Complexity

Analyzing the time complexity of computer programs

Why is complexity important?

- Complexity is a new metric for relating tasks to the resources required to complete them.
- Why do we need this new metric? Here is an example of a GOOD DEAL:
  - Paint a 6' * 100' fence for $20. It did not take long and was a fair deal.
  - Now paint a 6' * 200' fence for $50 — This is a Good Deal.

- Why do we need this new metric? Here is an example of a BAD DEAL:
  - You are given $20 to paint as much fence as you can until you finish an 2' diameter * 2' deep can of paint and it was a fair deal.
  - Now you are offered $50 to paint as much of another fence as you can with a can of paint measuring 4' diameter * 2' deep. Is this a good deal?

A really bad deal

- You blow up 20 balloons, each is 2' diameter. You get $20.
- Now you are asked to blow up 20 balloons each of which is 4' diameter for $50.
- Why does this deal stink?
  - Volume "increases" faster than diameter.
  - A 2' balloon has volume of (4/3 pi * 1^3 = 4.19 cubic ft.)
  - A 4' balloon has a volume of (4/3 pi * 2^3 = 33.5 cubic ft.)
- So, 8 times more effort for 2.5 times more pay?

What is complexity?

- The "relationship" between a measure n of the task and a measure m of the resources required to complete the task is called the m-complexity of the task with respect to n.
- Complexity ignores:
  - constants in the relationship
  - lower-order terms in the relationship
- Complexity is the shape of the graph relating n and m.

Big O notation — some examples

- Time complexity to paint fence with respect to length l is O(l)

- Time complexity to paint a fence with respect to the diameter d of the paint tin is O(d^2).

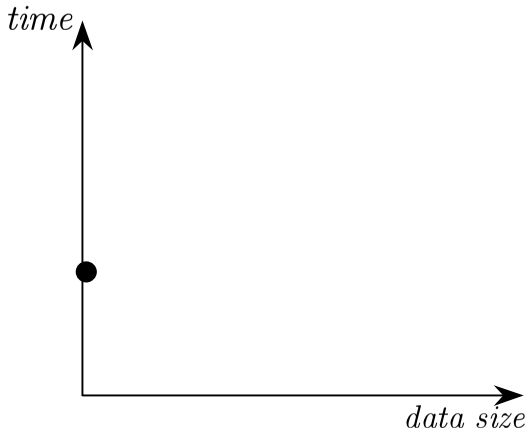
- Time complexity to blow up balloons with respect to diameter d is O(d^3).

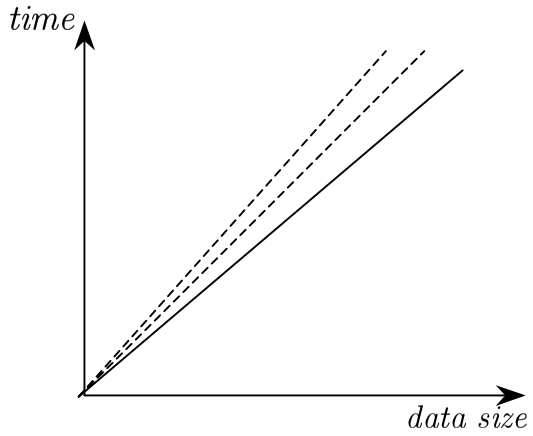
- Time complexity to blow up balloons with respect to volume v is O(v).

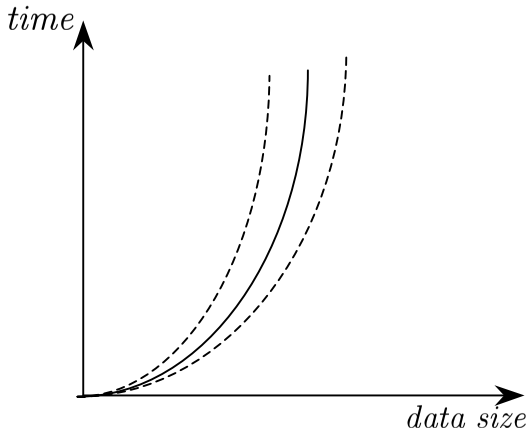
Examples fo complexity values

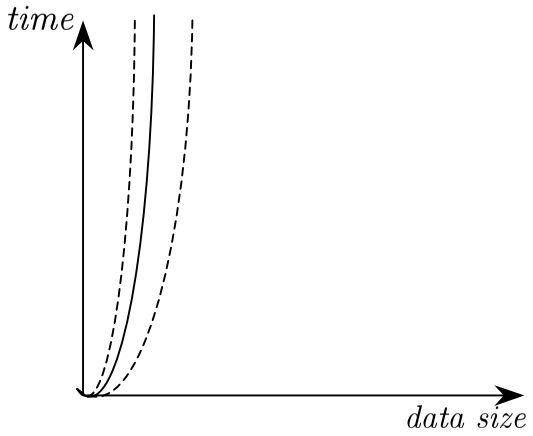
Example	Informal name
O(2^(n^n)	double exponential
O(2^n)	exponential
O(n^3)	quadratic - cubic
O(n^2)	quadratic
O(n log n)	n log n
O(n)	linear
O(log n)	logarithmic
O(k)	constant

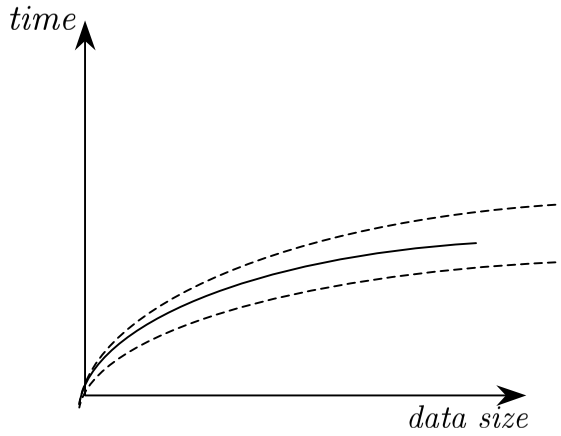
Graphs Related to Complexity



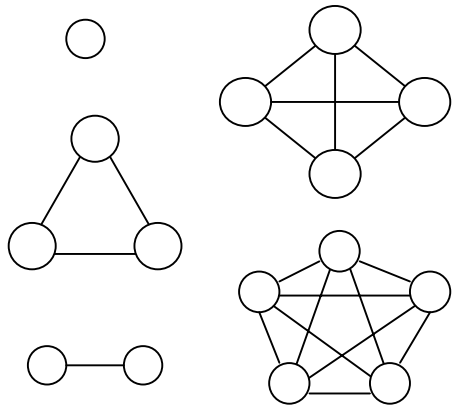




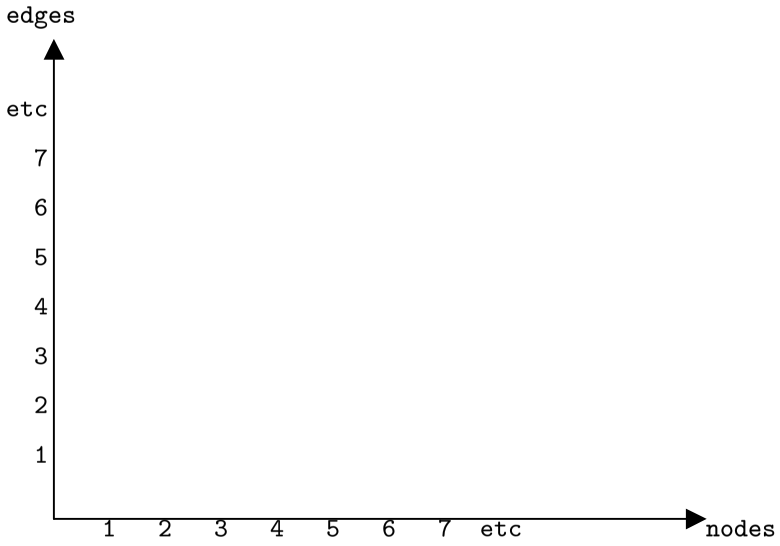




```
    if we have      we have

c0  0 nodes         0 edges
c1  1 node          0 edges
c2  2 nodes         1 edges
c3  3 nodes         3 edges
c4  4 nodes         6 edges
c5  5 nodes         10 edges
c6  6 nodes         15 edges
c7  etc...          etc...
```



```
numberOfEdges 1 = 0
numberOfEdges n = n-1 + numberOfEdges (n-1)
```

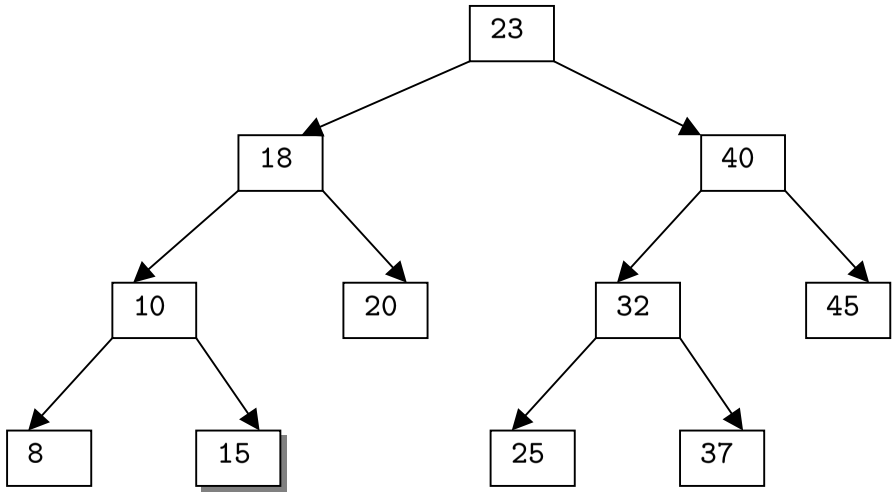


Just how bad is exponential complexity?

n	2^n
1	2
10	1024
20	1048576
50	1125899906842624
100	1267650600228229401496703205376

How many steps it would take (in the worst case) to to decide if '15' is in a list such as this: [23,18,40,10,20,8,15,32,25,37,45]

What if the list was sorted? `(or in a BINARY SEARCH TREE)`



- In the 1st step we remove m = n/2 possibilities from the search space.
- In the 2nd step we remove m/2 (i.e., (n/2)/2) possibilities.
- Etc...
- In n steps (worst case), we remove (...(((n/2)/2)/2)/2/2.../2) which is log(n)

- Very fast search. Of course, it assumes the tree is a binary search tree (i.e., the original list has been sorted!)

Complexity of (++)

```
x ++ y
Draw picture of computation
```

```
Therefore O(n)where n is length of
left hand argument
But O(k) with respect to right
 hand argument.
```

Complexity of Programs

```
p1 x = 2 * x
Plot assume constant time for mult.


Therefore O(k)
-----------------------------------
p2 x = (5 * 8) + (6 * x)
Plot



Therefore O(k)
```

Complexity of programs

```
p3 0 = 1
p3 n = n * p3 (n - 1)
Plot assume constant time for mult.
```

Therefore O(n)where n is input size

Complexity of linear recursion

```
p4 0 = 4 * 67 * 2
p4 n = 6 * (n * p4(n - 1))
Plot assume constant time for mult.
```

Therefore O(n)where n is input size

Complexity of (++)

```
x ++ y
Draw picture of computation
```

```
Therefore O(n)where n is length of
left hand argument
But O(k) with respect to right
 hand argument.
```

Complexity of reverse

```
rev [] = []
rev (x:xs) = rev xs ++ [x]
Plot - assume (++) depends on
         length of left argument
Draw "picture" of computation
```

```
Therefore O(n^2)
```

A complexity of composed programs

- `p . q . z`
  has complexity of whatever is the highest complexity of p, q, or z

- For example: `(++ [4]) . rev`

  Has complexity `O(n^2)`

- Note that the following has the same complexity

  `rev . (4:)`

Complexity Analysis (I)

BASIC RULES:

- N = the number of recursive calls needed to terminate a function:

  rec-func input = ...
  rec-func input = ... *rec-func input'*

  NOTE: must consider the rate at which the size of the input is decreasing (or, sometimes, increasing!)

- M = total amount of work inside each recursive call
- Complexity of is *rec-func* NxM

- The complexity of (*rec-func1* . *rec-func2*) is the maximum complexity of *rec-func1* and *rec-func2*

Complexity Analysis (Examples)

sq n = n * n	total number of calls = 1 cost inside every call = 1	O(1)
sumlist [ ] = 0 sumlist (e:es) = e + sumlist es	total number of calls = n cost inside every call = 1	O(n)
map f [ ] = [ ] map f (e:es) = (f e) : map f es	total number of calls = n cost inside every call = 1	O(n)
sumlist . map sq x	?	?
fact 0 = 1 fact n = n * fact (n-1)	total number of calls = n cost inside every call = 1	O(n)
factlist 0 = [1]	total number of calls = n	$O(n^2)$

factlist n = (fact n) : factlist (n-1)	cost inside every call = n	

Complexity Analysis (Examples)

(++) [ ] ys = ys (++) (x:xs) ys = x : (++) xs ys	total number of calls = n cost inside every call = 1	O(n) n = #xs
rev [ ] = [ ] rev (x:xs) = (rev xs) ++ [x]	total number of calls = n cost inside every call = n	$O(n^2)$
pc 1 = [1] pc n = 1 : pc (n/2)	total number of calls = $log\ n \begin{cases} n/2 \\ n/2/2 \\ n/2/2/2 1 \\ \dots \\ 1 \end{cases}$	$O(\log_2 n)$

	cost inside every call = 1	

Complexity Analysis (Sorting)

insert e [ ] = [e]
insert e (x:xs) = e : (x:xs), if e ≤ x
= x : (insert e xs), otherwise

sort [ ] = [ ]
sort (x:xs) = insert x (sort xs)

Complexity Analysis (Sorting)

```
qsort [] =  []
qsort xs =  qsort [a | a <- xs; a <= x]
            ++ [x] ++
            qsort [a | a <- xs; a > x]
```

$$\begin{aligned} \text{Best Case} &\Rightarrow O(n) \\ \text{Worst Case} &\Rightarrow O(n^2) \\ \text{Average} &\Rightarrow O(n\ log\ n) \end{aligned}$$

Complexity Analysis (Sorting)

```
msort xs = merge   (msrot (pick_fst_half (prod xs))
                   (msrot (pick_2nd_half (prod xs))

           where

           merge [] ys = ys
           merge xs [] = xs
           merge (x:xs) (y:ys) = x : merge xs (y:ys), x <= y
                               = y : merge (x:xs) ys, otherwise


           prod xs = [(a,b) | a <- xs; b <- [1..#xs]]

           pick_fst_half ps = [a | (a,n) <- ps; n <= #xs/2]
           pick_2nd_half ps = [a | (a,n) <- ps; n > #xs/2]
```

$$O(n \; log \; n)$$

Quadratic Complexity (Transitive Closure)

```
tc [] = []
tc (e:es) = (pair e es) ++ (tc es), #(pair e es) = #(e:es)
          = tc es, otherwise

where
```

$$pair\ (a,b)\ [\ ] = [\ ]$$

$$pair\ (a,b)\ ((x,y) : xs) = (a,b) : (x,y) : (a,y) : pair\ xs, (a = x)\ \&\ (not\ member\ (a,y)\ xs)$$
$$= (x,y) : pair\ (a,b)\ xs, otherwise$$

Exponential Complexity

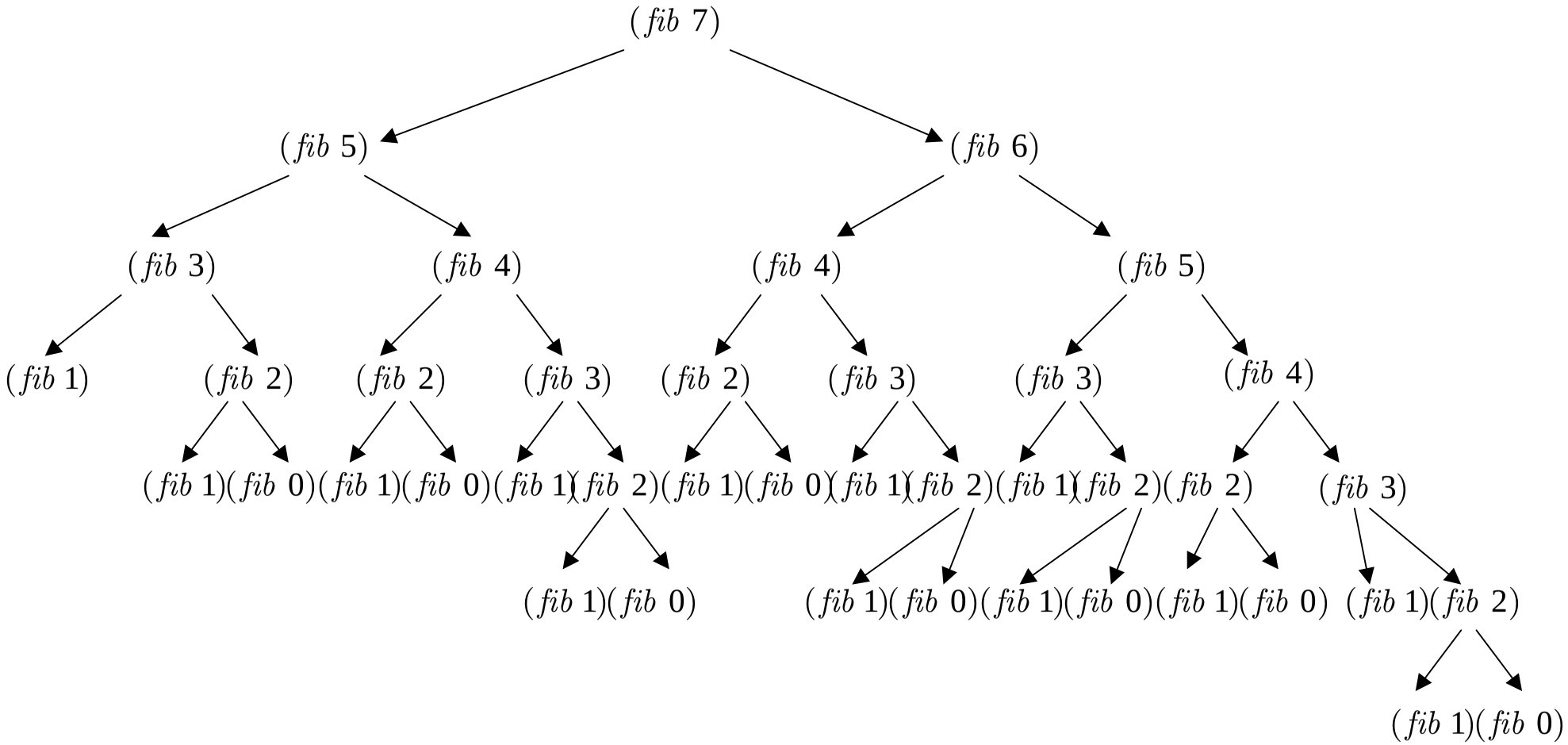
```
expand [] = [1]
expand (e:es) = expand es ++ expand es
```

```
expand [1,2,3,4] = expand [2,3,4] ++ expand [2,3,4]

= expand [3,4] ++ expand [3,4] ++ expand [3,4] ++ expand [3,4]

= expand [4] ++ expand [4] ++ expand [4] ++ expand [4] ++
  expand [4] ++ expand [4] ++ expand [4] ++ expand [4]

= expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++
  expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++
  expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++
  expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++
  expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++ expand [] ++
  expand [] ++ expand []

= [1] ++ [1] ++ [1] ++ [1] ++ [1] ... [1]
```

```
number of calls
#xs = 1 => 2(2)
#xs = 2 => 2(2(2)))
#xs = 3 => 2(2(2(2))))
...
#xs = n => 2(2(2(...))) = 2^(n+1)

cost inside: n
complexity: n * 2^(n+1) ~ 2^n
TEST IT on the above list!!!
n=4; total cost = 32 = 2^(4+1)
```

Complexity Analysis (Fibbonacci)

fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)



Reducing Time Complexity

$O(n^2)$

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

$O(n)$

```
reverse xs = reverse' xs []

reverse' [] c = c
reverse' (x:xs) = reverse xs (x:c)
```

Reducing Time Complexity

$O(n^2)$

```
factlist 0 = [1]
factlist n = (factorial n) : factlist (n-1)

                    where

                    factorial 0 = 1
                    factorial n = n * factorial (n-1)
```

Reducing Time Complexity

O(n)

```
factlist n = factlist2 n (factorial n)

                    where

                    factorial 0 = 1
                    factorial n = n * factorial (n-1)



factlist2 0 inp = [inp]
factlist2 n inp = inp : factlist (n-1) inp/n
```

Reducing Time Complexity

$O(n^2)$

```
binseq [] = 0
binseq (b:bs) = b*2^(len bs) + binseq es

                        where

                        len [] = 0
                        len (n:ns) = 1 + len ns
```

Reducing Time Complexity

O(n)

```
binseq seq = binseq2 seq (len seq)

                    where

                    len [] = 0
                    len (n:ns) = 1 + len ns


binseq2 [] inp = inp
binseq2 (b:bs) inp = b*2^(inp-1)+binseq es (inp-1)
```