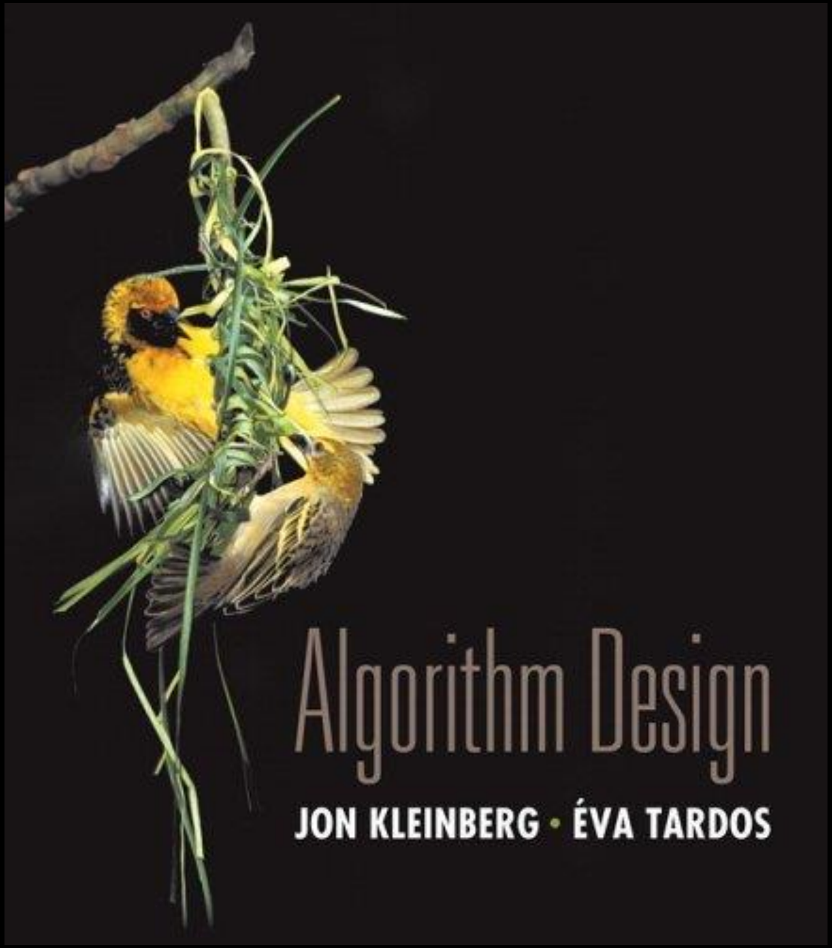


Chapter 2

Basics of
Algorithm Analysis

CS 350 Winter 2018

2.0 Some Preliminary Concepts

Log Function

(*) Log is a common “sub-linear” algorithm run-time (something of a best case scenario). ; the log function is 1-1.

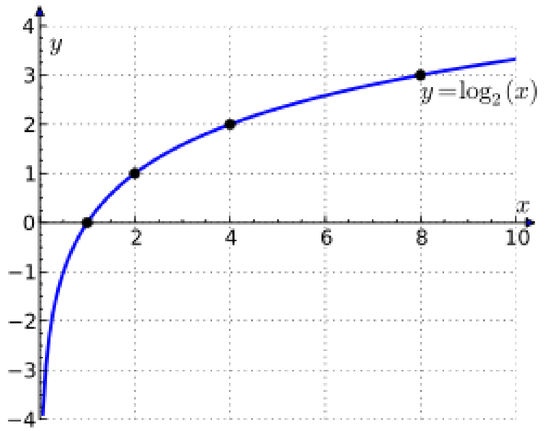
Some Useful Log Properties:

(1) $\log_b(xy)=\log_b(x)+\log_b(y)$

(2) $\log_b(x/y)=\log_b(x)-\log_b(y)$

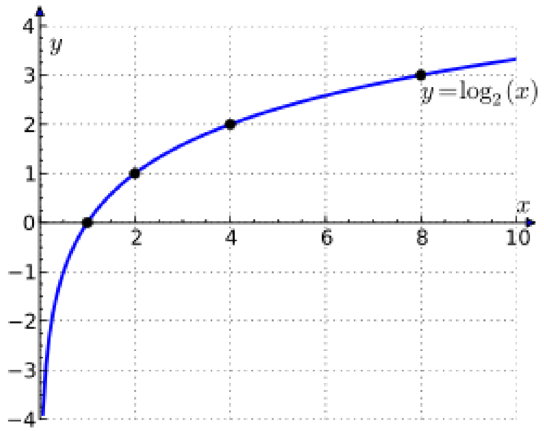
(3) $\log_b(x^y)=y\log_b(x)$

(4) $\log_b(x)=\ln(x)/\ln(b)$ (base conversion)



Log Function

(*) Log is a common "sub-linear" algorithm run-time (something of a best case scenario). ; the log function is 1-1.

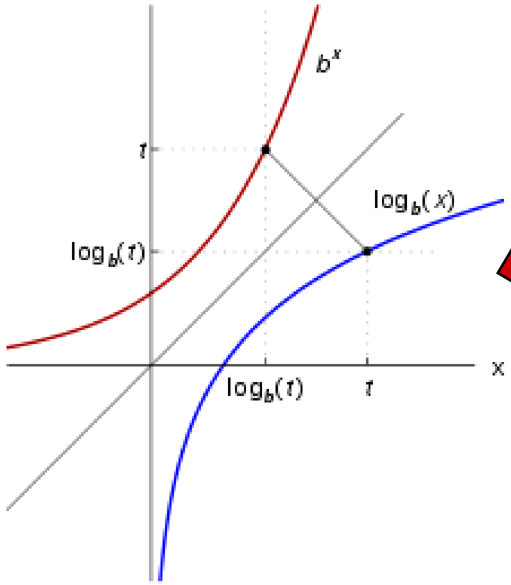


Some Useful Log Properties:

(1) $\log_b(xy)=\log_b(x)+\log_b(y)$

(2) $\log_b(x/y)=\log_b(x)-\log_b(y)$

(3) $\log_b(x^y)=y\log_b(x)$



log is the natural "inverse" of the exponential function

Counting Functions

(*) Counting functions serve as useful tools for algorithm analysis (e.g. analysis of run-time and space complexity).

Factorial Function: $n!=n(n-1)(n-2)\ldots(1)$. (NB: $0!=1$)

(*) Factorial counts there number of ways to order n distinct items.
(*) Factorial is "super exponential" (more on this later)

Choose: ${}_nC_k$ "n choose k" = $n!/(k!(n-k)!)$

(*) Choose counts the number of ways to <u>group</u> k elements from a set of n distinct elements ($n>=k$). (NB: <u>order is irrelevant here</u>)

Permutation: ${}_nP_k$ "n permute k" = $n!/(n-k)!$

(*) Permutations count the number of ways to <u>order</u> k elements from a set of n distinct elements ($n>=k$). (NB: <u>order matters</u>)

Comments on Series

(*) An arithmetic series is a sum of an arithmetic sequence – a progression of numbers such that the difference between consecutive terms is constant (d).

For arithmetic sequences/series:

$$a_n = a_1 + (n-1)d \qquad \sum_{i=1}^{n} a_i = \frac{n(a_1 + a_n)}{2}$$

Comments on Series

(*) An arithmetic series is a sum of an arithmetic sequence – a progression of numbers such that the difference between consecutive terms is constant (d).

For arithmetic sequences/series:

$$a_n = a_1 + (n-1)d \qquad \sum_{i=1}^{n} a_i = \frac{n(a_1 + a_n)}{2}$$

(*) A geometric series is a sum of a geometric sequence – a progression of numbers such that the quotient of consecutive terms is constant (r).

What is: 1/2+1/4+1/8+…? (Zeno)

Comments on Series

(*) An arithmetic series is a sum of an arithmetic sequence – a progression of numbers such that the difference between consecutive terms is constant (d).

For arithmetic sequences/series:

$$a_n = a_1 + (n-1)d \qquad \sum_{i=1}^{n} a_i = \frac{n(a_1 + a_n)}{2}$$

(*) A geometric series is a sum of a geometric sequence – a progression of numbers such that the quotient of consecutive terms is constant (r).

For geometric sequences/series in general:

$$a_n = ar^{n-1} \qquad a + ar + ar^2 + ar^3 + \ldots + ar^{n-1} = \sum_{i=1}^{n-1} ar^i = a\left(\frac{a - r^n}{1-r}\right)$$

$$a + ar + ar^2 + ar^3 + \ldots = \sum_{i=1}^{\infty} ar^i = \frac{a}{1-r}, \quad |r| < 1$$

Why?

Series

(*) Here are some more series from Calculus to keep in your "zoo".

$$(1-x)^{-1} = 1 + x + x^2 + x^3 + \cdots = \sum_{r=0}^{\infty} x^r \qquad (-1 < x < 1),$$

$$(1-x)^{-2} = 1 + 2x + 3x^2 + 4x^3 + \cdots = \sum_{r=0}^{\infty} (r+1)x^r \qquad (-1 < x < 1),$$

$$\log(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \cdots = \sum_{r=1}^{\infty}\left(-\frac{x^r}{r}\right) \qquad (-1 \le x < 1),$$

$$\tan^{-1} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots = \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m+1}}{2m+1} \qquad (-1 \le x \le 1),$$

$$e^x = 1 + x + \frac{x2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{r=0}^{\infty} \frac{x^r}{r!} \qquad (x \in \mathbb{R}),$$

$$\cosh x = 1 + x + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \cdots = \sum_{m=0}^{\infty} \frac{x^{2m}}{(2m)!} \qquad (x \in \mathbb{R}),$$

$$\sinh x = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \cdots = \sum_{m=0}^{\infty} \frac{x^{2m+1}}{(2m+1)!} \qquad (x \in \mathbb{R}),$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \cdots = \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m}}{(2m)!} \qquad (x \in \mathbb{R}),$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m+1}}{(2m+1)!} \qquad (x \in \mathbb{R}),$$

$$(1+x)^a = 1 + ax + \frac{a(a-1)}{2!}x^2 + \cdots = \sum_{r=0}^{\infty} \binom{a}{r} x^r \qquad (x \in V).$$

No need to memorize these – we can use them as needed. Q: Where do these formulae come from?

Basic Proof Techniques

Conditional Statement: P->Q

Converse: Q->P

Contrapositive: ~Q->~P

Which are "logically equivalent"? Why?

Basic Proof Techniques

Conditional Statement: P->Q

Converse: Q->P

Contrapositive: ~Q->~P ~Q->~P ⇔ P->Q (logically equivalent)

Direct Proof:

(1) Identify hypothesis, assumed hypothesis to be true.

(2) Using definitions, theorems, etc., make a series of deductions that eventually prove conclusion of conjecture.

Ex. The sum of two even integers is even.

Incidentally, what is the difference between deductive and inductive reasoning?

Basic Proof Techniques

Conditional Statement: P->Q

Converse: Q->P

Contrapositive: ~Q->~P   ~Q->~P ⇔ P->Q (logically equivalent)

Direct Proof:

(1) Identify hypothesis, assumed hypothesis to be true.

(2) Using definitions, theorems, etc., make a series of deductions that eventually prove conclusion of conjecture.

(Direct) Proof by exhaustion (i.e. proof by cases/brute force):

(1) Split statement to be proved into a finite number of cases or sets of equivalent cases.

(2) Check each type of case to see if the proposition in question holds.

Ex. Every integer that is a perfect cube is either a multiple of 9, or 1 more, or 1 less than a multiple of 9. (Hint: use mod 3)

Basic Proof Techniques

Direct Proof:

(1) Identify hypothesis, assumed hypothesis to be true.

(2) Using definitions, theorems, etc., make a series of deductions that eventually prove conclusion of conjecture.

Indirect Proof:

(1) Assume the opposite of the conjecture, or assume that the conjecture is false.

(2) Try to prove your assumption directly until you reach an irreconcilable contradiction.

(3) Since we get a contradiction, it must be the case that the assumption that the opposite of the hypothesis is true is false.

Ex. $\sqrt{2}$ is irrational.

Ex. There exists an infinitude of primes.

(Both original proofs are due to Euclid)

Basic Proof Techniques

Mathematical Induction:

(*) Powerful proof technique used frequently to prove properties over countable sets, e.g. the natural numbers (NB: we'll do the "weak" version here).

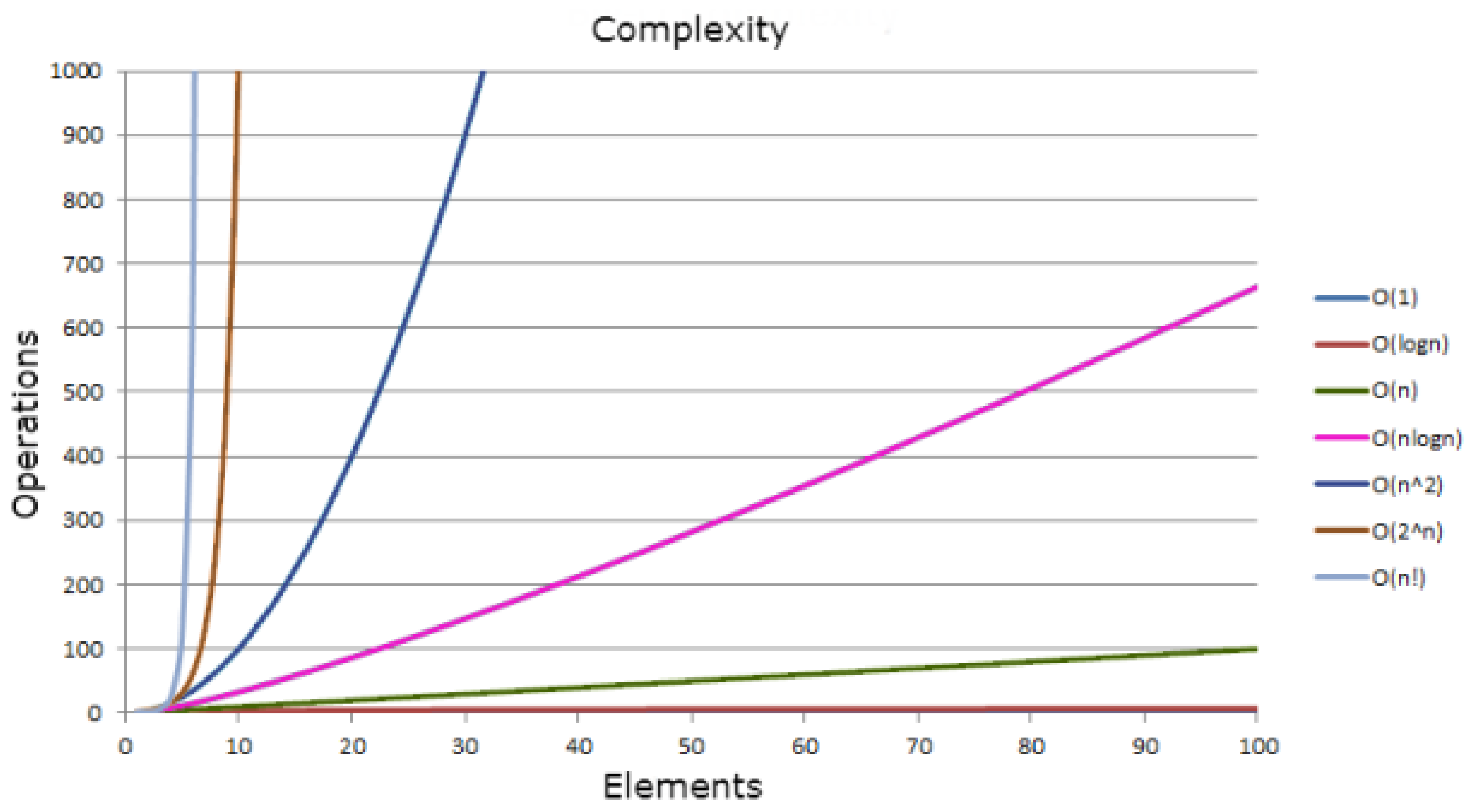
(1) Show the formula/procedure works for the basis case(s), e.g. n=1.
(2) Inductive Hypothesis: Assume the procedure holds in the arbitrary case: n=k.
(3) Using the inductive hypothesis, show that the procedure works for n=k+1.

Ex. $1+2+...+n=\frac{n(n+1)}{2} \quad n\geq 1$

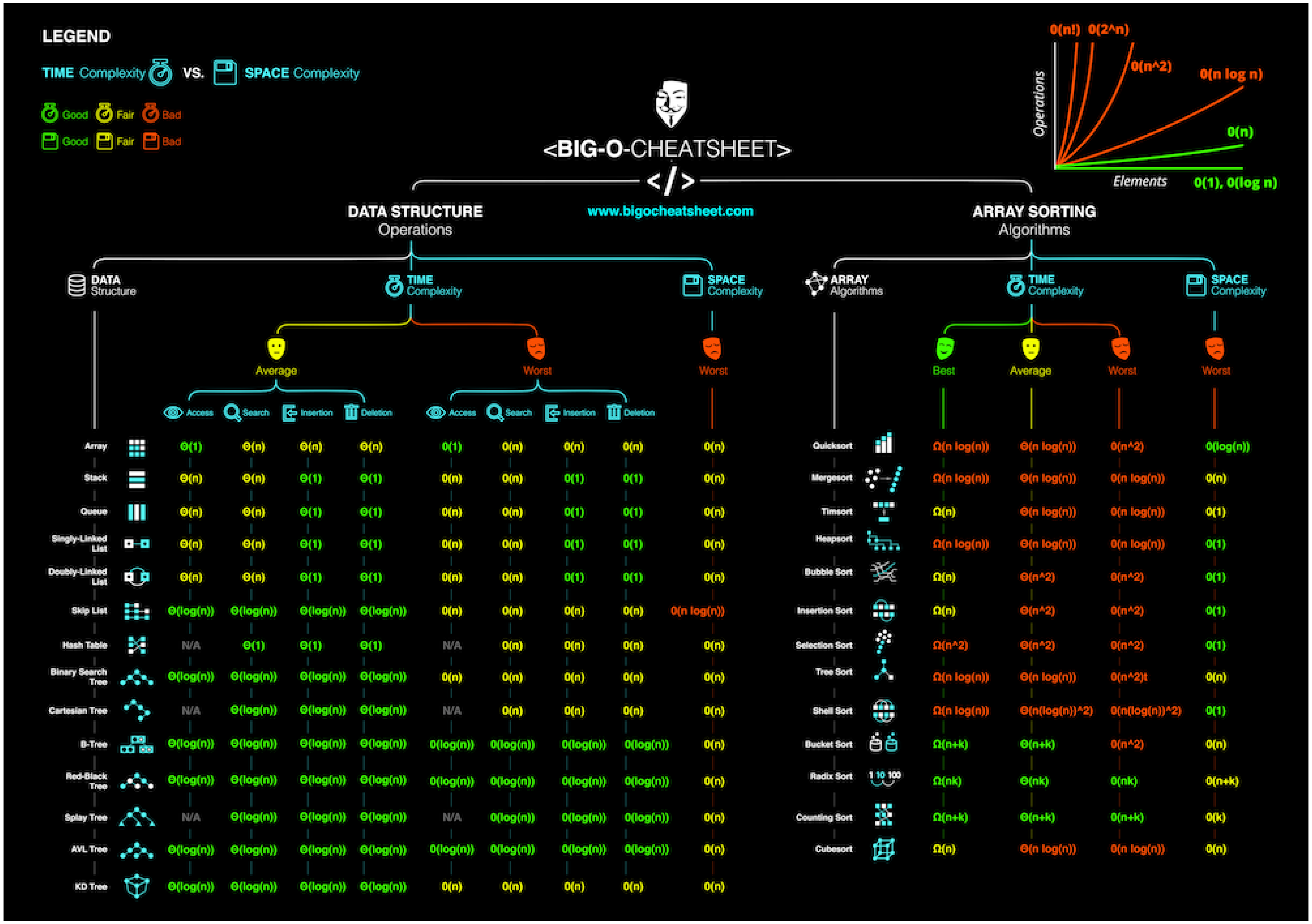
Ex. $5|\left(6^n+4\right) \quad n\geq 0$

Ex. Formula for the sum of the interior angles of an n-sided polygon, n≥3.

Complexity of Common Functions



Algorithm Complexity



LEGEND

TIME Complexity VS. SPACE Complexity

Good Fair Bad

Good Fair Bad

<BIG-O-CHEATSHEET>

</>

www.bigocheatsheet.com

DATA STRUCTURE Operations

ARRAY SORTING Algorithms

O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(1), O(log n)
Operations
Elements

DATA Structure	TIME Complexity Average				TIME Complexity Worst				SPACE Complexity Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

ARRAY Algorithms	TIME Complexity Best	TIME Complexity Average	TIME Complexity Worst	SPACE Complexity Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(1)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)t	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Here is a complexity "cheat sheet" for many common algorithms…stay tuned for details!

L'Hôpital's Rule

(*) Let's develop a deeper framework for analyzing and comparing function asymptotics.

(*) We say a function g "dominates" function f (g >> f) as x tends to infinity if:

$$\lim_{x\to\infty}\frac{f(x)}{g(x)}=0$$

(*) We can use elementary calculus (L'Hôpital's Rule) to determine when one function dominates another. Recall that L'H rule says that if a limit is in "indeterminate form" (as a quotient), then the limit is preserved under the derivative (in the numerator and denominator respectively).

Ex. $x \gg \log_b(x)$ Ex. $x^2 \gg x$

Ex. $b^x \gg x^n$ (any exponential function dominates a polynomial/power function)

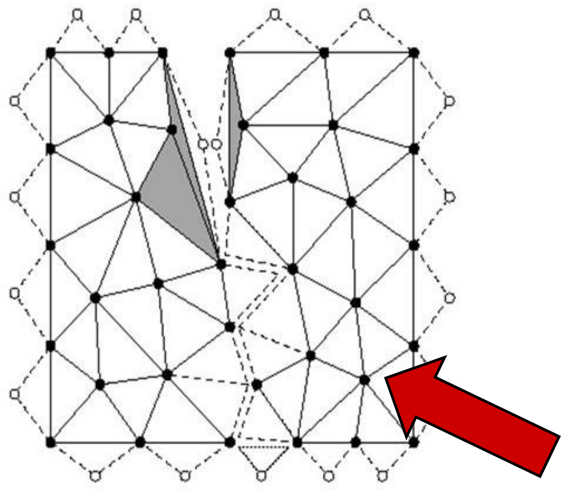
Ex. $x! \gg b^x$ (factorial functions dominate exponential functions)

Q: Does "anyone" beat x! ?

Divide-and-Conquer Example with Recurrences

(*) An archetypal and immensely powerful strategy in algorithm design called “divide and conquer” (e.g. “Binary Search” – more to come later) solves a problem by:

(1) Breaking it into subproblems that are themselves smaller instances of the same type of problem.
(2) Recursively solving these subproblems
(3) Appropriately combining their answers



Graph Triangulation Problem

Divide-and-Conquer Example with Recurrences

(*) Gauss once noticed that although the product of two complex numbers:

$$(a+bi)(c+di)=ac-bd+(bc+ad)i$$

Seems to involved four real-number multiplications, it can in fact be done with just three: ac, bd, and (a+b)(c+d), since:

$$bc+ad=(a+b)(c+d)-ac-bd$$

Surprisingly, perhaps, this modest improvement becomes very significant when applied recursively.

Let's move away from complex numbers and see how this helps with real-valued multiplication.

Divide-and-Conquer Example with Recurrences

(*)Suppose x and y are two n-bit integers, and assume, for convenience, that n is a power of 2 (the more general case is not too different).

As a first step toward multiplying x and y, split each of them into their left and right halves, which are n/2 bits long:

$$x = [x_L][x_R] = 2^{n/2}x_L + x_R$$
$$y = [y_L][y_R] = 2^{n/2}y_L + y_R$$

For instance, if x=101101102, then $x_L = 1011_2$ and $x_R = 0110_2$, and $x = 1011_2 * 2^4 + 0110_2$.

The product of x and y can thus be written:

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Divide-and-Conquer Example with Recurrences

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Consider the computation requirements of the RHS:

(*) The additions take linear time, as do the multiplications by powers of 2 (which are merely left-shifts).

(*) The significant operations are the four n/2-bit multiplications: $x_L y_L$, $x_L y_R$, $x_R y_L$, $x_R y_R$; these can be handled with four recursive calls.

(*) Thus our method for multipliying n-bit numbers starts by making recursive calls to multiply these four pairs of n/2-bit numbers (four subproblems of half the size), and then evaluates the preceding expression in O(n) time.

Writing T(n) for the overall running time on n-bit inputs, we get the recurrence relation:

$$T(n)=4T(n/2)+O(n)$$

Divide-and-Conquer Example with Recurrences

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Writing T(n) for the overall running time on n-bit inputs, we get the recurrence relation:

$$T(n)=4T(n/2)+O(n)$$

(*) In this course we will develop general strategies for solving such equations.

(*) In the meantime, this particular equation works out to $O(n^2)$, the same running-time as the traditional grade school multiplication technique.

Q: How can we speed up this method? A: Apply Gauss' trick.

Divide-and-Conquer Example with Recurrences

$$xy = (2^{n/2} x_L + x_R)(2^{n/2} y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

$$T(n)=4T(n/2)+O(n)$$

A: Apply Gauss' trick.

(*) Although the expression xy seems to demand four n/2-bit multiplications, as before just three will do: $x_L y_L$, $x_L y_R$, and $(x_L+x_R)(y_L+y_R)$, since $x_L y_R + x_L y_R = (x_L+x_R)(y_L+y_R) - x_L y_L - x_R y_R$.

The resulting algorithm has an improved running time of:

$$T(n)=3T(n/2)+O(n)$$

(*) The point is that now the constant factor improvement, from 4 to 3, occurs at every level of the recursion, and this compounding effect leads to a dramatically lower time bound of $O(n^{1.59})$.

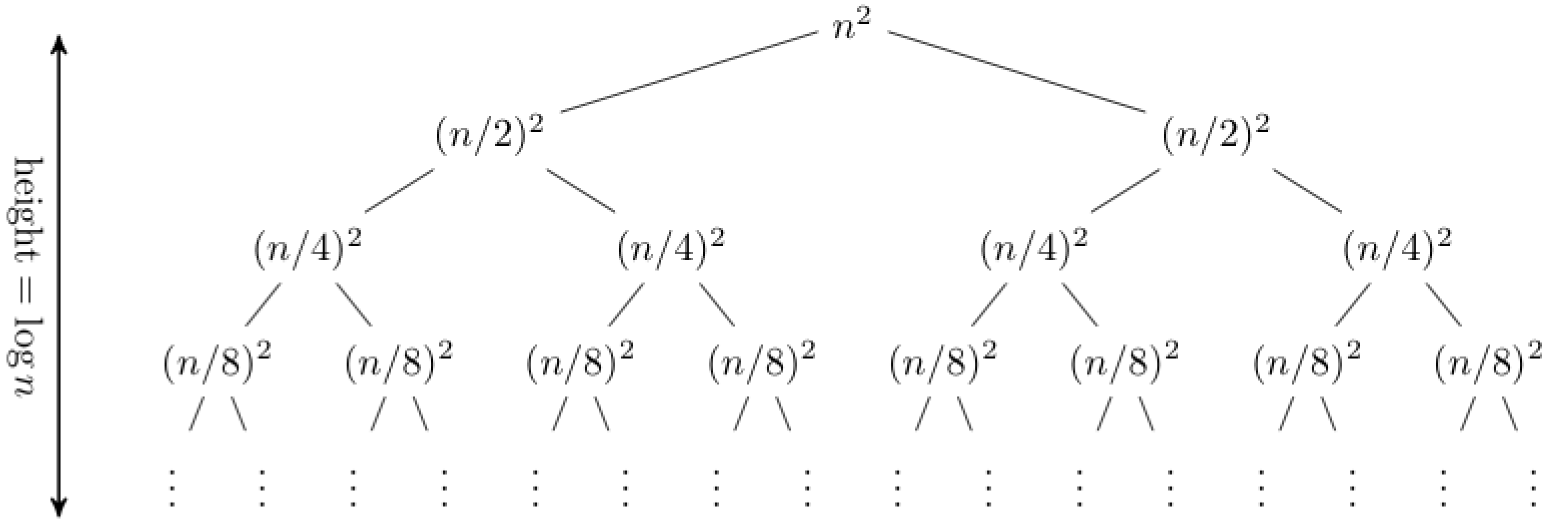
Q: How do we determine this bound (more later) – but for now, it is helpful to consider the recursive calls with respect to a tree structure (also: the "Master Theorem" can be used).

“Master Method” Example

(*) A *recursion tree* is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

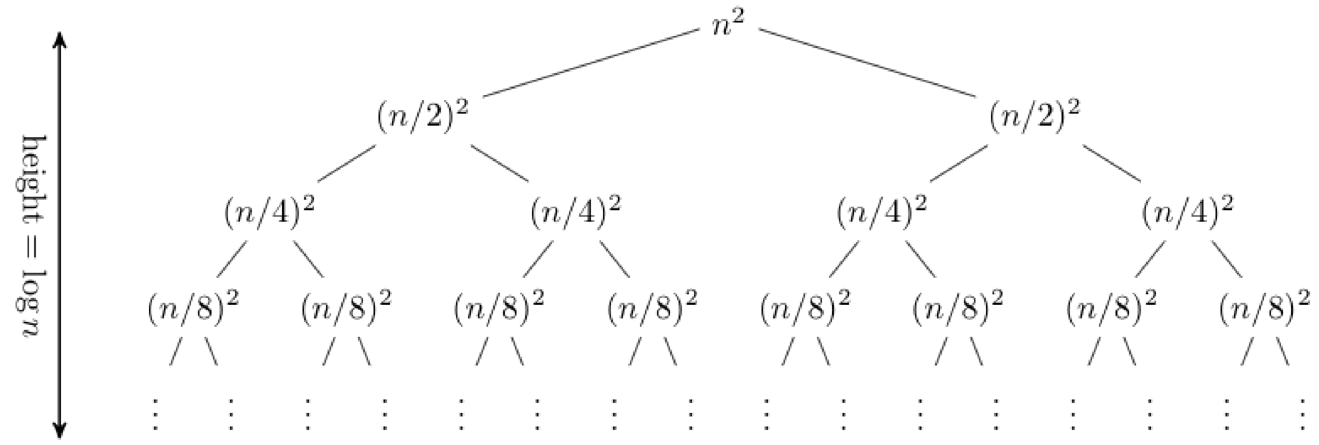
Consider the recurrence: $T(n)=2T(n/2)+n^2$

The corresponding recursion tree has the following form:

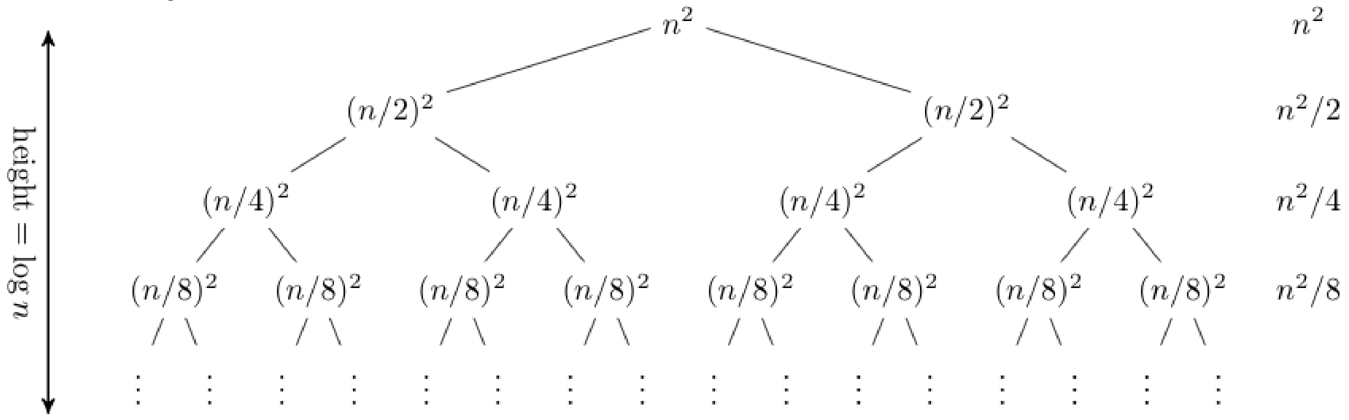


"Master Method" Example

Consider the recurrence: $T(n)=2T(n/2)+n^2$



Consider summing across each row:

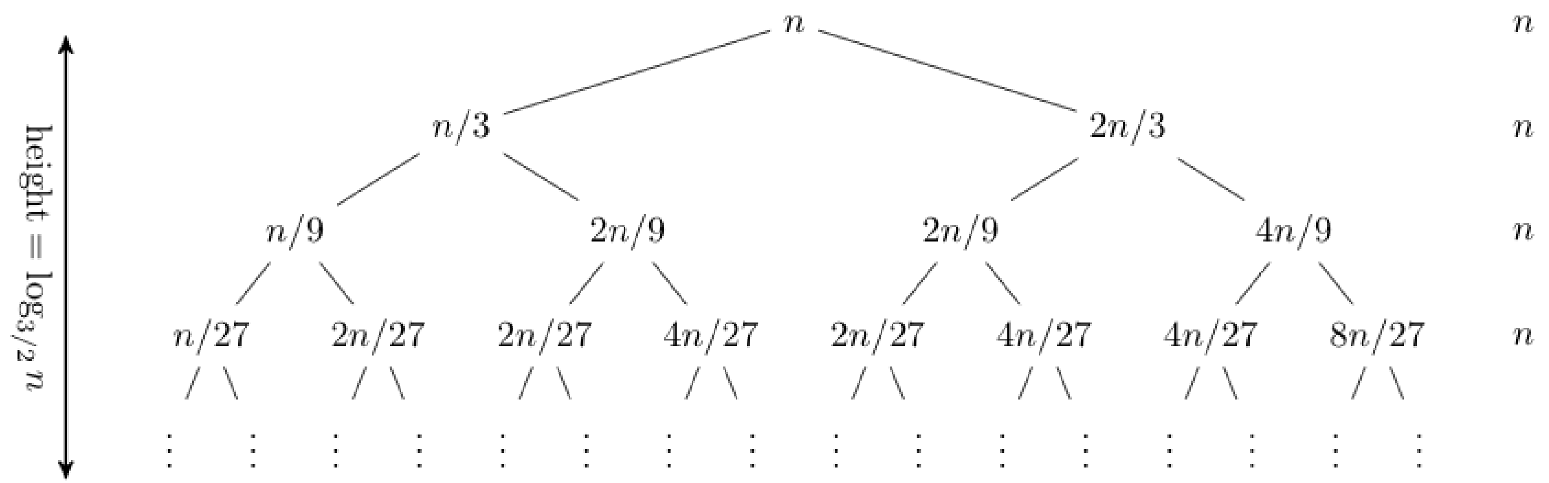


This yields a geometric series:

$$\sum_{i=1}^{n} \frac{n^2}{2^i} = ?$$

“Master Method” Example

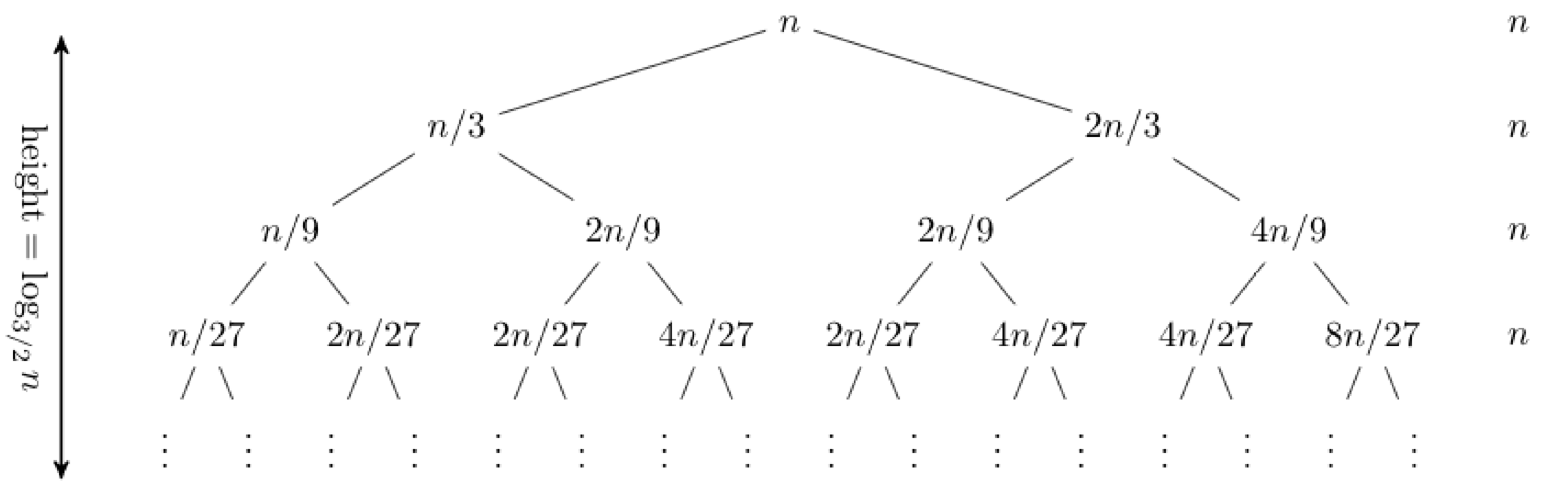
Consider the recurrence: T(n)=T(n/3)+T(2n/3)+n



Note that the recursion tree is not balance in this case, and that the longest path is the rightmost one.

"Master Method" Example

Consider the recurrence: $T(n)=T(n/3)+T(2n/3)+n$



Note that the recursion tree is not balance in this case, and that the longest path is the rightmost one.

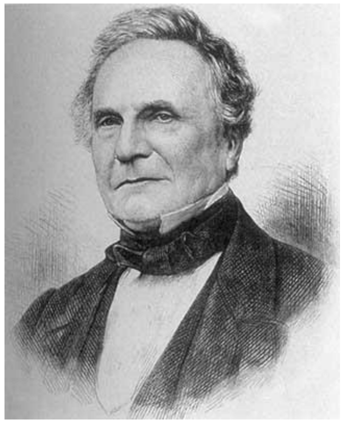
Since the longest path is $O(\log_{3/2}(n))$, our guess for the closed form solution to the recurrence is: $O(n \log n)$.

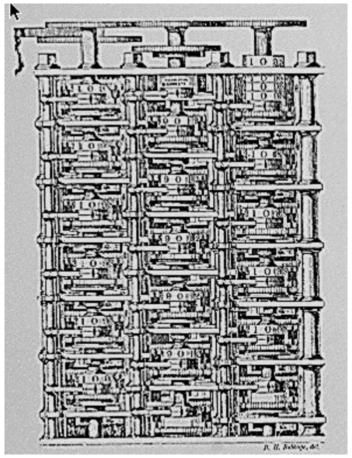
2.1 Computational Tractability

"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - *Francis Sullivan*

Computational Tractability

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - *Charles Babbage*


Charles Babbage (1864)


Analytic Engine (schematic)

Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

n! for stable matching with n men and n women

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C.

There exists constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by $c N^d$ steps.

Def. An algorithm is poly-time if the above scaling property holds.

choose $C = 2^d$

Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.

Worst-Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

simplex method
Unix grep

Why It Matters

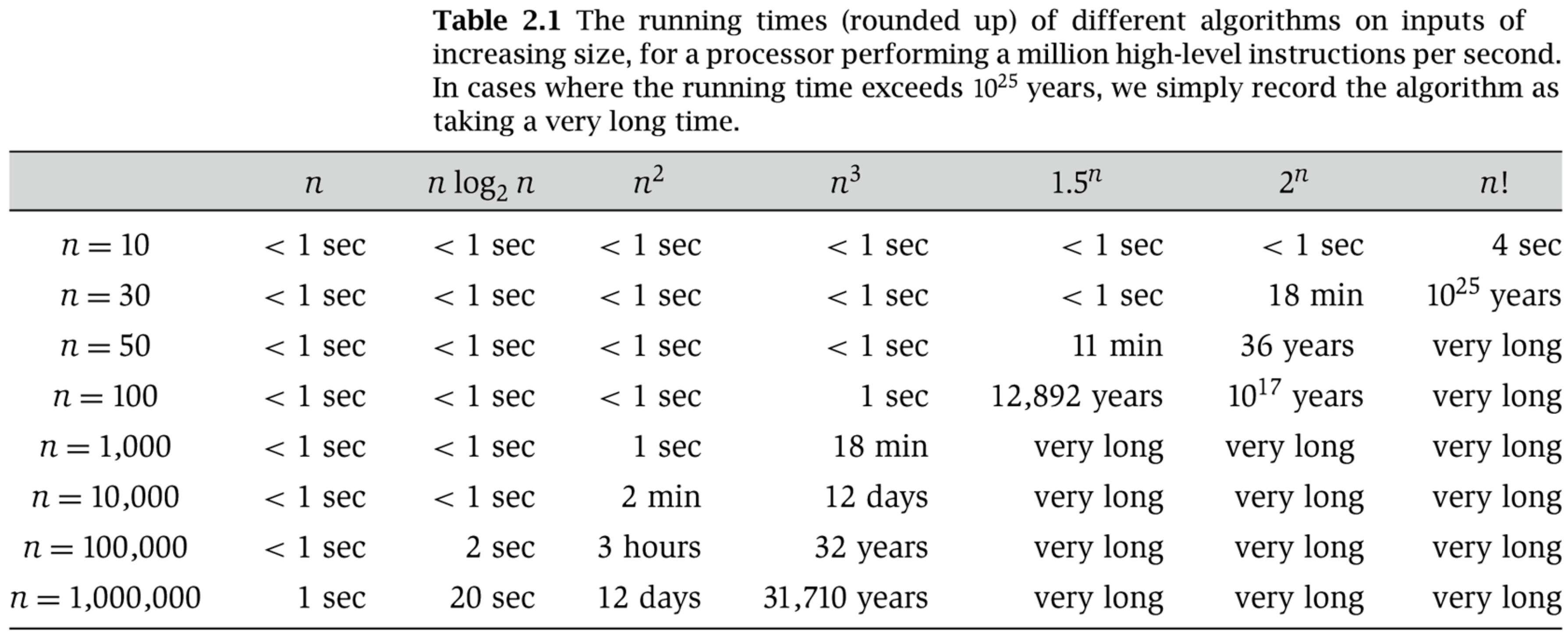


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

2.2 Asymptotic Order of Growth

Asymptotic Order of Growth

Upper bounds. T(n) is O(f(n)) if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. T(n) is Ω(f(n)) if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. T(n) is Θ(f(n)) if T(n) is both O(f(n)) and Ω(f(n)).

More formally:
Let f and g be two functions where:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Exists and is equal to some number $c > 0$. Then f(n)= Θ(g(n)).

Ex: $T(n) = 32n^2 + 17n + 32$.

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

Notation

Slight abuse of notation. $T(n) = O(f(n))$.

- Not transitive:
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - but $f(n) \neq g(n)$.
- Better notation: $T(n) \in O(f(n))$.

Meaningless statement. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.

- Statement doesn't "type-check."
- Use Ω for lower bounds.

Properties

Transitivity. If f = O(g) and g = O(h) then f = O(h).

Pf. f = O(g) implies there exist constants c and n_0, where $f(n) \leq cg(n)$ for all $n \geq n_0$. Also, for some (potentially different) constants c' and n_0' we have $g(n) \leq c'h(n)$ for all $n \geq n_0'$; why?

Q:What's the next step?

Properties

Transitivity. If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.

Pf. $f = O(g)$ implies there exist constants c and n_0, where $f(n) \leq cg(n)$ for all $n \geq n_0$. Also, for some (potentially different) constants c' and n_0' we have $g(n) \leq c'h(n)$ for all $n \geq n_0'$.

Next: it follows that $f(n) \leq cg(n) \leq cc'h(n)$, and so $f(n) \leq cc'h(n)$ for all $n \geq \max(n_0, n_0')$.

Properties

Additivity. If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.

Pf. We're given that for some constants c and n_0, we have $f(n) \leq ch(n)$ for all $n \geq n_0$. Also, for some (potentially different) constants c' and n_0', we have $g(n) \leq c'h(n)$ for all $n \geq n_0'$. So consider any number n that is at least as large as both n_0 and n_0'.

Properties

Additivity. If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.

Pf. We're given that for some constants c and n_0, we have $f(n) \le ch(n)$ for all $n \ge n_0$. Also, for some (potentially different) constants c' and n_0', we have $g(n) \le c'h(n)$ for all $n \ge n_0'$. So consider any number n that is at least as large as both n_0 and n_0'.

We have: $f(n)+g(n) \le ch(n)+c'h(n)$.

Thus, $f(n)+g(n) \le (c+c')h(n)$ for all $n \ge \max(n_0,n_0')$.

More Properties

Transitivity.

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$. (Proven)
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$. (Why?)
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$. (Why?)

Additivity.

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$. (Proven)
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$. (Why?)
- If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$. (Why?)

- In general: let $f_1,\ldots,f_k$ and h be functions such that $f_i=O(h)$. Then what property naturally follows?

More Properties

Transitivity.

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$. (Proven)
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$. (Why?)
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$. (Why?)

Additivity.

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$. (Proven)
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$. (Why?)
- If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$. (Why?)

- In general: let $f_1,\ldots,f_k$ and h be functions such that $f_i=O(h)$. Then what property naturally follows?

$$f_1+\ldots+f_k=O(h)$$

Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1 n + \dots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants a, b > 0. Why?

$\uparrow$ can avoid specifying the base

Logarithms. For every x > 0, $\log n = O(n^x)$.

$\uparrow$ log grows slower than every polynomial

Exponentials. For every r > 1 and every d > 0, $n^d = O(r^n)$.

$\uparrow$ every exponential grows faster than every polynomial

Examples

Ex1. Arrange the following functions in ascending order.

$F_1(n)=10^n$

$F_2(n)=n^{1/3}$

$F_3(n)=n^n$

$F_4(n)=\log_2 n$

$F_5(n)=2^{\text{root}(\log_2(n))}$

Examples

Ex1. Arrange the following functions in ascending order.

$F_1(n)=10^n$
$F_2(n)=n^{1/3}$
$F_3(n)=n^n$
$F_4(n)=\log_2 n$
$F_5(n)=2^{\text{root}(\log_2(n))}$

Most obviously, $F_2 \le F_1 \le F_3$; why?

Examples

Ex1. Arrange the following functions in ascending order.

$F_1(n)=10^n$
$F_2(n)=n^{1/3}$
$F_3(n)=n^n$
$F_4(n)=\log_2 n$
$F_5(n)=2^{\text{root}(\log_2(n))}$

$F_2 \leq F_1 \leq F_3$

Also, $F_4=O(F_2)$ and $F_2=O(F_1)$; why?

Examples

Ex1. Arrange the following functions in ascending order.

$F_1(n)=10^n$

$F_2(n)=n^{1/3}$

$F_3(n)=n^n$

$F_4(n)=\log_2 n$

$F_5(n)=2^{\mathrm{root}(\log_2(n))}$

$F_2 \leq F_1 \leq F_3$; $F_4 \leq F_2$; $F_2 \leq F_1$

What about F_5?

Take logs of F_2, F_4 and F_5 (why can we do this with impunity?)

$\log_2(F_5)=\mathrm{root}(\log_2(n))$; $\log_2(F_4)=\log_2(\log_2(n)))$; $\log(F_2)=1/3\log_2(n)$.

Examples

Ex1. Arrange the following functions in ascending order.

$F_1(n)=10^n$
$F_2(n)=n^{1/3}$
$F_3(n)=n^n$
$F_4(n)=\log_2 n$
$F_5(n)=2^{\mathrm{root}(\log_2(n))}$

$F_2 \leq F_1 \leq F_3$; $F_4 \leq F_2$; $F_2 \leq F_1$

$\log_2(F_5)=\mathrm{root}(\log_2(n))$; $\log_2(F_4)=\log_2(\log_2(n))$; $\log(F_2)=1/3\log_2(n)$

Let $z = \log_2(n)$.

This yields: $\log_2(F_5)=z^{1/2}$; $\log_2(F_4)=\log_2(z)$; $\log(F_2)=1/3z$.

Examples

Ex1. Arrange the following functions in ascending order.

$F_1(n)=10^n$
$F_2(n)=n^{1/3}$
$F_3(n)=n^n$
$F_4(n)=\log_2 n$
$F_5(n)=2^{root(\log_2(n))}$

$F_2 \leq F_1 \leq F_3$; $F_4 \leq F_2$; $F_2 \leq F_1$

$\log_2(F_5)=z^{1/2}$; $\log_2(F_4)=\log_2(z)$; $\log(F_2)=1/3z$

This implies: $F_4 \leq F_5 \leq F_2$ (why?)

Examples

Ex1. Arrange the following functions in ascending order.

$F_1(n)=10^n$

$F_2(n)=n^{1/3}$

$F_3(n)=n^n$

$F_4(n)=\log_2 n$

$F_5(n)=2^{\text{root}(\log_2(n))}$

Final Answer: $F_4 \leq F_5 \leq F_2 \leq F_1 \leq F_3$

Examples

Ex2. Prove: If f and g are two functions that take non-negative values, with $f=O(g)$, then $g=\Omega(f)$.

What does this show intuitively?

Examples

Ex2. Prove: If f and g are two functions that take non-negative values, with $f=O(g)$, then $g=\Omega(f)$.

What does this show intuitively? That O() and Ω() are "opposites."

How do we prove this?

Examples

Ex2. Prove: If f and g are two functions that take non-negative values, with $f=O(g)$, then $g=\Omega(f)$.

What does this show intuitively? That O() and Ω() are "opposites."

How do we prove this?

Pf. There exist constants c and n_0 with $f(n) \leq cg(n)$ for all $n \geq n_0$.

This implies: $g(n) \geq 1/cf(n)$; thus $g= \Omega(f)$, as was to be shown.

2.4 A Survey of Common Running Times

Linear Time: O(n)

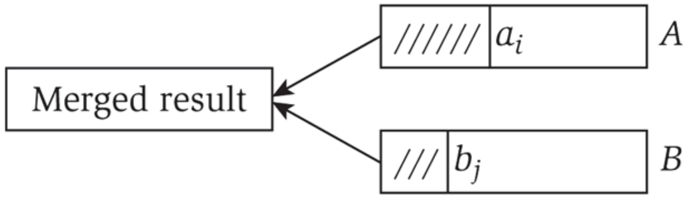
Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a₁
for i = 2 to n {
   if (aᵢ > max)
      max ← aᵢ
}
```

Linear Time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.



```
i = 1, j = 1
while (both lists are nonempty) {
   if ($a_i \le b_j$) append $a_i$ to output list and increment i
   else         append $b_j$ to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes O(n) time.
Pf. After each comparison, the length of output list increases by 1.

O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms.

↑ also referred to as linearithmic time

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← $(x_1 - x_2)^2 + (y_1 - y_2)^2$
for i = 1 to n {
   for j = i+1 to n {
      d ← $(x_i - x_j)^2 + (y_i - y_j)^2$
      if (d < min)
         min ← d
   }
}
```

← don't need to take square roots

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. ← see chapter 5

Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, …, S_n$ each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```
foreach set $S_i$ {
   foreach other set $S_j$ {
      foreach element p of $S_i$ {
         determine whether p also belongs to $S_j$
      }
      if (no element of $S_i$ belongs to $S_j$)
         report that $S_i$ and $S_j$ are disjoint
   }
}
```

Polynomial Time: O(n^k) Time

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

↑ k is a constant

O(n^k) solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set = O(k^2).
- Number of k element subsets = $\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \leq \frac{n^k}{k!}$
- O($k^2\ n^k\ /\ k!$) = O(n^k).

↑ poly-time for k=17, but not practical

Exponential Time

Independent set. Given a graph, what is maximum size of an independent set?

$O(n^2 \, 2^n)$ solution. Enumerate all subsets.

```
S* ← ϕ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

2.3 Data Structure: Priority Queues and Heaps

Priority Queues

A priority queue stores a collection of entries

Each **entry** is a pair (key, value)

Main methods of the Priority Queue ADT

- insert(k, x) inserts an entry with key k and value x
- removeMin() removes and returns the entry with smallest key

Additional methods

- min() returns, but does not remove, an entry with smallest key
- size(), isEmpty()

Applications:

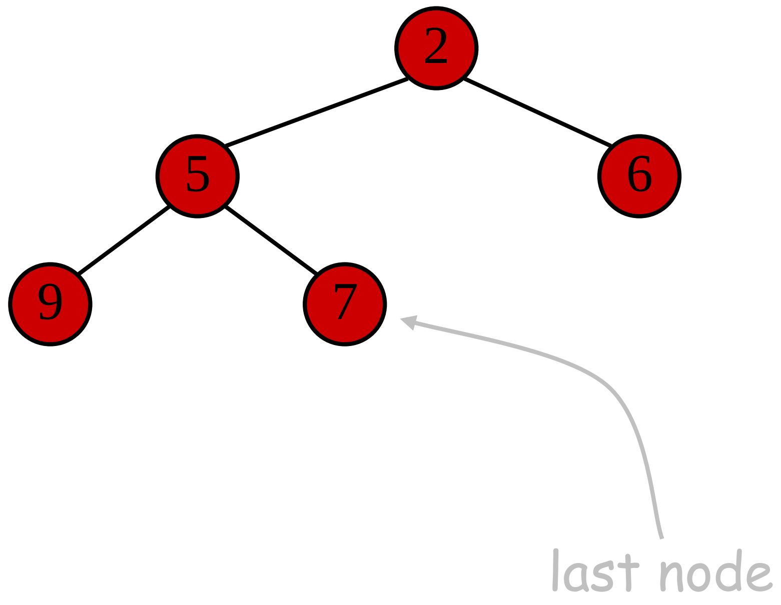
- Scheduling
- Auctions
- Stock market

Heaps

A heap is a binary tree storing keys at its nodes and satisfying the following properties:

- Heap-Order: for every internal node v other than the root,
  $key(v) \geq key(parent(v))$
- Complete Binary Tree: let h be the height of the heap
  - for $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
  - at depth $h - 1$, the internal nodes are to the left of the external nodes

The last node of a heap is the rightmost node of depth h

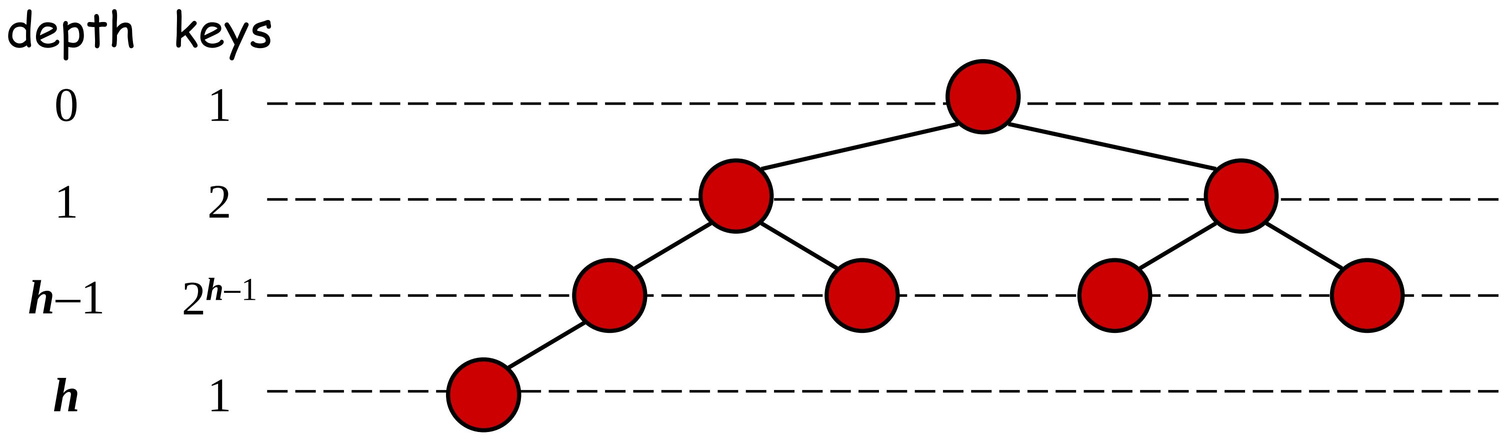


Heaps

Theorem: *A heap storing n keys has height $O(\log n)$*

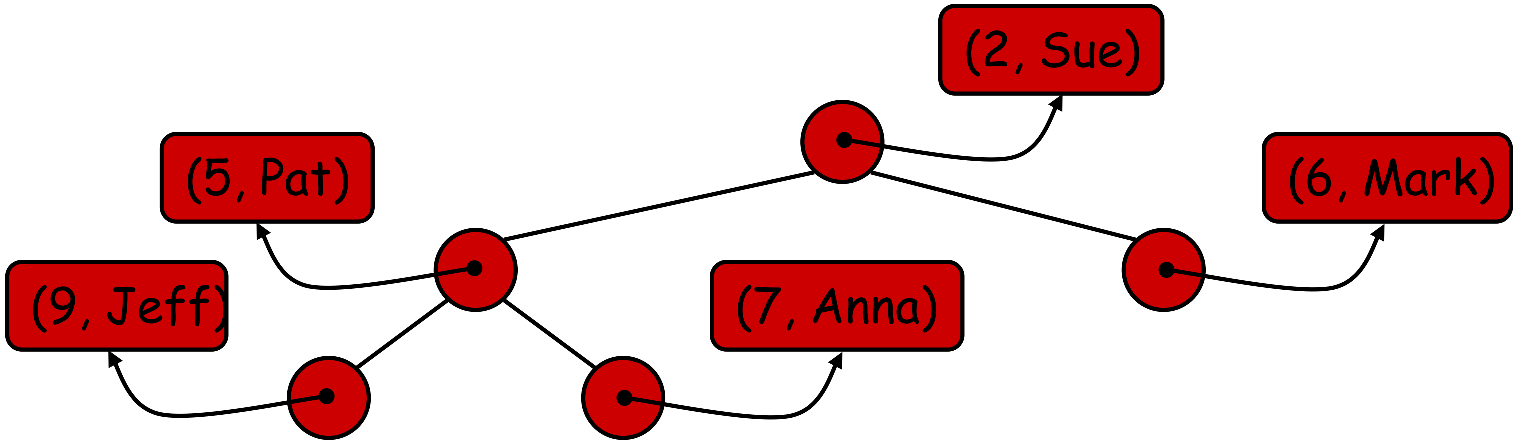
Pf. (we apply the complete binary tree property)

- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$, i.e., $h \leq \log n$



Heaps and Priority Queues

We can use a heap to implement a priority queue
We store a (key, element) item at each internal node
We keep track of the position of the last node
For simplicity, we show only the keys in the pictures

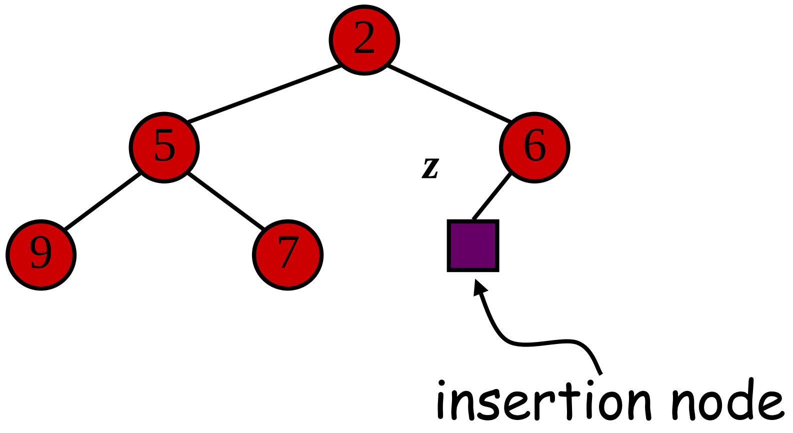


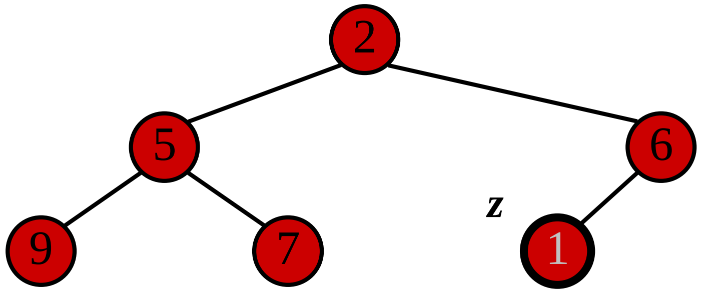
Heaps and Priority Queues

Method insertItem of the priority queue ADT corresponds to the insertion of a key k to the heap

The insertion algorithm consists of three steps

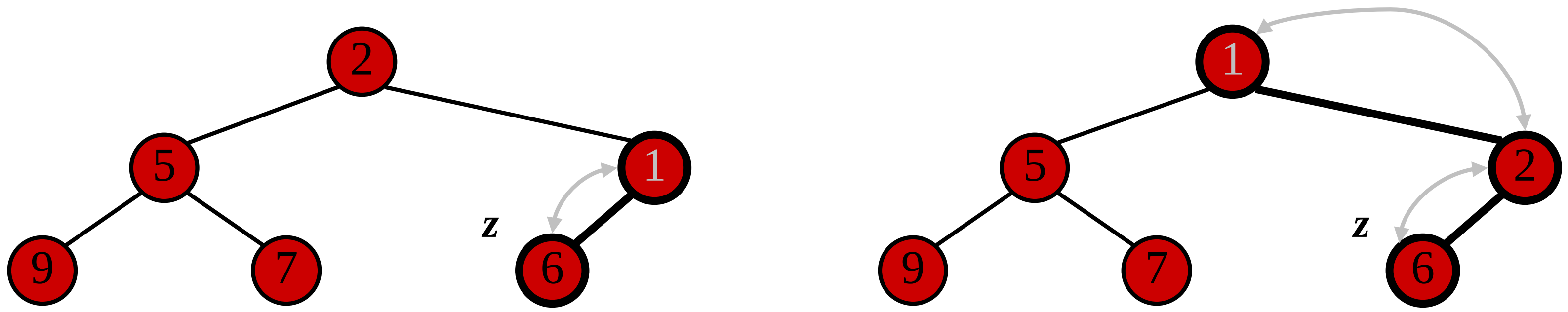
- Find the insertion node z (the new last node)
- Store k at z
- Restore the heap-order property (discussed next)





Upheap

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time

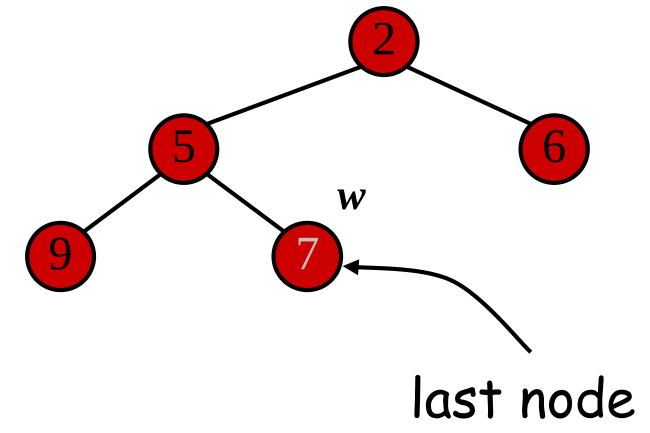


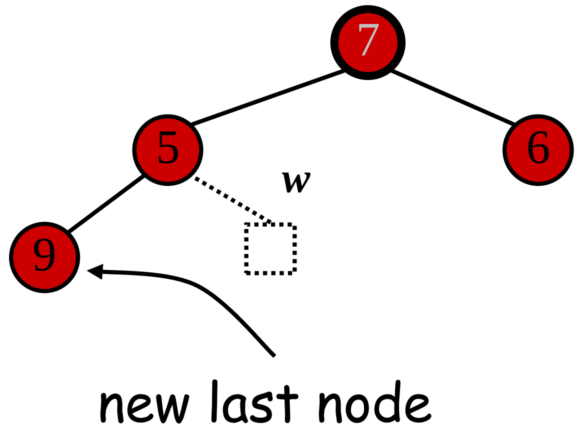
Remove Min (Heap)

Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap

The removal algorithm consists of three steps

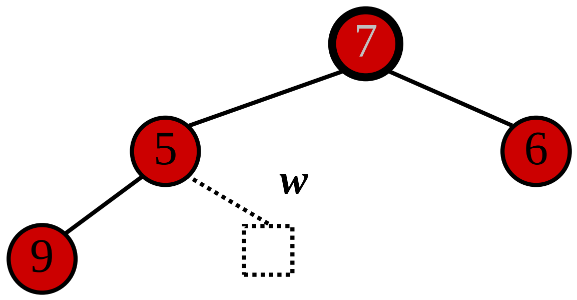
- Replace the root key with the key of the last node *w*
- Remove *w*
- Restore the heap-order property (discussed next)

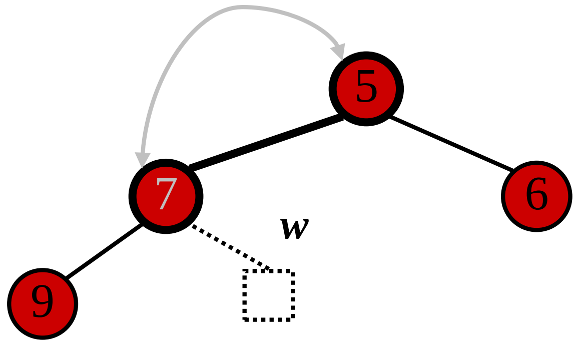




Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Upheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



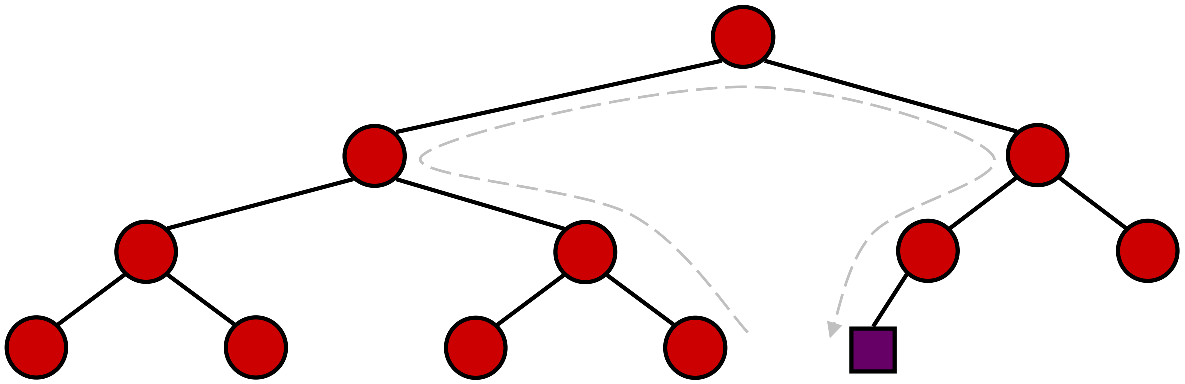


Updating the Last Node

The insertion node can be found by traversing a path of $O(\log n)$ nodes

- Go up until a left child or the root is reached
- If a left child is reached, go to the right child
- Go down left until a leaf is reached

Similar algorithm for updating the last node after a removal



)

Consider a priority queue with n items implemented by means of a heap

- the space used is $O(n)$
- methods insert and removeMin take $O(\log n)$ time
- methods size, isEmpty, and min take time $O(1)$ time

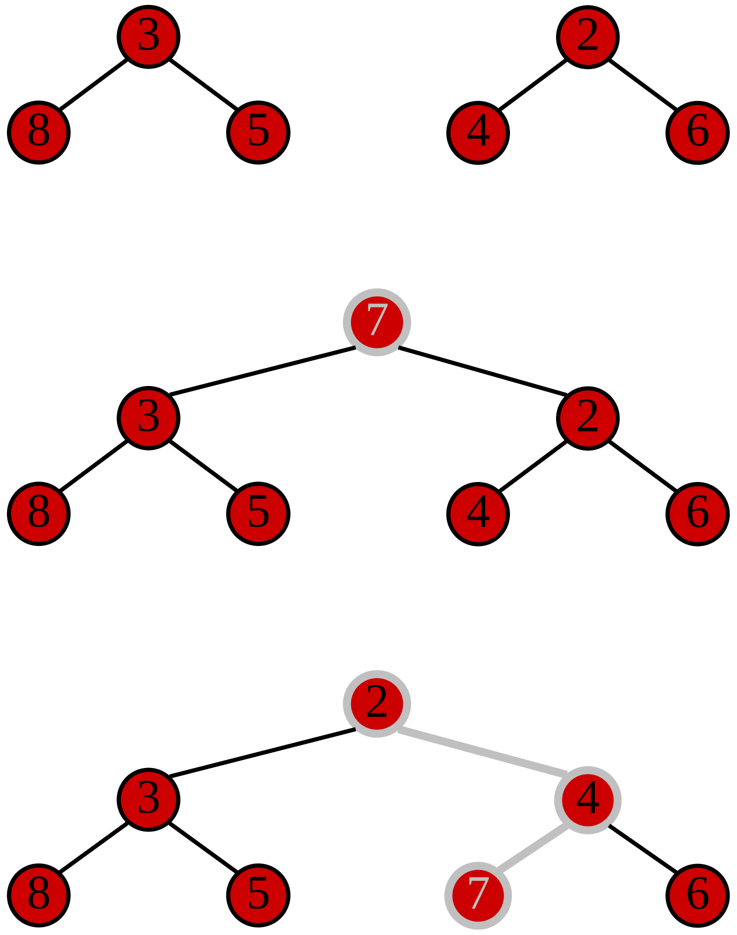
Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time

The resulting algorithm is called heap-sort

Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

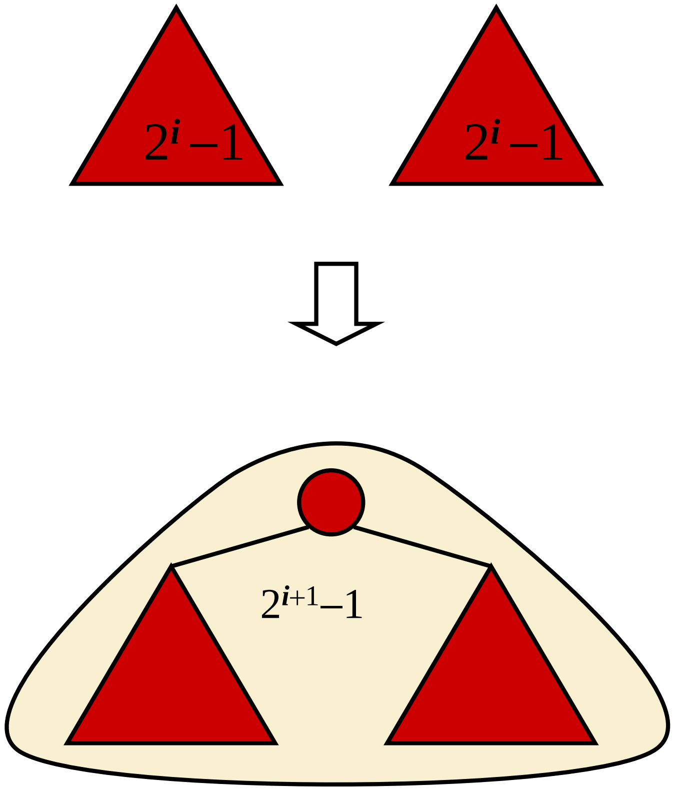
Merging Two Heaps

- We are given two two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform downheap to restore the heap-order property

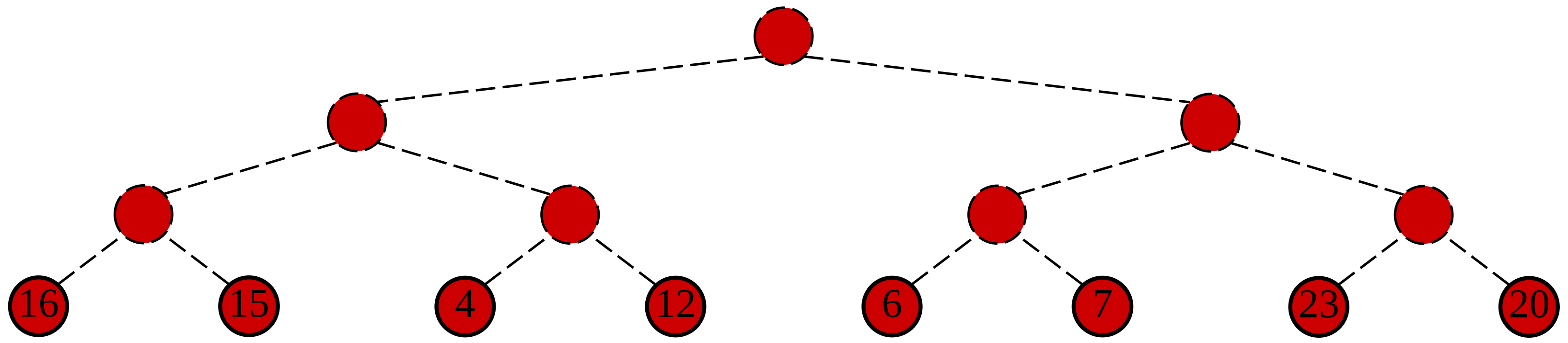


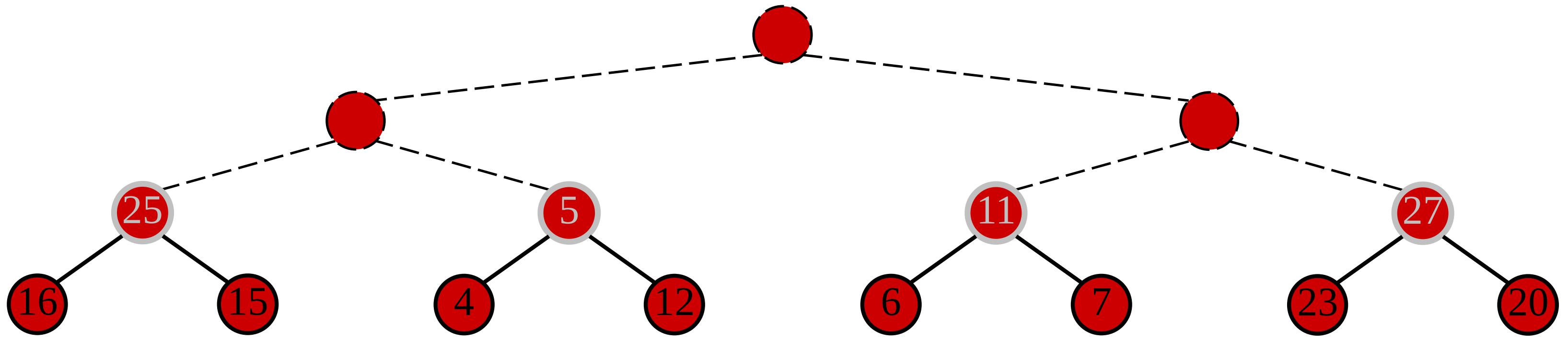
Bottom-up Heap Construction

We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys

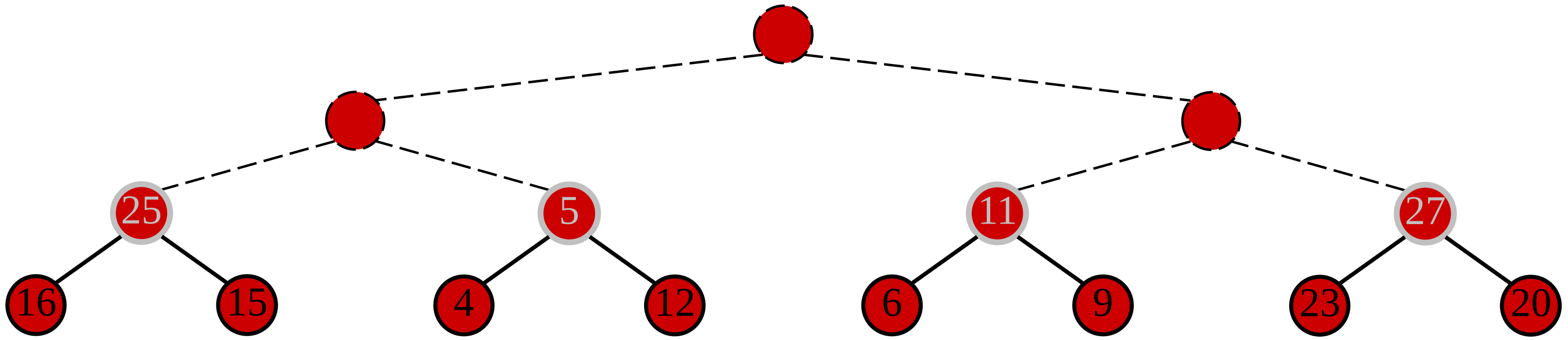


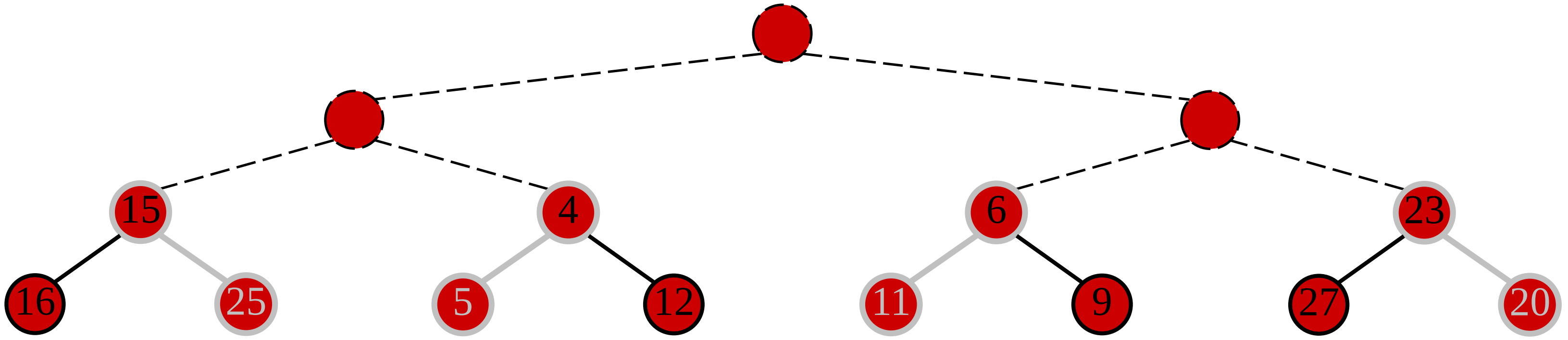
Example



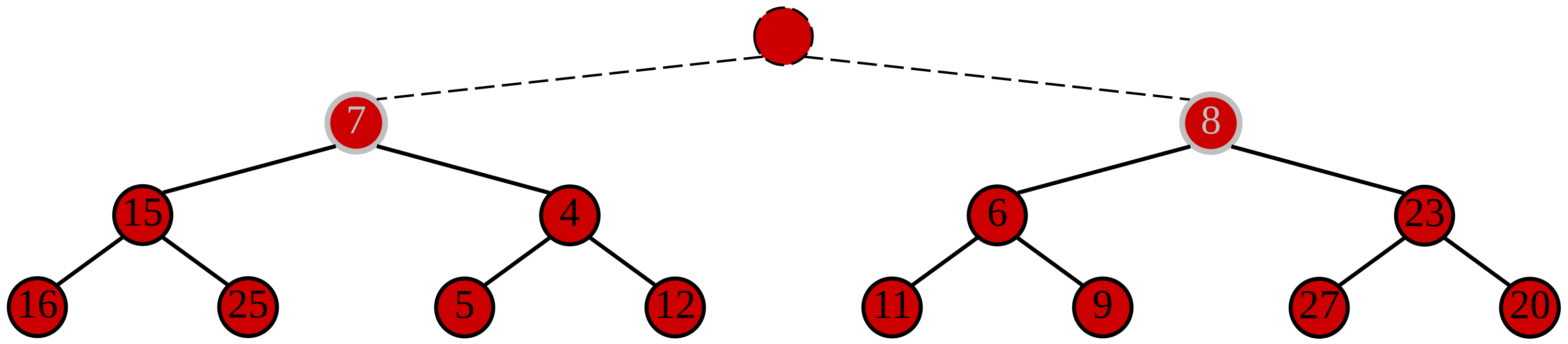


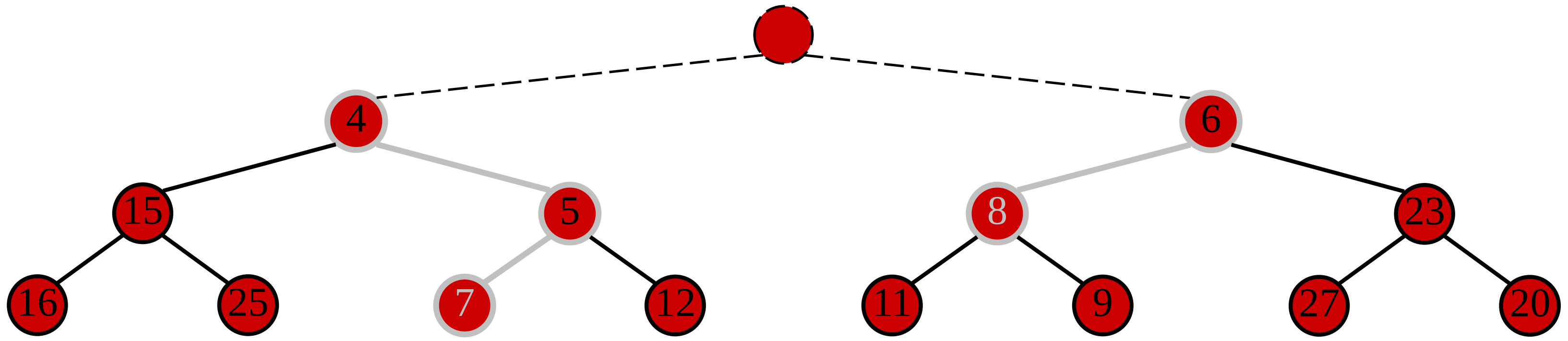
Example (contd.)



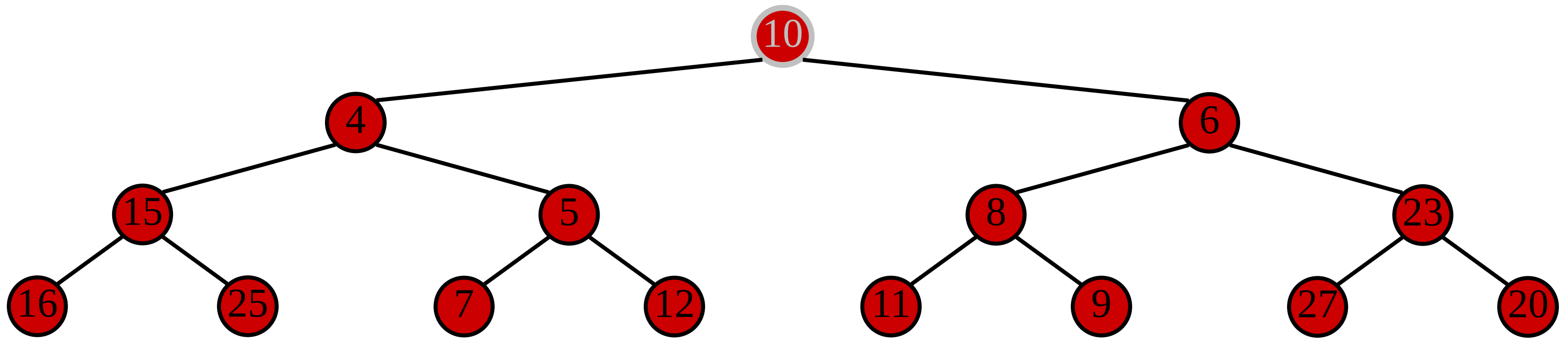


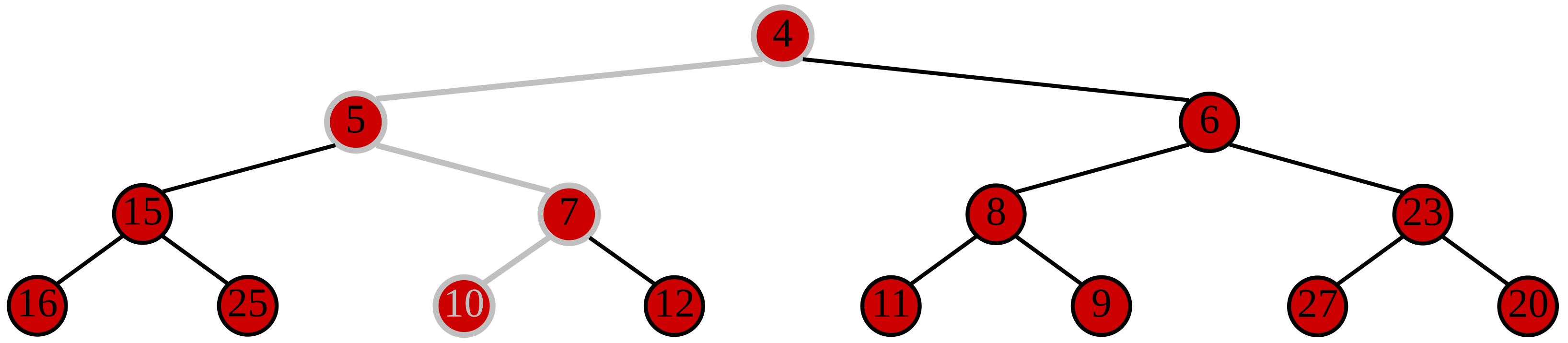
Example (contd.)





Example (end)





Analysis

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort