CSE 221: Algorithms

Introduction to algorithms

Mumit Khan

Computer Science and Engineering
BRAC University

References

1. Jon Kleinberg and Éva Tardos, *Algorithm Design*. Pearson Education, 2006.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

Last modified: January 13, 2011

Algorithm

Definition (from Wikipedia)

[. . .] an algorithm [. . .] is an effective method for solving a problem expressed as a finite sequence of steps. Algorithms are used for calculation, data processing, and many other fields.

Algorithm

Definition (from Wikipedia)

[...] an algorithm [...] is an effective method for solving a problem expressed as a finite sequence of steps. Algorithms are used for calculation, data processing, and many other fields.

Key concepts

Correctness Can you prove it's correct?

Algorithm

Definition (from Wikipedia)

[. . .] an algorithm [. . .] is an effective method for solving a problem expressed as a finite sequence of steps. Algorithms are used for calculation, data processing, and many other fields.

Key concepts

Correctness Can you prove it's correct?

Efficiency Can you quantify how much time (and space) it takes to solve a problem of a known size using your algorithm?

Algorithm

Definition (from Wikipedia)

[. . .] an algorithm [. . .] is an effective method for solving a problem expressed as a finite sequence of steps. Algorithms are used for calculation, data processing, and many other fields.

Key concepts

Correctness Can you prove it's correct?

Efficiency Can you quantify how much time (and space) it takes to solve a problem of a known size using your algorithm?

Elegance Ah, this is where the "art" in Computer Science comes in!

Contents

Contents

Exhaustive search vs. efficient algorithm

Exhaustive search

1. Enumerate all possible *configurations* (need to know what the natural search space is).
2. Pick *the* (or, *a* – there may be many solutions) *configuration* that satisfies the criteria for solution.

Exhaustive search vs. efficient algorithm

Exhaustive search

1. Enumerate all possible *configurations* (need to know what the natural search space is).
2. Pick *the* (or, *a* – there may be many solutions) *configuration* that satisfies the criteria for solution.

Problem with exhaustive search

The natural search space is often very large!

Exhaustive search vs. efficient algorithm

Exhaustive search

1. Enumerate all possible *configurations* (need to know what the natural search space is).
2. Pick *the* (or, *a* – there may be many solutions) *configuration* that satisfies the criteria for solution.

Problem with exhaustive search

The natural search space is often very large!

Efficient algorithm?

The goal of efficient algorithms is to **significantly shrink** the natural search space.

Exhaustive search vs. efficient algorithm

Exhaustive search

1. Enumerate all possible *configurations* (need to know what the natural search space is).
2. Pick *the* (or, *a* – there may be many solutions) *configuration* that satisfies the criteria for solution.

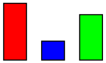
Problem with exhaustive search

The natural search space is often very large!

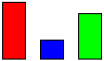
Efficient algorithm?

The goal of efficient algorithms is to **significantly shrink** the natural search space. Is it always possible?

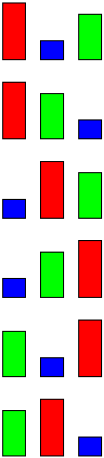
The sorting problem



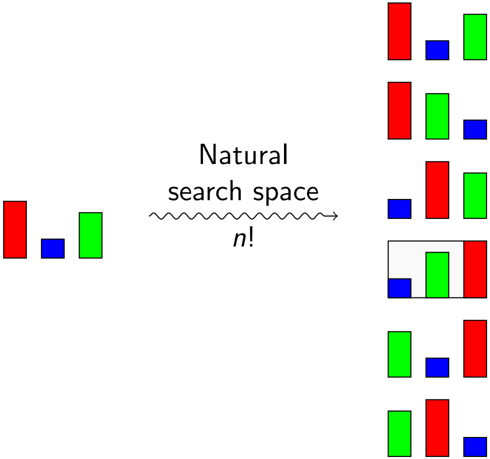
The sorting problem



Natural
search space
$\rightsquigarrow$
$n!$



The sorting problem



Search space

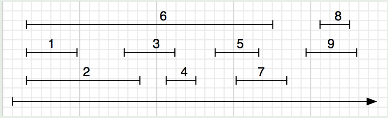
Natural search space is $n!$ (all possible permutations).

The interval scheduling problem

Definition

Given a set of schedules $I = \{I_i\}$, find the largest set $A \subseteq I$ such that the members of A are non-conflicting.

Example

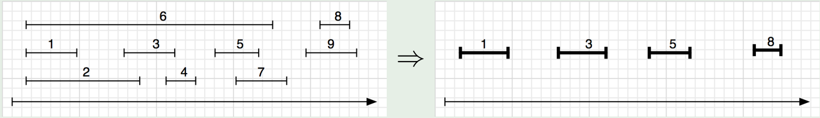


The interval scheduling problem

Definition

Given a set of schedules $I = \{I_i\}$, find the largest set $A \subseteq I$ such that the members of A are non-conflicting.

Example

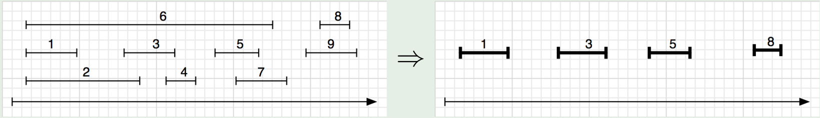


The interval scheduling problem

Definition

Given a set of schedules $I = \{I_i\}$, find the largest set $A \subseteq I$ such that the members of A are non-conflicting.

Example



Search space

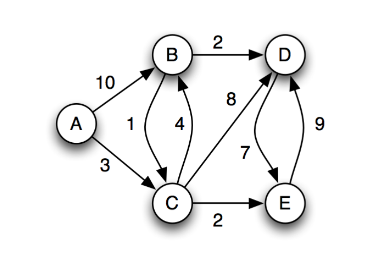
Natural search space is $2^n - 1$ (the set of non-empty subsets).

The shortest path problem

Definition

Given a weighted directed graph, find the shortest path from the source vertex to all the other vertices.

Example

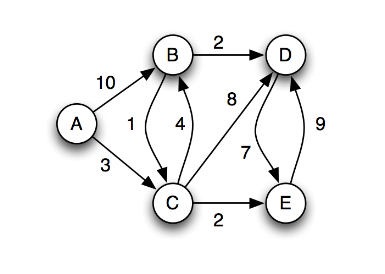


The shortest path problem

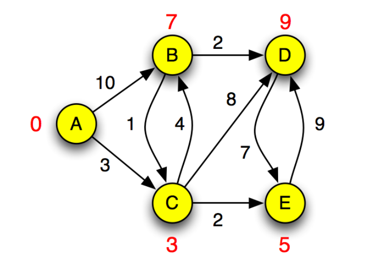
Definition

Given a weighted directed graph, find the shortest path from the source vertex to all the other vertices.

Example



$\Rightarrow$

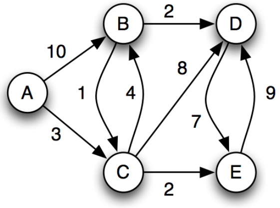


The shortest path problem

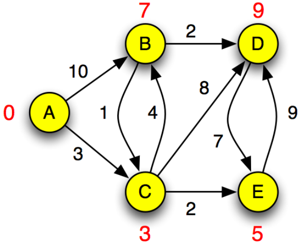
Definition

Given a weighted directed graph, find the shortest path from the source vertex to all the other vertices.

Example



⇒



Search space

Natural search space is exponential.

Contents

Finding the largest element in a sequence

The search for maximum problem

Find the largest element e in a sequence $A[1..n]$ of n elements.

Finding the largest element in a sequence

The search for maximum problem

Find the largest element e in a sequence $A[1\,..\,n]$ of n elements.
INPUT: Given the sequence

3	2	6	9	8
1	2	3	4	5

Finding the largest element in a sequence

The search for maximum problem

Find the largest element e in a sequence $A[1 \mathinner{..} n]$ of n elements.

INPUT: Given the sequence

3	2	6	9	8
1	2	3	4	5

The algorithm returns $\boxed{9}$.

Finding the largest element in a sequence

The search for maximum problem

Find the largest element e in a sequence $A[1\,..\,n]$ of n elements.

INPUT: Given the sequence

3	2	6	9	8
1	2	3	4	5

The algorithm returns $\boxed{9}$.

Algorithm

FIND-MAXIMUM$(A, n) \triangleright A[1\,..\,n]$

```
max ← A[1]
for i ← 2 to n
    do if A[i] > max
        then max ← A[i]
return max
```

Finding the largest element in a sequence: analysis

FIND-MAXIMUM$(A, n) \triangleright A[1 \,..\, n]$

	cost	*times*
$max \leftarrow A[1]$	c_1	1
for $i \leftarrow 2$ **to** n	c_2	n
do if $A[i] > max$	c_4	$n - 1$
then $max \leftarrow A[i]$	c_5	x
return max	c_6	1

Finding the largest element in a sequence: analysis

FIND-MAXIMUM(A, n) $\triangleright$ $A[1\,..\,n]$

	cost	*times*
$max \leftarrow A[1]$	c_1	1
for $i \leftarrow 2$ **to** n	c_2	n
do if $A[i] > max$	c_4	$n - 1$
then $max \leftarrow A[i]$	c_5	x
return max	c_6	1

Finding the largest element in a sequence: analysis

FIND-MAXIMUM$(A, n) \triangleright A[1 \mathinner{..} n]$

	cost	*times*
$max \leftarrow A[1]$	c_1	1
for $i \leftarrow 2$ **to** n	c_2	n
do if $A[i] > max$	c_4	$n-1$
then $max \leftarrow A[i]$	c_5	x
return max	c_6	1

Finding the largest element in a sequence: analysis

FIND-MAXIMUM$(A, n) \triangleright A[1 \,..\, n]$

		cost	*times*
1	$max \leftarrow A[1]$	c_1	1
2	**for** $i \leftarrow 2$ **to** n	c_2	n
3	**do if** $A[i] > max$	c_4	$n - 1$
4	**then** $max \leftarrow A[i]$	c_5	x
5	**return** max	c_6	1

Finding the largest element in a sequence: analysis

FIND-MAXIMUM(A, n) $\triangleright$ $A[1 \,..\, n]$

		cost	times
1	$max \leftarrow A[1]$	c_1	1
2	**for** $i \leftarrow 2$ **to** n	c_2	n
3	**do if** $A[i] > max$	c_4	$n - 1$
4	**then** $max \leftarrow A[i]$	c_5	x[a]
5	**return** max	c_6	1

[a] x is the number of times the max is assigned on line 4; x ranges between 0 (best-case, when $A[1]$ is the largest element) and $n - 1$ (worst-case, when A is sorted such that $A[n]$ is the largest element)

Finding the largest element in a sequence: analysis

FIND-MAXIMUM$(A, n) \triangleright A[1 \,..\, n]$

		cost	times
1	$max \leftarrow A[1]$	c_1	1
2	**for** $i \leftarrow 2$ **to** n	c_2	n
3	**do if** $A[i] > max$	c_4	$n - 1$
4	**then** $max \leftarrow A[i]$	c_5	x[a]
5	**return** max	c_6	1

[a] x is the number of times the max is assigned on line 4; x ranges between 0 (best-case, when $A[1]$ is the largest element) and $n - 1$ (worst-case, when A is sorted such that $A[n]$ is the largest element)

Finding the largest element in a sequence: analysis

FIND-MAXIMUM(A, n) $\triangleright$ $A[1\,.\,.\,n]$

	cost	*times*
$max \leftarrow A[1]$	c_1	1
for $i \leftarrow 2$ **to** n	c_2	n
do if $A[i] > max$	c_4	$n-1$
then $max \leftarrow A[i]$	c_5	x
return max	c_6	1

Total cost

$$T(n) = (c_1 - c_4 + c_6) + (c_2 + c_4)n + c_5 x$$

Finding the largest element in a sequence: analysis

$\text{FIND-MAXIMUM}(A, n) \triangleright A[1 \mathinner{.\,.} n]$

	cost	*times*
$max \leftarrow A[1]$	c_1	1
for $i \leftarrow 2$ **to** n	c_2	n
do if $A[i] > max$	c_4	$n-1$
then $max \leftarrow A[i]$	c_5	x
return max	c_6	1

Total cost

$$T(n) = (c_1 - c_4 + c_6) + (c_2 + c_4)n + c_5 x$$

Best-case cost: $x = 0$, when $A[1]$ is the largest element

$$\begin{aligned} T(n) &= (c_1 - c_4 + c_6) + (c_2 + c_4)n \\ &= cn + d \quad \text{where } c \text{ and } d \text{ are constants} \end{aligned}$$

Finding the largest element in a sequence: analysis

FIND-MAXIMUM$(A, n) \triangleright A[1\,..\,n]$

	cost	times
$max \leftarrow A[1]$	c_1	1
for $i \leftarrow 2$ **to** n	c_2	n
do if $A[i] > max$	c_4	$n-1$
then $max \leftarrow A[i]$	c_5	x
return max	c_6	1

Total cost

$$T(n) = (c_1 - c_4 + c_6) + (c_2 + c_4)n + c_5 x$$

Worst-case cost: $x = n - 1$, when A is sorted

$$\begin{aligned} T(n) &= (c_1 - c_4 + c_6) + (c_2 + c_4)n + c_5(n-1) \\ &= cn + d \quad \text{where } c \text{ and } d \text{ are constants} \end{aligned}$$

Finding the largest element in a sequence: analysis

FIND-MAXIMUM(A, n) $\triangleright$ $A[1 \,..\, n]$

	cost	*times*
$max \leftarrow A[1]$	c_1	1
for $i \leftarrow 2$ **to** n	c_2	n
do if $A[i] > max$	c_4	$n-1$
then $max \leftarrow A[i]$	c_5	x
return max	c_6	1

Total cost

$$T(n) = (c_1 - c_4 + c_6) + (c_2 + c_4)n + c_5 x$$

Average-case cost: $E[x] = \frac{n}{2}$

$$\begin{aligned} T(n) &= (c_1 - c_4 + c_6) + (c_2 + c_4)n + c_5 \tfrac{n}{2} \\ &= cn + d \quad \text{where } c \text{ and } d \text{ are constants} \end{aligned}$$

FIND-MAXIMUM analysis: summary

Best case Runs in linear time, when $A[1]$ is the largest element.

Worst case Runs in linear time, when A is sorted such that $A[n]$ is the largest element.

Average case Runs in linear time, if we assume randomly distributed input data.

FIND-MAXIMUM analysis: summary

Best case Runs in linear time, when $A[1]$ is the largest element.

Worst case Runs in linear time, when A is sorted such that $A[n]$ is the largest element.

Average case Runs in linear time, if we assume randomly distributed input data.

FIND-MAXIMUM analysis: summary

Best case Runs in linear time, when $A[1]$ is the largest element.

Worst case Runs in linear time, when A is sorted such that $A[n]$ is the largest element.

Average case Runs in linear time, if we assume randomly distributed input data.

FIND-MAXIMUM analysis: summary

Best case Runs in linear time, when $A[1]$ is the largest element.

Worst case Runs in linear time, when A is sorted such that $A[n]$ is the largest element.

Average case Runs in linear time, if we assume randomly distributed input data.

FIND-MAXIMUM analysis: summary

Best case Runs in linear time, when $A[1]$ is the largest element.

Worst case Runs in linear time, when A is sorted such that $A[n]$ is the largest element.

Average case Runs in linear time, if we assume randomly distributed input data.
Often as bad as the worst-case performance.

FIND-MAXIMUM analysis: summary

Best case Runs in linear time, when $A[1]$ is the largest element.

Worst case Runs in linear time, when A is sorted such that $A[n]$ is the largest element.

Average case Runs in linear time, if we assume randomly distributed input data.

Often as bad as the worst-case performance.

Question

Which one to use to analyze algorithms?

FIND-MAXIMUM analysis: summary

Best case Runs in linear time, when $A[1]$ is the largest element.

Worst case Runs in linear time, when A is sorted such that $A[n]$ is the largest element.

Average case Runs in linear time, if we assume randomly distributed input data.

Often as bad as the worst-case performance.

Question

Which one to use to analyze algorithms?
All are of the same degree, so which one to choose?
What is the problem with average-case analysis?

Inserting into a sorted sequence

The INSERT-SORTED problem

Insert the given *key* in a sorted sequence $A[1 \mathinner{..} n]$ of n numbers such that resulting sequence $A[1 \mathinner{..} n+1]$ remain sorted.

Inserting into a sorted sequence

The INSERT-SORTED problem

Insert the given *key* in a sorted sequence $A[1\,..\,n]$ of n numbers such that resulting sequence $A[1\,..\,n+1]$ remain sorted.

Example

INPUT: Given the following sorted sequence and $key = 4$

2	3	6	8	9
1	2	3	4	5

Inserting into a sorted sequence

The INSERT-SORTED problem

Insert the given *key* in a sorted sequence $A[1 \,..\, n]$ of n numbers such that resulting sequence $A[1 \,..\, n+1]$ remain sorted.

Example

INPUT: Given the following sorted sequence and $key = 4$

2	3	6	8	9
1	2	3	4	5

OUTPUT: A sorted sequence of $n+1$ numbers, with the $key = 4$ inserted in its proper position.

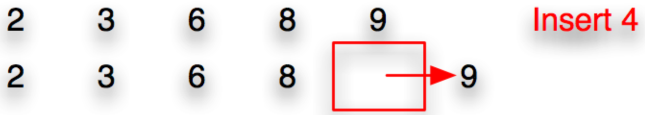
2	3	4	6	8	9
1	2	3	4	5	6

Inserting into a sorted sequence

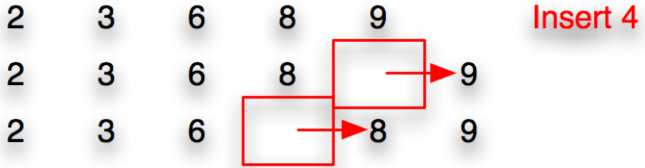
$$\underbrace{2 \quad 3 \quad 6 \quad 8 \quad 9}_{\text{sorted}}$$

Insert 4

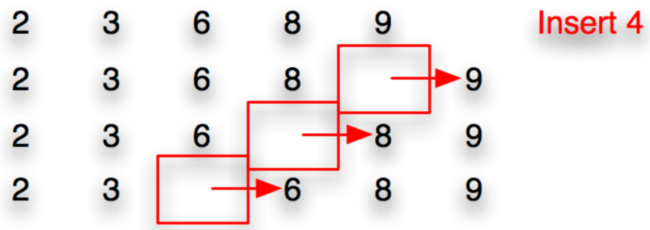
Inserting into a sorted sequence



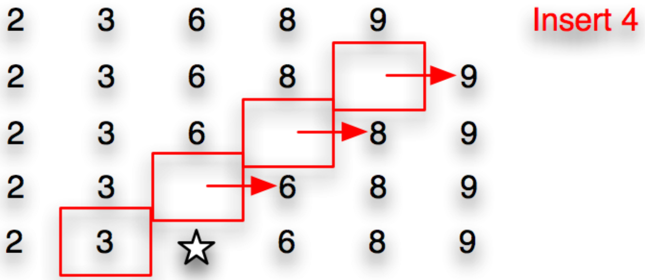
Inserting into a sorted sequence



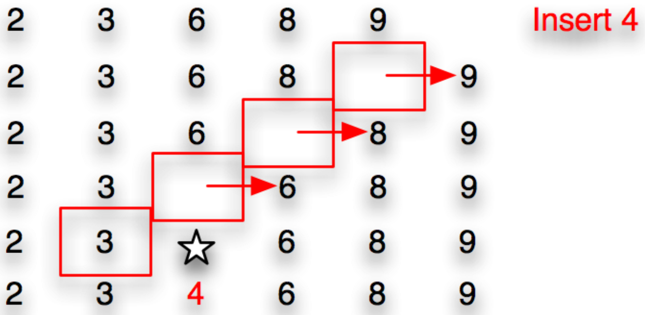
Inserting into a sorted sequence



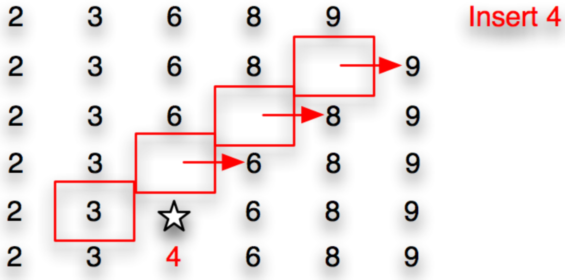
Inserting into a sorted sequence



Inserting into a sorted sequence



Inserting into a sorted sequence



Algorithm

INSERT-SORTED$(key, A, n) \triangleright A[1\,..\,n]$

```
i ← n
while i > 0 and A[i] > key
        do A[i + 1] ← A[i]
            i ← i − 1
A[i + 1] ← key
```

Analyzing the algorithm

INSERT-SORTED$(key, A, n) \triangleright A[1\,..\,n]$

	cost	times
$i \leftarrow n$	c_1	1
while $i > 0$ and $A[i] > key$	c_2	x
do $A[i+1] \leftarrow A[i]$	c_3	$x - 1$
$i \leftarrow i - 1$	c_4	$x - 1$
$A[i+1] \leftarrow key$	c_5	1

Analyzing the algorithm

$\textsc{Insert-Sorted}(key, A, n) \triangleright A[1\,..\,n]$

	cost	times
$i \leftarrow n$	c_1	1
while $i > 0$ and $A[i] > key$	c_2	x
do $A[i+1] \leftarrow A[i]$	c_3	$x-1$
$i \leftarrow i-1$	c_4	$x-1$
$A[i+1] \leftarrow key$	c_5	1

Analyzing the algorithm

$\text{INSERT-SORTED}(key, A, n) \triangleright A[1\,..\,n]$

		cost	times
1	$i \leftarrow n$	c_1	1
2	**while** $i > 0$ and $A[i] > key$	c_2	x[a]
3	**do** $A[i+1] \leftarrow A[i]$	c_3	$x - 1$
4	$i \leftarrow i - 1$	c_4	$x - 1$
5	$A[i+1] \leftarrow key$	c_5	1

[a] x is the number of times the **while** loop test executes; x ranges between 1 (best-case, when $key > A[n]$) and $n + 1$ (worst-case, when $key < A[1]$)

Analyzing the algorithm

$\text{INSERT-SORTED}(key, A, n) \triangleright A[1 \,..\, n]$

	cost	*times*
$i \leftarrow n$	c_1	1
while $i > 0$ and $A[i] > key$	c_2	x[a]
do $A[i+1] \leftarrow A[i]$	c_3	$x - 1$
$i \leftarrow i - 1$	c_4	$x - 1$
$A[i+1] \leftarrow key$	c_5	1

[a] x is the number of times the **while** loop test executes; x ranges between 1 (best-case, when $key > A[n]$) and $n + 1$ (worst-case, when $key < A[1]$)

Analyzing the algorithm

INSERT-SORTED$(key, A, n) \triangleright A[1\,..\,n]$

	cost	*times*
$i \leftarrow n$	c_1	1
while $i > 0$ and $A[i] > key$	c_2	x^a
do $A[i+1] \leftarrow A[i]$	c_3	$x - 1$
$i \leftarrow i - 1$	c_4	$x - 1$
$A[i+1] \leftarrow key$	c_5	1

[a] x is the number of times the **while** loop test executes; x ranges between 1 (best-case, when $key > A[n]$) and $n + 1$ (worst-case, when $key < A[1]$)

Analyzing the algorithm

$\text{INSERT-SORTED}(key, A, n) \triangleright A[1\,..\,n]$

	cost	*times*
$i \leftarrow n$	c_1	1
while $i > 0$ and $A[i] > key$	c_2	x^a
do $A[i+1] \leftarrow A[i]$	c_3	$x - 1$
$i \leftarrow i - 1$	c_4	$x - 1$
$A[i+1] \leftarrow key$	c_5	1

[a] x is the number of times the **while** loop test executes; x ranges between 1 (best-case, when $key > A[n]$) and $n + 1$ (worst-case, when $key < A[1]$)

Analyzing the algorithm

INSERT-SORTED$(key, A, n) \rhd A[1 \,..\, n]$

	cost	*times*
$i \leftarrow n$	c_1	1
while $i > 0$ and $A[i] > key$	c_2	x
do $A[i+1] \leftarrow A[i]$	c_3	$x - 1$
$i \leftarrow i - 1$	c_4	$x - 1$
$A[i+1] \leftarrow key$	c_5	1

Total cost

$$T(n) = c_1 + c_2 x + (c_3 + c_4)(x - 1) + c_5$$

Analyzing the algorithm

$\text{INSERT-SORTED}(key, A, n) \triangleright A[1 \,..\, n]$

	cost	*times*
$i \leftarrow n$	c_1	1
while $i > 0$ and $A[i] > key$	c_2	x
do $A[i+1] \leftarrow A[i]$	c_3	$x - 1$
$i \leftarrow i - 1$	c_4	$x - 1$
$A[i+1] \leftarrow key$	c_5	1

Total cost

$$T(n) = c_1 + c_2 x + (c_3 + c_4)(x - 1) + c_5$$

Best-case cost: $x = 1$, when $key > A[n]$

$$T(n) = c_1 + c_2 + c_5 = c \quad \text{where } c \text{ is a constant}$$

Analyzing the algorithm

INSERT-SORTED$(key, A, n) \triangleright A[1\,..\,n]$

	cost	times
$i \leftarrow n$	c_1	1
while $i > 0$ and $A[i] > key$	c_2	x
do $A[i+1] \leftarrow A[i]$	c_3	$x-1$
$i \leftarrow i - 1$	c_4	$x-1$
$A[i+1] \leftarrow key$	c_5	1

Total cost

$$T(n) = c_1 + c_2 x + (c_3 + c_4)(x-1) + c_5$$

Worst-case cost: $x = n + 1$, when $key < A[1]$

$$\begin{aligned} T(n) &= c_1 + c_2(n+1) + (c_3 + c_4)n + c_5 \\ &= cn + d \quad \text{where } c \text{ and } d \text{ are constants} \end{aligned}$$

Analyzing the algorithm

INSERT-SORTED$(key, A, n) \triangleright A[1\,..\,n]$

	cost	*times*
$i \leftarrow n$	c_1	1
while $i > 0$ and $A[i] > key$	c_2	x
do $A[i+1] \leftarrow A[i]$	c_3	$x - 1$
$i \leftarrow i - 1$	c_4	$x - 1$
$A[i+1] \leftarrow key$	c_5	1

Total cost

$$T(n) = c_1 + c_2 x + (c_3 + c_4)(x - 1) + c_5$$

Average-case cost: $E[x] = \frac{n}{2}$

$$\begin{aligned} T(n) &= c_1 + (c_2 + c_3 + c_4)\frac{n}{2} + c_5 \\ &= cn + d \quad \text{where } c \text{ and } d \text{ are constants} \end{aligned}$$

INSERT-SORTED analysis: summary

Best case Runs in constant time, when $key > A[n]$.

Worst case Runs in linear time, when $key < A[1]$.

Average case Runs in linear time, if we assume randomly distributed input data.

INSERT-SORTED analysis: summary

Best case Runs in constant time, when $key > A[n]$.

Worst case Runs in linear time, when $key < A[1]$.

Average case Runs in linear time, if we assume randomly distributed input data.

INSERT-SORTED analysis: summary

Best case Runs in constant time, when $key > A[n]$.

Worst case Runs in linear time, when $key < A[1]$.

Average case Runs in linear time, if we assume randomly distributed input data.

INSERT-SORTED analysis: summary

Best case Runs in constant time, when $key > A[n]$.

Worst case Runs in linear time, when $key < A[1]$.

Average case Runs in linear time, if we assume randomly distributed input data.

INSERT-SORTED analysis: summary

Best case Runs in constant time, when $key > A[n]$.

Worst case Runs in linear time, when $key < A[1]$.

Average case Runs in linear time, if we assume randomly distributed input data.

Often as bad as the worst-case performance.

INSERT-SORTED analysis: summary

Best case Runs in constant time, when $key > A[n]$.

Worst case Runs in linear time, when $key < A[1]$.

Average case Runs in linear time, if we assume randomly distributed input data.

Often as bad as the worst-case performance.

Question

Which one to use to analyze algorithms?

INSERT-SORTED analysis: summary

Best case Runs in constant time, when $key > A[n]$.

Worst case Runs in linear time, when $key < A[1]$.

Average case Runs in linear time, if we assume randomly distributed input data.

Often as bad as the worst-case performance.

Question

Which one to use to analyze algorithms?

Worst-case or average-case, but certainly *not* the best-case performance!

What is the problem with average-case analysis?

Sorting

The sorting problem

INPUT: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

5	2	10	4	3	6
1	2	3	4	5	6

Sorting

The sorting problem

INPUT: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

5	2	10	4	3	6
1	2	3	4	5	6

OUTPUT: A permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

2	3	4	5	6	10
1	2	3	4	5	6

Sorting

The sorting problem

INPUT: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

5	2	10	4	3	6
1	2	3	4	5	6

OUTPUT: A permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

2	3	4	5	6	10
1	2	3	4	5	6

Sorting algorithms

- Bubble, Selection, Insertion, Shell, ...
- Quicksort, Heapsort, Mergesort, ...

Insertion sort

Algorithm

INSERTION-SORT$(A, n) \triangleright A[1 \,..\, n]$

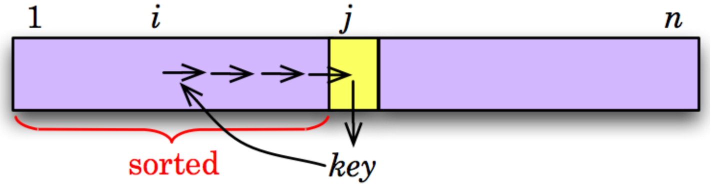
```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

Insertion sort

Algorithm

INSERTION-SORT(A, n) $\triangleright$ $A[1\,..\,n]$

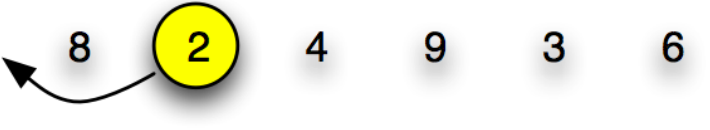
```
for j ← 2 to n
     do key ← A[j]
        i ← j − 1
        while i > 0 and A[i] > key
              do A[i + 1] ← A[i]
                 i ← i − 1
        A[i + 1] ← key
```



Sorting a sequence with Insertion sort

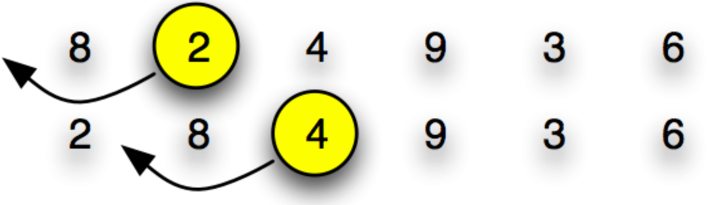
8 2 4 9 3 6

Sorting a sequence with Insertion sort

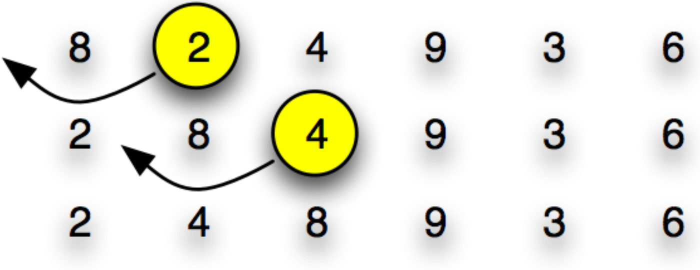


Sorting a sequence with Insertion sort

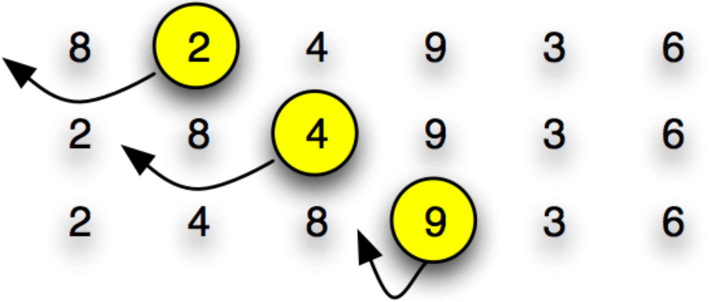
Sorting a sequence with Insertion sort



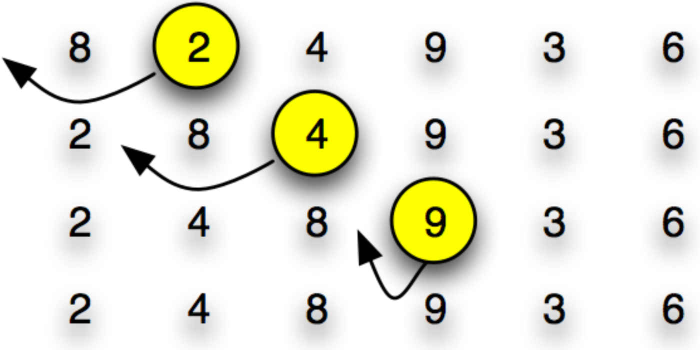
Sorting a sequence with Insertion sort



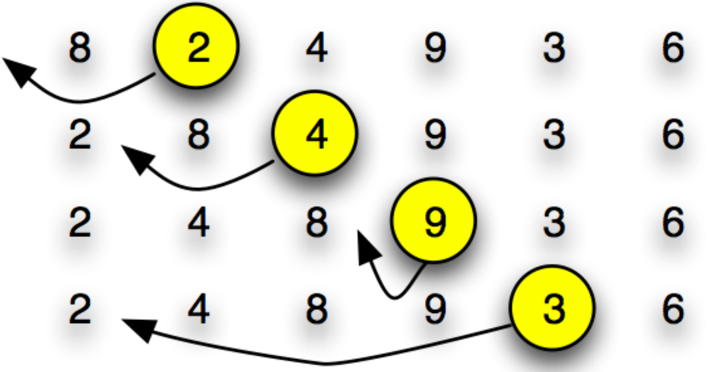
Sorting a sequence with Insertion sort



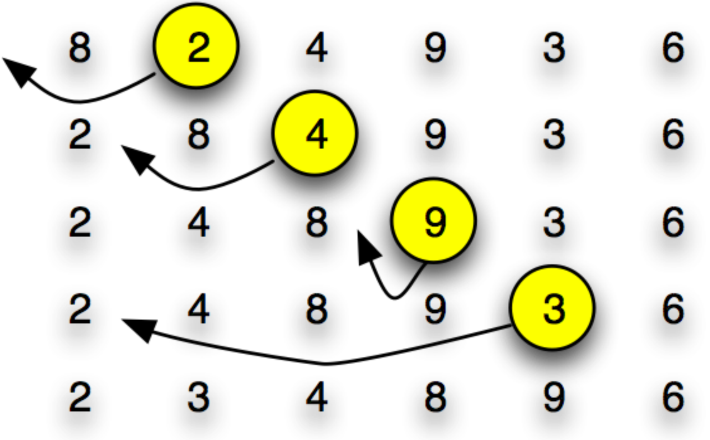
Sorting a sequence with Insertion sort



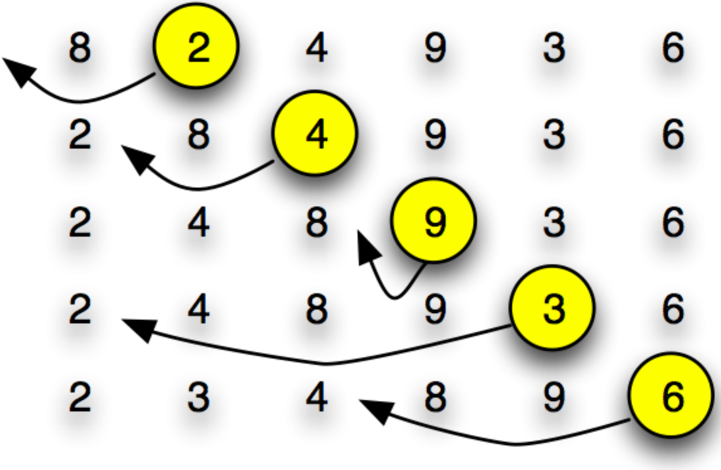
Sorting a sequence with Insertion sort



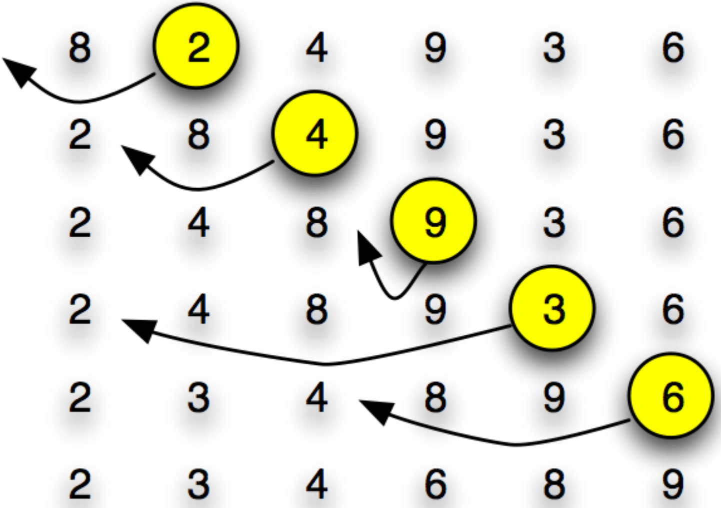
Sorting a sequence with Insertion sort



Sorting a sequence with Insertion sort



Sorting a sequence with Insertion sort

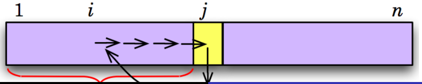


Insertion sort

Algorithm

INSERTION-SORT(A, n)

```
INPUT: A sequence of n numbers ⟨a_1, a_2, ..., a_n⟩
OUTPUT: A permutation ⟨a'_1, a'_2, ..., a'_n⟩ of the input
   sequence such that a'_1 ≤ a'_2 ≤ ... ≤ a'_n.
for j ← 2 to n
      do key ← A[j]
         ▷ Insert A[j] into sorted sequence A[1 .. j − 1].
         i ← j − 1
         while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i ← i − 1
         A[i + 1] ← key
```



Insertion sort analysis (CLRS 2.2)

INSERTION-SORT(A, n)

	cost	*times*
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into sorted		
sequence $A[1 \,.\,.\, j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Insertion sort analysis (CLRS 2.2)

INSERTION-SORT(A, n)

	cost	*times*
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into sorted		
sequence $A[1\,.\,.\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Insertion sort analysis (CLRS 2.2)

INSERTION-SORT(A, n)

	cost	times
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into sorted		
sequence $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Insertion sort analysis (CLRS 2.2)

INSERTION-SORT(A, n)

	cost	*times*
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into sorted		
sequence $A[1 \mathinner{.\,.} j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Insertion sort analysis (CLRS 2.2)

INSERTION-SORT(A, n)

	cost	times
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into sorted		
sequence $A[1\,.\,.\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Insertion sort analysis (CLRS 2.2)

INSERTION-SORT(A, n)

	cost	times
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into sorted		
sequence $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$ [a]
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

[a] t_j is the number of times the **while** loop test executes for that value of j; t_j ranges between 1 (best-case) and j (worst-case)

Insertion sort analysis (CLRS 2.2)

INSERTION-SORT(A, n)

	cost	times
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into sorted		
sequence $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$ [a]
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

[a] t_j is the number of times the **while** loop test executes for that value of j; t_j ranges between 1 (best-case) and j (worst-case)

Insertion sort analysis (CLRS 2.2)

INSERTION-SORT(A, n)

	cost	times
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into sorted		
sequence $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$ [a]
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

[a] t_j is the number of times the **while** loop test executes for that value of j; t_j ranges between 1 (best-case) and j (worst-case)

Insertion sort analysis (CLRS 2.2)

INSERTION-SORT(A, n)

	cost	times
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into sorted		
sequence $A[1 \, . \, . \, j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$ [a]
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

[a] t_j is the number of times the **while** loop test executes for that value of j; t_j ranges between 1 (best-case) and j (worst-case)

Insertion sort analysis (CLRS 2.2)

INSERTION-SORT(A, n)

	cost	times
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into sorted		
sequence $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Total cost

$$
\begin{aligned}
T(n) \;=\; & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\
& + c_6 \sum_{j=2}^{n}(t_j-1) + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1)
\end{aligned}
$$

Insertion sort analysis: best case

Runtime

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \end{aligned}$$

Best case

Insertion sort analysis: best case

Runtime

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \end{aligned}$$

Best case

Condition: Input already sorted.

Insertion sort analysis: best case

Runtime

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \textstyle\sum_{j=2}^{n} t_j \\ &\quad + c_6 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_7 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \end{aligned}$$

Best case

Condition: Input already sorted. $\Rightarrow t_j = 1 \text{ for } j = 2, 3, \ldots, n$.

Insertion sort analysis: best case

Runtime

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\
&\quad + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)
\end{aligned}
$$

Best case

Condition: Input already sorted. $\Rightarrow t_j = 1 \text{ for } j = 2, 3, \ldots, n.$

$$
T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)
$$

Insertion sort analysis: best case

Runtime

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Best case

Condition: Input already sorted. $\Rightarrow t_j = 1$ for $j = 2, 3, \ldots, n$.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Insertion sort analysis: best case

Runtime

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\
&\quad + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)
\end{aligned}
$$

Best case

Condition: Input already sorted. $\Rightarrow t_j = 1 \text{ for } j = 2, 3, \ldots, n.$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \\
&= cn + d \qquad (\text{where } c \text{ and } d \text{ are constants})
\end{aligned}
$$

Insertion sort analysis: best case

Runtime

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\
&\quad + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)
\end{aligned}
$$

Best case

Condition: Input already sorted. $\Rightarrow t_j = 1 \text{ for } j = 2, 3, \ldots, n.$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \\
&= cn + d \qquad \text{(where } c \text{ and } d \text{ are constants)}
\end{aligned}
$$

Observation

$T(n)$ is a ***linear function*** of n in the ***best case***.

Insertion sort analysis: worst case

Runtime

$$
\begin{aligned}
T(n) \;=\;& c_1 n + c_2(n-1) + c_4(n-1) + c_5 \textstyle\sum_{j=2}^{n} t_j \\
& + c_6 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_7 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)
\end{aligned}
$$

Worst case

Insertion sort analysis: worst case

Runtime

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \end{aligned}$$

Worst case

Condition: Input reverse sorted.

Insertion sort analysis: worst case

Runtime

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \end{aligned}$$

Worst case

Condition: Input reverse sorted. $\Rightarrow t_j = j \text{ for } j = 2, 3, \ldots, n$.

Insertion sort analysis: worst case

Runtime

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \textstyle\sum_{j=2}^{n} t_j \\ &\quad + c_6 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_7 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \end{aligned}$$

Note

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

Worst case

Condition: Input reverse sorted. $\Rightarrow t_j = j \text{ for } j = 2, 3, \ldots, n.$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \end{aligned}$$

Insertion sort analysis: worst case

Runtime

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Worst case

Condition: Input reverse sorted. $\Rightarrow t_j = j$ for $j = 2, 3, \ldots, n$.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Insertion sort analysis: worst case

Runtime

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \textstyle\sum_{j=2}^{n} t_j \\
&\quad + c_6 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_7 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)
\end{aligned}
$$

Worst case

Condition: Input reverse sorted. $\Rightarrow t_j = j$ for $j = 2, 3, \ldots, n$.

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\tfrac{n(n+1)}{2} - 1\right) \\
&\quad + c_6 \left(\tfrac{n(n-1)}{2}\right) + c_7 \left(\tfrac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\tfrac{c_5}{2} + \tfrac{c_6}{2} + \tfrac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \tfrac{c_5}{2} - \tfrac{c_6}{2} - \tfrac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8) \\
&= cn^2 + dn + e \qquad (\text{where } c,\ d, \text{ and } e \text{ are constants})
\end{aligned}
$$

Insertion sort analysis: worst case

Runtime

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \end{aligned}$$

Worst case

Condition: Input reverse sorted. $\Rightarrow t_j = j$ for $j = 2, 3, \ldots, n$.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \\ &= cn^2 + dn + e \qquad \text{(where } c,\ d, \text{ and } e \text{ are constants)} \end{aligned}$$

Observation

$T(n)$ is a ***quadratic function*** of n in the ***worst case***.

Insertion sort analysis: average case

Runtime

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Average case

Insertion sort analysis: average case

Runtime

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Average case

Condition: On the average, half the elements in $A[1 \,..\, j-1]$ are less than $A[j]$.

Insertion sort analysis: average case

Runtime

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Average case

Condition: On the average, half the elements in $A[1\,..\,j-1]$ are less than $A[j]$. $\Rightarrow E\left[t_j\right] = \frac{j}{2}$ for $j = 2, 3, \ldots, n$.

Insertion sort analysis: average case

Runtime

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \textstyle\sum_{j=2}^{n} t_j \\ &\quad + c_6 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_7 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \end{aligned}$$

Average case

Condition: On the average, half the elements in $A[1\,..\,j-1]$ are less than $A[j]$. $\Rightarrow E\left[t_j\right] = \frac{j}{2}$ for $j = 2, 3, \ldots, n$.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + \tfrac{c_5}{2}\left(\tfrac{n(n+1)}{2} - 1\right) \\ &\quad + \tfrac{c_6}{2}\left(\tfrac{n(n-1)}{2}\right) + \tfrac{c_7}{2}\left(\tfrac{n(n-1)}{2}\right) + c_8(n-1) \end{aligned}$$

Insertion sort analysis: average case

Runtime

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \textstyle\sum_{j=2}^{n} t_j \\
&\quad + c_6 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_7 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)
\end{aligned}
$$

Average case

Condition: On the average, half the elements in $A[1\,..\,j-1]$ are less than $A[j]$. $\Rightarrow E[t_j] = \frac{j}{2}$ for $j = 2, 3, \ldots, n$.

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + \tfrac{c_5}{2}\left(\tfrac{n(n+1)}{2} - 1\right) \\
&\quad + \tfrac{c_6}{2}\left(\tfrac{n(n-1)}{2}\right) + \tfrac{c_7}{2}\left(\tfrac{n(n-1)}{2}\right) + c_8(n-1) \\
&= cn^2 + dn + e \qquad \text{(where } c\text{, } d\text{, and } e \text{ are constants)}
\end{aligned}
$$

Insertion sort analysis: average case

Runtime

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \end{aligned}$$

Average case

Condition: On the average, half the elements in $A[1\,..\,j-1]$ are less than $A[j]$. $\Rightarrow E\left[t_j\right] = \frac{j}{2}$ for $j = 2, 3, \ldots, n$.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + \frac{c_5}{2}\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + \frac{c_6}{2}\left(\frac{n(n-1)}{2}\right) + \frac{c_7}{2}\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= cn^2 + dn + e \qquad \text{(where } c, d, \text{ and } e \text{ are constants)} \end{aligned}$$

Observation

$T(n)$ is a ***quadratic function*** of n in the ***average case***.

Insertion sort analysis: summary

Best case Runs in linear time, when the input is already sorted.

Worst case Runs in quadratic time, when the input is already sorted, but in the wrong order.

Average case Runs in quadratic time, if we assume randomly distributed input data.

Insertion sort analysis: summary

Best case Runs in linear time, when the input is already sorted.

Worst case Runs in quadratic time, when the input is already sorted, but in the wrong order.

Average case Runs in quadratic time, if we assume randomly distributed input data.

Insertion sort analysis: summary

Best case Runs in linear time, when the input is already sorted.

Worst case Runs in quadratic time, when the input is already sorted, but in the wrong order.

Average case Runs in quadratic time, if we assume randomly distributed input data.

Insertion sort analysis: summary

Best case Runs in linear time, when the input is already sorted.

Worst case Runs in quadratic time, when the input is already sorted, but in the wrong order.

Average case Runs in quadratic time, if we assume randomly distributed input data.

Insertion sort analysis: summary

Best case Runs in linear time, when the input is already sorted.

Worst case Runs in quadratic time, when the input is already sorted, but in the wrong order.

Average case Runs in quadratic time, if we assume randomly distributed input data.

Often as bad as the worst-case performance.

Insertion sort analysis: summary

Best case Runs in linear time, when the input is already sorted.

Worst case Runs in quadratic time, when the input is already sorted, but in the wrong order.

Average case Runs in quadratic time, if we assume randomly distributed input data.

Often as bad as the worst-case performance.

Question

Which one to use to analyze algorithms?

Insertion sort analysis: summary

Best case Runs in linear time, when the input is already sorted.

Worst case Runs in quadratic time, when the input is already sorted, but in the wrong order.

Average case Runs in quadratic time, if we assume randomly distributed input data.

Often as bad as the worst-case performance.

Question

Which one to use to analyze algorithms?

Worst-case or average-case, but certainly *not* the best-case performance!

What is the problem with average-case analysis?

Selection sort

Algorithm

SELECTION-SORT$(A, n) \rhd A[1 \,..\, n]$

```
for j ← 1 to n − 1
▷ Find the minimum element in A[j . . n],
      and exchange the element with A[j].
       do i_min ← j
           for i ← j + 1 to n
                do if A[i] < A[i_min]
                       then i_min ← i
           if j ≠ i_min
              then exchange A[j] ↔ A[i_min]
```

Selection sort analysis

$\textsc{Selection-Sort}(A, n) \triangleright A[1\,..\,n]$

	cost	*times*
for $j \leftarrow 1$ **to** $n-1$	c_1	n
$\triangleright$ Find the minimum element in $A[j\,..\,n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n-1$
for $i \leftarrow j+1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n-1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n-1$

Selection sort analysis

SELECTION-SORT$(A, n) \triangleright A[1\,..\,n]$

	cost	times
for $j \leftarrow 1$ **to** $n-1$	c_1	n
$\triangleright$ Find the minimum element in $A[j\,..\,n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n-1$
for $i \leftarrow j+1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n-1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n-1$

Selection sort analysis

$\textsc{Selection-Sort}(A, n) \rhd A[1 \,.\,.\, n]$

	cost	times
for $j \leftarrow 1$ **to** $n-1$	c_1	n
$\rhd$ Find the minimum element in $A[j \,.\,.\, n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n-1$
for $i \leftarrow j+1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n-1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n-1$

Selection sort analysis

SELECTION-SORT(A, n) $\triangleright$ $A[1 \,..\, n]$

	cost	times
for $j \leftarrow 1$ **to** $n-1$	c_1	n
$\triangleright$ Find the minimum element in $A[j \,..\, n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n-1$
for $i \leftarrow j+1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n-1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n-1$

Selection sort analysis

SELECTION-SORT(A, n) $\triangleright$ $A[1 \mathinner{.\,.} n]$

	cost	times
for $j \leftarrow 1$ **to** $n-1$	c_1	n
$\triangleright$ Find the minimum element in $A[j \mathinner{.\,.} n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n-1$
for $i \leftarrow j+1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n-1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n-1$

Selection sort analysis

SELECTION-SORT$(A, n) \triangleright A[1 \,..\, n]$

	cost	times
for $j \leftarrow 1$ **to** $n - 1$	c_1	n
$\triangleright$ Find the minimum element in $A[j \,..\, n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n - 1$
for $i \leftarrow j + 1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n - 1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n - 1$

Selection sort analysis

$\text{SELECTION-SORT}(A, n) \triangleright A[1 \,.\,.\, n]$

	cost	times
for $j \leftarrow 1$ **to** $n - 1$	c_1	n
$\triangleright$ Find the minimum element in $A[j \,.\,.\, n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n - 1$
for $i \leftarrow j + 1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n - 1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n - 1$

Selection sort analysis

SELECTION-SORT$(A, n) \rhd A[1 \,..\, n]$

	cost	*times*
for $j \leftarrow 1$ **to** $n-1$	c_1	n
$\rhd$ Find the minimum element in $A[j \,..\, n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n-1$
for $i \leftarrow j+1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n-1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n-1$

Selection sort analysis

SELECTION-SORT(A, n) $\triangleright$ $A[1 \,..\, n]$

	cost	times
for $j \leftarrow 1$ **to** $n - 1$	c_1	n
$\triangleright$ Find the minimum element in $A[j \,..\, n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n - 1$
for $i \leftarrow j + 1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n - 1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n - 1$

Selection sort analysis

SELECTION-SORT$(A, n) \triangleright A[1\,..\,n]$

	cost	times
for $j \leftarrow 1$ **to** $n-1$	c_1	n
$\triangleright$ Find the minimum element in $A[j\,..\,n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n-1$
for $i \leftarrow j+1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n-1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n-1$

Worst-case cost

$$T(n) = c_1 n + c_2(n-1) + c_3 \sum_{k=0}^{n} k + c_4 \sum_{k=0}^{n-1} k + c_5 \sum_{k=0}^{n-1} k + c_6(n-1) + c_7(n-1)$$

Selection sort analysis

SELECTION-SORT(A, n) $\triangleright$ $A[1\,..\,n]$

	cost	times
for $j \leftarrow 1$ **to** $n-1$	c_1	n
$\triangleright$ Find the minimum element in $A[j\,..\,n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n-1$
for $i \leftarrow j+1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n-1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n-1$

Worst-case cost

$$\begin{aligned} T(n) &= (c_1 + c_2 + c_6 + c_7)n + c_3\frac{n(n+1)}{2} + c_4\frac{n(n-1)}{2} \\ &\quad + c_5\frac{n(n-1)}{2} - (c_2 + c_6 + c_7) \end{aligned}$$

Selection sort analysis

$\textsc{Selection-Sort}(A, n) \triangleright A[1 \,..\, n]$

	cost	*times*
for $j \leftarrow 1$ **to** $n-1$	c_1	n
$\triangleright$ Find the minimum element in $A[j \,..\, n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n-1$
for $i \leftarrow j+1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n-1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n-1$

Worst-case cost

$$T(n) \;=\; cn^2 + dn + e \qquad \text{(where } c,\ d, \text{ and } e \text{ are constants)}$$

Selection sort analysis

SELECTION-SORT(A, n) $\triangleright$ $A[1\,..\,n]$

	cost	times
for $j \leftarrow 1$ **to** $n-1$	c_1	n
$\triangleright$ Find the minimum element in $A[j\,..\,n]$,		
and exchange the element with $A[j]$.	0	n
do $i_{min} \leftarrow j$	c_2	$n-1$
for $i \leftarrow j+1$ **to** n	c_3	$\sum_{k=0}^{n} k$
do if $A[i] < A[i_{min}]$	c_4	$\sum_{k=0}^{n-1} k$
then $i_{min} \leftarrow i$	c_5	$\sum_{k=0}^{n-1} k$
if $j \neq i_{min}$	c_6	$n-1$
then exchange $A[j] \leftrightarrow A[i_{min}]$	c_7	$n-1$

Observation

$T(n) = cn^2 + dn + e$

Selection sort is a quadratic algorithm in the worst- and best-cases!

Summing a sequence using *Divide and Conquer*

Algorithm

$\text{RECURSIVE-SUM}(A, p, q) \triangleright A[p \,..\, q]$

if $p > q$
  then return 0
elseif $p = q$
  then return $A[p]$
else $mid \leftarrow \frac{p+q}{2}$
  return $\text{RECURSIVE-SUM}(A, p, mid) +$
    $\text{RECURSIVE-SUM}(A, mid + 1, q)$

Analyzing *Divide and Conquer* recursive algorithms

$\text{RECURSIVE-SUM}(A, p, q) \triangleright A[p \mathinner{.\,.} q]$

	cost	*times*
if $p > q$	c_1	1
then return 0	c_2	1
elseif $p = q$	c_3	1
then return $A[p]$	c_4	1
else $mid \leftarrow \frac{p+q}{2}$	c_5	1
return $\text{RECURSIVE-SUM}(A, p, mid) +$		
$\text{RECURSIVE-SUM}(A, mid+1, q)$		
	1	$T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$

Analyzing *Divide and Conquer* recursive algorithms

$\textsc{Recursive-Sum}(A, p, q) \triangleright A[p \mathinner{..} q]$

	cost	*times*
if $p > q$	c_1	1
then return 0	c_2	1
elseif $p = q$	c_3	1
then return $A[p]$	c_4	1
else $mid \leftarrow \frac{p+q}{2}$	c_5	1
return $\textsc{Recursive-Sum}(A, p, mid) +$		
$\textsc{Recursive-Sum}(A, mid+1, q)$		
	1	$T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$

Total cost

$$\begin{aligned} T(n) &= (c_1 + c_2 + c_3 + c_4 + c_5) + T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) \\ &= T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + c \quad \text{(where } c \text{ is a constant)} \\ &= 2T(\tfrac{n}{2}) + c \quad \text{(letting } n = 2^k \text{ for some k)} \end{aligned}$$

Analyzing *Divide and Conquer* recursive algorithms

$\textsc{Recursive-Sum}(A, p, q) \triangleright A[p \,..\, q]$

	cost	*times*
if $p > q$	c_1	1
then return 0	c_2	1
elseif $p = q$	c_3	1
then return $A[p]$	c_4	1
else $mid \leftarrow \frac{p+q}{2}$	c_5	1
return $\textsc{Recursive-Sum}(A, p, mid) +$		
$\textsc{Recursive-Sum}(A, mid + 1, q)$		
	1	$T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$

Solving recurrences

How do you solve recurrences such as $T(n) = 2T(n/2) + c$?

Solving recurrences: *iterative substitution* method

$$
\begin{aligned}
T(n) &= 2T(n/2) + c \\
&= 2(2T(n/4) + c) + c = 4T(n/4) + 3c \\
&= 4(2T(n/8) + c) + 3c = 8T(n/8) + 7c \\
&= 8(2T(n/16) + c) + 7c = 16T(n/16) + 15c \\
&= 2^4 T(n/2^4) + (2^4 - 1)c \\
&\vdots \\
&= 2^k T(n/2^k) + (2^k - 1)c \\
&\triangleright \text{ setting } 2^k = n, \text{ so } k = \log_2 n \\
&= 2^{\log_2 n} T(n/n) + (n-1)c \\
&= nT(1) + (n-1)c \\
&= nd + (n-1)c \quad \text{where } T(1) = d, \text{ a constant} \\
&= (c+d)n - c
\end{aligned}
$$

Solving recurrences: *iterative substitution* method

$$
\begin{aligned}
T(n) &= 2T(n/2) + c \\
&= 2(2T(n/4) + c) + c = 4T(n/4) + 3c \\
&= 4(2T(n/8) + c) + 3c = 8T(n/8) + 7c \\
&= 8(2T(n/16) + c) + 7c = 16T(n/16) + 15c \\
&= 2^4 T(n/2^4) + (2^4 - 1)c \\
&\vdots \\
&= 2^k T(n/2^k) + (2^k - 1)c \\
&\triangleright \ \text{setting } 2^k = n, \text{ so } k = \log_2 n \\
&= 2^{\log_2 n} T(n/n) + (n-1)c \\
&= nT(1) + (n-1)c \\
&= nd + (n-1)c \quad \text{where } T(1) = d, \text{ a constant} \\
&= (c+d)n - c
\end{aligned}
$$

▷ $T(n)$ **is a *linear function* of n.**

Mathematical preliminaries – summations

Arithmetic series For $n \geq 0$,

$$\sum_{i=0}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \Theta(n^2)$$

.

Geometric series Let $c \neq 1$ be any constant, then for $n \geq 0$,

$$\sum_{i=0}^{n} c^i = 1 + c + c^2 + \ldots + c^n = \frac{c^{n+1} - 1}{c - 1}$$

.

if $0 < c < 1$, then $\Theta(1)$; if $c > 1$, then $\Theta(c^n)$.

Linear geometric series Let $c \neq 1$ be any constant, then for $n \geq 0$,

$$\sum_{i=0}^{n-1} ic^i = c + 2c^2 + 3c^3 + \ldots + nc^n = \frac{(n-1)c^{n+1} - nc^n + c}{(c-1)^2}$$

.

$= \Theta(nc^n)$

Harmonic series For $n \geq 0$,

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = (\ln n) + O(1)$$

Mathematical preliminaries

Polynomials

Given a nonnegative integer d, a ***polynomial in*** n ***of degree*** d is a function $p(n)$ of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

where the constants $a_0, a_1, \ldots, a_d$ are the ***coefficients*** of the polynomial and $a_d \neq 0$.

Exponentials

$$\begin{aligned}
a^0 &= 1, \\
a^1 &= a, \\
a^{-1} &= 1/a, \\
(a^m)^n &= a^{mn}, \\
(a^n)^m &= (a^m)^n, \\
a^m a^n &= a^{m+n}.
\end{aligned}$$

Logarithms

$$\begin{aligned}
a &= b^{\log_b a}, \\
\log_c(ab) &= \log_c a + \log_c b, \\
\log_b a^n &= n \log_b a, \\
\log_b a &= \frac{\log_c a}{\log_c b}, \\
\log_b(1/a) &= -\log_b a, \\
\log_b a &= \frac{1}{\log_a b}, \\
a^{\log_b c} &= c^{\log_b a}.
\end{aligned}$$

Contents

Growth of functions

Question

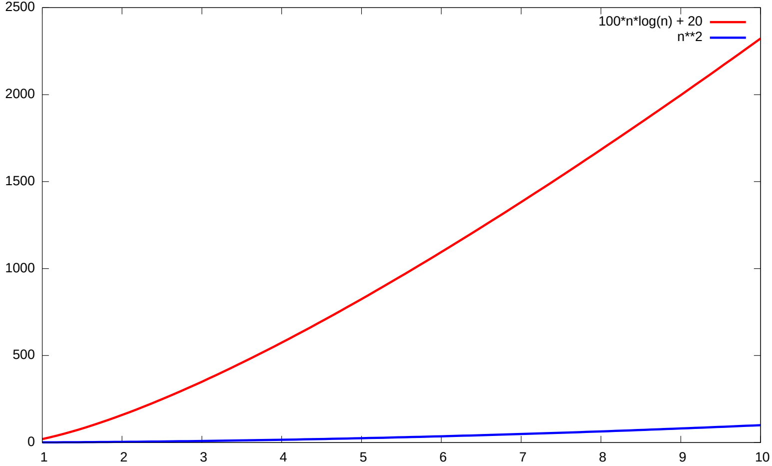
Which of the following two functions grows faster?

1. $T_1(n) = 100n \log n + 20$
2. $T_2(n) = n^2$

Growth of functions

1. $T_1(n) = 100n \log n + 20$
2. $T_2(n) = n^2$

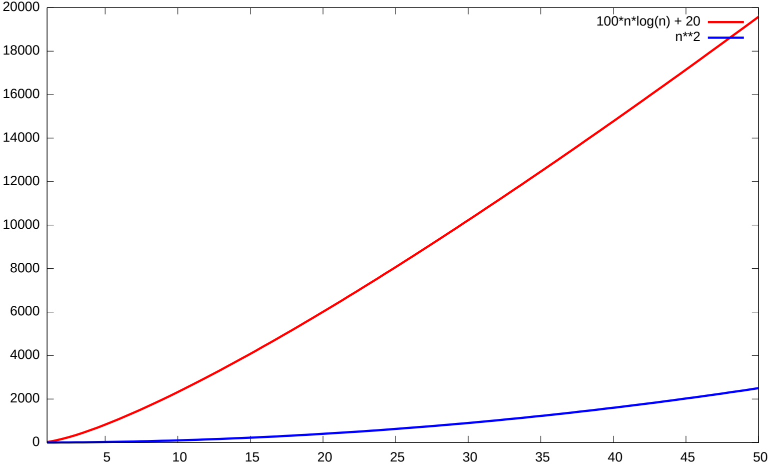
$\boxed{n = [1..10]}$



Growth of functions

1. $T_1(n) = 100n \log n + 20$
2. $T_2(n) = n^2$

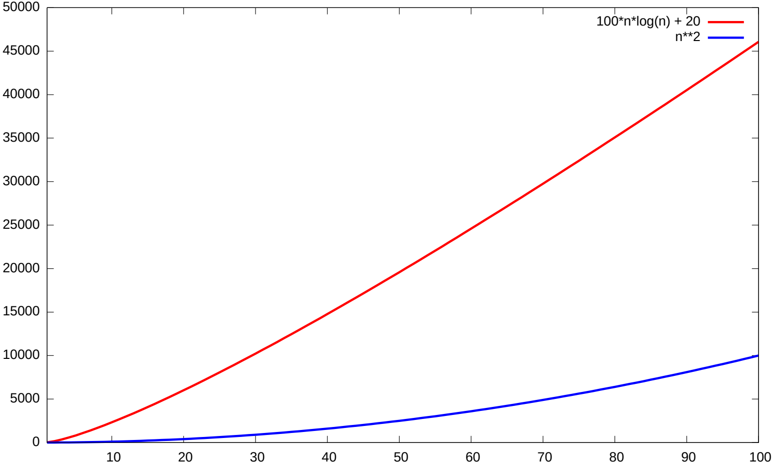
$\boxed{n = [1..50]}$



Growth of functions

1. $T_1(n) = 100n \log n + 20$
2. $T_2(n) = n^2$

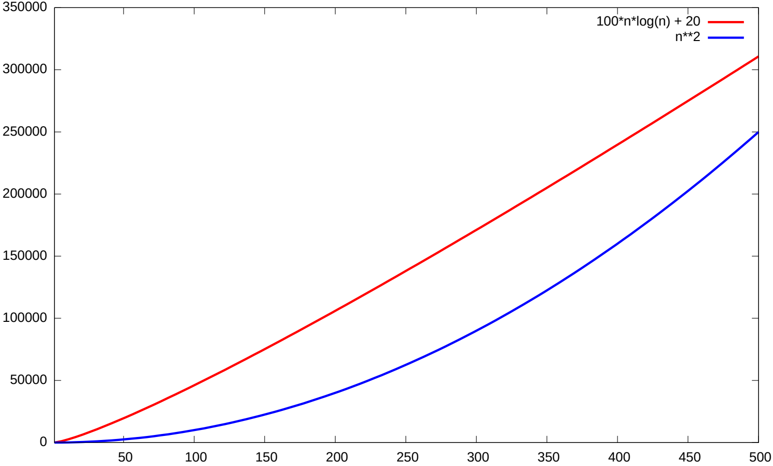
$\boxed{n = [1 \,..\, 100]}$



Growth of functions

1. $T_1(n) = 100n \log n + 20$
2. $T_2(n) = n^2$

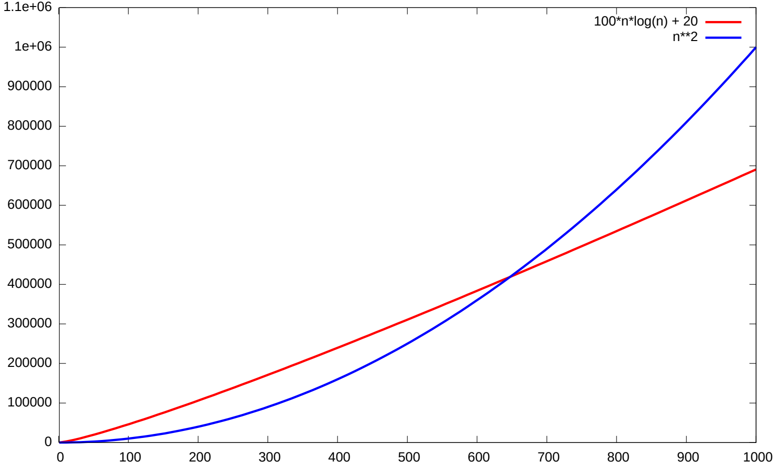
$\boxed{n = [1\,..\,500]}$



Growth of functions

1. $T_1(n) = 100n \log n + 20$
2. $T_2(n) = n^2$

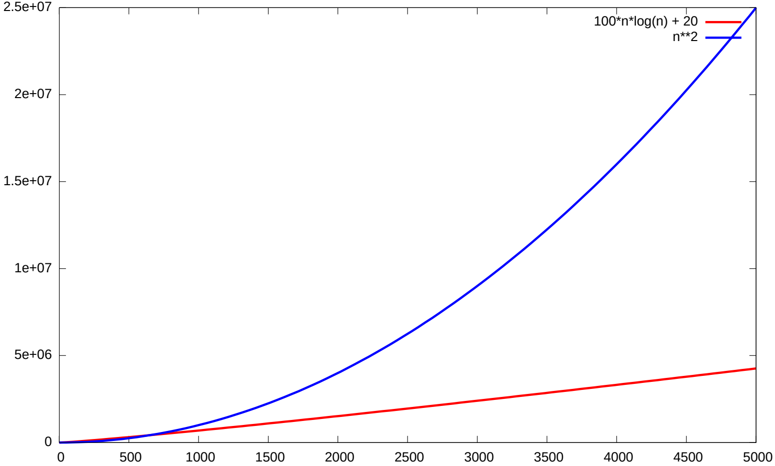
$\boxed{n = [1 \,..\, 1000]}$



Growth of functions

1. $T_1(n) = 100n \log n + 20$
2. $T_2(n) = n^2$

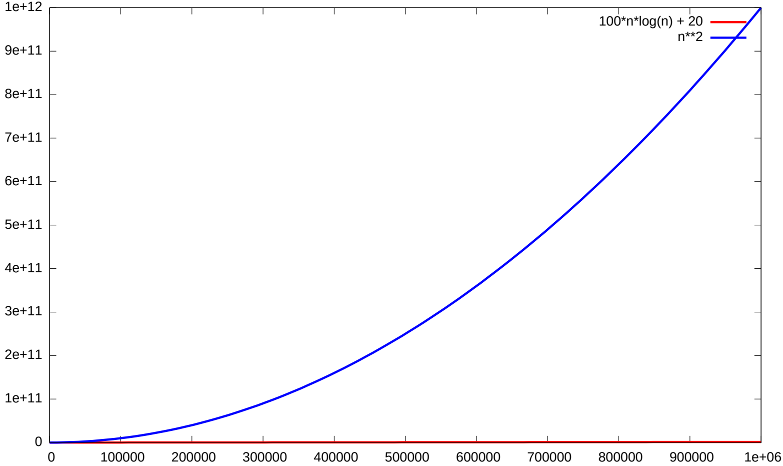
$\boxed{n = [1 \mathinner{..} 5000]}$



Growth of functions

1. $T_1(n) = 100n \log n + 20$
2. $T_2(n) = n^2$

$n \longrightarrow \infty$



Running times of different algorithms

size	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
10	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 4 s
30	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 m	10^{25} y
50	< 1 s	< 1 s	< 1 s	< 1 s	11 m	36 y	VL
100	< 1 s	< 1 s	< 1 s	1 s	12,892 y	10^{17} y	VL
1,000	< 1 s	< 1 s	1 s	18 m	VL	VL	VL
10,000	< 1 s	< 1 s	1 m	12 d	VL	VL	VL
100,000	< 1 s	2 s	3 h	32 y	VL	VL	VL
1,000,000	1 s	20 s	12 d	32,710 y	VL	VL	VL

1. *Assuming 1 Million high-level instructions per second*
2. *s: seconds, m: minutes, d: days, y: years,* VL*: very long!*

Running times of different algorithms

size	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
10	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 4 s
30	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 m	10^{25} y
50	< 1 s	< 1 s	< 1 s	< 1 s	11 m	36 y	VL
100	< 1 s	< 1 s	< 1 s	1 s	12,892 y	10^{17} y	VL
1,000	< 1 s	< 1 s	1 s	18 m	VL	VL	VL
10,000	< 1 s	< 1 s	1 m	12 d	VL	VL	VL
100,000	< 1 s	2 s	3 h	32 y	VL	VL	VL
1,000,000	1 s	20 s	12 d	32,710 y	VL	VL	VL

1. *Assuming 1 Million high-level instructions per second*
2. *s: seconds, m: minutes, d: days, y: years,* VL*: very long!*

Asymptotic complexity

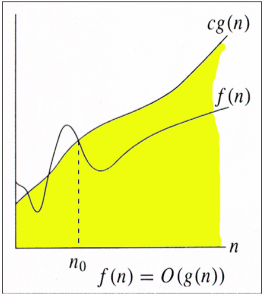
- Need a formalism to express the running time of an algorithm as a function of the input size n for large n.
- Expressed using only the highest-order term in the expression for the exact running time. For example, if running time is $13n^2 + 2n - 14$, say $\Theta(n^2)$.
- Describes behavior of function in the limit $n \to \infty$.
- Written using asymptotic notation Θ, O, and Ω (and their "distant cousins" o and ω), which define a set of functions.

  Θ or "Big-Theta" Describes the tight bound.
  O or "Big-Oh" Describes the upper bound.
  Ω or "Big-Omega" Describes the lower bound.

Asymptotic notation

Upper bound | Lower bound | Tight bound

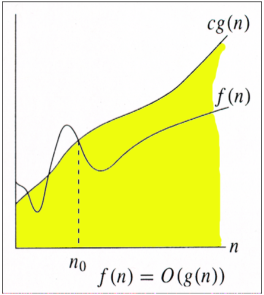
Can you find a function $g(n)$ that grows at least as fast as your algorithm $f(n)$ in the worst-case?



Asymptotic notation

Upper bound Lower bound Tight bound

Can you find a function $g(n)$ that grows at least as fast as your algorithm $f(n)$ in the worst-case?



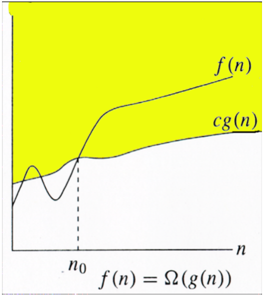
Definition

$O(\cdot)$: $f(n)$ is $O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, 0 \leq f(n) \leq cg(n)$.

Asymptotic notation

Upper bound **Lower bound** Tight bound

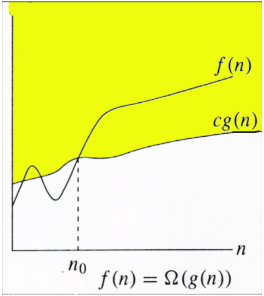
Can you find a function $g(n)$ that grows no faster than your algorithm $f(n)$ in the worst-case?



Asymptotic notation

Upper bound | **Lower bound** | Tight bound

Can you find a function $g(n)$ that grows no faster than your algorithm $f(n)$ in the worst-case?



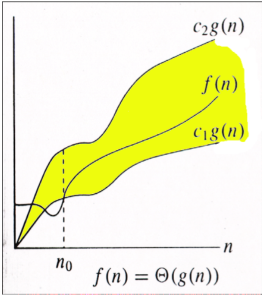
Definition

$\Omega(\cdot)$: $f(n)$ is $\Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$, $f(n) \geq cg(n)$.

Asymptotic notation

Upper bound Lower bound **Tight bound**

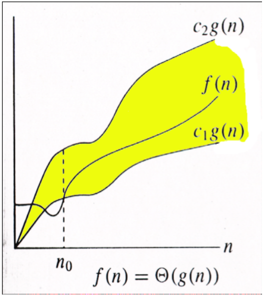
Can you find a function $g(n)$ that grows at the same rate as your algorithm $f(n)$ in the worst-case?



Asymptotic notation

Upper bound Lower bound **Tight bound**

Can you find a function $g(n)$ that grows at the same rate as your algorithm $f(n)$ in the worst-case?



Definition

$\Theta(\cdot)$: $f(n)$ is $\Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ and $n_0 > 0$ such that for all $n \geq n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$.

Asymptotic notation

Definition

- $O(\cdot)$ – upper bound. $f(n)$ is $O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, 0 \leq f(n) \leq cg(n)$.

Asymptotic notation

Definition

- $O(\cdot)$ – upper bound. $f(n)$ is $O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, 0 \leq f(n) \leq cg(n)$.
- $\Omega(\cdot)$ – lower bound. $f(n)$ is $\Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, f(n) \geq cg(n)$.

Asymptotic notation

Definition

- $O(\cdot)$ – upper bound. $f(n)$ is $O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, 0 \leq f(n) \leq cg(n)$.
- $\Omega(\cdot)$ – lower bound. $f(n)$ is $\Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, f(n) \geq cg(n)$.
- $\Theta(\cdot)$ – tight bound. $f(n)$ is $\Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ and $n_0 > 0$ such that for all $n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$.

Asymptotic notation

Definition

- $O(\cdot)$ – upper bound. $f(n)$ is $O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, 0 \leq f(n) \leq cg(n)$.
- $\Omega(\cdot)$ – lower bound. $f(n)$ is $\Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, f(n) \geq cg(n)$.
- $\Theta(\cdot)$ – tight bound. $f(n)$ is $\Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ and $n_0 > 0$ such that for all $n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$.

  $f(n)$ is $\Theta(g(n))$ **iff** $f(n)$ is $O(g(n))$ **and** $f(n)$ is $\Omega(g(n))$.

Asymptotic notation

Definition

- $O(\cdot)$ – upper bound. $f(n)$ is $O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, 0 \leq f(n) \leq cg(n)$.
- $\Omega(\cdot)$ – lower bound. $f(n)$ is $\Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, f(n) \geq cg(n)$.
- $\Theta(\cdot)$ – tight bound. $f(n)$ is $\Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ and $n_0 > 0$ such that for all $n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$.

  $f(n)$ is $\Theta(g(n))$ **iff** $f(n)$ is $O(g(n))$ **and** $f(n)$ is $\Omega(g(n))$.

Example

$f(n) = 32n^2 + 17n + 32$.

Asymptotic notation

Definition

- $O(\cdot)$ – upper bound. $f(n)$ is $O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, 0 \leq f(n) \leq cg(n)$.
- $\Omega(\cdot)$ – lower bound. $f(n)$ is $\Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, f(n) \geq cg(n)$.
- $\Theta(\cdot)$ – tight bound. $f(n)$ is $\Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ and $n_0 > 0$ such that for all $n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$.

  $f(n)$ is $\Theta(g(n))$ **iff** $f(n)$ is $O(g(n))$ **and** $f(n)$ is $\Omega(g(n))$.

Example

$f(n) = 32n^2 + 17n + 32$.

- $f(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.

Asymptotic notation

Definition

- $O(\cdot)$ – upper bound. $f(n)$ is $O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, 0 \leq f(n) \leq cg(n)$.
- $\Omega(\cdot)$ – lower bound. $f(n)$ is $\Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, f(n) \geq cg(n)$.
- $\Theta(\cdot)$ – tight bound. $f(n)$ is $\Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ and $n_0 > 0$ such that for all $n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$.

  $f(n)$ is $\Theta(g(n))$ **iff** $f(n)$ is $O(g(n))$ **and** $f(n)$ is $\Omega(g(n))$.

Example

$f(n) = 32n^2 + 17n + 32$.

- $f(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $f(n)$ is **not** $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Asymptotic notation summary

Notation	means ...	think ...	e.g.,	$\lim \frac{f(n)}{g(n)}$ [1]
$f(n) = O(g(n))$	$\exists c > 0, n_0 > 0 : \forall n \geq n_0, 0 \leq f(n) \leq cg(n)$.	Upper bound	$100n^2 = O(n^3)$	$\neq \infty$
$f(n) = \Omega(g(n))$	$\exists c > 0, n_0 > 0 : \forall n \geq n_0, f(n) \geq cg(n)$.	Lower bound	$100n^2 = \Omega(n)$	> 0
$f(n) = \Theta(g(n))$	$\exists c_1, c_2 > 0, n_0 > 0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$.	Tight bound	$100n^2 = \Theta(n^2)$	$= \text{CONST}$
$f(n) = o(g(n))$	$\exists n_0 > 0 : \forall c > 0, n \geq n_0, 0 \leq f(n) \leq cg(n)$.	Weak upper bound	$100n^2 = o(n^6)$	$= 0$
$f(n) = \omega(g(n))$	$\exists n_0 > 0 : \forall c > 0, n \geq n_0, f(n) \geq cg(n)$.	Weak lower bound	$100n^2 = \omega(n)$	$= \infty$

[1] if the limit $\lim_{n\to\infty} f(n)/g(n)$ exists

Properties of asymptotic notations

Transitivity:

$$\begin{array}{lllllllll} f(n) & = & \Theta(g(n)) & \text{and} & g(n) & = & \Theta(h(n)) & \text{imply} & f(n) = \Theta(h(n)), \\ f(n) & = & O(g(n)) & \text{and} & g(n) & = & O(h(n)) & \text{imply} & f(n) = O(h(n)), \\ f(n) & = & \Omega(g(n)) & \text{and} & g(n) & = & \Omega(h(n)) & \text{imply} & f(n) = \Omega(h(n)). \end{array}$$

Reflexivity:

$$\begin{array}{lll} f(n) & = & \Theta(f(n)), \\ f(n) & = & O(f(n)), \\ f(n) & = & \Omega(f(n)). \end{array}$$

Symmetry:

$$f(n) = \Theta(g(n)) \quad \text{if and only if} \quad g(n) = \Theta(f(n)).$$

Transpose Symmetry:

$$f(n) = O(g(n)) \quad \text{if and only if} \quad g(n) = \Omega(f(n)).$$

Linearity:

$$\textstyle\sum_{k=1}^{n} \Theta(f_k) = \Theta(\sum_{k=1}^{n} f_k)$$

Examples of asymptotic growth

1. $10n + 3 = O(n)$, if $c \geq 11$ and $n_0 \geq 2$.

Examples of asymptotic growth

1. $10n + 3 = O(n)$, if $c \geq 11$ and $n_0 \geq 2$.
2. $10n + 3 = O(n^2)$, if $c \geq 3$ and $n_0 \geq 4$.

Examples of asymptotic growth

1. $10n + 3 = O(n)$, if $c \geq 11$ and $n_0 \geq 2$.
2. $10n + 3 = O(n^2)$, if $c \geq 3$ and $n_0 \geq 4$.
3. $n^3 = \Omega(n^2)$, if $c = 1$ and $n_0 = 0$.

Examples of asymptotic growth

1. $10n + 3 = O(n)$, if $c \geq 11$ and $n_0 \geq 2$.
2. $10n + 3 = O(n^2)$, if $c \geq 3$ and $n_0 \geq 4$.
3. $n^3 = \Omega(n^2)$, if $c = 1$ and $n_0 = 0$.
4. $\frac{n(n-1)}{2} = \Theta(n^2)$.
   Upper bound: $\frac{n^2}{2} - \frac{n}{2} \leq \frac{n^2}{2}$ for all n, so $c_1 = \frac{1}{2}$;
   Lower bound: $\frac{1}{2}n^2 - \frac{n}{2} > \frac{n^2}{2} - \frac{n^2}{4} = \frac{n^2}{4}$ for all $n \geq 2$, so $c_2 = \frac{1}{4}$, and $n_0 = 2$.

Examples of asymptotic growth

1. $10n + 3 = O(n)$, if $c \geq 11$ and $n_0 \geq 2$.
2. $10n + 3 = O(n^2)$, if $c \geq 3$ and $n_0 \geq 4$.
3. $n^3 = \Omega(n^2)$, if $c = 1$ and $n_0 = 0$.
4. $\frac{n(n-1)}{2} = \Theta(n^2)$.
   Upper bound: $\frac{n^2}{2} - \frac{n}{2} \leq \frac{n^2}{2}$ for all n, so $c_1 = \frac{1}{2}$;
   Lower bound: $\frac{1}{2}n^2 - \frac{n}{2} > \frac{n^2}{2} - \frac{n^2}{4} = \frac{n^2}{4}$ for all $n \geq 2$, so $c_2 = \frac{1}{4}$, and $n_0 = 2$.
5. $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.
   $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ for all $n \geq n_0$. Dividing by n^2 yields:
   $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$.
   $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$, and $n_0 = 7$.

Examples of asymptotic growth

1. $10n + 3 = O(n)$, if $c \geq 11$ and $n_0 \geq 2$.
2. $10n + 3 = O(n^2)$, if $c \geq 3$ and $n_0 \geq 4$.
3. $n^3 = \Omega(n^2)$, if $c = 1$ and $n_0 = 0$.
4. $\frac{n(n-1)}{2} = \Theta(n^2)$.
   Upper bound: $\frac{n^2}{2} - \frac{n}{2} \leq \frac{n^2}{2}$ for all n, so $c_1 = \frac{1}{2}$;
   Lower bound: $\frac{1}{2}n^2 - \frac{n}{2} > \frac{n^2}{2} - \frac{n^2}{4} = \frac{n^2}{4}$ for all $n \geq 2$, so $c_2 = \frac{1}{4}$, and $n_0 = 2$.
5. $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.
   $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \quad$ for all $n \geq n_0$. Dividing by n^2 yields:
   $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$.
   $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$, and $n_0 = 7$.
6. $2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$.

Examples of asymptotic growth

1. $n^2/2 - 3n = O(n^2)$

Examples of asymptotic growth

1. $n^2/2 - 3n = O(n^2)$
2. $1 + 4n = O(n)$

Examples of asymptotic growth

1. $n^2/2 - 3n = O(n^2)$
2. $1 + 4n = O(n)$
3. $\log_{10} n = \frac{\log_2 n}{\log_2 10} = O(\log_2 n) = O(\log n)$

Examples of asymptotic growth

1. $n^2/2 - 3n = O(n^2)$
2. $1 + 4n = O(n)$
3. $\log_{10} n = \frac{\log_2 n}{\log_2 10} = O(\log_2 n) = O(\log n)$
4. $\sin n = O(1)$, $10 = O(1)$, $10^{10} = O(1)$

Examples of asymptotic growth

1. $n^2/2 - 3n = O(n^2)$
2. $1 + 4n = O(n)$
3. $\log_{10} n = \frac{\log_2 n}{\log_2 10} = O(\log_2 n) = O(\log n)$
4. $\sin n = O(1)$, $10 = O(1)$, $10^{10} = O(1)$
5. $\sum_{i=1}^{n} i^2 \leq n \cdot n^2 = O(n^3)$

Examples of asymptotic growth

1. $n^2/2 - 3n = O(n^2)$
2. $1 + 4n = O(n)$
3. $\log_{10} n = \frac{\log_2 n}{\log_2 10} = O(\log_2 n) = O(\log n)$
4. $\sin n = O(1)$, $10 = O(1)$, $10^{10} = O(1)$
5. $\sum_{i=1}^{n} i^2 \leq n \cdot n^2 = O(n^3)$
6. $\sum_{i=1}^{n} i \leq n \cdot n = O(n^2)$

Examples of asymptotic growth

1. $n^2/2 - 3n = O(n^2)$
2. $1 + 4n = O(n)$
3. $\log_{10} n = \frac{\log_2 n}{\log_2 10} = O(\log_2 n) = O(\log n)$
4. $\sin n = O(1)$, $10 = O(1)$, $10^{10} = O(1)$
5. $\sum_{i=1}^{n} i^2 \leq n \cdot n^2 = O(n^3)$
6. $\sum_{i=1}^{n} i \leq n \cdot n = O(n^2)$
7. 2^{10n} is not $O(2^n)$

Examples of asymptotic growth

1. $n^2/2 - 3n = O(n^2)$
2. $1 + 4n = O(n)$
3. $\log_{10} n = \frac{\log_2 n}{\log_2 10} = O(\log_2 n) = O(\log n)$
4. $\sin n = O(1)$, $10 = O(1)$, $10^{10} = O(1)$
5. $\sum_{i=1}^{n} i^2 \leq n \cdot n^2 = O(n^3)$
6. $\sum_{i=1}^{n} i \leq n \cdot n = O(n^2)$
7. 2^{10n} is not $O(2^n)$
8. $\log(n!) = \log(n) + \log(n-1) + \cdots + \log 1 = O(n \log n)$

Examples of asymptotic growth

1. $n^2/2 - 3n = O(n^2)$
2. $1 + 4n = O(n)$
3. $\log_{10} n = \frac{\log_2 n}{\log_2 10} = O(\log_2 n) = O(\log n)$
4. $\sin n = O(1)$, $10 = O(1)$, $10^{10} = O(1)$
5. $\sum_{i=1}^{n} i^2 \leq n \cdot n^2 = O(n^3)$
6. $\sum_{i=1}^{n} i \leq n \cdot n = O(n^2)$
7. 2^{10n} is not $O(2^n)$
8. $\log(n!) = \log(n) + \log(n-1) + \cdots + \log 1 = O(n \log n)$
9. $\sum_{i=1}^{n} \frac{1}{i} = O(\log n)$

Ordering by asymptotic growth

- $n \log n$
- 2^n
- $\log n$
- n^2
- $n^{1,000,000}$
- $n!$
- n^4
- $\sqrt{n}$
- n

$\xrightarrow{\text{Order by asymptotic growth}}$

Ordering by asymptotic growth

- $n \log n$
- 2^n
- $\log n$
- n^2
- $n^{1,000,000}$
- $n!$
- n^4
- $\sqrt{n}$
- n

Order by asymptotic growth $\rightsquigarrow$

- $\log n$

Ordering by asymptotic growth

- $n \log n$
- 2^n
- $\log n$
- n^2
- $n^{1,000,000}$
- $n!$
- n^4
- $\sqrt{n}$
- n

Order by asymptotic growth $\rightsquigarrow$

- $\log n$
- $\sqrt{n}$

Ordering by asymptotic growth

- $n \log n$
- 2^n
- $\log n$
- n^2
- $n^{1,000,000}$
- $n!$
- n^4
- $\sqrt{n}$
- n

Order by asymptotic growth $\rightsquigarrow$

- $\log n$
- $\sqrt{n}$
- n

Ordering by asymptotic growth

- $n \log n$
- 2^n
- $\log n$
- n^2
- $n^{1,000,000}$
- $n!$
- n^4
- $\sqrt{n}$
- n

Order by asymptotic growth
$\rightsquigarrow$

- $\log n$
- $\sqrt{n}$
- n
- $n \log n$

Ordering by asymptotic growth

- $n \log n$
- 2^n
- $\log n$
- n^2
- $n^{1,000,000}$
- $n!$
- n^4
- $\sqrt{n}$
- n

Order by asymptotic growth $\rightsquigarrow$

- $\log n$
- $\sqrt{n}$
- n
- $n \log n$
- n^2

Ordering by asymptotic growth

- $n \log n$
- 2^n
- $\log n$
- n^2
- $n^{1,000,000}$
- $n!$
- n^4
- $\sqrt{n}$
- n

Order by asymptotic growth $\rightsquigarrow$

- $\log n$
- $\sqrt{n}$
- n
- $n \log n$
- n^2
- n^4

Ordering by asymptotic growth

- $n \log n$
- 2^n
- $\log n$
- n^2
- $n^{1,000,000}$
- $n!$
- n^4
- $\sqrt{n}$
- n

$\xrightarrow{\text{Order by asymptotic growth}}$

- $\log n$
- $\sqrt{n}$
- n
- $n \log n$
- n^2
- n^4
- $n^{1,000,000}$

Ordering by asymptotic growth

- $n \log n$
- 2^n
- $\log n$
- n^2
- $n^{1,000,000}$
- $n!$
- n^4
- $\sqrt{n}$
- n

$\xrightarrow{\text{Order by asymptotic growth}}$

- $\log n$
- $\sqrt{n}$
- n
- $n \log n$
- n^2
- n^4
- $n^{1,000,000}$
- 2^n

Ordering by asymptotic growth

- $n \log n$
- 2^n
- $\log n$
- n^2
- $n^{1,000,000}$
- $n!$
- n^4
- $\sqrt{n}$
- n

$\xrightarrow{\text{Order by asymptotic growth}}$

- $\log n$
- $\sqrt{n}$
- n
- $n \log n$
- n^2
- n^4
- $n^{1,000,000}$
- 2^n
- $n!$

Ordering by asymptotic growth

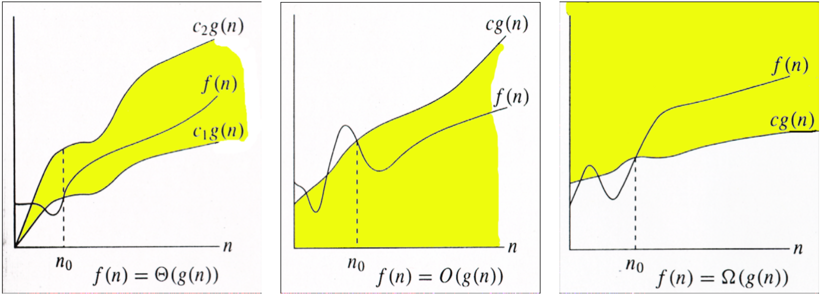
- $n \log n$
- 2^n
- $\log n$
- n^2
- $n^{1,000,000}$
- $n!$
- n^4
- $\sqrt{n}$
- n

Order by asymptotic growth $\rightsquigarrow$

- $\log n$
- $\sqrt{n}$
- n
- $n \log n$
- n^2
- n^4
- $n^{1,000,000}$
- $\boxed{2^n}$
- $n!$

x^k beats n^k for any fixed k and $x > 1$

Relationship of Θ, O and Ω



$O(\cdot)$ summary

$O(1)$	:	Great. Constant time. Can't beat this!
$O(\log\log n)$	:	Very fast, almost constant time.
$O(\log n)$	:	*logarithmic time*. Very good.
$O((\log n)^k)$	:	(where k is a constant) *polylogarithmic time*. Not bad.
$O(n^p)$	:	(where $0 < p < 1$ is a constant) Beats $O((\log n)^k)$ regardless of how large k is or how small p is.
$O(n)$	:	*linear time*. About the best you can do if your algorithm has to look at all the data.
$O(n \log n)$	:	*log-linear time*. Shows up in many places.
$O(n^2)$	:	*quadratic time*.
$O(n^k)$	:	(where k is a constant) *polynomial time*. Only if k is not too large.
$O(2^n)$, $O(n!)$	:	*exponential time*. Unusable for any problem of reasonable size ($n > 20$?).

Contents

Correctness proofs

- Proving, beyond any doubt, that an algorithm is correct.
  1. **Partial correctness:** Prove that the algorithm producess correct output when it terminates.
  2. **Total correctness:** Prove that the algorithm will necessarily terminate.
- Proof techniques
  1. Proof by Construction.
  2. Proof by Induction.
  3. Proof by Contradiction.

Loop invariants

Definition

Loop invariants are logical expressions with the following properties:

1. **Initialization:** Holds true before the first iteration of a loop.
2. **Maintenance:** If it's true before an iteration of a loop, it holds true at the beginning of the next iteration.
3. **Termination:** When the loop terminates, the invariant – along with the fact that the loop terminated – gives a useful property that helps to show that the loop is correct.

Similar to Mathematical induction. (How?)

Example of loop invariant

Algorithm to find the maximum value in a sequence

FIND-MAXIMUM$(A, n) \triangleright A[1 \,..\, n]$

```
max ← A[1]
for i ← 2 to n
    do if A[i] > max
        then max ← A[i]
return max
```


Example of loop invariant

Algorithm to find the maximum value in a sequence

FIND-MAXIMUM(A, n) $\triangleright$ $A[1 \,..\, n]$

```
1   max ← A[1]
2   for i ← 2 to n
3          do if A[i] > max
4                 then max ← A[i]
5   return max
```

Loop invariant

$\triangleright$ **At the start of each for loop, *max* contains the largest element in $A[1 \,..\, i-1]$.**

Example of loop invariant

Algorithm to find the maximum value in a sequence

FIND-MAXIMUM(A, n) $\triangleright$ $A[1 \,..\, n]$

```
max ← A[1]
for i ← 2 to n
    do if A[i] > max
        then max ← A[i]
return max
```

Loop invariant

$\triangleright$ **At the start of each for loop, *max* contains the largest element in $A[1 \,..\, i-1]$.**

Initialization: Before the first iteration, $max = A[1]$, so the loop invariant trivially holds. $\surd$

Example of loop invariant

Algorithm to find the maximum value in a sequence

FIND-MAXIMUM$(A, n) \triangleright A[1 \,..\, n]$

```
1   max ← A[1]
2   for i ← 2 to n
3         do if A[i] > max
4               then max ← A[i]
5   return max
```

Loop invariant

$\triangleright$ At the start of each for loop, max contains the largest element in $A[1 \,..\, i-1]$.

Maintenance: At the end of $i - 1^{th}$ iteration, the value of max is updated to hold the larger of max and $A[i]$ (see line 4), so max contains the largest value in $A[1 \,..\, i-1]$ in the beginning of the next (i^{th}) iteration. $\surd$

Example of loop invariant

Algorithm to find the maximum value in a sequence

FIND-MAXIMUM(A, n) $\triangleright$ $A[1 \, . \, . \, n]$

```
1   max ← A[1]
2   for i ← 2 to n
3       do if A[i] > max
4           then max ← A[i]
5   return max
```

Loop invariant

$\triangleright$ **At the start of each for loop, *max* contains the largest element in $A[1 \, . \, . \, i-1]$.**

Termination: Since the value of *max* is updated to hold the larger of *max* and $A[i]$ (see line 4) just before the loop terminated, and since $i = n + 1$ after the loop terminated, *max* contains the largest value in $A[1 \, . \, . \, n]$ or $A[1 \, . \, . \, i-1]$ after the loop. $\surd$

Another example of loop invariant

Algorithm to sort a sequence using insertion sort

INSERTION-SORT$(A, n) \triangleright A[1 \,..\, n]$

```
for j ← 2 to n
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

Another example of loop invariant

Algorithm to sort a sequence using insertion sort

INSERTION-SORT$(A, n) \triangleright A[1 \,..\, n]$

```
for j ← 2 to n
        do key ← A[j]
           i ← j − 1
           while i > 0 and A[i] > key
                 do A[i + 1] ← A[i]
                    i ← i − 1
           A[i + 1] ← key
```

Loop invariant

$\triangleright$ **At the start of each for loop, $A[1 \,..\, j-1]$ consists of elements originally in $A[1 \,..\, j-1]$ but in sorted order.**

Another example of loop invariant

Algorithm to sort a sequence using insertion sort

INSERTION-SORT(A, n) $\triangleright$ $A[1 \mathinner{..} n]$

```
for j ← 2 to n
        do key ← A[j]
           i ← j − 1
           while i > 0 and A[i] > key
                 do A[i + 1] ← A[i]
                    i ← i − 1
           A[i + 1] ← key
```

Loop invariant

$\triangleright$ **At the start of each for loop, $A[1 \mathinner{..} j-1]$ consists of elements originally in $A[1 \mathinner{..} j-1]$ but in sorted order.**

Initialization: Before the first iteration, $j = 2$, and so the loop invariant trivially holds. $\surd$

Another example of loop invariant

Algorithm to sort a sequence using insertion sort

INSERTION-SORT$(A, n) \triangleright A[1 \,.\,.\, n]$

```
for j ← 2 to n
      do key ← A[j]
         i ← j − 1
         while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i ← i − 1
         A[i + 1] ← key
```

Loop invariant

$\triangleright$ **At the start of each for loop, $A[1 \,.\,.\, j-1]$ consists of elements originally in $A[1 \,.\,.\, j-1]$ but in sorted order.**

Maintenance: The inner while loop finds the position i with $A[i] \leq key$, and shifts $A[j-1], A[j-2], \ldots, A[i+1]$ right by one position. Then *key*, formerly known as $A[j]$, is placed in position $i+1$ so that $A[i] \leq A[i+1] < A[i+2]$.

$A[1 \,.\,.\, j-1]$ sorted $+ A[j] \rightarrow A[1 \,.\,.\, j]$ sorted $\checkmark$

Another example of loop invariant

Algorithm to sort a sequence using insertion sort

```
INSERTION-SORT(A, n) ▷ A[1 . . n]
for j ← 2 to n
     do key ← A[j]
        i ← j − 1
        while i > 0 and A[i] > key
             do A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← key
```

Loop invariant

▷ **At the start of each for loop, $A[1\,..\,j-1]$ consists of elements originally in $A[1\,..\,j-1]$ but in sorted order.**

Termination: The loop terminates, when $j = n + 1$. Then the invariant states: "$A[1\,..\,n]$ consists of elements originally in $A[1\,..\,n]$ but in sorted order." $\surd$

Contents

Recurrences

Definition

A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.

Recurrences

Definition

A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.

Example

The worst-case running time for MERGE-SORT can be described using the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

The closed-form solution is $T(n) = \Theta(n \log n)$.

Recurrences

Definition

A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.

Example

The worst-case running time for MERGE-SORT can be described using the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

The closed-form solution is $T(n) = \Theta(n \log n)$.

Question

How do we get the closed form solutions of such recurrences?

Recurrence solution methods

1. **Substitution method:** Use algebraic manipulation to compute bounds.
   1. Guess and Test: Guess a bound, and then use mathematical induction to prove our guess correct. Must start with a good guess.
   2. Iterative substitution: Algebraically expand the recurrence, until a pattern emerges, which you can use to solve for the correct bound. Often involves very elaborate algebraic manipulation.
2. **Recursion-tree method:** Convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion, and then use the tree to solve the recurrence. Often very intuitive.
3. **Master method:** Provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1, b > 1$, and $f(n)$ is a given function. Requires memorization of three cases.

Recurrence solution methods

1. Substitution method: Use algebraic manipulation to compute bounds.
   1. Guess and Test: Guess a bound, and then use mathematical induction to prove our guess correct. Must start with a good guess.
   2. Iterative substitution: Algebraically expand the recurrence, until a pattern emerges, which you can use to solve for the correct bound. Often involves very elaborate algebraic manipulation.
2. Recursion-tree method: Convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion, and then use the tree to solve the recurrence. Often very intuitive.
3. Master method: Provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1, b > 1$, and $f(n)$ is a given function. Requires memorization of three cases.

Recurrence solution methods

1. Substitution method: Use algebraic manipulation to compute bounds.
   1. Guess and Test: Guess a bound, and then use mathematical induction to prove our guess correct. Must start with a good guess.
   2. Iterative substitution: Algebraically expand the recurrence, until a pattern emerges, which you can use to solve for the correct bound. Often involves very elaborate algebraic manipulation.
2. Recursion-tree method: Convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion, and then use the tree to solve the recurrence. Often very intuitive.
3. Master method: Provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1, b > 1$, and $f(n)$ is a given function. Requires memorization of three cases.

Recurrence solution methods

1. Substitution method: Use algebraic manipulation to compute bounds.
   1. Guess and Test: Guess a bound, and then use mathematical induction to prove our guess correct. Must start with a good guess.
   2. Iterative substitution: Algebraically expand the recurrence, until a pattern emerges, which you can use to solve for the correct bound. Often involves very elaborate algebraic manipulation.
2. Recursion-tree method: Convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion, and then use the tree to solve the recurrence. Often very intuitive.
3. Master method: Provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1, b > 1$, and $f(n)$ is a given function. Requires memorization of three cases.

Recurrence solution methods

1. Substitution method: Use algebraic manipulation to compute bounds.
    1. Guess and Test: Guess a bound, and then use mathematical induction to prove our guess correct. Must start with a good guess.
    2. Iterative substitution: Algebraically expand the recurrence, until a pattern emerges, which you can use to solve for the correct bound. Often involves very elaborate algebraic manipulation.
2. Recursion-tree method: Convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion, and then use the tree to solve the recurrence. Often very intuitive.
3. Master method: Provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1, b > 1$, and $f(n)$ is a given function. Requires memorization of three cases.

"Guess and Test" substitution example

Recurrence: MERGE-SORT $T(n) = 2T(n/2) + n, n > 1$, with $T(1) = 1$.

Guess: $T(n) = n \lg n + n$.

Induction:

Basis: $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$.

Hypothesis: $T(k) = k \lg k + k, \text{ for all } k < n$.

Inductive step:

$$\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(n/2 \lg(n/2) + (n/2)) + n \\
&= n(\lg(n/2)) + 2n \\
&= n \lg n - n \lg 2 + 2n \\
&= n \lg n - n + 2n \\
&= n \lg n + n
\end{aligned}$$

Iterative substitution example: Binary Search

BINARY-SEARCH $T(n) = T(n/2) + 1$, with $T(0) = T(1) = 1$.

$$
\begin{aligned}
T(n) &= T(n/2) + 1 \\
&= (T(n/4) + 1) + 1 = T(n/4) + 2 \\
&= (T(n/8) + 1) + 2 = T(n/8) + 3 \\
&\vdots \\
&= T(n/2^k) + k \\
&\triangleright \text{ setting } 2^k = n, \text{ so } k = \log_2 n \\
&= T(n/n) + \log_2 n \\
&= T(1) + \log_2 n \\
&= \log_2 n \\
&= \boldsymbol{\Theta(\log n)}
\end{aligned}
$$

Iterative substitution example: Merge Sort

MERGE-SORT $T(n) = 2T(n/2) + cn$, with $T(0) = T(1) = 1$.

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2(2T(n/4) + cn/2) + cn = 4T(n/4) + 2cn \\
&= 4(2T(n/8) + cn/4) + 2cn = 8T(n/8) + 3n \\
&= 8(2T(n/16) + cn/8) + 3cn = 16T(n/16) + 4n \\
&\vdots \\
&= 2^k T(n/2^k) + kn \\
&\triangleright \text{ setting } 2^k = n, \text{ so } k = \log_2 n \\
&= nT(1) + \log_2 nn \\
&= n + n\log_2 n = n(\log_2 n + 1) \\
&= \boldsymbol{\Theta(n \log n)}
\end{aligned}
$$

Recursion tree example: Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

- Expand the tree until you reach the base case (problem size of 1 in this case).
- In this case, the cost per step is cn **plus** the cost of the two recursive calls.

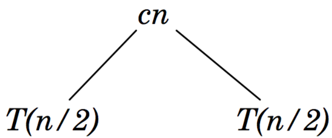
Recursion tree example: Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$T(n)$$

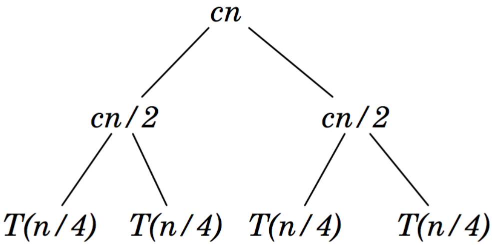
Recursion tree example: Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



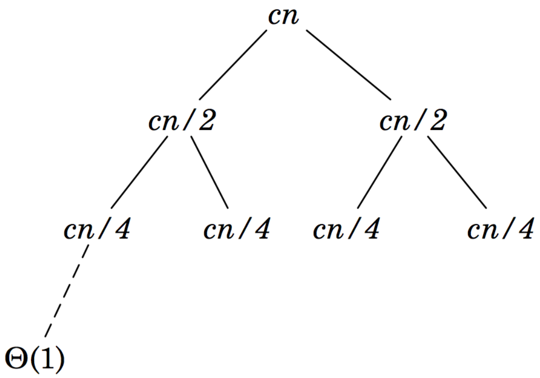
Recursion tree example: Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



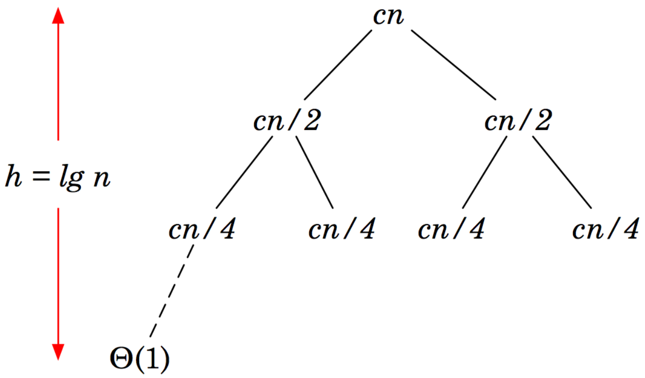
Recursion tree example: Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



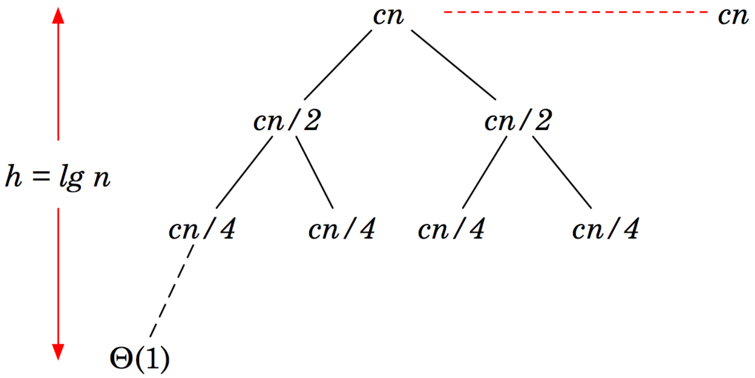
Recursion tree example: Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



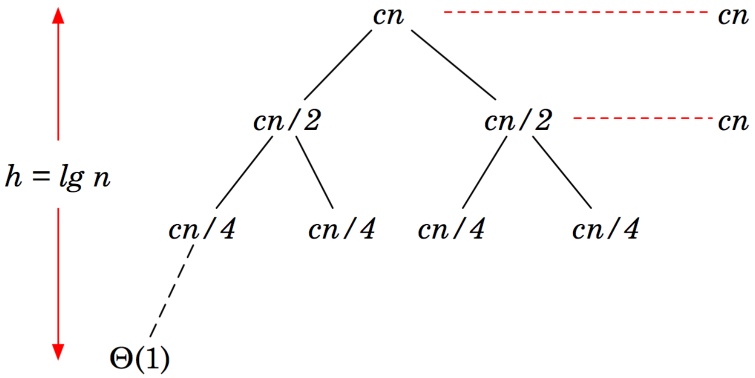
Recursion tree example: Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



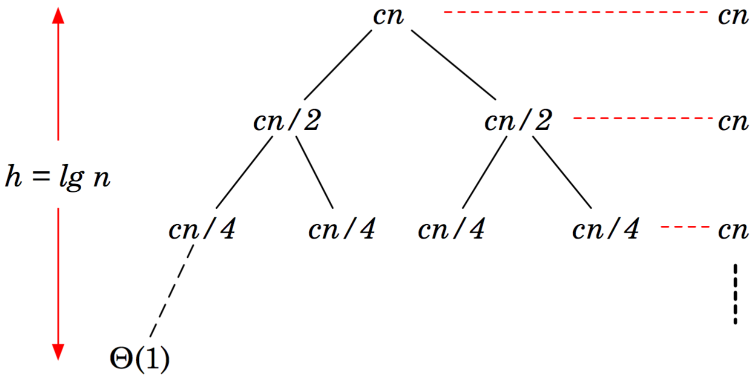
Recursion tree example: Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



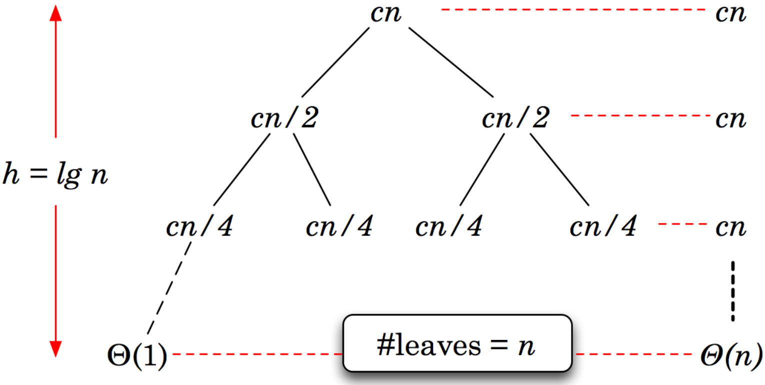
Recursion tree example: Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



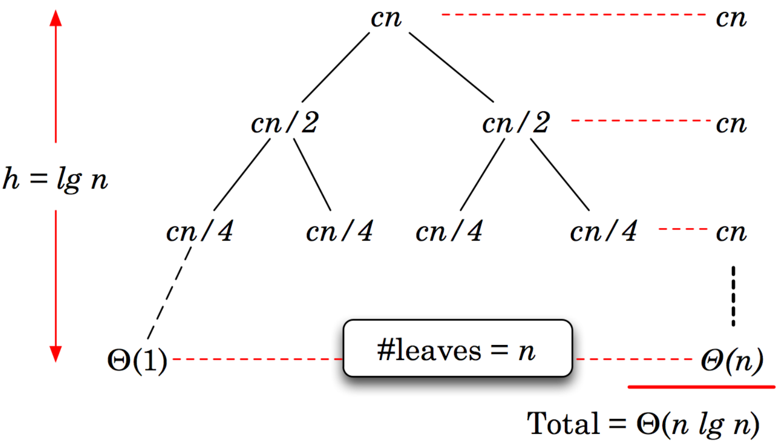
Recursion tree example: Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree example: Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Master method: solving "Divide and Conquer" recurrences

Theorem (Master Theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $\boxed{T(n) = aT(n/b) + f(n)}$. *Then $T(n)$ can be bounded asymptotically as follows.*

- Case 1 *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then* $\boxed{T(n) = \Theta(n^{\log_b a})}$.
- Case 2 *If $f(n) = \Theta(n^{\log_b a})$, then* $\boxed{T(n) = \Theta(n^{\log_b a} \lg n).}$
- Case 3 *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n (this is the regularity condition), then* $\boxed{T(n) = \Theta(f(n)).}$

Intuition behind the master method

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) = O(n^{log_b a - \epsilon}), \epsilon > 0 \\ \Theta(n^{\log_b a} \log n) & \text{if } f(n) = \Theta(n^{log_b a}) \\ \Theta(f(n)) & \text{if } f(n) = \Omega(n^{log_b a + \epsilon}), \epsilon > 0 \\ & \text{and if } af(n/b) \le cf(n),\ c < 1. \end{cases}$$

Comparing $f(n)$ with the special function $n^{log_b a}$.

Case 1 If $f(n)$ is *polynomially* smaller than $n^{log_b a}$, then $T(n) = \Theta(n^{log_b a})$.

Case 2 If $f(n)$ and $n^{log_b a}$ are of the "same size", then we multiply by a logarithmic factor, and $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \lg n)$.

Case 3 If $f(n)$ is *polynomially* larger than $n^{log_b a}$, and $af(n/b)$ is a decreasing function, then $T(n) = \Theta(f(n))$. The *regularity condition* – that $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n – must hold for case 3.

Using the master method

1. $T(n) = 9T(n/3) + n$. $a = 9, b = 3, f(n) = n$. $n^{\log_b a} = n^{\log_3 9} = n^2 = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, falls under Case 1. Solution is $T(n) = \Theta(n^2)$.
2. $T(n) = T(2n/3) + 1$. $a = 1, b = 3/2$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and solution is $T(n) = \Theta(\lg n)$.
3. $T(n) = 3T(n/4) + n \lg n$. $a = 3, b = 4, f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if the *regularity condition* holds. For sufficiently large n,
   $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ for $c = 3/4$.
   So, under case 3, $T(n) = \Theta(n \lg n)$.

Pitfalls in using the master method

Consider $T(n) = 2T(n/2) + n \lg n$. $a = 2, b = 2, f(n) = n \lg n$, and $n^{\log_b a} = n^{log_2 2} = n$. Case 3 *should apply* since $f(n) = n \lg n$ is asymptotically larger than $n^{log_b a} = n$; however, it is not *polynomially* larger! The ratio $f(n)/n^{log_b a} = (n \lg n)/n$ is asymptotically less than n^{ϵ} for any positive constant ϵ.
Falls in the gap between case 2 and 3.

Where does this "special function" $n^{\log_b a}$ come from?

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2) + f(n/b)) + f(n) = a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^2(aT(n/b^3) + f(n/b^2)) + af(n/b) + f(n) = a^3T(n/b^3) + a^2f(n \\
&= a^4T(n/b^4) + a^3f(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\
&\vdots \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) \quad \triangleright\ b^k = n\ (k = \log_b n) \\
&= \boxed{n^{\log_b a}T(1)} + \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) \quad \triangleright\ a^{\log_b n} = n^{\log_b a}
\end{aligned}
$$