Asymptotic notation

forget about constants and lower order terms

asymptotic efficiency of algorithms

$100n + 10000 = \Theta(n)$

$\frac{1}{1000} n^2 + 100 = \Theta(n^2)$

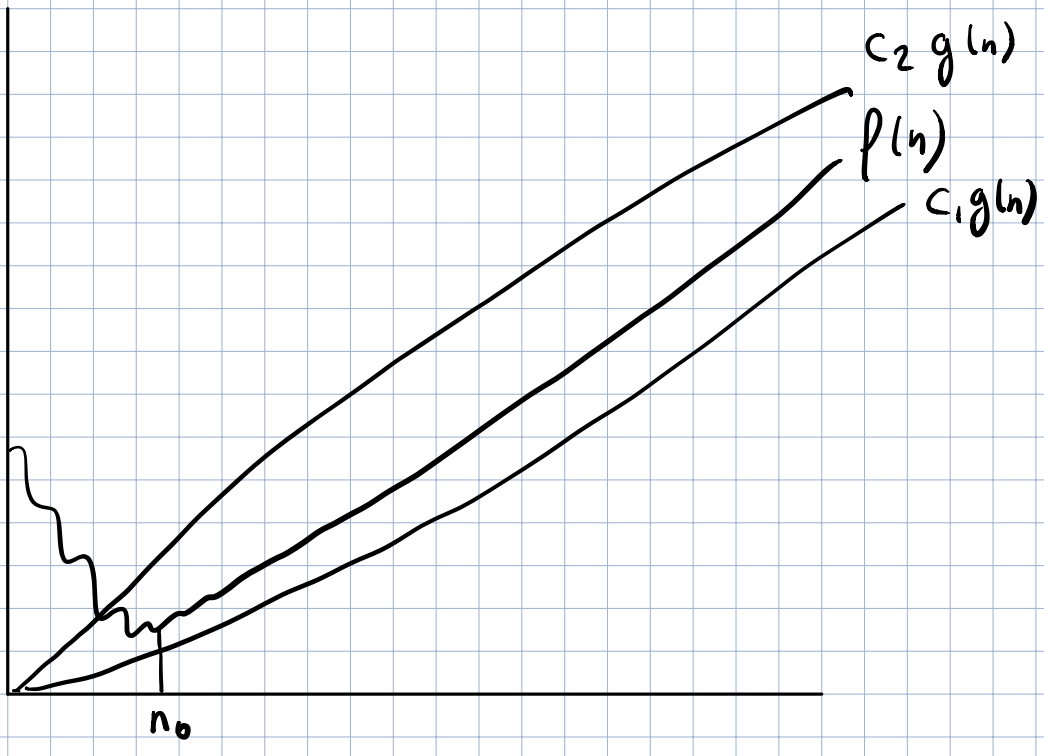
we consider functions $f(n)$ (or $g(n)$) which take positive integers n (i.e., input size)

Θ - notation

For a given function $g(n)$

$\Theta(g(n))$ denotes set of functions

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$



$\Theta(n^2)$
$\|$
$g(n)$

$100 n^2 \in \Theta(n^2)$

$c_2 = 101$

$c_1 = 99$

$\Theta(n) = \{$ set of polynomi of degree $n\}$

$f(n) \in \Theta(g(n))$

we say that $g(n)$ is an asymptotical tight bound for $f(n)$

$f(n) = \frac{1}{2} n^2 - 3n \in \Theta(n^2) = g(n)$

$\frac{1}{100} n^3 + n^2 \notin \Theta(n^2)$

$\Uparrow \Theta(n^3)$

$\Theta(n^2)$

The running time of an algorithm is $\Theta(g(n))$

① $\forall$ instance of size n (for large enough n) the algorithm take time $f(n) \in O(g(n))$

② $\exists$ at least an instance for which it take "at least" $g(n)$

IS $\Theta(n^2)$ time

① $\forall$ instance it take "at most" n^2

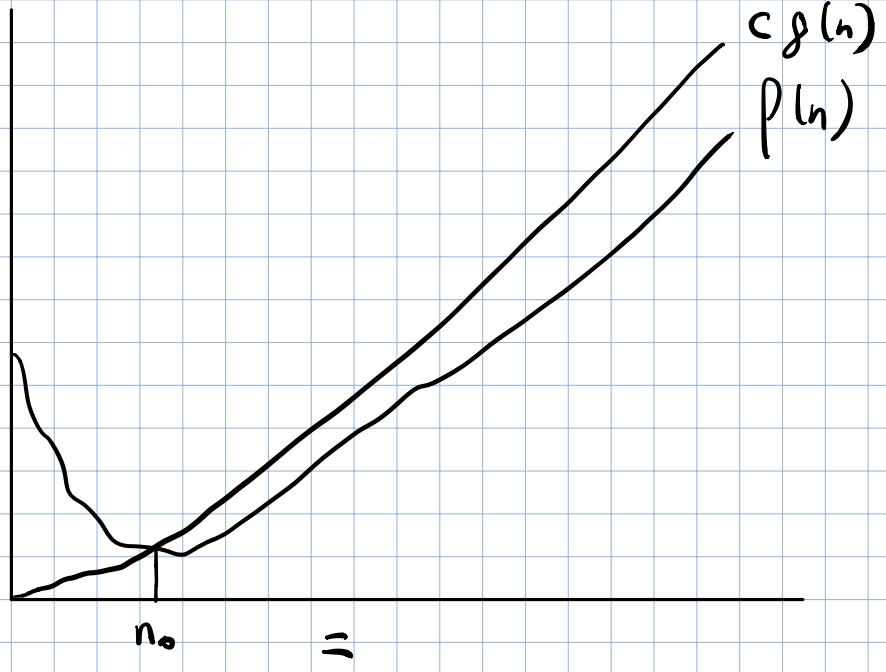
② $\exists$ a worst case instance which take n^2 time

O - notation Asymtotic upper bound

$O(g(n))$ pronounced big-Oh of g of n

$O(g(n)) = \{ p(n) :$ there exist positive constants c and n_0 such that

$0 \leq p(n) \leq cg(n)$ for all $n \geq n_0 \}$



$3n^2 + 5n \overset{b}{=} \in O(n^2)$

$5n + 3 \in O(n)$

~~$\in$~~ $\ni \in O(n^2$

$O(1) O(n^3)$

$p(n) \in \Theta(g(n)) \Rightarrow p(n) \in O(g(n))$

$\not\Leftarrow$

$O(1)$ $\quad$ is $n \in O(n^2) \not\Rightarrow$ is $n \in \Theta(n^2)$

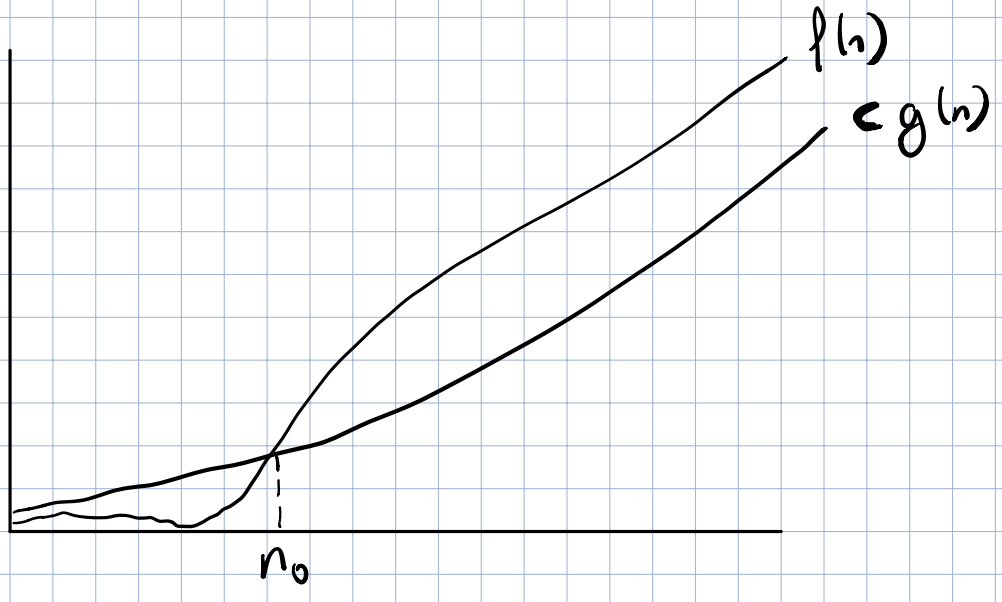
$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2)$

very fast $\qquad$ slow

$\Theta(g(n))$ time

Ω - notation Asymtotic lowerbounds

$\Omega(g(n)) = \{ f(n)$: there exist constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$



$2n^2 + 4n \in \Omega(n^2)$

$\notin \Omega(n)$

For any functions $g(n)$ and $f(n)$

we have $f(n) = \Theta(g(n))$

if and only if

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Exercise

Recall that MergeSort recurrence is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n == 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise} \end{cases}$$

and it solves to $\Theta(n \log n)$

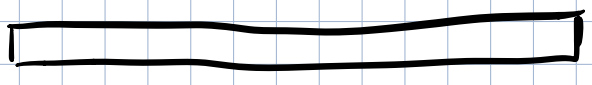
Find a recurrence that solves to $\Theta(n)$

Solutions

1

$$T(n) = \begin{cases} \Theta(1) & \text{if } n == 1 \\ T(1) + \Theta(n) \end{cases}$$

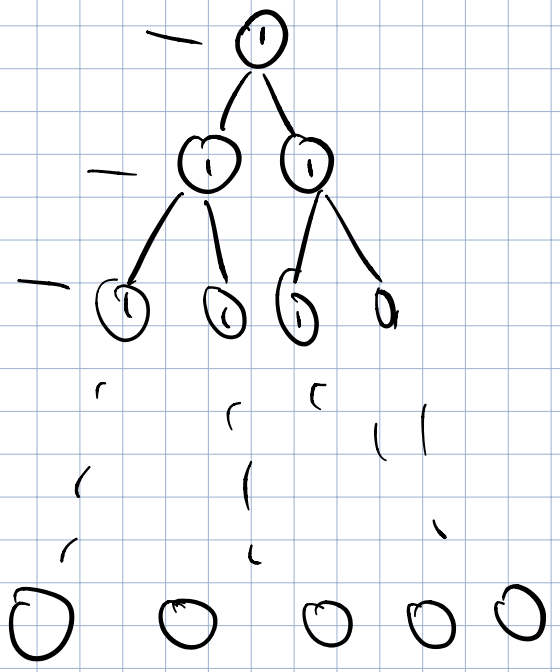
A

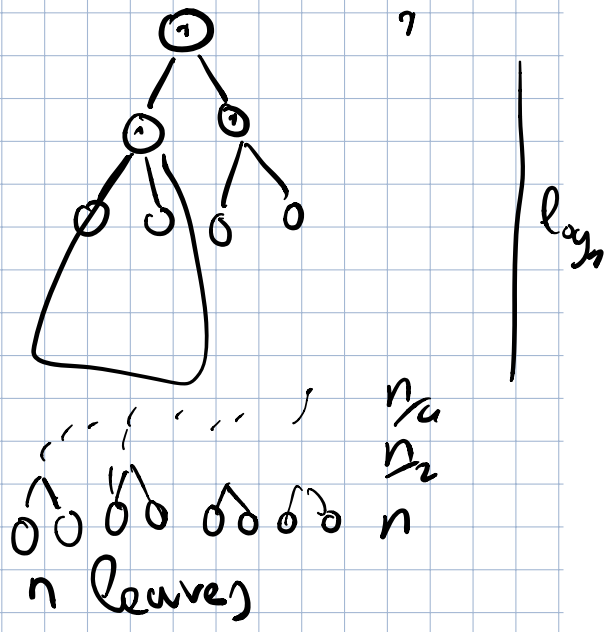


```
f(A, 1, n)
    if n == 1   return A[1]
     M  = A[1]
    for i = 1 to n-1
        if M < A[i]
            M = A[i]
    return max(M, f(A, n, n))
```

$$T(n) = 2T(\tfrac{n}{2}) + \Theta(1)$$

(2)





$$n + \frac{n}{2} + \frac{n}{4} + \dots + 4 + 2 + 1$$

$$\simeq 2n$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(1) \end{cases}$$

$P(A, p, r)$

if $p < r$

$q = \lceil \frac{p+r}{2} \rceil$ // $\Theta(1)$

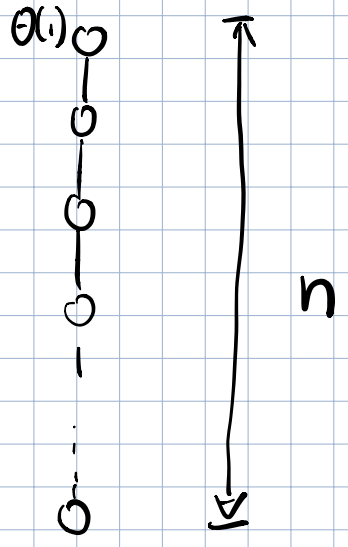
$M_1 = P(A, p, q)$ // $2T(\frac{n}{2})$

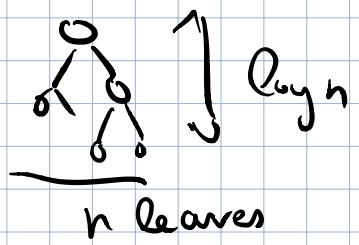
$M_2 = P(A, q+1, r)$ //

return $\max(M_1, M_2)$ // $\Theta(1)$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$$

3






```
fact(n)
    if n == 1 return 1
    return fact(n-1)*n
```

$$T(n) = \begin{cases} \Theta(1) & \text{if } n == 1 \\ T(n-1) + \Theta(1) & \text{otherwise} \end{cases}$$

nine sub problem

just one child

```
f(A, p, r)
    if p == r return A[p]
    return max(A[p], f(A, p+1, r))
```

1	2	3	4
1	10	5	20

max(1, f(A, 2, 4))

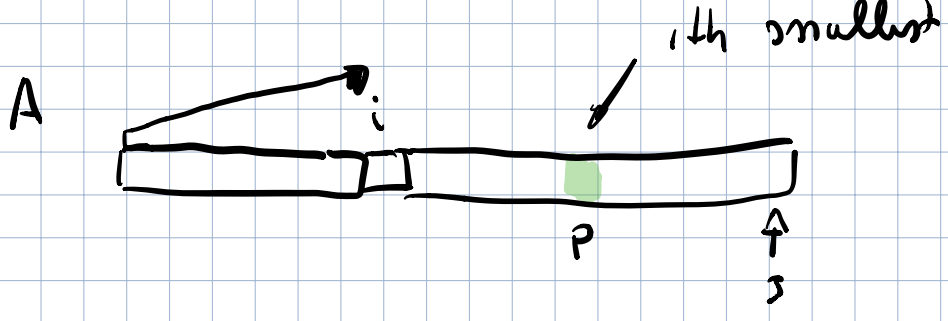
Exercise

BubbleSort(A)

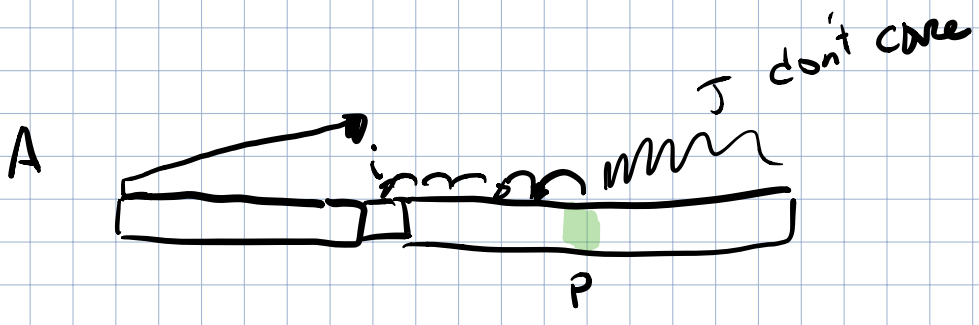
```
for i = 1 to A.length
    for j = A.length downto i+1
        if A[j] < A[j-1]
            swap (A[j-1], A[j])
```

LOOP-INVARIANT:

At ith iteration, prefix $A[1 \ldots i-1]$ contains $(i-1)$ smallest elements of A sorted



we want to prove that ith smallest element will be placed in position A[i]



what's the complexity of Bubble Sort?

```
BubbleSort(A)
    for i = 1 to A.length                                   | Θ(n-i)
        for j = A.length downto i+1                         | Θ(n-i)
            if A[j] < A[j-1]                                | Θ(1)
                swap (A[j-1], A[j])                         | c(n-i)
```

$$T(n) = \sum_{i=1}^{n} c(n-i) = \Theta(n^2)$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} c$$

$$\| \quad c(n-i)$$

```
for i = 1 to A.length
    for j = A.length downto i+1
        if A[j] < A[j-1]
            swap (A[j-1], A[j])
```

$c(n-1)$

$$T(n) = n \cdot c(n-1) = \Theta(n^2)$$

Exercise

Searching a key in a collection

Problem

Input: Array A of n integers, a key k

Output: $\begin{cases} i & \text{such that } A[i] == k \\ -1 & \text{if no } i \text{ exists} \end{cases}$

① Provide pseudo code of LinearSearch and analyze it

```
LinearSearch ( A, k)
    For i = 1 to A.length
        if A[i] == k                    | Θ(1)
            return i

    return -1
```

Analysis

Best case: $A[1] = k$ $\Theta(1)$

Worst case: k is not in A $\Theta(n)$

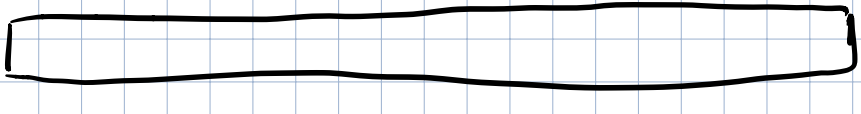
Can you prove that linear time is the best you can hope for? (without any assumption on A)

There is an algorithm which (always) takes at most $\ell < n$ steps.

Prove this is not possible

# of entries of A the algo checks is $\leq \ell < n$

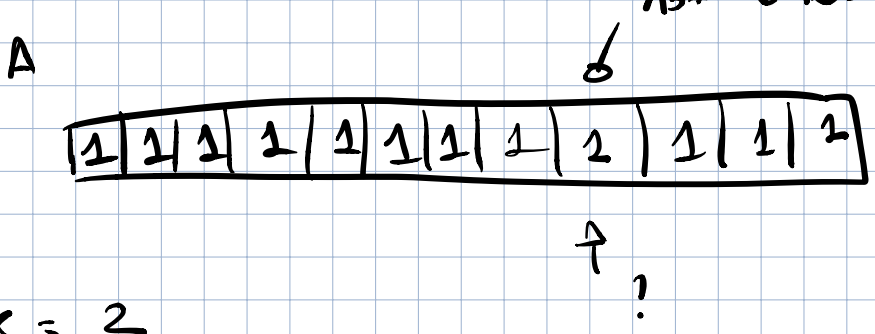
A



If $\ell < n$, I can make it wrong (at least once)

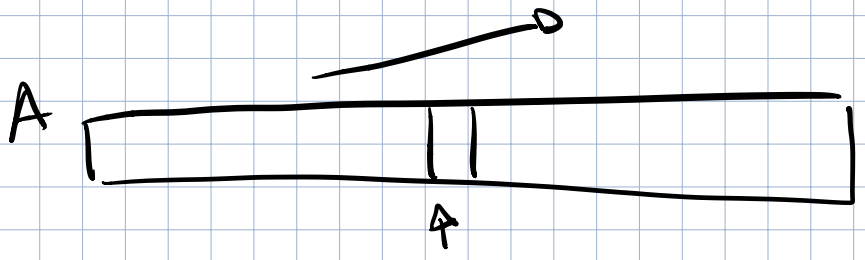
not checked

$l < n$



$K = 2$

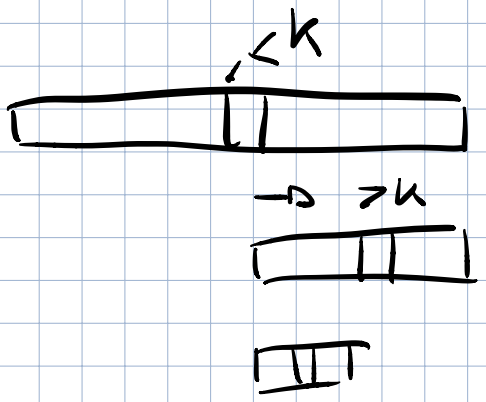
① Sort A and use Dirde & Conquer Approach to search for K



Dirde: splHing the array in two parts of size $\frac{n}{2}$

Conquer: recursively search k in the part that may contain K

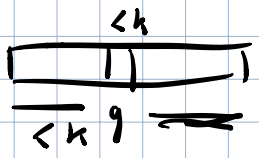
Combine Just report the result



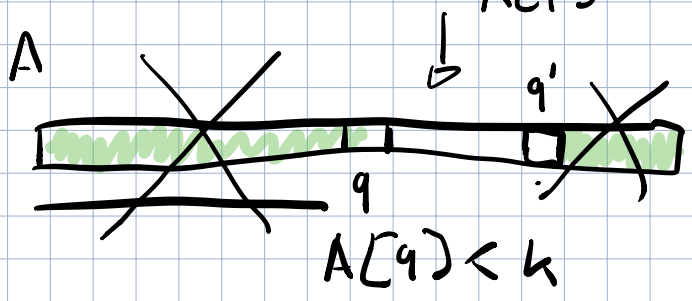
① Write pseudo code

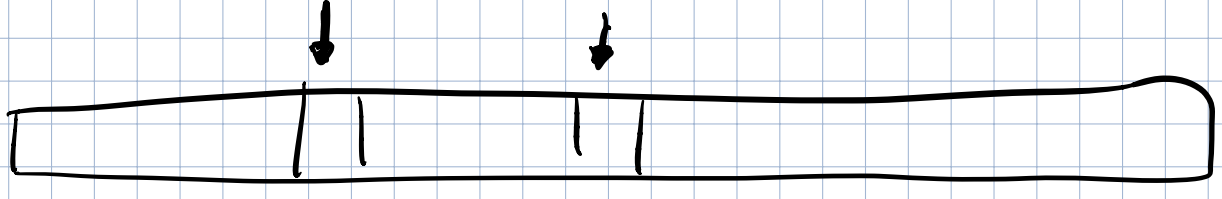
BinarySearch (A, p, r, k) — p, r: first and last element in the sub array; k: key we are looking for

```
if  p > r      return -1
if  p == r
        if A[p] == k    return p
        else            return -1
q = ⌈(p+r)/2⌉   // divide
if  A[q] == k   return q   // just to get better best case
if  A[q] < k
        return  BinarySearch(A, q+1, r, k)
else
        return  BinarySearch(A, p, q-1, k)
```



A[q'] > k





Running examples by yourself

	1	2	3	4	5	6	7	8
A	14	43	76	100	115	290	500	511

$k = 76$

$k = 80$

p	r	q
1	8	

(3) Find and solve BS recurrence

$$T(n) \begin{cases} \Theta(1) & \text{if } n = 1 \\ \underbrace{T\left(\frac{n}{2}\right)}_{a} + \Theta(1) \end{cases}$$

a = 2

b = 2

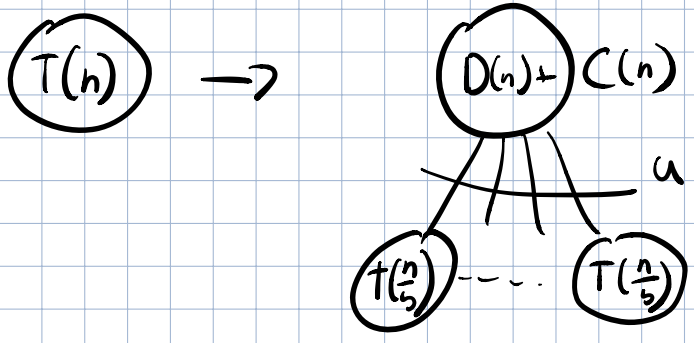
$$a\,T\left(\frac{n}{b}\right) + D(n) + C(n)$$

- a: # of subproblems
- $\frac{n}{b}$: size of a subproblem
- $D(n)$: time to divide step
- $C(n)$: time to combine step

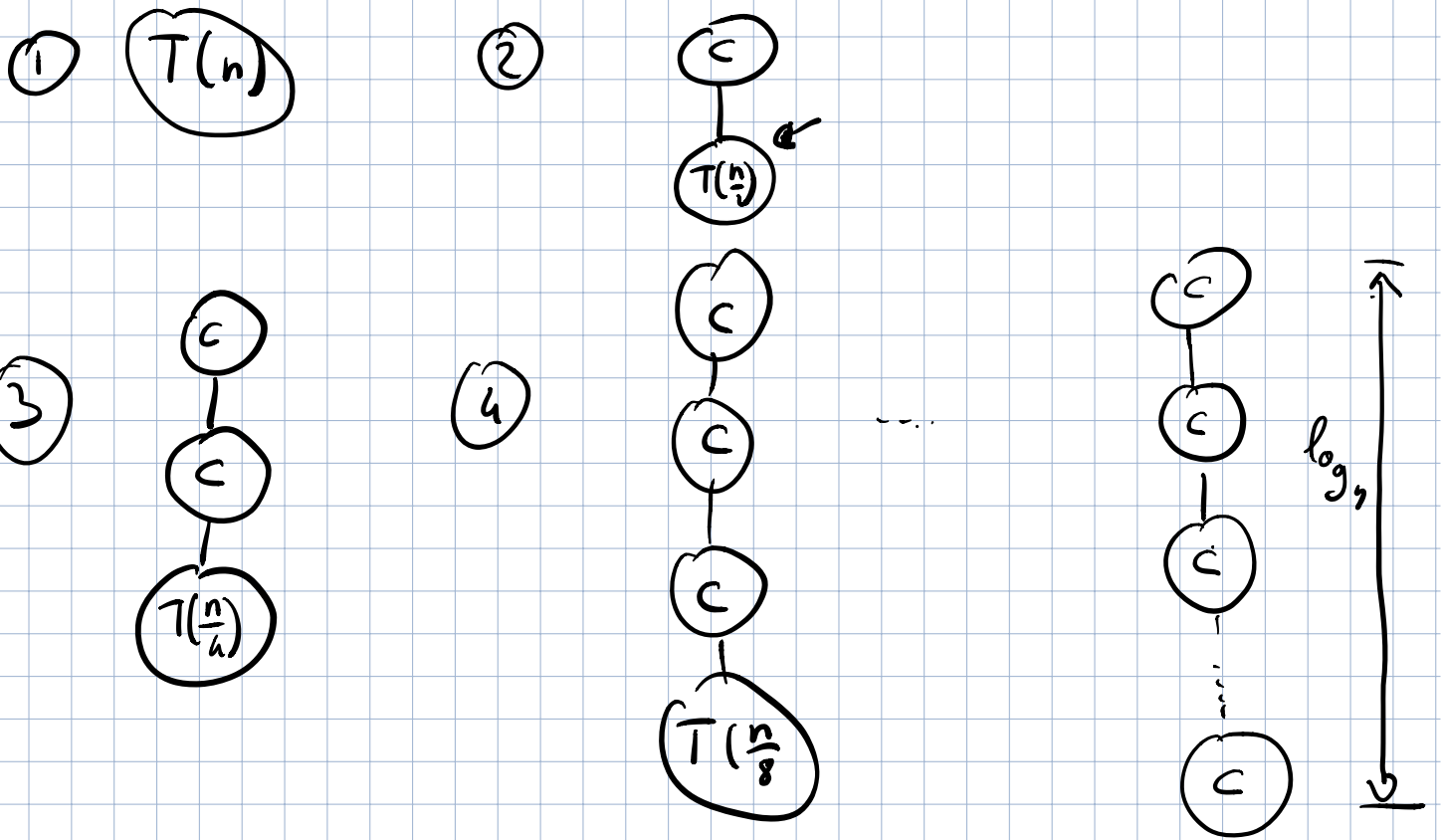
$$T(n) = a\,T\left(\frac{n}{b}\right) + D(n) + C(n)$$

- a: # of children
- $T\left(\frac{n}{b}\right)$: label of each child
- $D(n) + C(n)$: cost in the node

Expand a node



$$T(n) = T\left(\frac{n}{2}\right) + c$$



$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$ vs $T(n) = T(n-1) + \Theta(1)$

$\Theta(\log n)$ $\Theta(n)$

$n = 4$ billion $= 2^{32}$

$\log n = 32$

$n = 2^{32}$

Exercise

$f(n) = 3n^2 \overset{?}{\in}$

- $O(n^2)$ ✓
- $O(n^4)$ ✓
- $O(n!)$ ✓
- $O(2^n)$ ✓
- $O(n \log n)$ ✗
- $O(n^2 \log n)$ ✓
- $\Theta(n^2)$ ✓
- $\Theta(n^2 \log n)$ ✗
- $\Omega(n^2)$ ✓
- $\Omega(n)$ ✓
- $\Omega(n^3)$ ✗