtime:

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

Runtime - Heapsort

Heapify

`Heapify` total runtime:

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

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

Runtime - Heapsort

Heapify

`Heapify` total runtime:

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

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

Runtime - Heapsort

Heapify

`Heapify` total runtime:

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

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

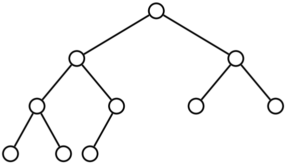


Figure: Partial binary tree

Runtime - Heapsort

Heapify

`Heapify` total runtime:

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

- A binary tree of depth d has $2^{d-1} \leq n$ nodes Why?
- $2^{d-1} - 1$ nodes in full tree till layer $d - 1$

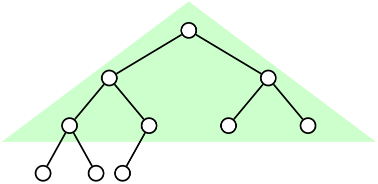


Figure: Partial binary tree

Runtime - Heapsort
Heapify

`Heapify` total runtime:

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

- A binary tree of depth d has $2^{d-1} \leq n$ nodes Why?
- $2^{d-1} - 1$ nodes in full tree till layer $d - 1$
- At least 1 node in layer d

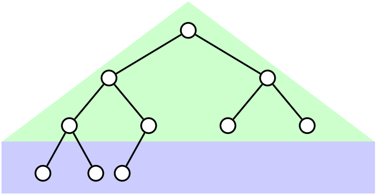


Figure: Partial binary tree

Runtime - Heapsort

Heapify

`Heapify` total runtime:

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

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

- $2^{d-1} - 1$ nodes in full tree till layer $d - 1$
- At least 1 node in layer d

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

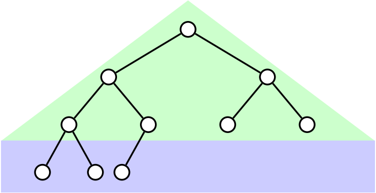


Figure: Partial binary tree

Runtime - Heapsort

Heapify

`Heapify` total runtime:

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

- A binary tree of depth d has $2^{d-1} \leq n$ nodes Why?
- $2^{d-1} - 1$ 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`:
  $\Rightarrow T(n) \leq C \cdot 4 \cdot n$

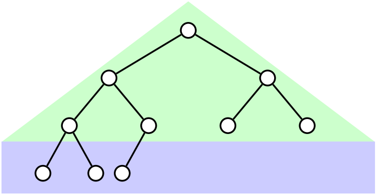


Figure: Partial binary tree

Structure

Induction - Example 2

- **We want to proof (induction assumption):**

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

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

Induction - Example 2

- **Induction basis:** $d := 1$:

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

Induction - Example 2

- **Induction basis:** $d := 1$:

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

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

Induction - Example 2

- **Induction basis:** $d := 1$:

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

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

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

Induction - Example 2

- **Induction basis:** $d := 1$:

$$\begin{aligned} A(d) &\leq B(d) \\ \sum_{i=1}^{d} \left(i \cdot 2^{d-i}\right) &\leq 2^{d+1} \\ \sum_{i=1}^{1} \left(i \cdot 2^{1-i}\right) &\leq 2^{1+1} \\ 2^0 &\leq 2^2 \checkmark \end{aligned}$$

Induction - Example 2

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

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

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

Induction - Example 2

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

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

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

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

Induction - Example 2

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

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

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

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

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

$$\vdots$$

Induction - Example 2

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

$$\vdots$$

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

Induction - Example 2

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

$$\vdots$$

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

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

Induction - Example 2

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

$$\vdots$$

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

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

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

Induction - Example 2

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

$$\vdots$$

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

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

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

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

Induction - Example 2

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

$$\vdots$$

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

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

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

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

- **Problem:** does not work but claim still holds

Induction - Example 2

Working proof:

- Show a little bit stronger claim

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

Induction - Example 2

Working proof:

- Show a little bit stronger claim

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

- **Advantage:** results in a stronger induction assumption

$\Rightarrow$ exercise

Structure

Runtime of the other operations:

Runtime of the other operations:

- $n \times$ taking out maximum (each constant cost)

Runtime of the other operations:

- $n \times$ taking out maximum (each constant cost)
- Maximum of d steps for each of $n \times$ heap repair

Runtime of the other operations:

- $n \times$ taking out maximum (each constant cost)
- Maximum of d steps for each of $n \times$ heap repair

  $\Rightarrow$ Depth d of initial heap is $\leq 1 + \log_2 n$ Why?

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

Runtime of the other operations:

- $n \times$ taking out maximum (each constant cost)
- Maximum of d steps for each of $n \times$ heap repair

  $\Rightarrow$ Depth d of initial heap is $\leq 1 + \log_2 n$ Why?

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

- **Recall**: the depth and number of elements is decreasing

Runtime of the other operations:

- $n \times$ taking out maximum (each constant cost)
- Maximum of d steps for each of $n \times$ heap repair

  $\Rightarrow$ Depth d of initial heap is $\leq 1 + \log_2 n$ Why?

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

- **Recall**: the depth and number of elements is decreasing
  - **Hence**: $T(n) \leq n \cdot d \cdot C \leq n \cdot (1 + \log_2 n) \cdot C$

Runtime of the other operations:

- $n \times$ taking out maximum (each constant cost)
- Maximum of d steps for each of $n \times$ heap repair

  $\Rightarrow$ Depth d of initial heap is $\leq 1 + \log_2 n$ Why?

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

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

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

Runtime - Heapsort

Runtime costs:

- `Heapify`: $T(n) \leq 4 \cdot n \cdot C$

Runtime - Heapsort

Runtime costs:

- `Heapify`: $T(n) \leq 4 \cdot n \cdot C$
- `Remove`: $T(n) \leq 2 \cdot n \log_2 n \cdot C$

Runtime - Heapsort

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: $T(n) \leq 6 \cdot n \log_2 n \cdot C$

Runtime - Heapsort

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 $n \geq 2$)

Runtime - Heapsort

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

Structure

Base of Logarithms

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:

- $\log_2 4 = \log_{10} 4 \cdot \frac{1}{\log_2 10} = 0.602 \ldots \cdot 3.322 \ldots = 2$ ✓
- $\log_{10} 1000 = \log_{\mathrm{e}} 1000 \cdot \frac{1}{\log_{\mathrm{e}} 10} = \ln 1000 \cdot \frac{1}{\ln 10} = 3$ ✓

Runtime Example

Runtime of $n \log_2 n$:

- Assume we have constants C_1 and C_2 with

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

Runtime Example

Runtime of $n\ \log_2 n$:

- Assume we have constants C_1 and C_2 with

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

- $2 \times$ elements $\Rightarrow$ only slightly larger than $2 \times$ runtime

Runtime Example

Runtime of $n\ \log_2 n$:

- Assume we have constants C_1 and C_2 with

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

- $2\times$ elements $\Rightarrow$ only slightly larger than $2\times$ runtime
  - $C = 1\,\text{ns}$ (1 simple instruction $\approx 1\,\text{ns}$)

Runtime Example

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
  - $C = 1\,\text{ns}$ (1 simple instruction $\approx 1\,\text{ns}$)
  - $n = 2^{20}$ (1 million numbers $= 4\,\text{MB}$ with $4\,\text{B/number}$)
    - $C \cdot n \cdot log_2 n = 10^{-9}\,\text{s} \cdot 2^{20} \cdot 20 = 21.0\,\text{ms}$

Runtime Example

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
  - $C = 1\,\text{ns}$ (1 simple instruction $\approx 1\,\text{ns}$)
  - $n = 2^{20}$ (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 Example

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
  - $C = 1\,\text{ns}$ (1 simple instruction $\approx 1\,\text{ns}$)
  - $n = 2^{20}$ (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!**

Further Literature

- **Course literature**

[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`.

Further Literature

- **Mathematical Induction**

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