Algorithms and Data Structures

Analyzing Algorithms: Runnning Time

Britta Peis

Chair of Management Science

DDS Winter Semester 2018/19

Analyzing algorithms: key points

Correctness:

Does the algorithm terminate for each input with a correct answer?

Analyzing algorithms: key points

Correctness:

Does the algorithm terminate for each input with a correct answer?

Efficiency:

How much computation time and space is required?

Analyzing algorithms: key points

Correctness:

Does the algorithm terminate for each input with a correct answer?

Efficiency:

How much computation time and space is required?

Data structure:

How is the data stored and organized?

Data structures

A data structure is a way to store and organize data in order to facilitate access and modifications.

Data structures

A data structure is a way to store and organize data in order to facilitate access and modifications.

No single data structure works well for all purposes, and so it is important to know the strengths and limitations of them.

Data structures

A data structure is a way to store and organize data in order to facilitate access and modifications.

No single data structure works well for all purposes, and so it is important to know the strengths and limitations of them.

Examples of simple data structure:

- arrays,
- stacks and queues,
- linked lists,
- hash tables,
- search trees, ...

$\longrightarrow$ defined and discussed later.

Running time

The running time of an algorithm depends on the input.

Running time

The running time of an algorithm depends on the input.

For example, insertion sort is faster on inputs that are already sorted.

Running time

The running time of an algorithm depends on the input.

For example, insertion sort is faster on inputs that are already sorted.

In particular, the running time depends on the input size, like

- the number of items (sorting),
- the number of vertices and edges (graph algorithms),
- the number of jobs that needs be assigned to resources (scheduling)
- …

Running time

The running time of an algorithm depends on the input.

For example, insertion sort is faster on inputs that are already sorted.

In particular, the running time depends on the input size, like

- the number of items (sorting),
- the number of vertices and edges (graph algorithms),
- the number of jobs that needs be assigned to resources (scheduling)
- ...

Generally, we seek upper bounds on the running time of an algorithm.

$\longrightarrow$ Guarantees!

Different kinds of analysis

Worst-case: (usually)

- $T(n) =$ maximal number of elementary steps the algorithm needs on an input of size n.

Different kinds of analysis

Worst-case: (usually)

- $T(n) =$ maximal number of elementary steps the algorithm needs on an input of size n.

Average-case: (sometimes)

- $T(n) =$ expected number of elementary steps the algorithm needs on an input of size n.
- Requires information on statistical distribution of inputs.

Different kinds of analysis

Worst-case: (usually)

- $T(n) =$ maximal number of elementary steps the algorithm needs on an input of size n.

Average-case: (sometimes)

- $T(n) =$ expected number of elementary steps the algorithm needs on an input of size n.
- Requires information on statistical distribution of inputs.

Best-case: (bogus)

- $T(n) =$ minimal number of elementary steps the algorithm needs on an input of size n.
- Cheat with a slow algorithm that works fast on <u>some</u> input.

Insertion sort: best- and worst-case analysis

Best-case running time

- Best case: the sequence is already sorted.

Insertion sort: best- and worst-case analysis

Best-case running time

- Best case: the sequence is already sorted.
- Each item requires only one comparison $\Rightarrow$ $n - 1$ comparisons in total.

Insertion sort: best- and worst-case analysis

Best-case running time

- Best case: the sequence is already sorted.
- Each item requires only one comparison $\Rightarrow n - 1$ comparisons in total.

Worst-case running time

- Worst case: each item needs to be inserted at the left-most position, so that each item needs to be compared with every predecessor.

Insertion sort: best- and worst-case analysis

Best-case running time

- Best case: the sequence is already sorted.
- Each item requires only one comparison $\Rightarrow n-1$ comparisons in total.

Worst-case running time

- Worst case: each item needs to be inserted at the left-most position, so that each item needs to be compared with every predecessor.
- That is, insertion of the i-th item requires $i-1$ comparisons.
- Thus, $1+2+\ldots+n-1=\frac{n(n-1)}{2}$ comparisons in total.

Insertion sort: best- and worst-case analysis

Best-case running time

- Best case: the sequence is already sorted.
- Each item requires only one comparison $\Rightarrow n - 1$ comparisons in total.

Worst-case running time

- Worst case: each item needs to be inserted at the left-most position, so that each item needs to be compared with every predecessor.
- That is, insertion of the i-th item requires $i - 1$ comparisons.
- Thus, $1 + 2 + \ldots + n - 1 = \frac{n(n-1)}{2}$ comparisons in total.

Remark

Insertion sort does not require any additional space.

Machine-independent analysis

What is the worst-case running time of insertion sort?

$\longrightarrow$ depends certainly on the speed of the computer!

Machine-independent analysis

What is the worst-case running time of insertion sort?

$\longrightarrow$ depends certainly on the speed of the computer!

Big idea:

- Ignore machine-dependent constants.
- Analyse the growth of $T(n)$ as $n \to \infty$.

Machine-independent analysis

What is the worst-case running time of insertion sort?

$\longrightarrow$ depends certainly on the speed of the computer!

Big idea:

- Ignore machine-dependent constants.
- Analyse the growth of $T(n)$ as $n \to \infty$.

$\longrightarrow$ Asymptotic Analysis

Machine-independent analysis

What is the worst-case running time of insertion sort?

$\longrightarrow$ depends certainly on the speed of the computer!

Big idea:

- Ignore machine-dependent constants.
- Analyse the growth of $T(n)$ as $n \to \infty$.

$\longrightarrow$ Asymptotic Analysis

Asymptotic analysis provides upper, lower, and tight bounds on the asymptotic growth of $T(n)$ using

- O-notation (”Big-Oh”),
- Ω-notation (”Big-Omega”), and
- Θ-notation (”Big-Omega”).

$O-, \Omega-,$ and Θ-notation are defined later.

Common running times

Actual running times corresponding to $T(n)$:

Input size n	Running time function $T(n)$:				
	$33n$	$46n\log(n)$	$13n^2$	$3,4n^3$	2^n
10	$0,00033s$	$0,0015s$	$0,0013s$	$0,0034s$	$0,001s$
10^2	$0,0033s$	$0,03s$	$0,13s$	$3,4s$	$4 \cdot 10^{16}y$
10^3	$0,033s$	$0,45s$	$13s$	$0,94h$	
10^4	$0,33s$	$6,1s$	$1300s$	$39d$	
10^5	$3,3s$	$1,3m$	$1,5d$	$108y$	

s	seconds
h	hours
d	days
y	years

Assumption: 1.000.000 operations per second

Common running times

Actual running times corresponding to $T(n)$:

Input size n	Running time function $T(n)$:				
	$33n$	$46n\log(n)$	$13n^2$	$3,4n^3$	2^n
10	$0,00033s$	$0,0015s$	$0,0013s$	$0,0034s$	$0,001s$
10^2	$0,0033s$	$0,03s$	$0,13s$	$3,4s$	$4 \cdot 10^{16}y$
10^3	$0,033s$	$0,45s$	$13s$	$0,94h$	
10^4	$0,33s$	$6,1s$	$1300s$	$39d$	
10^5	$3,3s$	$1,3m$	$1,5d$	$108y$	

s seconds
h hours
d days
y years

Assumption: 1.000.000 operations per second

Observation:

The impact of constant factors decreases with growing n.

Impact of faster computers

Even a super computer cannot save a badly designed algorithm.

Impact of faster computers

Even a super computer cannot save a badly designed algorithm.

For large enough instances, the better algorithm on the slower computer always beats the worse algorithm on a faster computer!

Impact of faster computers

Even a super computer cannot save a badly designed algorithm.

For large enough instances, the better algorithm on the slower computer always beats the worse algorithm on a faster computer!

Example

- Let's say that a certain algorithm with running time $T(n)$ implemented on our computer is able to solve instances of size N within one hour.

Impact of faster computers

Even a super computer cannot save a badly designed algorithm.

For large enough instances, the better algorithm on the slower computer always beats the worse algorithm on a faster computer!

Example

- Let's say that a certain algorithm with running time $T(n)$ implemented on our computer is able to solve instances of size N within one hour.
- Suppose we get a new computer being K-times faster.

Impact of faster computers

Even a super computer cannot save a badly designed algorithm.

For large enough instances, the better algorithm on the slower computer always beats the worse algorithm on a faster computer!

Example

- Let's say that a certain algorithm with running time $T(n)$ implemented on our computer is able to solve instances of size N within one hour.
- Suppose we get a new computer being K-times faster.
- Q: How large are the instances the K-times faster algorithm can solve within one hour for $T(n) \in \{\log n, n, n^2, 2^n\}$?

Impact of faster computers

Even a super computer cannot save a badly designed algorithm.

For large enough instances, the better algorithm on the slower computer always beats the worse algorithm on a faster computer!

Example

- Let's say that a certain algorithm with running time $T(n)$ implemented on our computer is able to solve instances of size N within one hour.
- Suppose we get a new computer being K-times faster.
- Q: How large are the instances the K-times faster algorithm can solve within one hour for $T(n) \in \{\log n, n, n^2, 2^n\}$?

$T(n)$	New maximal input size
$\log(n)$	N^K
n	$K \cdot N$
n^2	$\sqrt{K} \cdot N$
2^n	$N + \log(K)$

Asymptotic Analysis

$\mathcal{O}$ ("big-Oh")-notation

Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be two functions and $c > 0$.

Informal definition of $\mathcal{O}(f)$

$\mathcal{O}(f)$ is the set of all functions growing not faster than f.

- $g \in \mathcal{O}(f)$ means $c \cdot f(n)$ is an upper bound for $g(n)$.

$\mathcal{O}$ ("big-Oh")-notation

Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be two functions and $c > 0$.

Informal definition of $\mathcal{O}(f)$

$\mathcal{O}(f)$ is the set of all functions growing not faster than f.

- $g \in \mathcal{O}(f)$ means $c \cdot f(n)$ is an upper bound for $g(n)$.

This properties holds for all $n \in \mathbb{N}$ with $n \geq n_0$ for some integer n_0.

$\mathcal{O}$ ("big-Oh")-notation

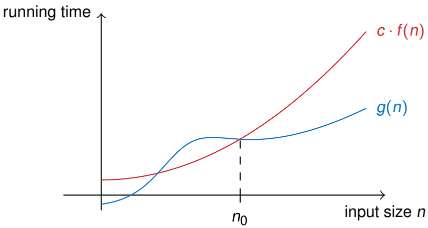
Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be two functions and $c > 0$.

Informal definition of $\mathcal{O}(f)$

$\mathcal{O}(f)$ is the set of all functions growing not faster than f.

- $g \in \mathcal{O}(f)$ means $c \cdot f(n)$ is an upper bound for $g(n)$.

This properties holds for all $n \in \mathbb{N}$ with $n \geq n_0$ for some integer n_0.

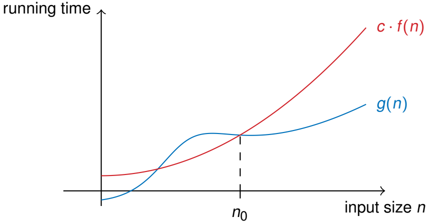


$\mathcal{O}$-notation

$\mathcal{O}$ provides an upper bound on the asymptotic growth of a functions.

Mathematical definition of $\mathcal{O}(f)$

$g \in \mathcal{O}(f)$ if and only if $\exists\ c > 0, n_0 \in \mathbb{N}$ with $\forall\ n \geq n_0 : 0 \leq g(n) \leq c \cdot f(n)$

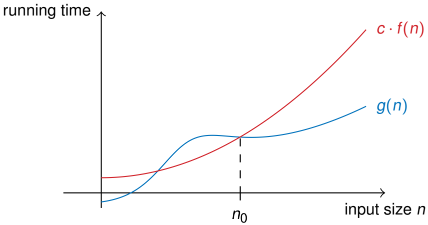


$\mathcal{O}$-notation

$\mathcal{O}$ provides an upper bound on the asymptotic growth of a functions.

Mathematical definition of $\mathcal{O}(f)$

$g \in \mathcal{O}(f)$ if and only if $\exists\, c > 0, n_0 \in \mathbb{N}$ with $\forall\, n \geq n_0 : 0 \leq g(n) \leq c \cdot f(n)$



Certainly, the smaller upper bounds the better. For example,

$$g \in \mathcal{O}(n^2) \text{ provides more information than } g \in \mathcal{O}(n^3).$$

Ω (”big Omega”)- notation

Informal definition of $\Omega(f)$

$\Omega(f)$ is the set of functions growing not smaller than f:

- $g \in \Omega(f)$ means $c \cdot f(n)$ is a lower bound for $g(n)$.

Ω ("big Omega")- notation

Informal definition of $\Omega(f)$

$\Omega(f)$ is the set of functions growing not smaller than f:

- $g \in \Omega(f)$ means $c \cdot f(n)$ is a lower bound for $g(n)$.

This properties holds for all $n \in \mathbb{N}$ with $n \geq n_0$ for some integer n_0.

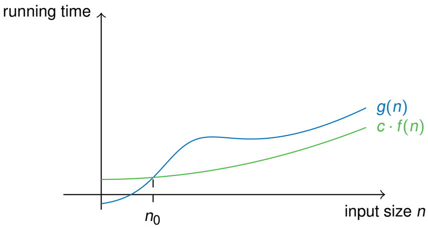
Ω ("big Omega")- notation

Informal definition of $\Omega(f)$

$\Omega(f)$ is the set of functions growing not smaller than f:

- $g \in \Omega(f)$ means $c \cdot f(n)$ is a lower bound for $g(n)$.

This properties holds for all $n \in \mathbb{N}$ with $n \geq n_0$ for some integer n_0.



Ω-notation

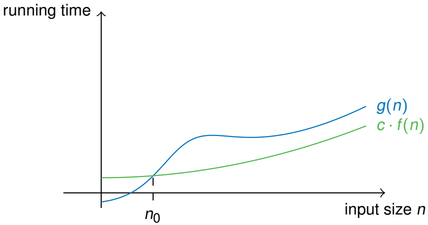
Ω provides a lower bound on the asymptotic growth of a function.

Ω-notation

Ω provides a lower bound on the asymptotic growth of a function.

Mathematical definition of $\Omega(f)$

$g \in \Omega(f)$ if and only if $\exists\ c > 0, n_0 \in \mathbb{N}$ with $\forall\ n \geq n_0 : g(n) \geq c \cdot f(n)$

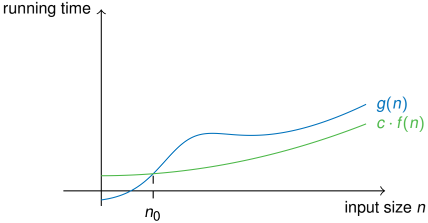


Ω-notation

Ω provides a lower bound on the asymptotic growth of a function.

Mathematical definition of $\Omega(f)$

$g \in \Omega(f)$ if and only if $\exists\ c > 0, n_0 \in \mathbb{N}$ with $\forall\ n \geq n_0 : g(n) \geq c \cdot f(n)$



Certainly, the greater lower bounds the better. For example,

$$g \in \Omega(n^2) \text{ gives more information than } g \in \Omega(n).$$

Θ ("big Theta")-notation

$\Theta(f)$ denotes the set of all functions growing equally fast as f.

Θ ("big Theta")-notation

$\Theta(f)$ denotes the set of all functions growing equally fast as f.
Let $c_1, c_2 > 0$ be two constants.

Θ ("big Theta")-notation

$\Theta(f)$ denotes the set of all functions growing equally fast as f.
Let $c_1, c_2 > 0$ be two constants.

Informal definition of $\Theta(f)$

$\Theta(f)$ is the set of all functions growing equally fast as f.

- $g \in \Theta(f)$ means: $c_1 \cdot f(n)$ is lower, $c_2 \cdot f(n)$ is upper bound on $g(n)$.

Θ ("big Theta")-notation

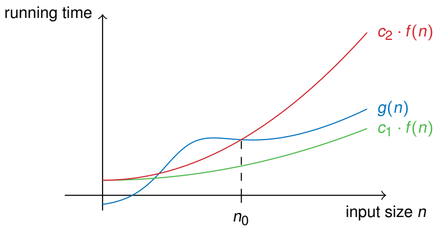
$\Theta(f)$ denotes the set of all functions growing equally fast as f.
Let $c_1, c_2 > 0$ be two constants.

Informal definition of $\Theta(f)$

$\Theta(f)$ is the set of all functions growing equally fast as f.

- $g \in \Theta(f)$ means: $c_1 \cdot f(n)$ is lower, $c_2 \cdot f(n)$ is upper bound on $g(n)$.

This properties holds for all $n \in \mathbb{N}$ with $n \geq n_0$ for some integer n_0.



Θ-notation

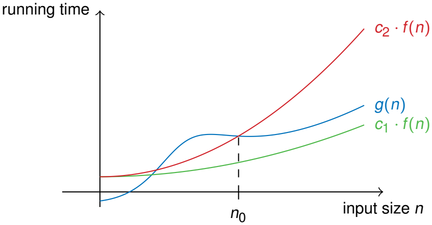
Θ provides a tight bound on the asymptotic growth of a function.

Θ-notation

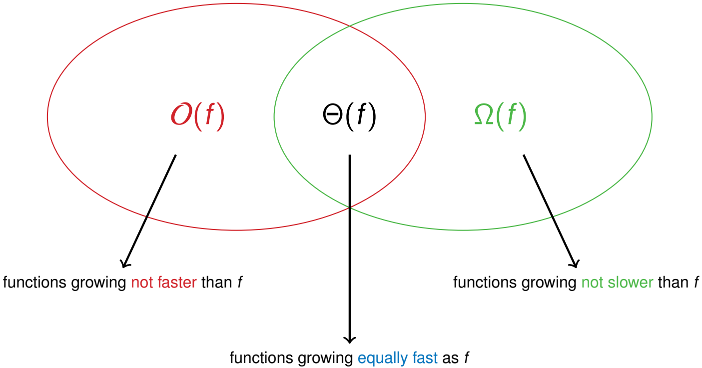
Θ provides a tight bound on the asymptotic growth of a function.

Mathematical definition of $\Theta(f)$

$g \in \Theta(f)$ if and only if
$\exists\ c_1, c_2 > 0, n_0$ with $\forall\ n \geq n_0 : c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$



Relationship between the three sets



Running Time of Insertion Sort and Merge Sort

Running time of insertion sort (worst case)

Recall: Arithmetic series (cf. exercises)

$$\sum_{j=1}^{n} j = \frac{1}{2}n(n+1).$$

Running time of insertion sort (worst case)

Recall: Arithmetic series (cf. exercises)

$$\sum_{j=1}^{n} j = \frac{1}{2}n(n+1).$$

$$\Rightarrow \sum_{j=2}^{n} (j-1) =$$

Running time of insertion sort (worst case)

Recall: Arithmetic series (cf. exercises)

$$\sum_{j=1}^{n} j = \frac{1}{2} n(n+1).$$

$$\Rightarrow \sum_{j=2}^{n} (j-1) = \sum_{j=1}^{n-1} j =$$

Running time of insertion sort (worst case)

Recall: Arithmetic series (cf. exercises)

$$\sum_{j=1}^{n} j = \frac{1}{2}n(n+1).$$

$$\Rightarrow \sum_{j=2}^{n}(j-1) = \sum_{j=1}^{n-1} j = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$$

Running time of insertion sort (worst case)

Recall: Arithmetic series (cf. exercises)

$$\sum_{j=1}^{n} j = \frac{1}{2} n(n+1).$$

$$\Rightarrow \sum_{j=2}^{n} (j-1) = \sum_{j=1}^{n-1} j = \frac{1}{2} n(n-1) = \frac{1}{2} n^2 - \frac{1}{2} n \in \Theta(n^2).$$

Running time of insertion sort (worst case)

Recall: Arithmetic series (cf. exercises)

$$\sum_{j=1}^{n} j = \frac{1}{2}n(n+1).$$

$$\Rightarrow \sum_{j=2}^{n}(j-1) = \sum_{j=1}^{n-1} j = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \in \Theta(n^2).$$

Worst case: Input reverse sorted.

For some constant $c > 0$,

$$T(n) = \sum_{j=2}^{n} c(j-1) = c\sum_{j=1}^{n-1} j \in \Theta(n^2).$$

Running time of insertion sort (average case)

Suppose that we randomly choose n numbers and apply insertion sort.

Average case: all permutations equally likely.

For some constant $c > 0$,

$$T(n) = \sum_{j=2}^{n} \frac{c}{2}(j-1) = \frac{c}{2}\sum_{j=1}^{n-1} j = \frac{c}{4}n^2 - \frac{c}{4}n \in \Theta(n^2).$$

Running time of insertion sort (average case)

Suppose that we randomly choose n numbers and apply insertion sort.

Average case: all permutations equally likely.

For some constant $c > 0$,

$$T(n) = \sum_{j=2}^{n} \frac{c}{2}(j-1) = \frac{c}{2}\sum_{j=1}^{n-1} j = \frac{c}{4}n^2 - \frac{c}{4}n \in \Theta(n^2).$$

Thus, average case is asymptotically as bad as worst case.

Running time of insertion sort (average case)

Suppose that we randomly choose n numbers and apply insertion sort.

Average case: all permutations equally likely.

For some constant $c > 0$,

$$T(n) = \sum_{j=2}^{n} \frac{c}{2}(j-1) = \frac{c}{2}\sum_{j=1}^{n-1} j = \frac{c}{4}n^2 - \frac{c}{4}n \in \Theta(n^2).$$

Thus, average case is asymptotically as bad as worst case.

Usual notation

We often write $g(n) = \Theta(f(n))$ instead of $g(n) \in \Theta(f(n))$.

Running time of insertion sort (average case)

Suppose that we randomly choose n numbers and apply insertion sort.

Average case: all permutations equally likely.

For some constant $c > 0$,

$$T(n) = \sum_{j=2}^{n} \frac{c}{2}(j-1) = \frac{c}{2}\sum_{j=1}^{n-1} j = \frac{c}{4}n^2 - \frac{c}{4}n \in \Theta(n^2).$$

Thus, average case is asymptotically as bad as worst case.

Usual notation

We often write $g(n) = \Theta(f(n))$ instead of $g(n) \in \Theta(f(n))$.
Analog for $\mathcal{O}(f)$ and $\Omega(f)$.

Running time of merge sort (worst-case)

MergeSort $(A[1..n])$	$T(n)$	
IF $(n = 1)$	$\Theta(1)$	
RETURN A	$\Theta(1)$	
Sort $A[1\ ..\ \lfloor \frac{n}{2} \rfloor]$ und $A[\lfloor \frac{n}{2} \rfloor + 1\ ..\ n]$ recursively	$2T(\frac{n}{2})$	"sloppy"
Merge the two sorted subarrays	$\Theta(n)$	
RETURN A	$\Theta(1)$	

"sloppy" should be $T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$, but it turns out not to matter asymptotically

Running time of merge sort (worst-case)

MergeSort ($A[1..n]$)	$T(n)$	
IF ($n = 1$)	$\Theta(1)$	
RETURN A	$\Theta(1)$	
Sort $A[1 .. \lfloor \frac{n}{2} \rfloor]$ und $A[\lfloor \frac{n}{2} \rfloor + 1 .. n]$ recursively	$2T(\frac{n}{2})$	"sloppy"
Merge the two sorted subarrays	$\Theta(n)$	
RETURN A	$\Theta(1)$	

"sloppy" should be $T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$, but it turns out not to matter asymptotically

We derive the following recurrence for running time function $T(n)$:

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \textbf{if } n > 1 \end{cases}$$

Recurrence for merge sort

Recurrence for merge sort:

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \textbf{if } n > 1 \end{cases}$$

Recurrence for merge sort

Recurrence for merge sort:

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \textbf{if } n > 1 \end{cases}$$

Remark: We will usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n.

Recurrence for merge sort

Recurrence for merge sort:

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \textbf{if } n > 1 \end{cases}$$

Remark: We will usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n. So, the recurrence for merge sort is

$$T(n) = 2T(n/2) + \Theta(n).$$

Recurrence for merge sort

Recurrence for merge sort:

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \textbf{if } n > 1 \end{cases}$$

Remark: We will usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n. So, the recurrence for merge sort is

$$T(n) = 2T(n/2) + \Theta(n).$$

Next lecture: several ways to find upper bounds for recurrences of type

$$T(n) = aT(n/2) + f(n),$$

where a, b are some constants, and $f(n)$ is some function.

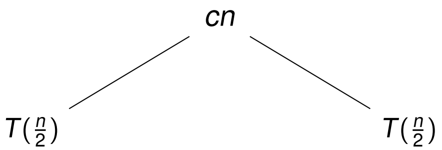
Recursion tree (rough idea)

Question: How can we solve a recurrence of type $T(n) = 2T(\frac{n}{2}) + cn$ where $c > 0$ is some constant?

$$T(n)$$

Recursion tree (rough idea)

Question: How can we solve a recurrence of type $T(n) = 2T(\frac{n}{2}) + cn$ where $c > 0$ is some constant?



Recursion tree (rough idea)

Question: How can we solve a recurrence of type $T(n) = 2T(\frac{n}{2}) + cn$ where $c > 0$ is some constant?

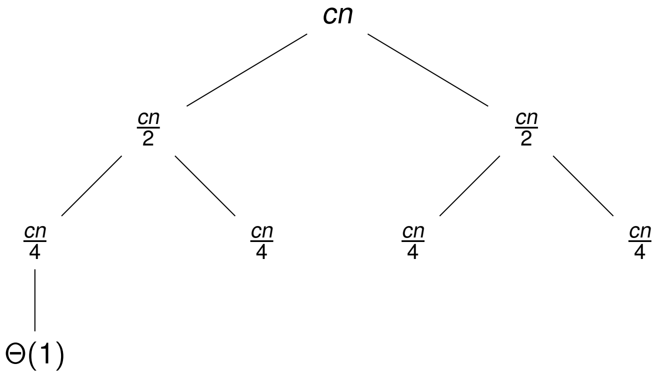
$$cn$$

$$\frac{cn}{2} \qquad \frac{cn}{2}$$

$$T(\tfrac{n}{4}) \quad T(\tfrac{n}{4}) \quad T(\tfrac{n}{4}) \quad T(\tfrac{n}{4})$$

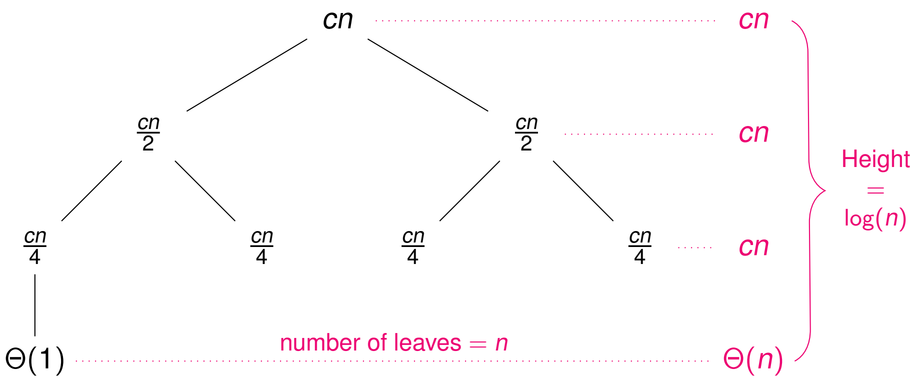
Recursion tree (rough idea)

Question: How can we solve a recurrence of type $T(n) = 2T(\frac{n}{2}) + cn$ where $c > 0$ is some constant?



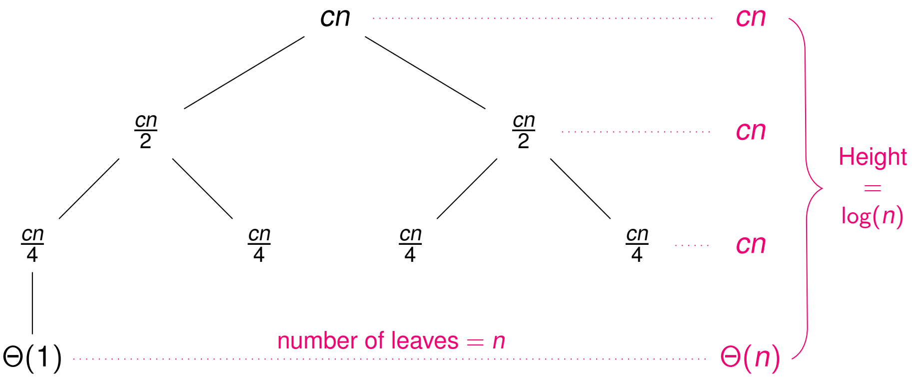
Recursion tree (rough idea)

Question: How can we solve a recurrence of type $T(n) = 2T(\frac{n}{2}) + cn$ where $c > 0$ is some constant?



Recursion tree (rough idea)

Question: How can we solve a recurrence of type $T(n) = 2T(\frac{n}{2}) + cn$ where $c > 0$ is some constant?



$$\Rightarrow T(n) = \Theta(n \cdot \log(n))$$

Summary

$\Theta(n \cdot \log(n))$ grows more slowly than $\Theta(n^2)$.

Summary

$\Theta(n \cdot \log(n))$ grows more slowly than $\Theta(n^2)$.

Thus, merge sort beats insertion sort asymptotically.

Summary

$\Theta(n \cdot \log(n))$ grows more slowly than $\Theta(n^2)$.

Thus, merge sort beats insertion sort asymptotically.

In practice, merge sort beats insertion sort already on inputs of size around 30.

Summary

$\Theta(n \cdot \log(n))$ grows more slowly than $\Theta(n^2)$.

Thus, merge sort beats insertion sort asymptotically.

In practice, merge sort beats insertion sort already on inputs of size around 30.

There are several sorting algorithms (selection sort, quick sort, bubble sort, heap sort, bucket sort, ...)

Summary

$\Theta(n \cdot \log(n))$ grows more slowly than $\Theta(n^2)$.

Thus, merge sort beats insertion sort asymptotically.

In practice, merge sort beats insertion sort already on inputs of size around 30.

There are several sorting algorithms (selection sort, quick sort, bubble sort, heap sort, bucket sort, . . .)

But: without any additional information on the input instance it is not possible to beast the worst case running time of $\Theta(n \cdot \log(n))$.