

COL 351 : ANALYSIS & DESIGN OF ALGORITHMS

LECTURE 3

DIVIDE & CONQUER II :

MERGE SORT (CONTD.) AND ASYMPTOTIC ANALYSIS

JULY 29, 2026

|

ROHIT VAISH

INTEGER MULTIPLICATION

input: two n digit numbers x and y

output: the product $x \cdot y$

INTEGER MULTIPLICATION

input: two n digit numbers x and y

output: the product $x \cdot y$

Grade-school multiplication

$\leq 9n^2$ basic operations

INTEGER MULTIPLICATION

input: two n digit numbers x and y

output: the product $x \cdot y$

Grade-school multiplication

$\leq 9n^2$ basic operations

Recursive algorithm (4 calls)

?

karatsuba algorithm (3 calls)

?

MERGE SORT

input: array A of n distinct integers

output: array with the same numbers sorted in increasing order

MERGE SORT

input: array A of n distinct integers

output: array with the same numbers sorted in increasing order

if $n \leq 1$

return A

else

$L :=$ left half of A sorted recursively

$R :=$ right " " " "

return $\text{MERGE}(L, R)$

MERGE SORT

input: array A of n distinct integers

output: array with the same numbers sorted in increasing order

if $n \leq 1$

return A

else

$L :=$ left half of A sorted recursively

$R :=$ right " " " "

return **MERGE**(L, R)

MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

assume n is even

MERGE SUBROUTINE

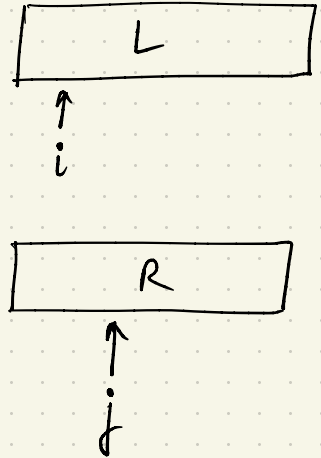
input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

initialize $i := 1$

initialize $j := 1$

assume n is even



MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

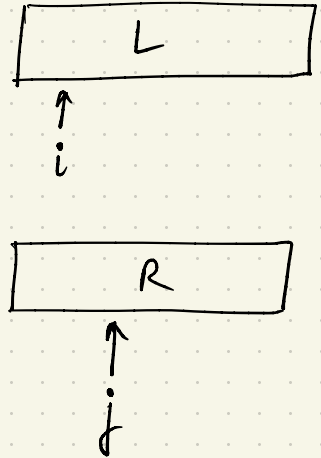
output: sorted array B of length n

initialize $i := 1$

initialize $j := 1$

for $k := 1$ to n

assume n is even



MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

assume n is even

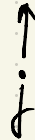
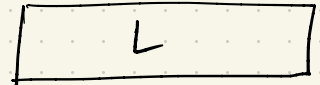
initialize $i := 1$

initialize $j := 1$

for $k := 1$ to n

if $L[i] < R[j]$ then

else



MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

assume n is even

initialize $i := 1$

initialize $j := 1$

for $k := 1$ to n

if $L[i] < R[j]$ then

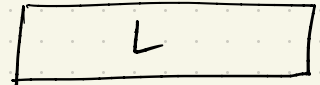
$B[k] := L[i]$

increment i by 1

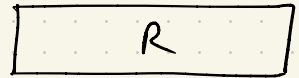
else

$B[k] := R[j]$

increment j by 1



$\uparrow$
 i



$\uparrow$
 j

MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

assume n is even

initialize $i := 1$

initialize $j := 1$

for $k := 1$ to n

if $L[i] < R[j]$ then

$B[k] := L[i]$

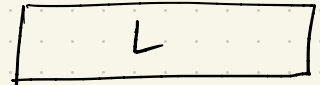
increment i by 1

else

$B[k] := R[j]$

increment j by 1

ignoring end cases
when i or j falls off



MERGE SORT RUNNING TIME

What is the basic operation?

MERGE SORT RUNNING TIME

What is the **basic operation**?

- commonly, any pairwise comparison
- for now, a line-by-line implementation using a debugger

MERGE SORT RUNNING TIME

What is the **basic operation**?

- commonly, any pairwise comparison
- for now, a line-by-line implementation using a debugger

Let's start by analyzing MERGE subroutine.

MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

assume n is even

initialize $i := 1$

initialize $j := 1$

for $k := 1$ to n

if $L[i] < R[j]$ then

$B[k] := L[i]$

increment i by 1

else

$B[k] := R[j]$

increment j by 1

MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

assume n is even

initialize $i := 1$
initialize $j := 1$ } 2 operations

for $k := 1$ to n

if $L[i] < R[j]$ then

$B[k] := L[i]$

increment i by 1

else

$B[k] := R[j]$

increment j by 1

MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

assume n is even

initialize $i := 1$

initialize $j := 1$

} 2 operations

for $k := 1$ to n

executes n times

if $L[i] < R[j]$ then

$B[k] := L[i]$

increment i by 1

else

$B[k] := R[j]$

increment j by 1

MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

assume n is even

initialize $i := 1$

initialize $j := 1$

} 2 operations

for $k := 1$ to n

executes n times

if $L[i] < R[j]$ then

$B[k] := L[i]$

increment i by 1

} ?? operations

else

$B[k] := R[j]$

increment j by 1

MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

assume n is even

initialize $i := 1$

initialize $j := 1$

} 2 operations

for $k := 1$ to n

executes n times

if $L[i] < R[j]$ then

$B[k] := L[i]$

increment i by 1

} $3 + 1 = 4$ operations
└ incrementing k

else

$B[k] := R[j]$

increment j by 1

MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

assume n is even

initialize $i := 1$

initialize $j := 1$

} 2 operations

for $k := 1$ to n

executes n times

if $L[i] < R[j]$ then

$B[k] := L[i]$

increment i by 1

} $3 + 1 = 4$ operations
└ incrementing k

else

$B[k] := R[j]$

increment j by 1

} 4 operations

MERGE SUBROUTINE

input: sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

initialize $i := 1$

initialize $j := 1$

} 2 operations

for $k := 1$ to n

executes n times

if $L[i] < R[j]$ then

$B[k] := L[i]$

increment i by 1

else

$B[k] := R[j]$

increment j by 1

assume n is even

$4n + 2$ operations

$3 + 1 = 4$ operations
└ incrementing k

4 operations

MERGE SUBROUTINE

Lemma: Running time of MERGE subroutine on input arrays of length $n/2$ each is $\leq 4n + 2$.

MERGE SUBROUTINE

Lemma: Running time of MERGE subroutine on input arrays of length $n/2$ each is $\leq 4n + 2$.

$$\leq 6n \quad \text{since } n \geq 1$$

MERGE SUBROUTINE

Lemma: Running time of MERGE subroutine on input arrays of length $n/2$ each is $\leq 4n + 2$.

$\leq 6n$ since $n \geq 1$

MERGE SUBROUTINE

Lemma: Running time of MERGE subroutine on input arrays of length $l/2$ each is $\leq 4l + 2$.

$$\leq 6l \quad \text{since } l \geq 1$$

MERGE SUBROUTINE

Lemma: Running time of MERGE subroutine on input arrays of length $l/2$ each is $\leq 4l + 2$.

$$\leq 6l \quad \text{since } l \geq 1$$

(So we can plug in other values of l later)

MERGE SORT RUNNING TIME

MERGE SORT RUNNING TIME

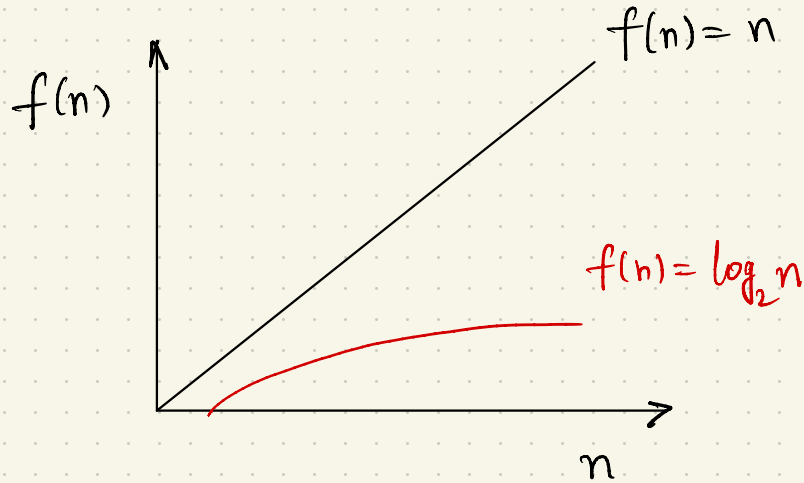


MERGE SORT RUNNING TIME

Theorem: For every input array of length $n \geq 1$, Merge Sort performs at most $6n \log_2 n + 6n$ operations.

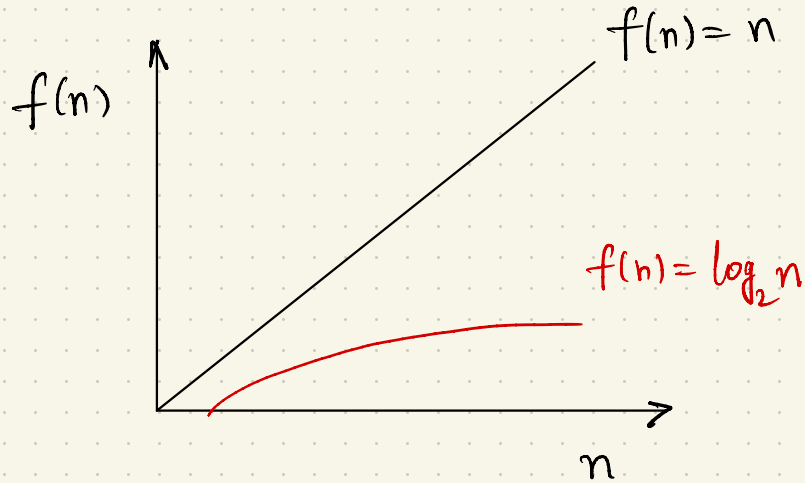
MERGE SORT RUNNING TIME

Theorem: For every input array of length $n \geq 1$, Merge Sort performs at most $6n \log_2 n + 6n$ operations.



MERGE SORT RUNNING TIME

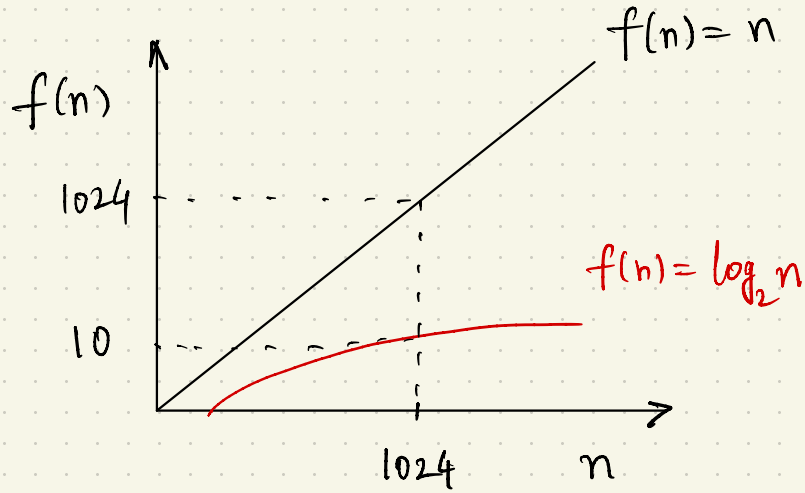
Theorem: For every input array of length $n \geq 1$, Merge Sort performs at most $6n \log_2 n + 6n$ operations.



$\log_2$: how many "divide by 2" operations are needed to bring the answer to ≤ 1 .

MERGE SORT RUNNING TIME

Theorem: For every input array of length $n \geq 1$, Merge Sort performs at most $6n \log_2 n + 6n$ operations.



$\log_2$: how many "divide by 2" operations are needed to bring the answer to ≤ 1 .

MERGE SORT RUNNING TIME

Theorem: For every input array of length $n \geq 1$, Merge Sort performs at most $6n \log_2 n + 6n$ operations.

Proof via recursion tree

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)

root
●

level 0 outmost call

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)

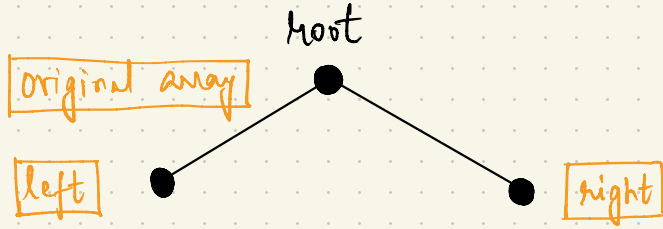
original array

root
●

level 0 outermost call

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)

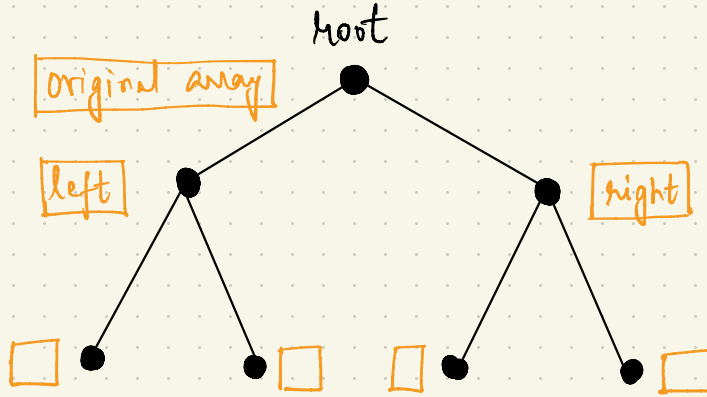


level 0 outermost call

level 1 first recursive calls

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



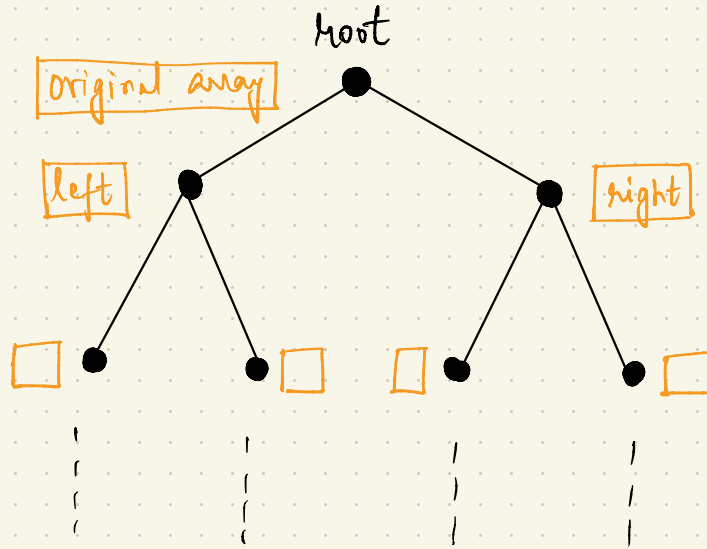
level 0 outmost call

level 1 first recursive calls

level 2

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



level 0 outermost call

level 1 first recursive calls

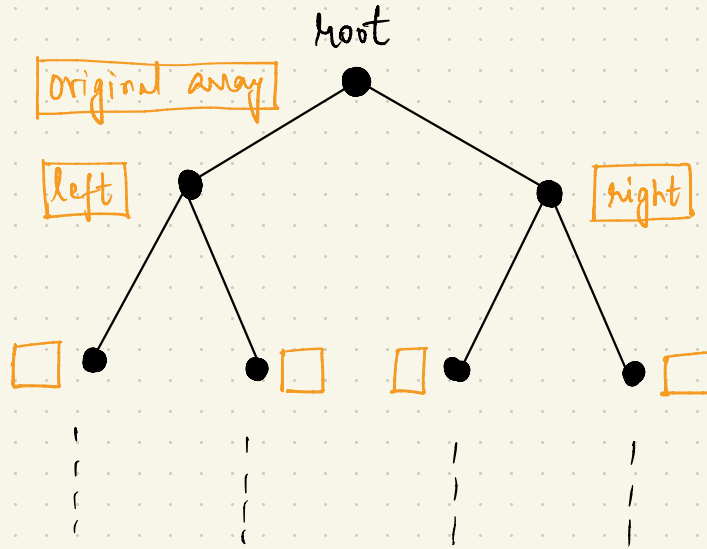
level 2

.....

leaves (single-element arrays)

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



.....
leaves (single-element arrays)

level 0 outermost call

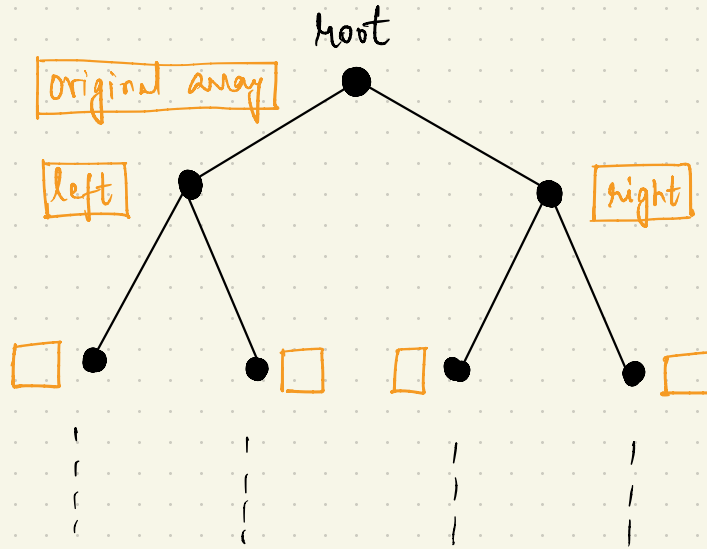
level 1 first recursive calls

level 2

level ??

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



.....

leaves (single-element arrays)

level 0 outermost call

level 1 first recursive calls

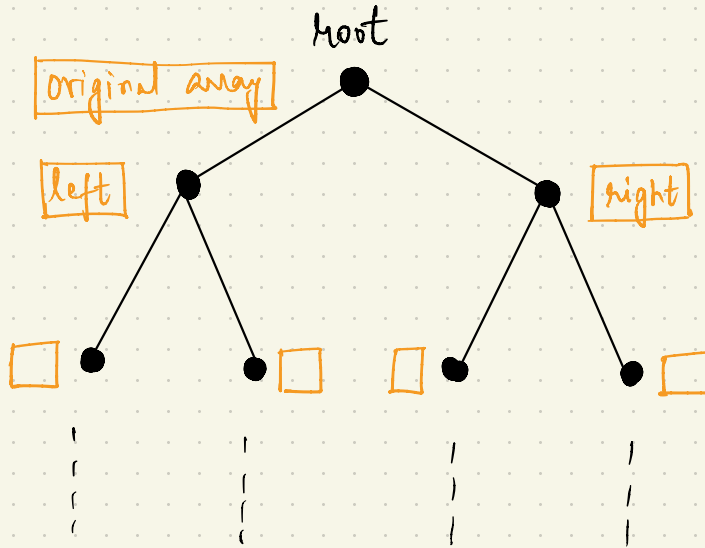
level 2

level $\log_2 n$

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)

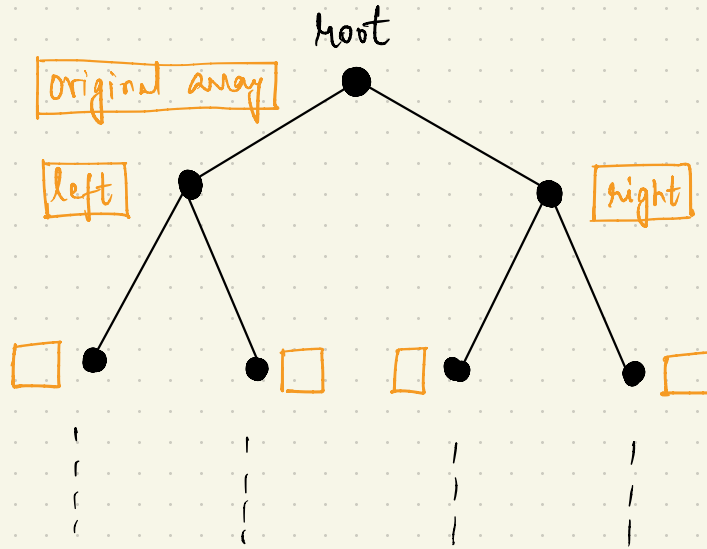
$\log_2 n + 1$ levels in total



.....
leaves (single-element arrays)

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



.....
leaves (single-element arrays)

$\log_2 n + 1$ levels in total

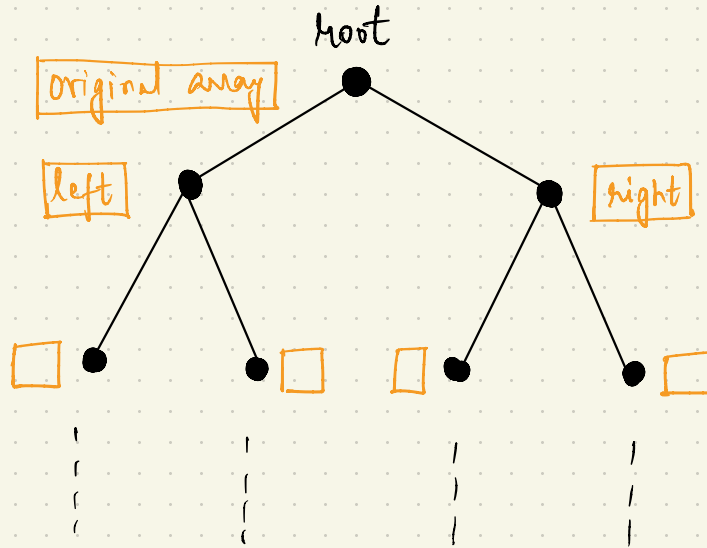
At level $j \in \{0, 1, \dots, \log_2 n\}$

?? subproblems

each of size ??

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



.....
leaves (single-element arrays)

$\log_2 n + 1$ levels in total

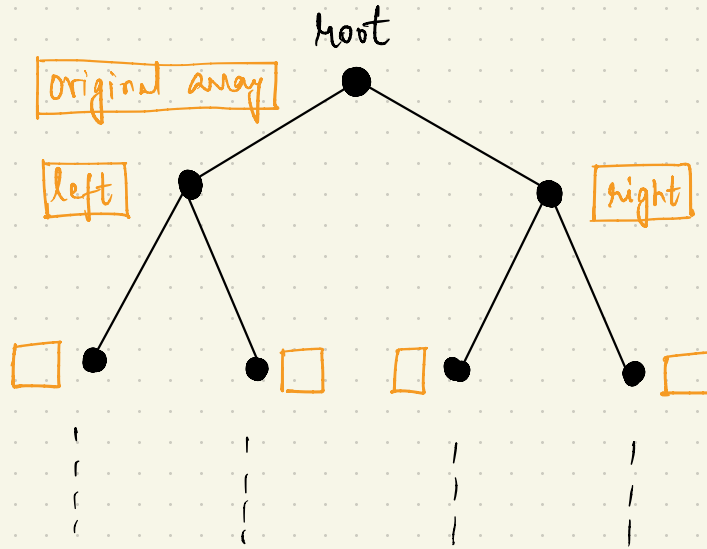
At level $j \in \{0, 1, \dots, \log_2 n\}$

2^j subproblems

each of size $n/2^j$

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



How much work does Merge Sort do at level j ?

.....
leaves (single-element arrays)

MERGE SORT : PSEUDOCODE

if $n \leq 1$

return A

else

$L :=$ left half of A sorted recursively

$R :=$ right " " "

return MERGE(L, R)

MERGE SORT : PSEUDOCODE

if $n \leq 1$

return A

else

$L :=$ left half of A sorted recursively

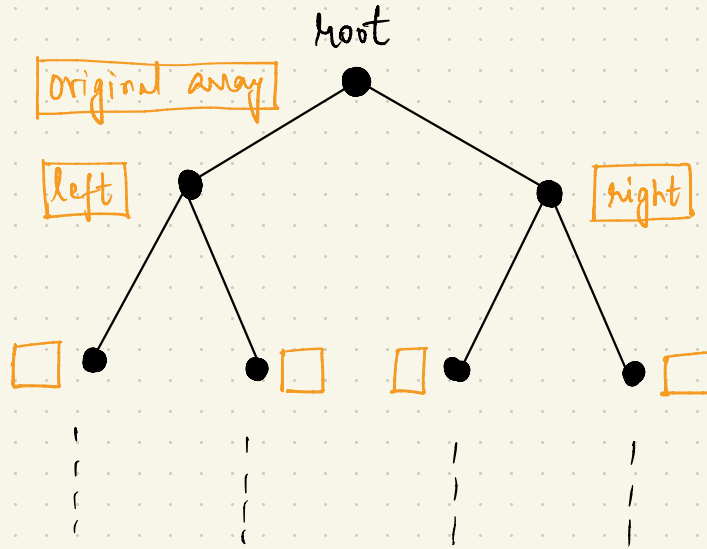
$R :=$ right " " "

return MERGE(L, R)

↖ only need to count this

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)

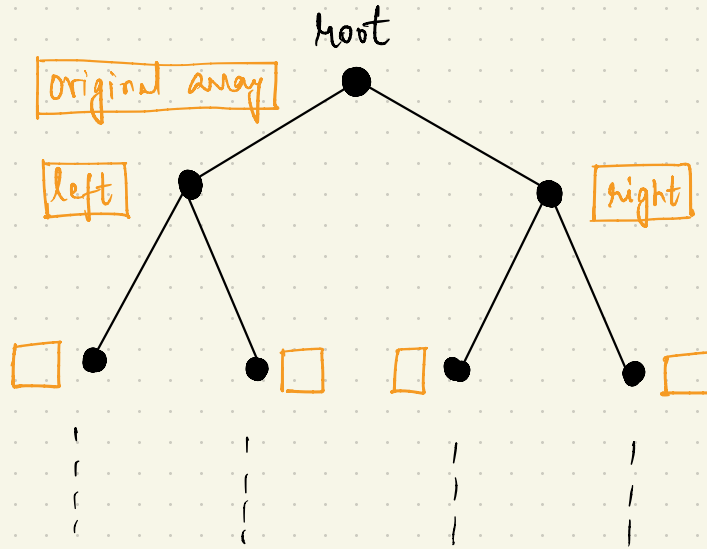


How much work does Merge Sort do at level j ?

.....
leaves (single-element arrays)

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



.....
leaves (single-element arrays)

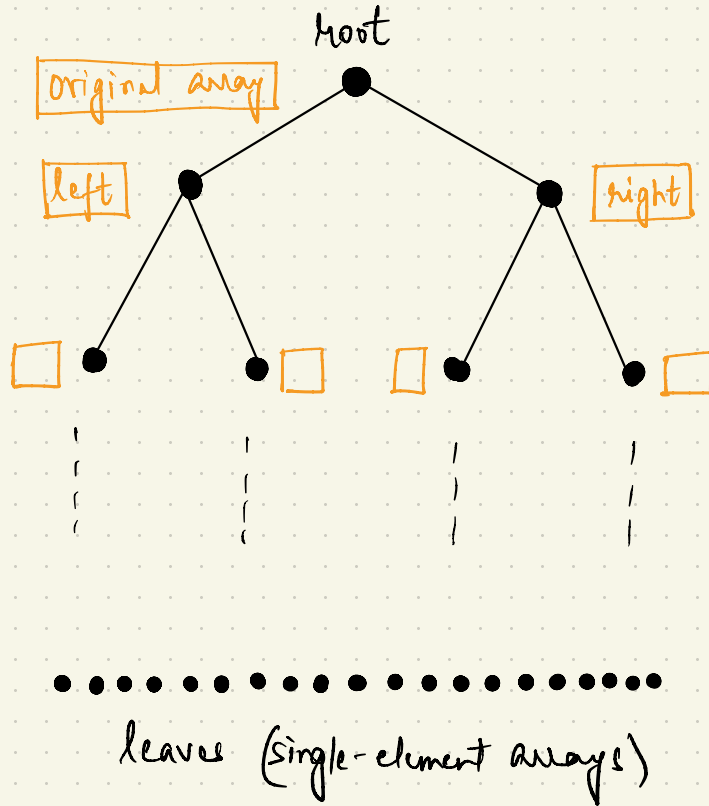
How much work does Merge Sort do at level j ?

=

Work done by merge at level j

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



How much work does Merge Sort do at level j ?

=

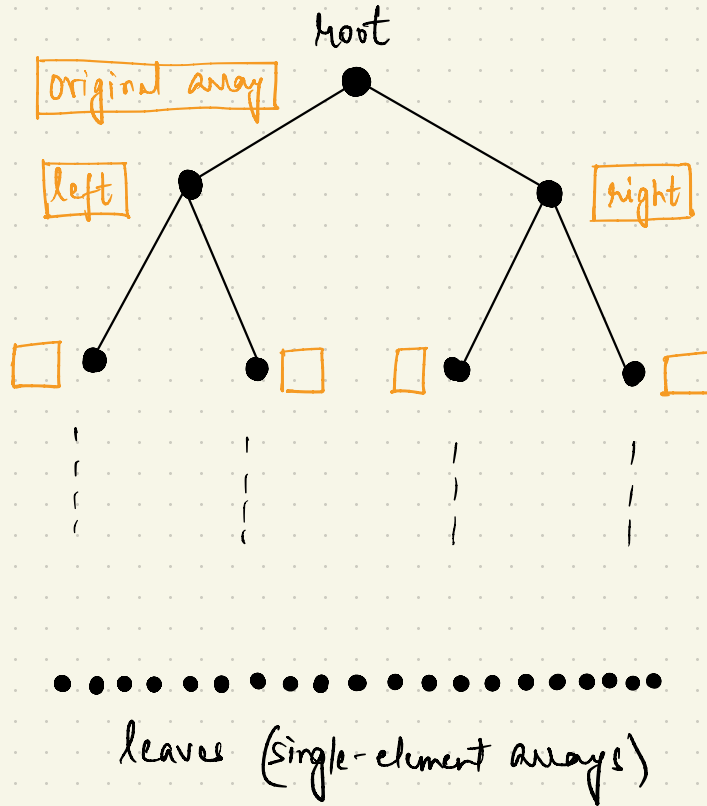
Work done by merge at level j

$$\leq 2^j$$

subproblems

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



How much work does Merge Sort do at level j ?

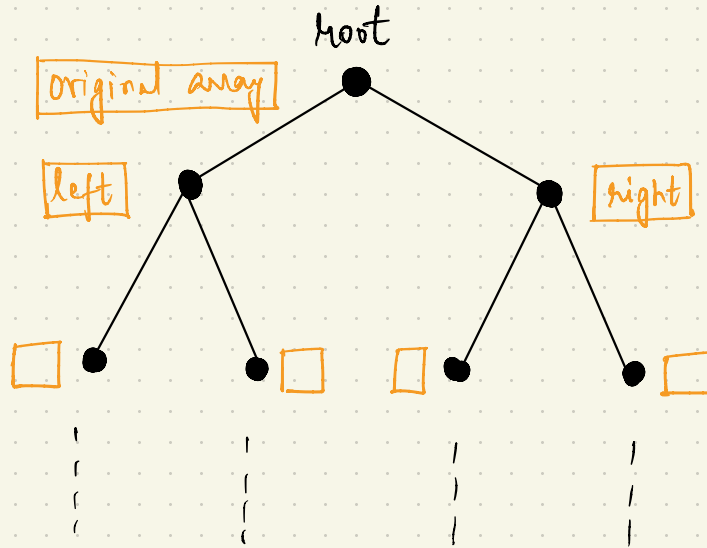
=

Work done by merge at level j

$$\leq \underbrace{2^j}_{\text{\# subproblems}} \times \underbrace{6 \left(\frac{n}{2^j} \right)}_{\text{size } l = n/2^j \text{ (from Lemma)}}$$

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



.....
leaves (single-element arrays)

How much work does Merge Sort do at level j ?

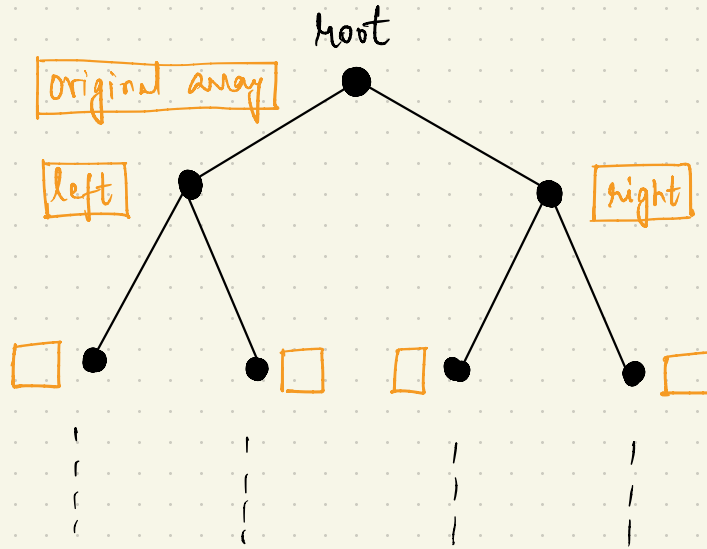
=

Work done by merge at level j

$$\leq \cancel{2^j} \times 6 \left(\cancel{2^j} \right)^{n/j}$$

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



.....
leaves (single-element arrays)

How much work does Merge Sort do at level j ?

=

Work done by merge at level j

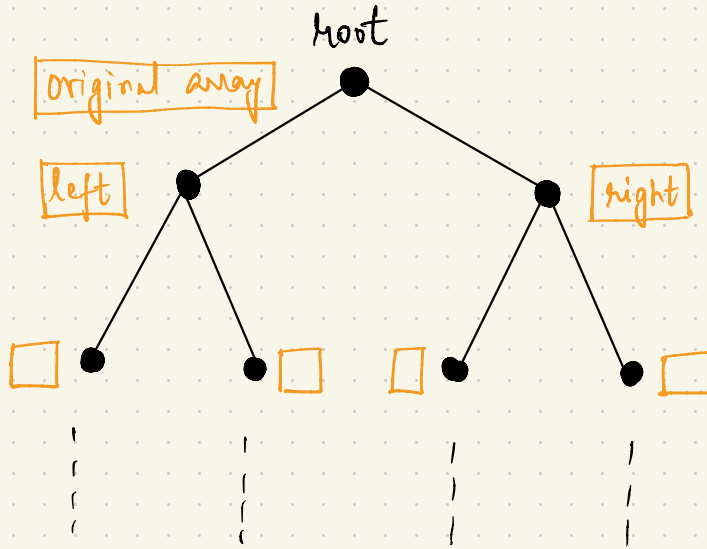
$$\leq \cancel{2^j} \times 6 \left(\frac{n}{\cancel{2^j}} \right)$$

$$= 6n$$

independent of j

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



.....
leaves (single-element arrays)

How much work does Merge Sort do at level j ?

=

Work done by merge at level j

$$\leq \cancel{2^j} \times 6 \left(\cancel{2^j} \right)^{n/j}$$

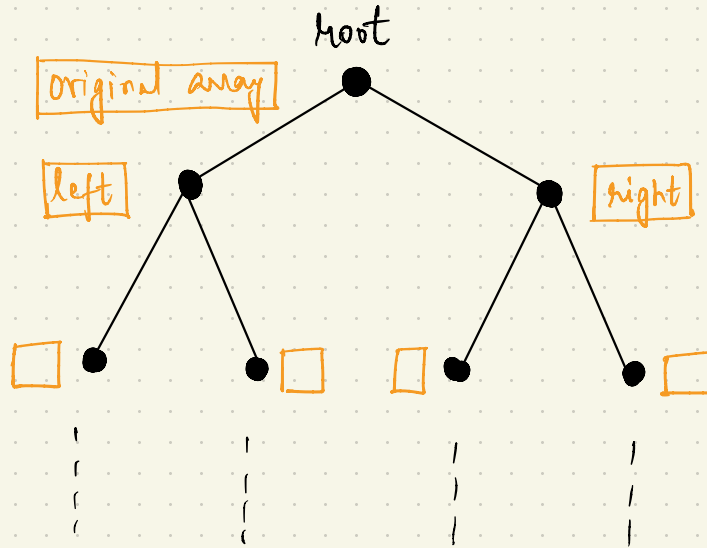
$$= 6n$$

subproblems $\uparrow$ by $2\times$

Size/subproblem $\downarrow$ by $2\times$

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



.....
leaves (single-element arrays)

How much work does Merge Sort do at level j ?

=

Work done by merge at level j

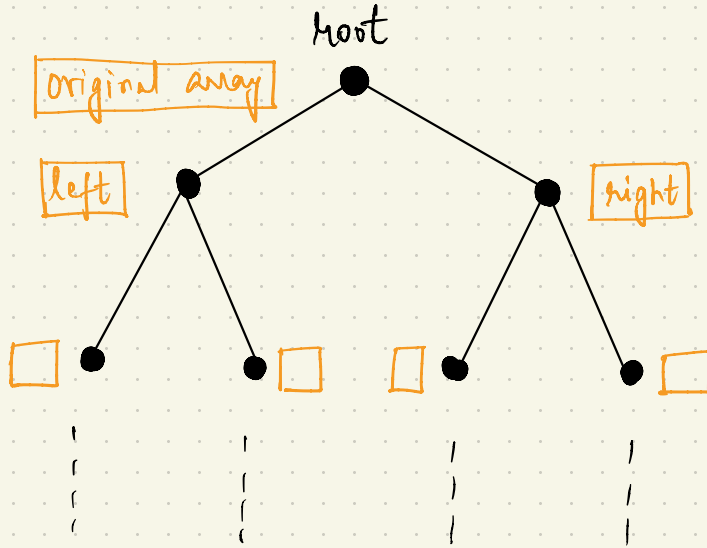
$$\leq \cancel{2^j} \times 6 \left(\frac{n}{\cancel{2^j}} \right)$$

$$= 6n$$

Total work done =

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



.....
leaves (single-element arrays)

How much work does Merge Sort do at level j ?

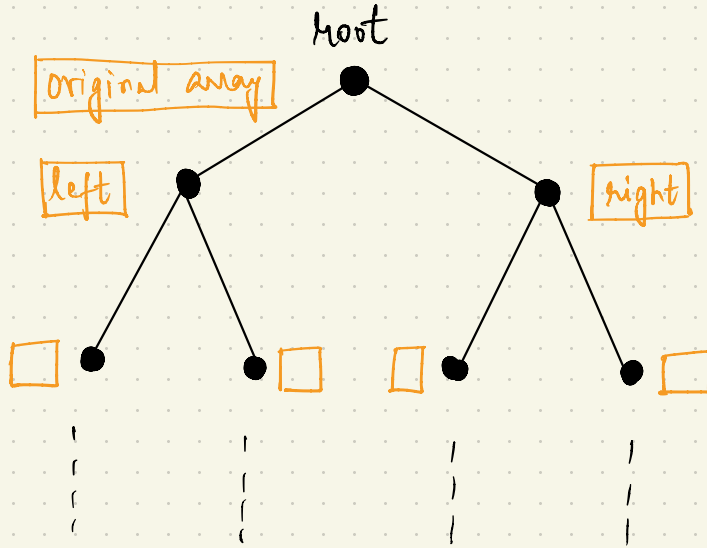
=
Work done by merge
at level j

$$\leq \cancel{2^j} \times 6 \left(\cancel{2^j} \right) \\ = 6n$$

$$\text{Total work done} = 6n \cdot \underbrace{(\log_2 n + 1)}_{\# \text{ levels}}$$

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



.....
leaves (single-element arrays)

How much work does Merge Sort do at level j ?

=
Work done by merge
at level j

$$\leq \cancel{2^j} \times 6 \left(\frac{n}{\cancel{2^j}} \right)$$
$$= 6n$$

$$\text{Total work done} = 6n \cdot (\log_2 n + 1)$$



MAKING SOME ASSUMPTIONS EXPLICIT

MAKING SOME ASSUMPTIONS EXPLICIT

Worst-case (or adversarial) analysis

MAKING SOME ASSUMPTIONS EXPLICIT

Worst-case (or adversarial) analysis

- " $6n \log_2 n + 6n$ " bound holds for every single input of size n
- no assumption on where the input comes from

MAKING SOME ASSUMPTIONS EXPLICIT

Worst-case (or adversarial) analysis

- " $6n \log_2 n + 6n$ " bound holds for every single input of size n
- no assumption on where the input comes from

Not too worried about precise constants

MAKING SOME ASSUMPTIONS EXPLICIT

Worst-case (or adversarial) analysis

- " $6n \log_2 n + 6n$ " bound holds for **every single input** of size n
- **no assumption** on where the input comes from

Not too worried about precise constants

- used " $6l$ " instead of " $4l + 2$ " for MERGE
- helps focus on properties of algorithms that **transcend** dependence on machine, programming language, architecture, etc.
- mathematically **easier** and **no loss** in predictive power

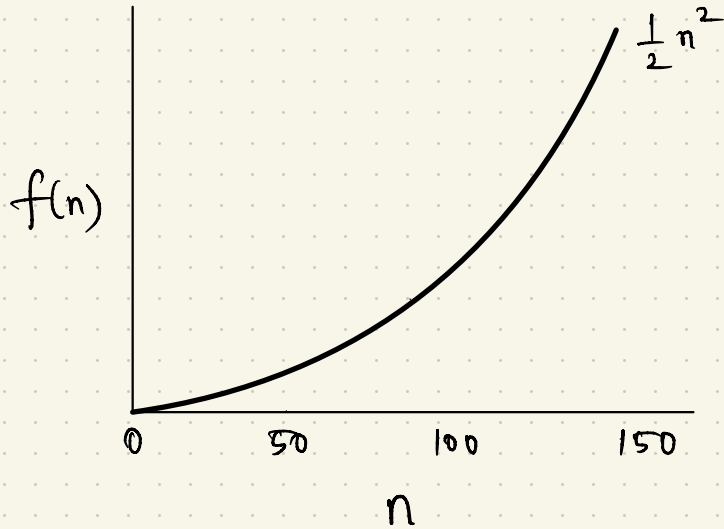
ASYMPTOTIC ANALYSIS

ASYMPTOTIC ANALYSIS

Which running time is better : $6n \log_2 n + 6n$ or $\frac{1}{2}n^2$?

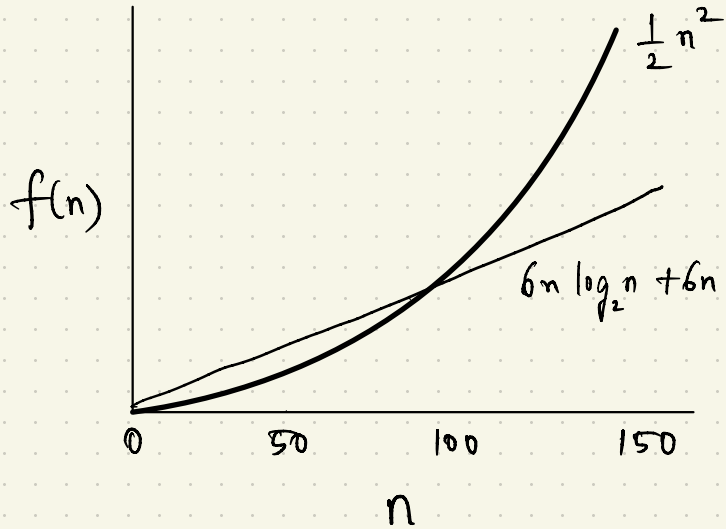
ASYMPTOTIC ANALYSIS

Which running time is better : $6n \log_2 n + 6n$ or $\frac{1}{2}n^2$?



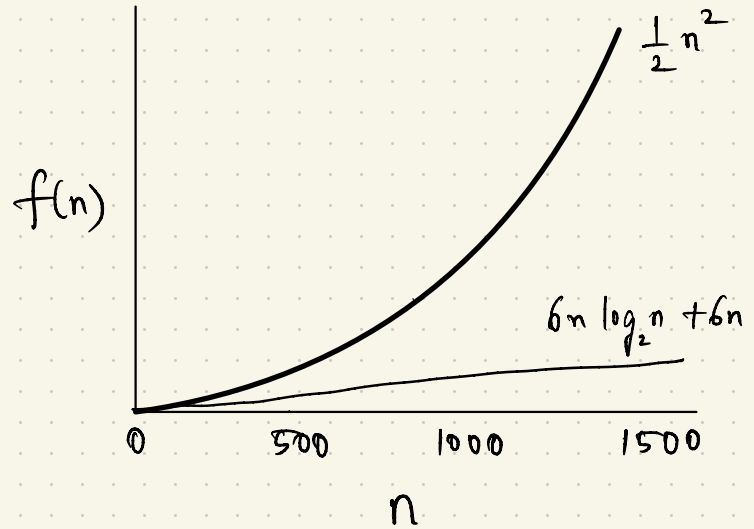
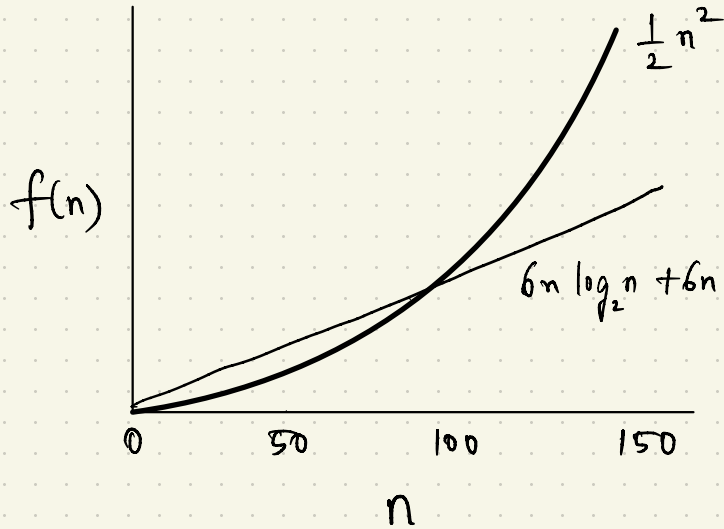
ASYMPTOTIC ANALYSIS

Which running time is better : $6n \log_2 n + 6n$ or $\frac{1}{2}n^2$?



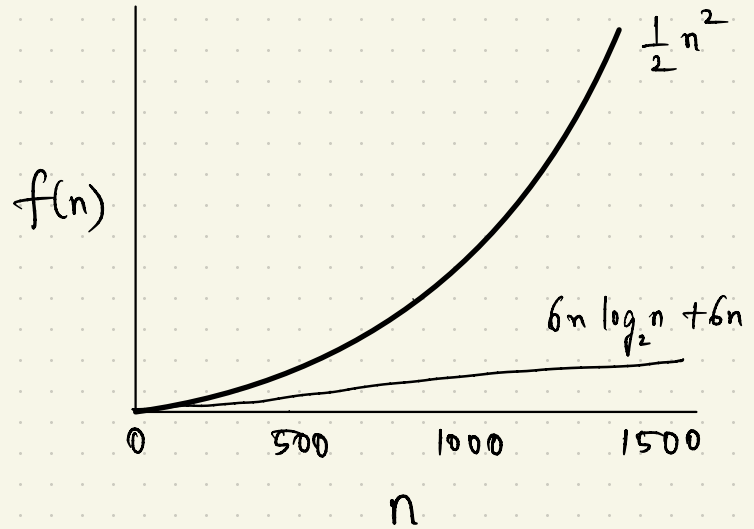
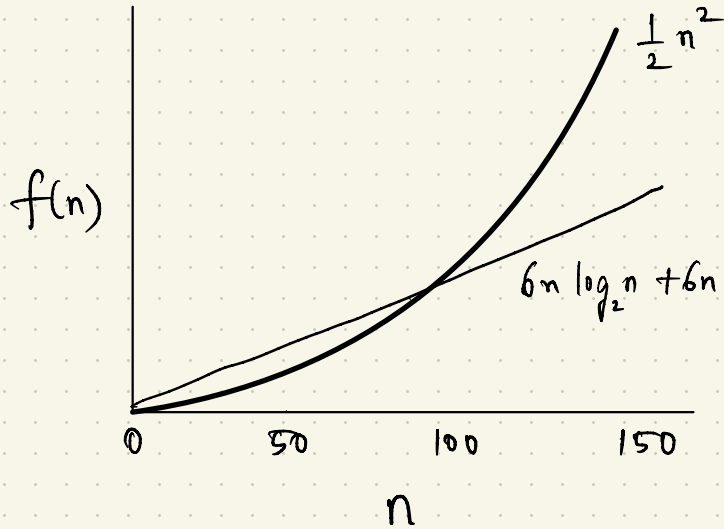
ASYMPTOTIC ANALYSIS

Which running time is better : $6n \log_2 n + 6n$ or $\frac{1}{2}n^2$?



ASYMPTOTIC ANALYSIS

Which running time is better : $6n \log_2 n + 6n$ or $\frac{1}{2}n^2$?



$6n \log_2 n + 6n$ is faster for sufficiently large inputs

ASYMPTOTIC ANALYSIS

Why care about large inputs?

ASYMPTOTIC ANALYSIS

Why care about large inputs?

- most algorithms can easily handle small inputs ;
it's the large inputs that require *algorithmic cleverness*
- Our ambitions grow with technological progress !

Need fast algorithms for large inputs

WHAT IS A "FAST" ALGORITHM ?

WHAT IS A "FAST" ALGORITHM ?

An algorithm whose **worst-case** running time
grows **slowly** with input size

WHAT IS A "FAST" ALGORITHM ?

An algorithm whose **worst-case** running time
grows **slowly** with input size

polynomial time

e.g., n , n^2 , $100n^3 + 35n^2 + 475$ etc.

WHAT IS A "FAST" ALGORITHM ?

An algorithm whose **worst-case** running time
grows **slowly** with input size

polynomial time

e.g., n , n^2 , $100n^3 + 35n^2 + 475$ etc.

WHAT IS A "FAST" ALGORITHM ?

An algorithm whose **worst-case** running time
grows **slowly** with input size

Poly-time in theory $\implies$ fast in practice
**often but
not always**

Better Balance by Being Biased: A 0.8776-Approximation for Max Bisection

Per Austrin^{*}, Siavosh Benabbas^{*}, and Konstantinos Georgiou[†]

has a lot of flexibility, indicating that further improvements may be possible. We remark that, while polynomial, the running time of the algorithm is somewhat abysmal; loose estimates places it somewhere around $O(n^{10^{100}})$; the running time of the algorithm of [RT12] is similar.

Picture-Hanging Puzzles

Erik D. Demaine · Martin L. Demaine ·
Yair N. Minsky · Joseph S.B. Mitchell ·
Ronald L. Rivest · Mihai Pătraşcu

Theorem 7 *For any $n \geq k \geq 1$, there is a picture hanging on n nails, of length $n^{c'}$ for a constant c' , that falls upon the removal of any k of the nails.*

of sorting networks is under $6,100 \log_2 n$. Applying Theorem 3, we obtain a picture hanging of length $c^{6,100 \log_2 n} = n^{6,100 \log_2 c}$. Using the $c \leq 1,078$ upper bound, we obtain an upper bound of $c' \leq 6,575,800$. Using the $c \leq 256 + o(1)$ upper bound, we obtain an upper bound of $c' \leq 1,561,600 + o(1)$.

So, while this construction is polynomial, it is a rather large polynomial. For small values of n , we can use known small sorting networks to obtain somewhat reasonable constructions.



WIKIPEDIA
The Free Encyclopedia

Simplex algorithm

🌐 33 languages ▾

Article [Talk](#)

Read [Edit](#) [View history](#) [Tools](#) ▾

From Wikipedia, the free encyclopedia

This article is about the linear programming algorithm. For the non-linear optimization heuristic, see [Nelder–Mead method](#).

In [mathematical optimization](#), [Dantzig's](#) **simplex algorithm** (or **simplex method**) is a popular [algorithm](#) for [linear programming](#).^[1]

Not poly-time in the worst case!