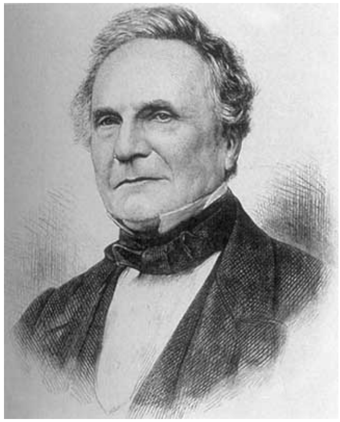
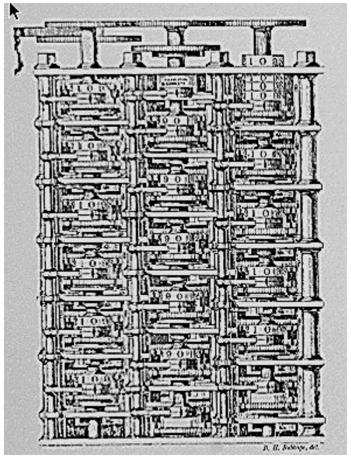
Algorithm runtime analysis and computational tractability

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)

Time Complexity of an Algorithm

How do we measure the **complexity** (time, space requirements) of an algorithm.

The **size** of the problem: an integer n

- \# inputs (for sorting problem)
- \#digits of input (for the primality problem)
- sometimes more than one integer

We want to characterize the running time of an algorithm for increasing problem sizes by a function T(n)

Units of time

1 microsecond ?

1 machine instruction?

# of code fragments that take constant time?

Units of time

1 microsecond ?

no, too specific and machine dependent

1 machine instruction?

no, still too specific and machine dependent

# of code fragments that take constant time?

yes

what kind of instructions take constant time?

arithmetic op, memory access

unit of space

bit?

int?

unit of space

bit?

very detailed but sometimes necessary

int?

nicer, but dangerous: we can code a whole program or array (or disk) in one **arbitrary** int, so we have to be careful with space analysis (take value ranges into account when needed). Better to think in terms of machine **words**

i.e. fixed size, e.g. 64, collections of bits

Worst-Case Analysis

Worst case running time.

A bound on largest possible running time of algorithm on inputs of size n.

- Generally captures efficiency in practice, but can be an overestimate.

Same for worst case space complexity

Average case

Average case running time. A bound on the average running time of algorithm on random inputs as a function of input size n. In other words: the **expected** number of steps an algorithm takes.

$$\sum_{i \in I_n} P_i . C_i$$

$$P_i : probability \;\; input \;\; i \;\; occurs$$

$$C_i : complexity \;\; given \;\; input \;\; i$$

$$I_n : \;\; all \;\; possible \;\; inputs \;\; of \;\; size \;\; n$$

- Hard to model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
- Often hard to compute.

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

A definition of tractability: Polynomial-Time

Brute force. For many 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.
  - Permutations, TSP

> An algorithm is said to be polynomial if there exist constants $c > 0$ and $d > 0$ such that on every input of size n, its running time is bounded by $c\,n^d$ steps.

- What about an n log n algorithm?

Worst-Case Polynomial-Time

On the one hand:

- Possible objection: 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 typically have **low constants and low exponents**.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

On the other:

- Some exponential-time (or worse) algorithms are widely used because the worst-case (exponential) instances seem to be rare.
  - simplex method solving **linear programming problems**

Comparing algorithm running times

Suppose that algorithm A has a running time bounded by

$$T(n) = 1.62\ n^2 + 3.5\ n + 8$$

- It is hard to get this kind of exact statement
  - It is probably machine dependent
- There is more detail than is useful
- We want to quantify running time in a way that will allow us to identify broad classes of algorithms
- I.e., we only care about Orders of Magnitude
  - in this case : $T(n) = O(n^2)$

Asymptotic Growth Rates

Upper bounds

T(n) is **O(f(n))** if there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$\text{for all } n \geq n_0 : \; T(n) \leq c \cdot f(n)$$

Example: $T(n) = 32n^2 + 16n + 32$.

- $T(n)$ is $O(n^2)$
- BUT ALSO: $T(n)$ is $O(n^3)$, $T(n)$ is $O(2^n)$.

There are many possible upper bounds for one function! We always look for a tight bound **Θ(f(n)) later,** but it is not always easy to establish

Expressing Lower Bounds

Big O Doesn't always express what we want:

Any comparison-based sorting algorithm **requires at least** $c(n \log n)$ comparisons, for some constant c.

- Use Ω for **lower bounds.**

$T(n)$ is $\mathbf{\Omega(f(n))}$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0 : T(n) \geq c \cdot f(n)$

Example: $T(n) = 32n^2 + 16n + 32$.

- $T(n)$ is $\Omega(n^2)$, $\Omega(n)$.

Tight Bounds

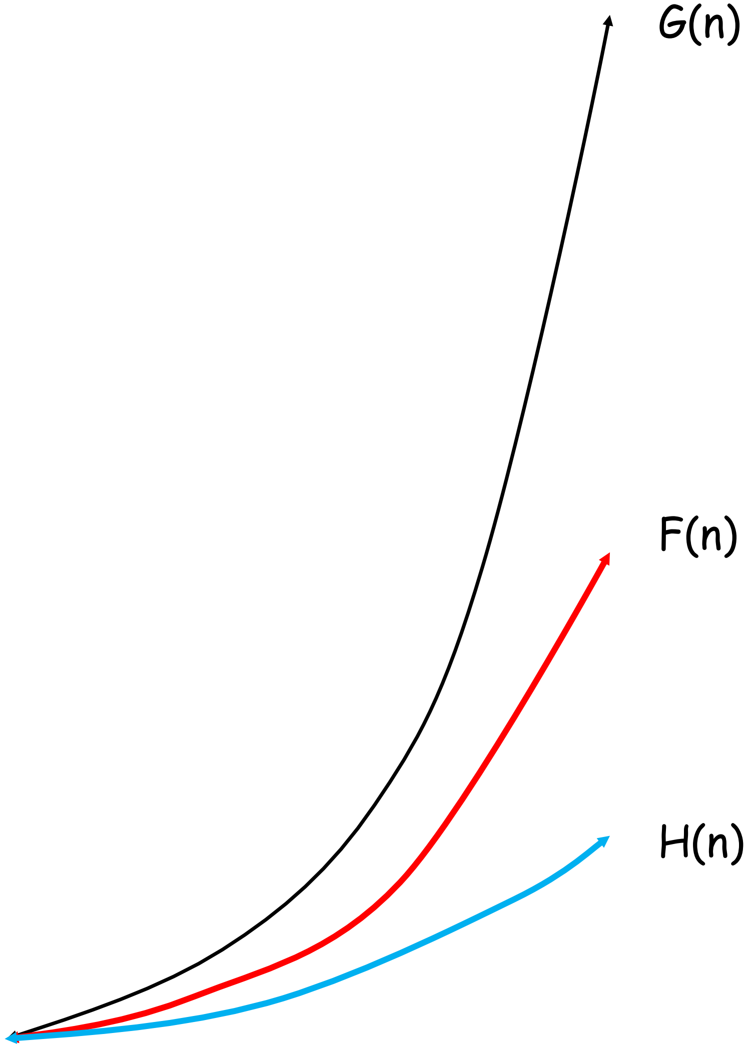
$T(n)$ is **$\Theta(f(n))$** if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

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

- $T(n)$ is $\Theta(n^2)$.

If we show that the running time of an algorithm is $\Theta(f(n))$, we have closed the problem and found a bound for the problem and its algorithm solving it.

- excursion: heap sort and priority queues



F(n) is O(G(n))

F(n) is Ω(H(n))

if G(n) = c.H(n)
then F(n) is Θ(G(n))

Priority Queue

priority Queue: data structure that maintains a set S of elements.

Each element v in S has a key **key(v)** that denotes the **priority** of v.

Priority Queue provides support for

- **adding**, **deleting** elements,
- **selection / extraction of**
  - **smallest** (Min prioQ) / **largest** (Max prioQ) key element,
- **changing** key value.

Applications

E.g. used in managing real time events where we want to get the earliest next event and events are added / deleted on the fly.

Sorting

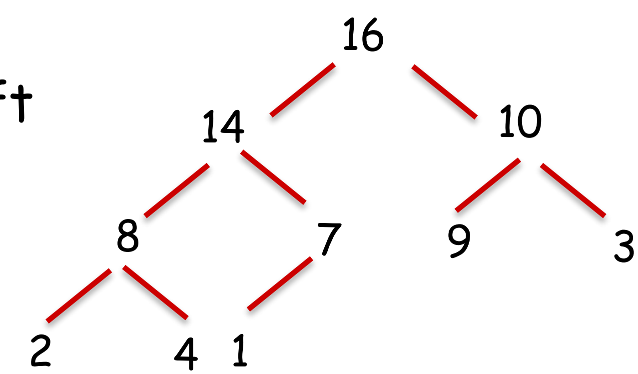
- build a prioQ
- Iteratively extract the smallest element

PrioQs can be implemented using **heaps**

Heaps

Heap: array representation of a **complete** binary tree

- every level is completely filled except the bottom level: filled from left to right
- Can compute the index of parent and children, for 1 based arrays:
  - parent(i) = floor((i-1)/2)
    leftChild(i)= 2i+1
    rightChild(i)=2(i+1)



16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9

Max Heap property:

A[parent(i)] >= A[i]

Min heaps have the min at the root

Heapify(A,i,n)

To create a heap at index i, assuming left(i) and right(i) are heaps, **bubble A[i] down**: swap with max child until heap property holds

```
heapify(A,i,n):
# precondition
# n is the size of the heap
# tree left(i) and tree right(i) are heaps
```

Building a heap

heapify performs at most lg n swaps

why? what is n?

building a heap out of an array:

- the leaves are all heaps
- heapify **backwards** starting at last internal node

WHY backwards?

Complexity buildheap

Suggestions? ...

Complexity buildheap

Initial thought: O(n lgn), but

half of the heaps are height 1
quarter are height 2
only one is height log n

It turns out that O(n lgn) is not tight!

complexity buildheap

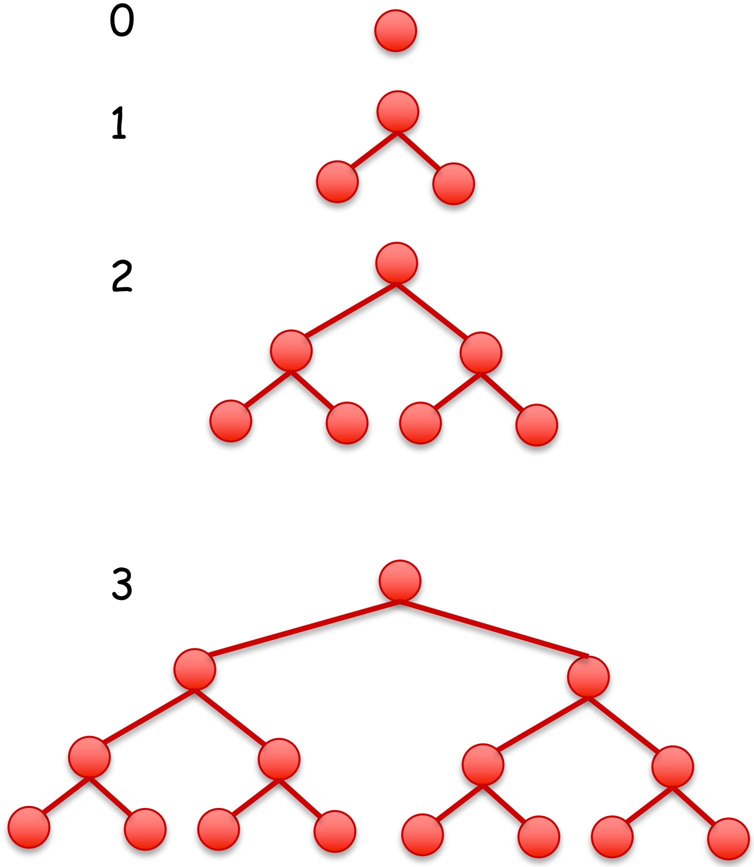
height | max #swaps ?

0

1

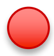
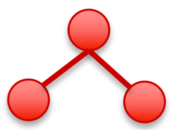
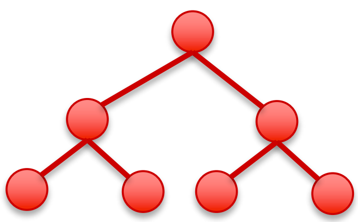
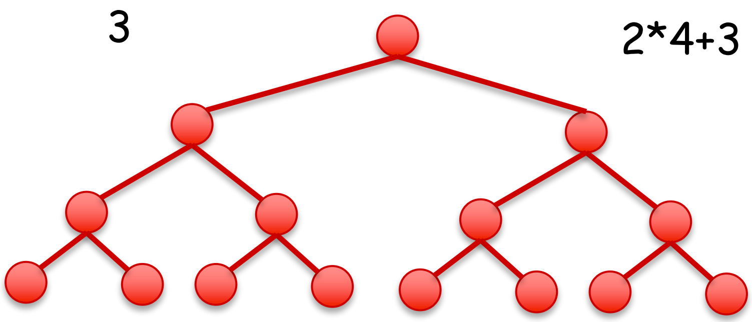
2

3

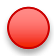
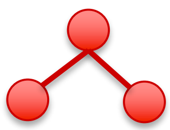
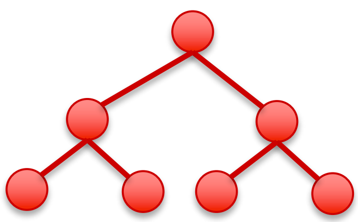
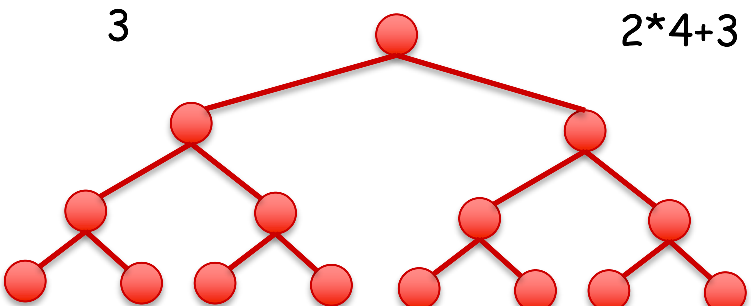


complexity buildheap

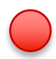
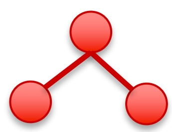
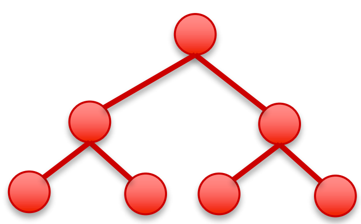
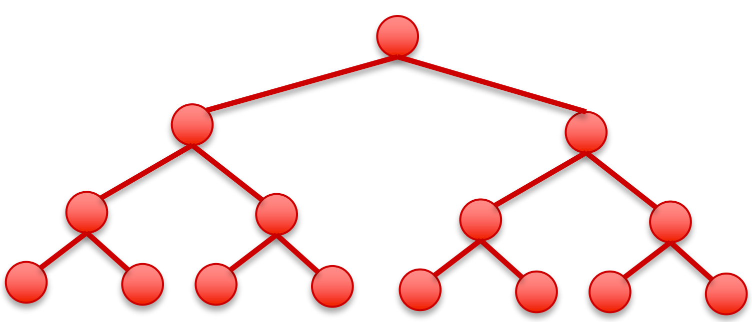
max #swaps, see a pattern?
(What kind of growth function do you expect ?)

height		max #swaps
0		0
1		1
2		2*1+2 = 4
3		2*4+3 = 11

complexity buildheap

height		max #swaps
0		$0 = 2^1-2$
1		$1 = 2^2-3$
2		$2*1+2 = 4 = 2^3-4$
3		$2*4+3 = 11 = 2^4-5$

complexity buildheap

height		max #swaps
0		$0 = 2^1-2$
1		$1 = 2^2-3$
2		$2*1+2 = 4 = 2^3-4$
3		$2*4+3 = 11 = 2^4-5$

Conjecture:
  height = h
  max #swaps = $2^{h+1}-(h+2)$

Proof: induction
  base?
  step:
    height = (h+1)
    max #swaps:

$$
\begin{aligned}
& 2*(2^{h+1}-(h+2))+(h+1) \\
&= 2^{h+2}-2h-4+h+1 \\
&= 2^{h+2}-(h+3) \\
&= 2^{(h+1)+1}-((h+1)+2)
\end{aligned}
$$

n nodes $\rightarrow \Theta(n)$ swaps

See it the Master theorem way

T(n) = 2*T(n/2)+ lg n

Master theorem $\Theta(n^{\lg_2 2}) = \Theta(n)$

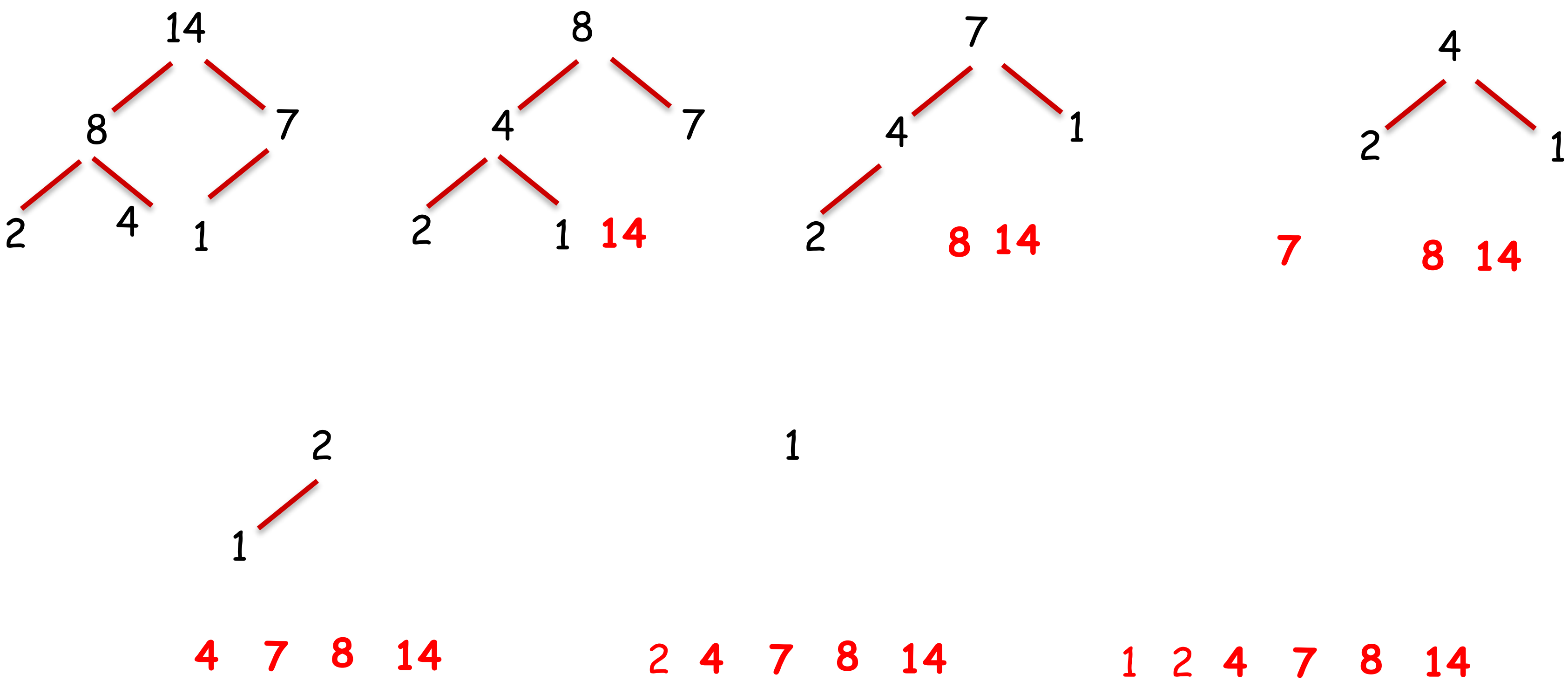
Heapsort, complexity

```
heapsort(A):
   buildheap(A)
   for i = n-1 downto 1 :
     # put max at end array

     # max is removed from heap
     n=n-1

     # reinstate heap property
```

- buildheap: $\Theta(n)$
- heapsort: $\Theta(n \lg n)$

- space: in place: $\Theta(n)$

DO IT, DO IT!



Priority Queues

heaps can be used to implement priority queues:

- each value associated with a key
- max priority queue S has operations that maintain the heap property of S
  - max(S) returning max element
  - Extract-max(S) extracting and returning max element
  - increase key(S,x,k) increasing the key value of x
  - insert(S,x)
    - put x at end of S
    - bubble x (see Increase-key)

Back to O, Properties

Transitivity.

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

Additivity.

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

Asymptotic Bounds for Some Common Functions

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

Polynomial time. Running time is $O(n^d)$ for some constant d

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

can avoid specifying the base

for every $x > 0$, $\log n$ is $O(n^x)$.

log grows slower than any polynomial

Combinations. Merge sort, Heap sort $O(n\log n)$

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

exponential grows faster than any polynomial

Problems have lower bounds

A **problem** has a lower bound **$\Omega(f(n))$**, which means :

Any algorithm solving this problem takes

at least $\Omega(f(n))$ steps

We can often show that an algorithm has to "touch" all elements of a data structure, or produce a certain sized output. This then gives rise to an easy lower bound.

Sometimes we can prove better (higher, stronger) lower bounds (eg Searching and Sorting (cs420)).

Closed / open problems

Problems have lower bounds, algorithms have upper bounds. A closed problem has a lower bound $\Omega(f(n))$ and at least one algorithm with upper bound $O(f(n))$

- Example: sorting is $\Omega(n\log n)$ and there are $O(n\log n)$ sorting algorithms.

  To show this, we need to reason about lower bounds of problems (cs420)

An open problem has lower bound < upper bound

- Example: matrix multiplication (multiply two n x n matrices).
  - Takes $\Omega(n^2)$ why?
  - Naïve algorithm: $O(n^3)$
  - Coppersmith-Winograd algorithm: $O(n^{2.376})$

A Survey of Common Running Times

Constant time: $O(1)$

A single line of code that involves "simple" expressions, e.g.:

- Arithmetical operations (+,-,*,/) for fixed size inputs
- assignments (x = simple expression)
- conditionals with simple sub-expressions
- function calls (excluding the time spent in the called function)

Logarithmic time

Example of a problem with O(log(n)) bound: binary search

How did we get that bound?

log(n) and algorithms

When in each step of an algorithm we halve the size of the problem then it takes $\log_2 n$ steps to get to the base case

We often use log(n) when we should use floor(log(n)). That's OK since floor(log(n)) is $\Theta(\log(n))$

Similarly, if we divide a problem into k parts the number of steps is $\log_k n$. For the purposes of big-O analysis it doesn't matter since

$\log_a n$ is $O(\log_b n$ **WHY?**

Logarithms

definition:

$b^x = a \rightarrow x = \log_b a$, eg $2^3=8$, $\log_2 8=3$

$$b^{\log_b a} = a \qquad \log_b b = 1 \qquad \log 1 = 0$$

- $\log(x*y) = \log x + \log y$ because $b^x\, b^y = b^{x+y}$
- $\log(x/y) = \log x - \log y$
- $\log x^a = a \log x$
- $\log x$ is a 1-to-1 monotonically (slow) growing function

$$\log x = \log y \quad \Leftrightarrow \quad x = y$$

- $\log_a x = \log_b x \,/\, \log_b a$
- $y^{\log x} = x^{\log y}$

$\log_a x = \log_b x / \log_b a$

$$b^{\log_b x} = x = a^{\log_a x} = b^{(\log_b a)(\log_a x)}$$

$$\log_b x = (\log_b a)(\log_a x)$$

$$\log_a x = \log_b x / \log_b a$$

therefore $\log_a x = O(\log_b x)$ for any a and b

$y^{\log x} = x^{\log y}$

$$x^{\log_b y} =$$

$$y^{\log_y x \log_b y} =$$

$$y^{(\log_b x / \log_b y) \log_b y} =$$

$$y^{\log_b x}$$

Linear Time: O(n)

Linear time. Running time is proportional to the size of the input.

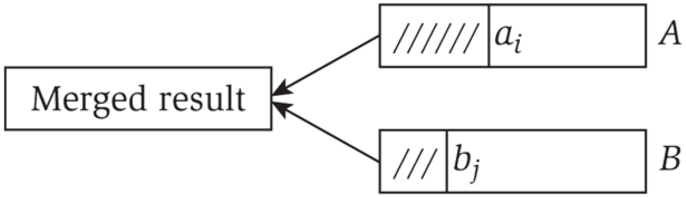
Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

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

Also $\Theta(n)$?

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 a single sorted list.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (ai ≤ bj) append ai to output list and increment i
   else         append bj 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.

Linear Time: O(n)

Polynomial evaluation. Given

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \quad (a_n != 0)$$

Evaluate $A(x)$

How not to do it:

$$a_n * \exp(x,n) + a_{n-1} * \exp(x,n-1) + \ldots + a_1 * x + a_0$$

Why not?

How to do it: Horner's rule

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 =$$

$$(a_n x^{n-1} + a_{n-1} x^{n-2} + \ldots + a_1) x + a_0 = \ldots =$$

$$(\ldots(a_n x + a_{n-1}) x + a_{n-2}) x \ldots + a_1) x + a_0$$

```
y=a[n]
for (i=n-1;i>=0;i--)
    y = y *x  + a[i]
```

Polynomial evaluation using Horner: complexity

Lower bound: $\Omega(n)$ because we need to access each a[i] at least once

Upper bound: $O(n)$

Closed problem!

But what if $A(x) = x^n$

$A(x)=x^n$

Recurrence:

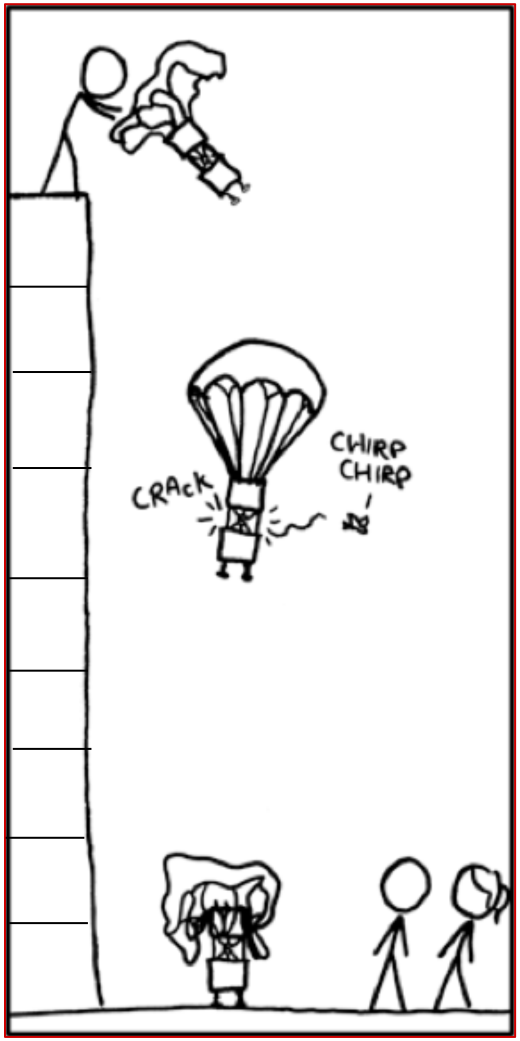
$x^{2n}=x^n * x^n$ $x^{2n+1}=x * x^{2n}$

```
def pwr(x, n) :
    if (n==0) : return 1
    if odd(n) :
        return x * pwr(x, n-1)
    else :
        a = pwr(x, n/2)
        return a * a
```

Complexity?

A glass-dropping experiment

- You are testing a model of glass jars, and want to know from what height you can drop a jar without its breaking. You can drop the jar from heights of 1,…,n foot heights. Higher means faster means more likely to break.
- You want to minimize the amount of work (number of heights you drop a jar from). Your strategy would depend on the number of jars you have available.
- If you have a single jar:
  - do linear search (O(n) work).
- If you have an unlimited number of jars:
  - do binary search (O(log n) work)
- Can you design a strategy for the case you have 2 jars, resulting in a bound that is strictly less than O(n)?



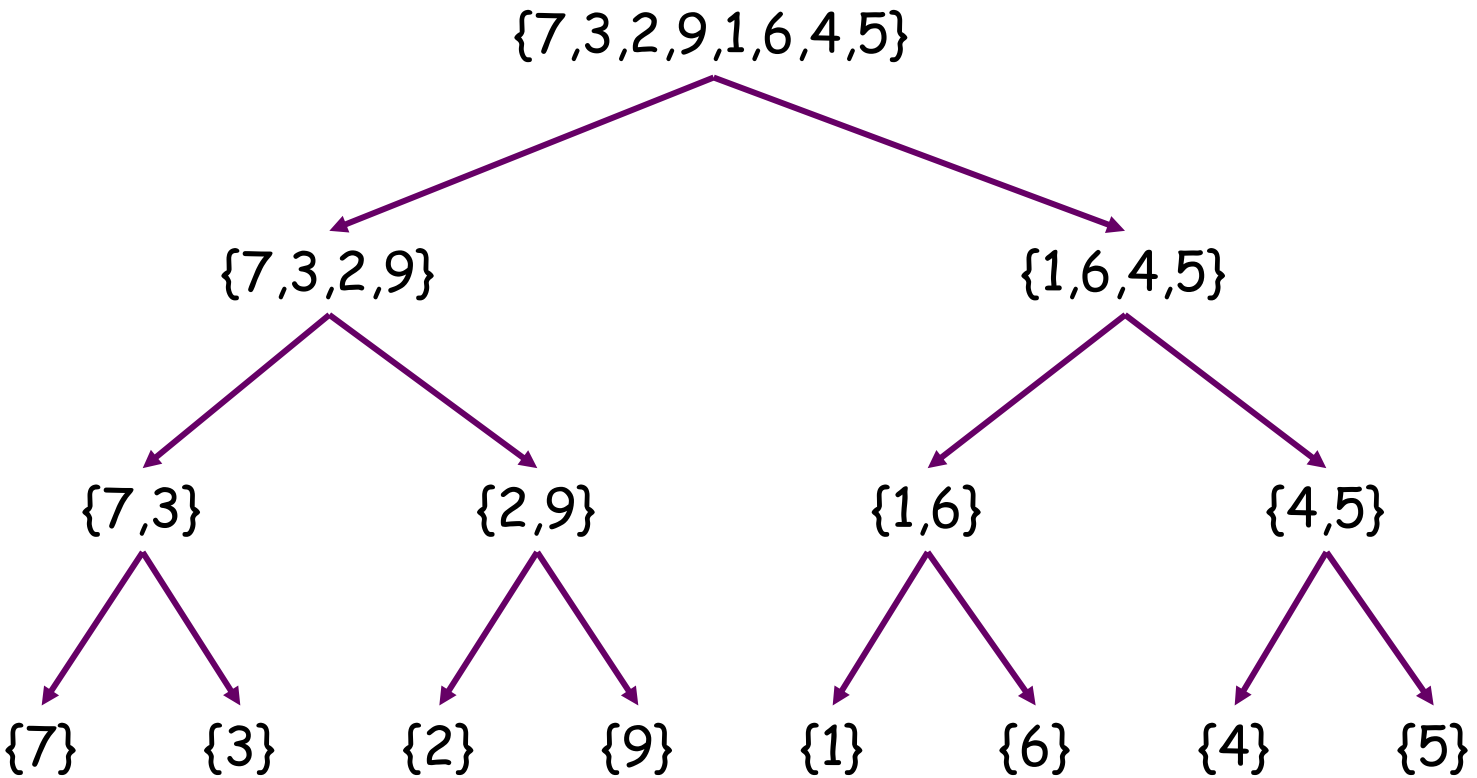
http://xkcd.com/510/

O(n log n) Time

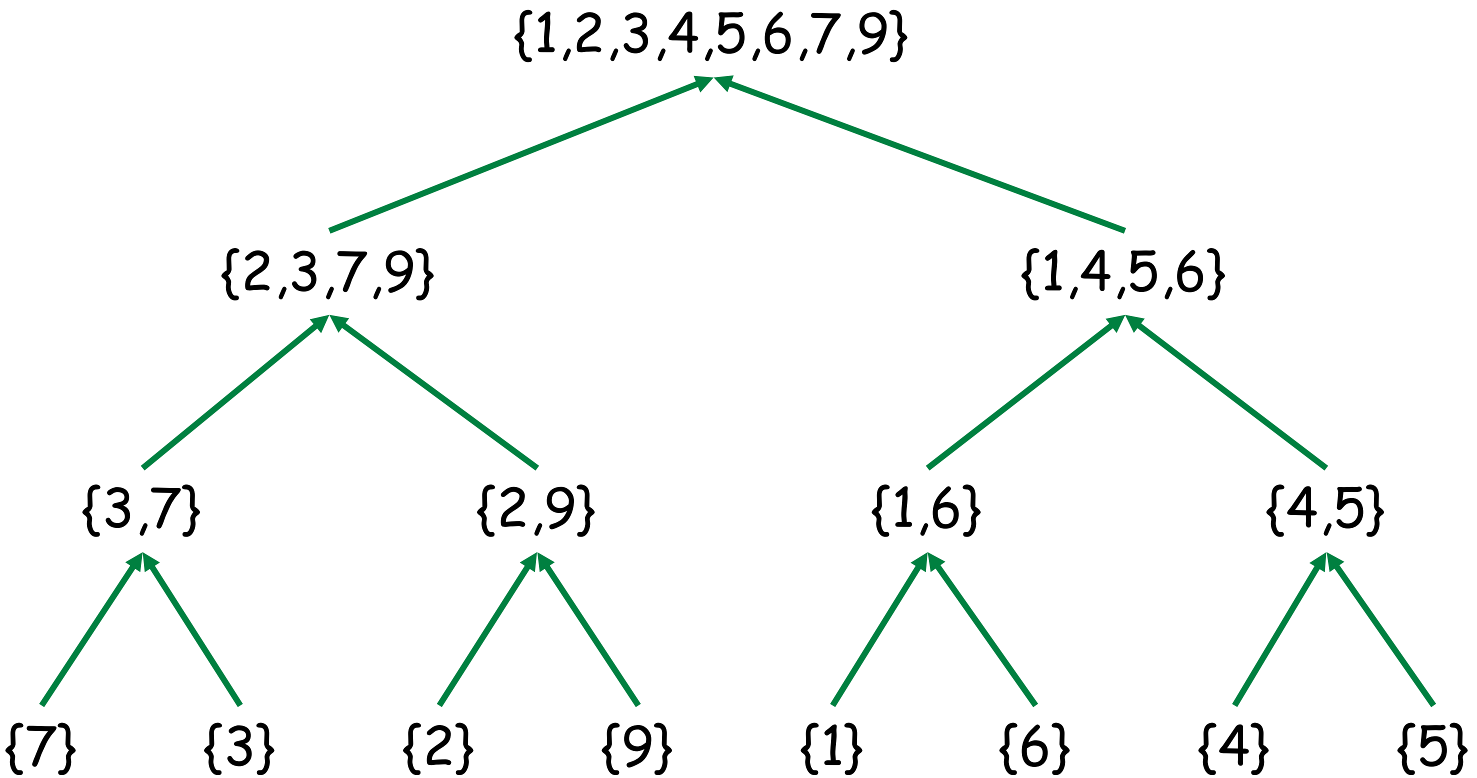
Often arises in divide-and-conquer algorithms like mergesort.

```
mergesort(A) :
    if len(A) <= 1 return A
    else return merge(mergesort(left half(A)),
                      mergesort(right half(A)))
```

Merge Sort - Divide

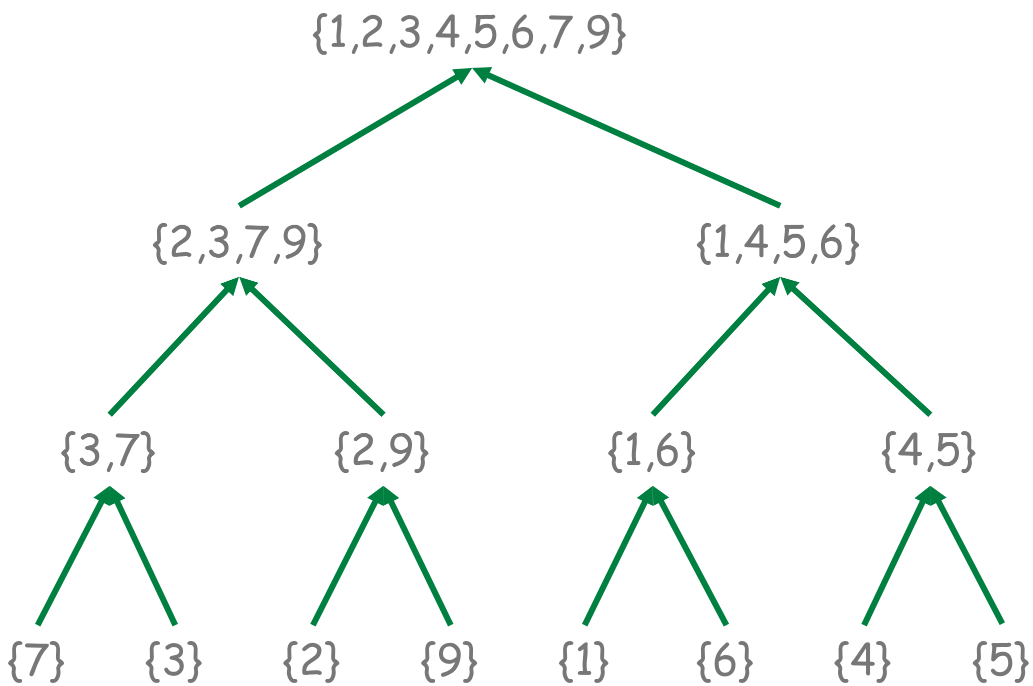


Merge Sort - Merge



O(n log n)

```
mergesort(A) :
    if len(A) <= 1 return A
    else return merge(mergesort(left half(A)),
                      mergesort(right half(A)))
```



At depth i

- work done
  - split
  - merge
- total work?

Total depth?

Total work?

Quadratic Time: $O(n^2)$

Quadratic time example. 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 ← (x1 - x2)^2 + (y1 - y2)^2
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)^2 + (yi - yj)^2
      if (d < min)
         min ← d
   }
}
```

see chapter 5

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

Cubic Time: $O(n^3)$

Example 1: Matrix multiplication

Tight?

Example 2: Set disjoint-ness. Given n sets $S_1, \ldots, S_n$ 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 Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
   }
}
```

what do we need for this to be $O(n^3)$?

Largest interval sum

Given an array A[0],…,A[n – 1], find indices i,j such that the sum A[i] +… +A[j] is maximized.

Example:

A = [2, -3, 4, 2, 5, 7, -10, -8, 12]

Naïve algorithm :

```
maximum_sum = - infinity
for i in range(n - 1) :
    for j in range(i, n) :
        current_sum = A[i] +… +A[j]
        if current_sum >= maximum_sum :
            maximum_sum = current_sum
            save the values of i and j
```

big O bound?

Can we do better?

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 = $\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \le \frac{n^k}{k!}$
- $O(k^2\, n^k / k!) = O(n^k)$.

poly-time for k=17,
but not practical

Exponential Time

Independent set. Given a graph, what is the 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
   }
}
```

For some problems (e.g. TSP) we need to consider all permutations. The factorial function ($n!$) grows much faster than 2^n

Questions

1. Is 2^n $O(3^n)$?
2. Is 3^n $O(2^n)$
3. Is 2^n $O(n!)$?
4. Is $n!$ $O(2^n)$

Polynomial, NP, Exponential

Some problems (such as matrix multiply) have a polynomial complexity solution: an **$O(n^p)$** time algorithm solving them. (p constant)

Some problems (such as Hanoi) take an exponential time to solve: **$\Theta(p^n)$** (p constant)

For some problems we only have an exponential solution, **but we don't know if there exists a polynomial solution**. Trial and error algorithms are the only ones we have so far to find an exact solution. If we would always make the right guess these algorithms would take polynomial time. Therefore we call these problems NP (we will discuss NP later)

Some NP problems

TSP: Travelling Salesman
given cities $c_1, c_2, \ldots, c_n$ and distances between all of these, find a minimal tour connecting all cities.

SAT: Satisfiability
given a boolean expression E with boolean variables $x_1, x_2, \ldots, x_n$ determine a truth assignment to all x_i making E true

Back tracking

Back tracking searches (walks) a state space, at each choice point it guesses a choice.

In a leaf (no further choices) if solution found OK, else go back to last choice point and pick another move.

NP is the class of problems for which we can check in polynomial time whether it is correct (certificates, later)

Coping with intractability

NP problems become intractable quickly

TSP for 100 cities?

How would you enumerate all possible tours? How many?

Coping with intractability:

- Approximation: Find a nearly optimal tour
- Randomization: use a probabilistic algorithm using "coin tosses" (eg prime witnesses)