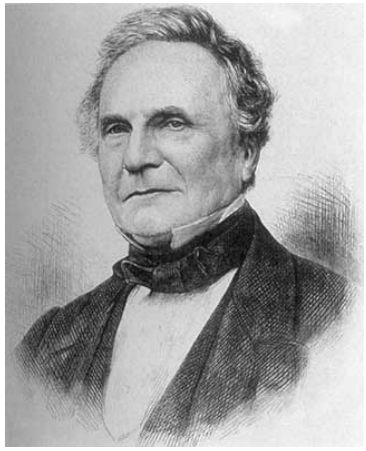
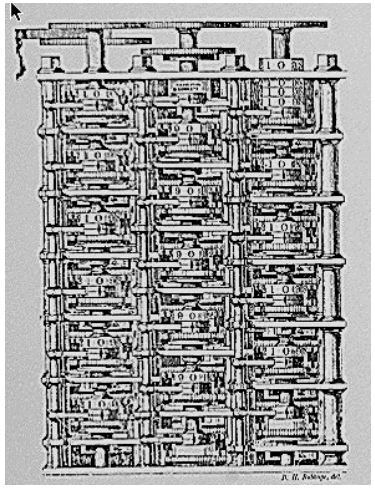
Algorithm runtime analysis and computational tractability

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)

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?

Guessing game

I have a number between 0 and 63

How many (Y/N) questions do you need to find it?

What's the number?

What (kind of) questions would you ask?

Guessing game

I have a number between 0 and 63

How many (Y/N) questions do you need to find it?

is it >= 32 N
is it >= 16 Y
is it >= 24 N
is it >= 20 N
is it >= 18 Y
is it >= 19 Y

What's the number?

19

Take N=0 and Y=1, what is 010011 ?

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 equal 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 $O(\log_b n)$

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

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

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

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

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

$$y^{\log_b x}$$

Combinations of functions /code fragments

AdditiveTheorem:

Suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$.

Then $(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x))$.

Sequences of code are additive in complexity:

```
int c = 0;
for(int i=0; i<n; i++)
      c++;
for(int j=0; j<m; j++)
      c++;
```

Complexity?

What is counting the complexity?

Combinations of functions /code fragments

Multiplicative Theorem:

Suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$.

Then $(f_1 f_2)(x)$ is $O(g_1(x)g_2(x))$.

Nested code is multiplicative in complexity

```
for(int i=0; i<n; i++)
    for(int j=0; j<m; j++)
            c++;
```

Complexity?

BUT, be careful with nests where the inner loop depends outer loop:

```
int b = n;
while(b>0){
    b/=2;
    for(int i=0; i<b; i++)
        c++;
}
```

Recursive Code

Draw the call tree, and assert the number of nodes in the tree and their individual complexity, as a function of n.

Recursive Code

Draw the call tree, and assert the number of nodes in the tree and their individual complexity, as a function of n.

```
public int divCo(int n){
        if(n<=1)
                return 1;
        else
                return 1 + divCo(n-1) + divCo(n-1);
}
```

How many recursive calls?

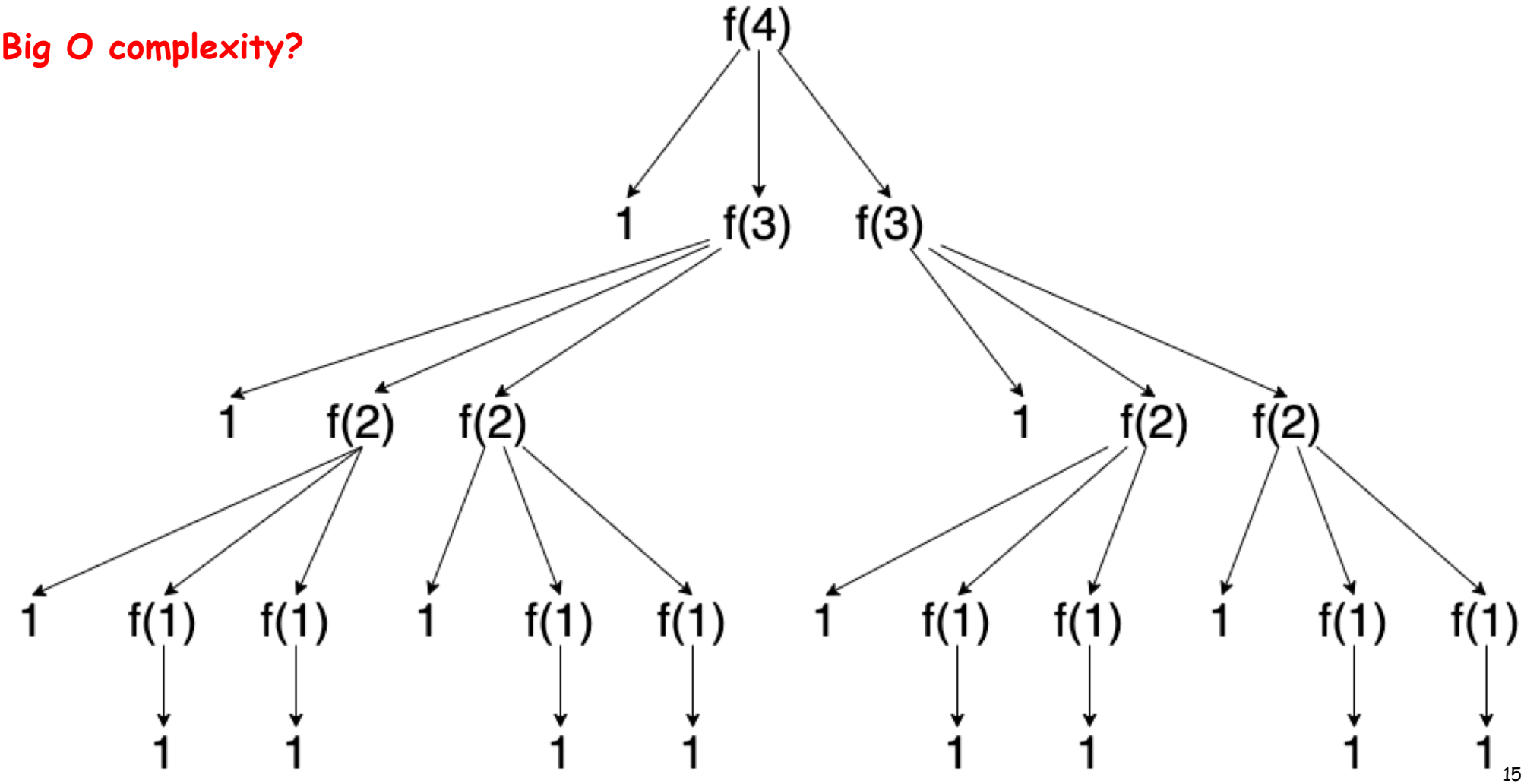
How much work per call?
What is the role of "return 1" and return 1+…" ?
So what does this function count?

Big O complexity?

How many recursive calls?

How much work per call?
What is the role of "return 1" and return 1+…" ?
So what does this function count?

Big O complexity?



Linear Time: O(n)

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

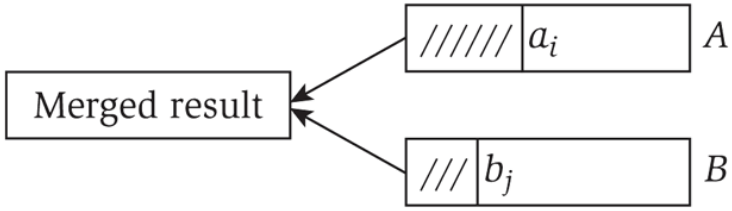
Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a_1
for i = 2 to n {
    if (a_i > max)
        max ← a_i
}
```

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 (a_i ≤ 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.

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:

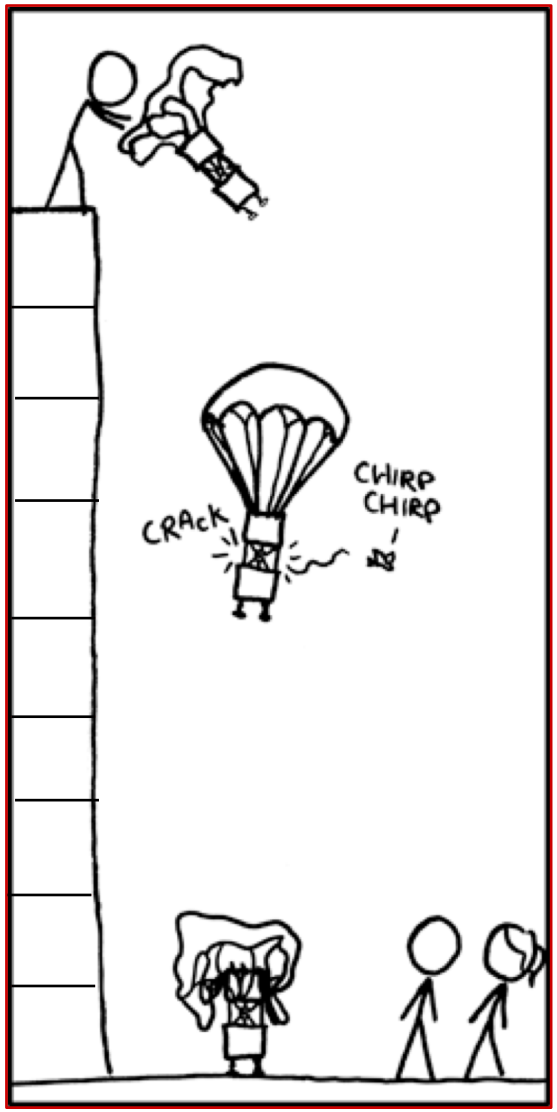
$x^{2n}=x^n * x^n$ $\quad$ $x^{2n+1}=x * x^{2n}$

```
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
```

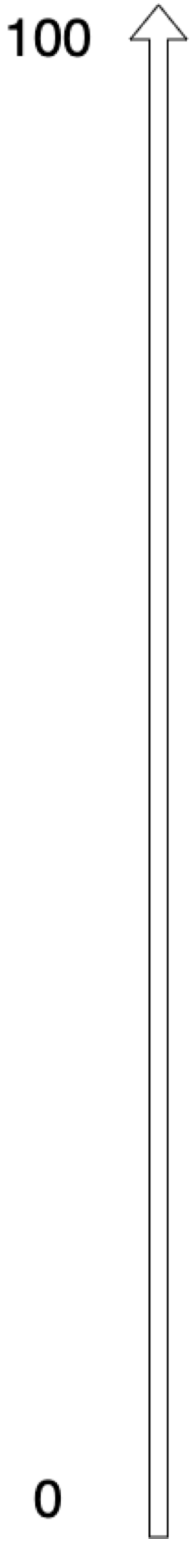
Complexity?

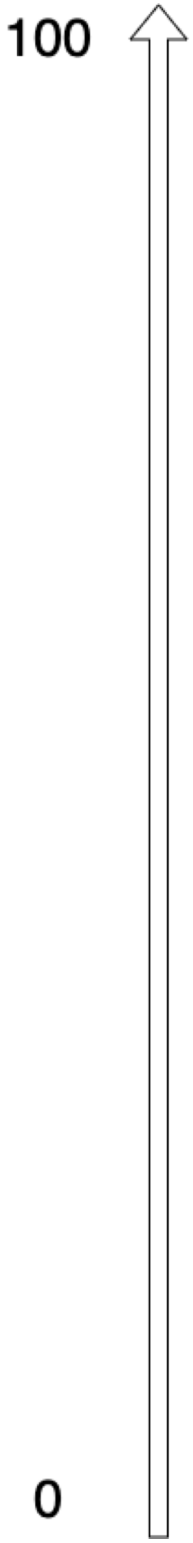
A glass-dropping experiment

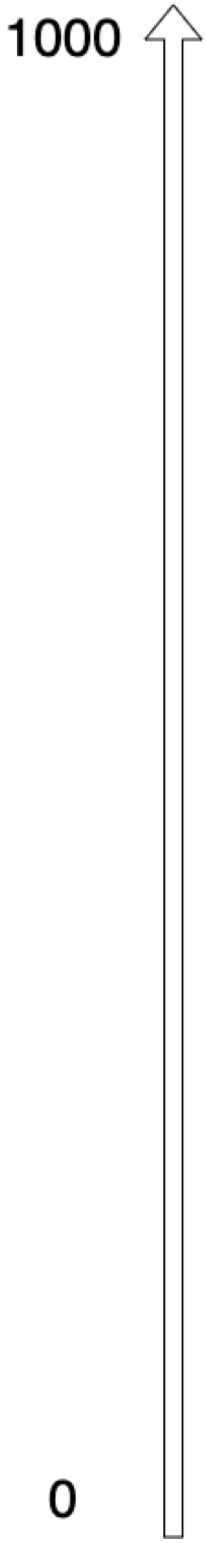
- You are testing a model of glass jars, and want to know from what height you can drop a jar without it breaking. You can drop the jar from heights of 1,…,n foot heights. Higher means faster means more likely to break.
- You want to minimize the amount of work (number of heights you drop a jar from). 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/


100
0


100
0

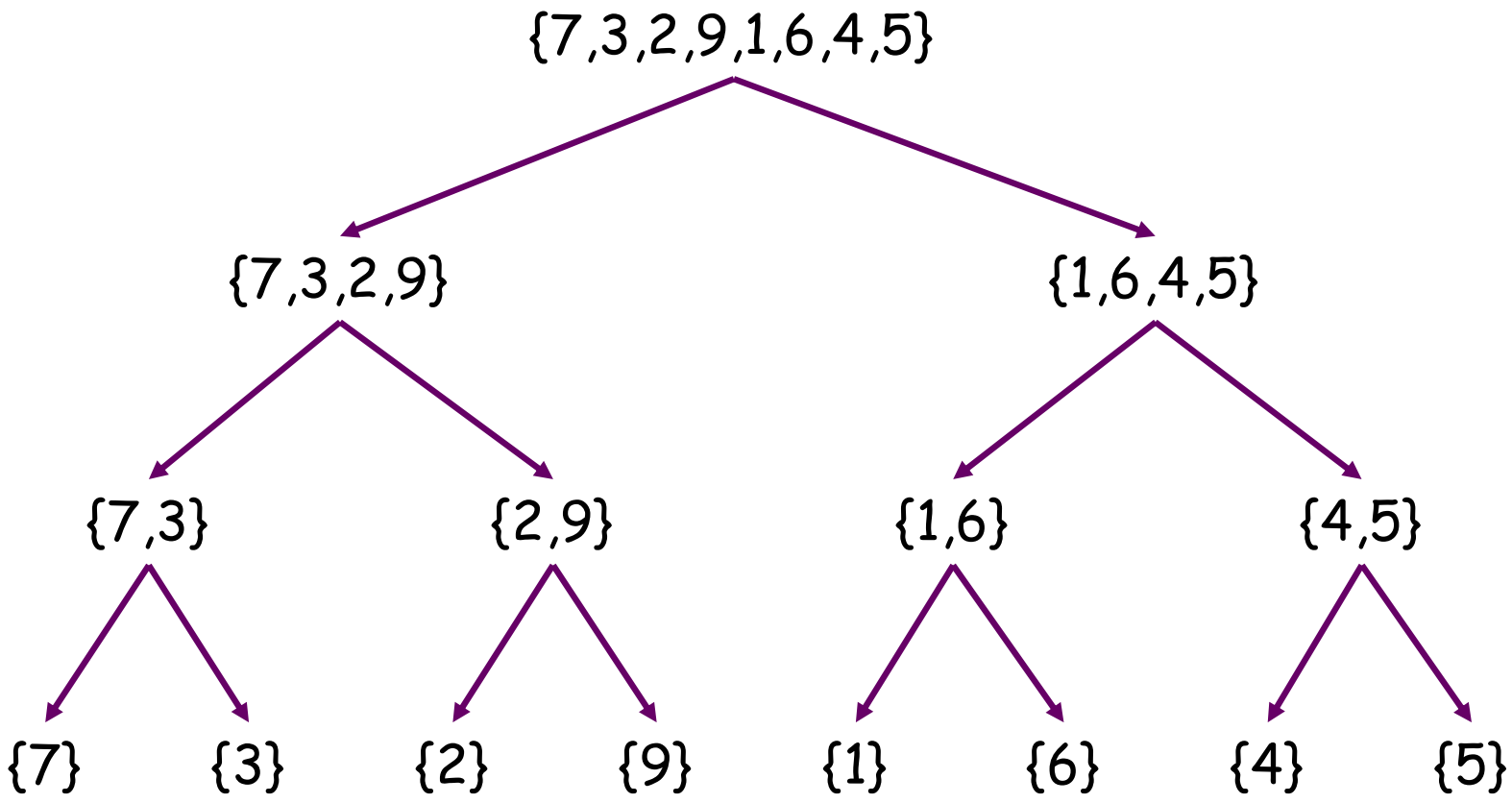

1000
0

O(n log n) Time

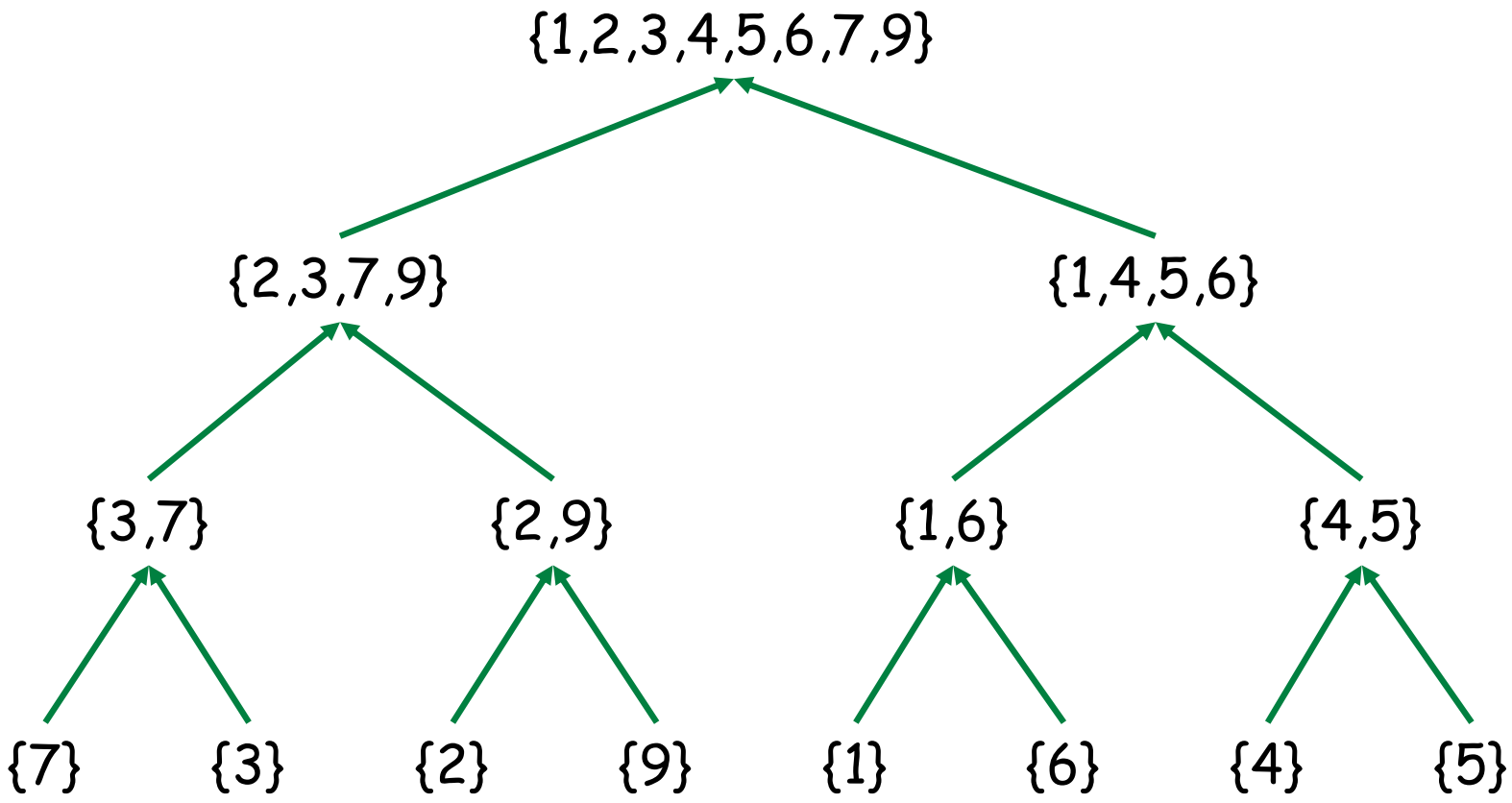
Often arises in divide-and-conquer algorithms like mergesort.

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

Merge Sort - Divide

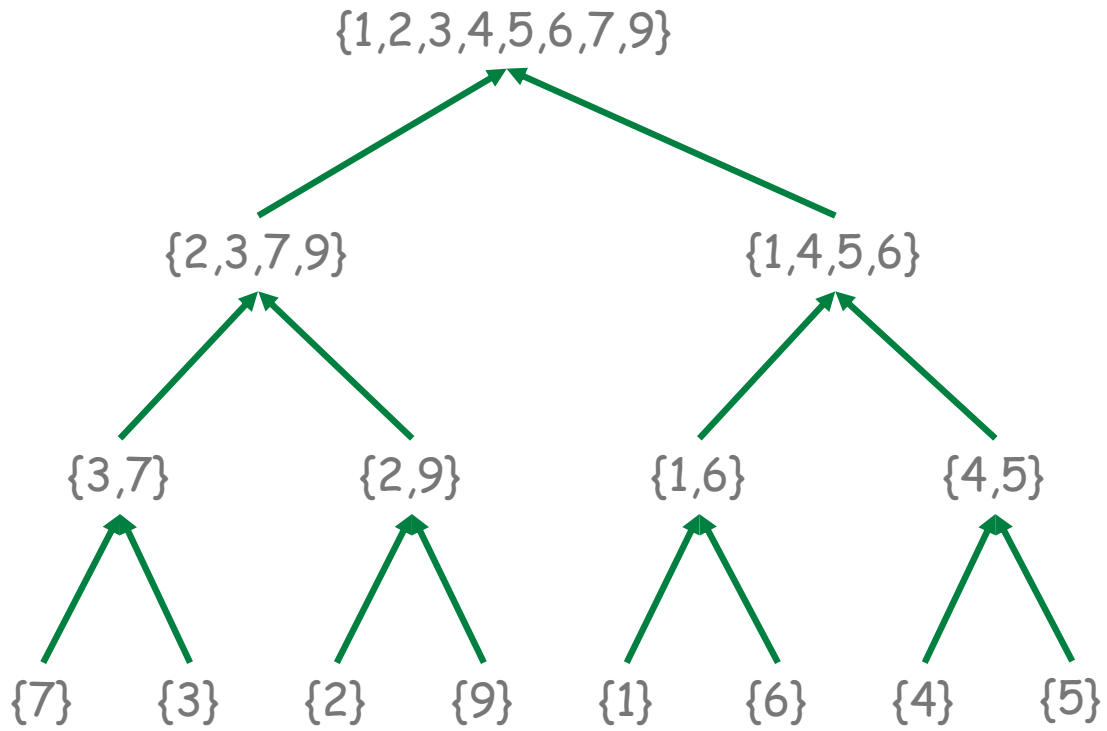


Merge Sort - Merge



O(n log n)

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



How many levels?

WHY?

At level 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
   }
}
```

Remark. $\Omega(n^2)$ seems inevitable, but . . . .

Cubic Time: $O(n^3)$

Example 1: Matrix multiplication

Tight?

Example 2: Set disjoint-ness. 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 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 (maximum segment 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 $n!$ $O(2^n)$?
5. Is $\log_2 n$ $O(\log_3 n)$?
6. Is $\log_3 n$ $O(\log_2 n)$?

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, and if we would always make the right guess, these algorithms would take polynomial time.

We call these problems NP (non deterministic polynomial) 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)