COL 351 : Analysis & Design of Algorithms

Lecture 3

Divide & Conquer II :

Merge Sort (contd.) and Asymptotic Analysis

July 26, 2024 | Rohit Vaish

ANNOUNCEMENTS

Tutorial Sheet 1 is available.

Tutorials start on Monday, July 29

Groups will be announced by Saturday (July 27)

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 ≤ 1
        return A
else
        L := left half of A sorted recursively
        R := right    "          "          "

        return MERGE(L, R)
```

MERGE SORT

input : array A of n distinct integers

output: array with the same numbers sorted in increasing order

```
if n ≤ 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

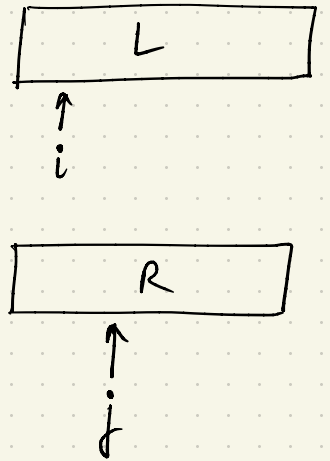
input : sorted arrays L and R of length $n/2$ each

output: sorted array B of length n

assume n is even

initialize $i := 1$

initialize $j := 1$



MERGE SUBROUTINE

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

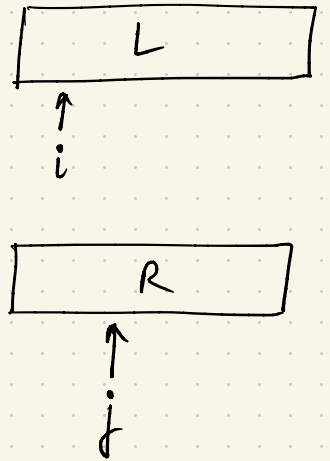
output: sorted array B of length n

assume n is even

initialize $i := 1$

initialize $j := 1$

for $k := 1$ to n



MERGE SUBROUTINE

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

output: sorted array B of length n

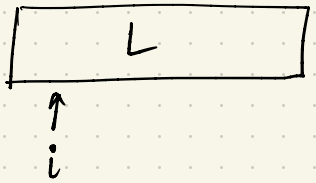
assume n is even

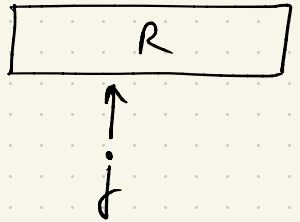
initialize $i := 1$

initialize $j := 1$

for $k := 1$ to n

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





else

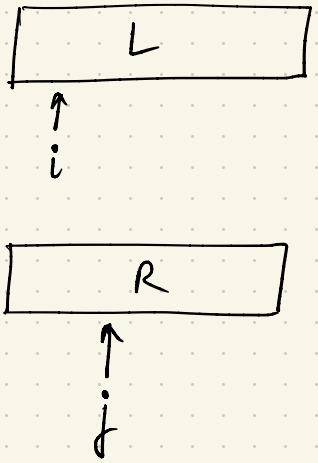
MERGE SUBROUTINE

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
```



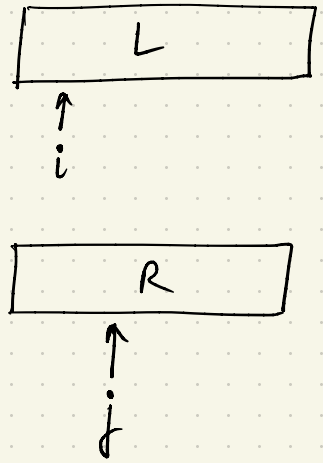
MERGE SUBROUTINE

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]         }  ?? operations
                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          } 3+1 = 4 operations
                B[k] := L[i]            (+1 = incrementing k)
                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]               } 3+1 = 4 operations
                increment i by 1           }   └ 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

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]               } 3+1 = 4 operations
                increment i by 1           }   └ incrementing k
        else                               }
                B[k] := R[j]               } 4 operations
                increment j by 1           }
```

$4n + 2$ 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 $\leq 4n + 2$.

$\leq 6n$ 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 $\leq 4l + 2$.

$\leq 6l$ since $l \geq 1$

MERGE SUBROUTINE

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

$\leq 6l$ since $l \geq 1$

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

MERGE SORT RUNNING TIME

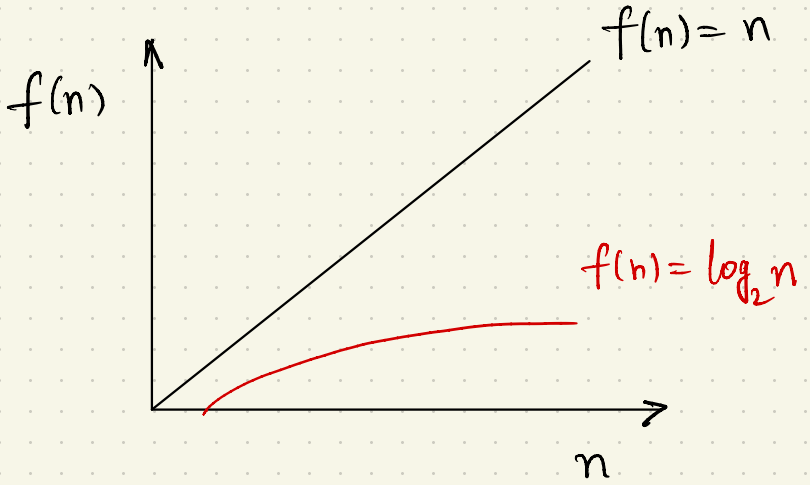
MERGE SORT RUNNING TIME

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.

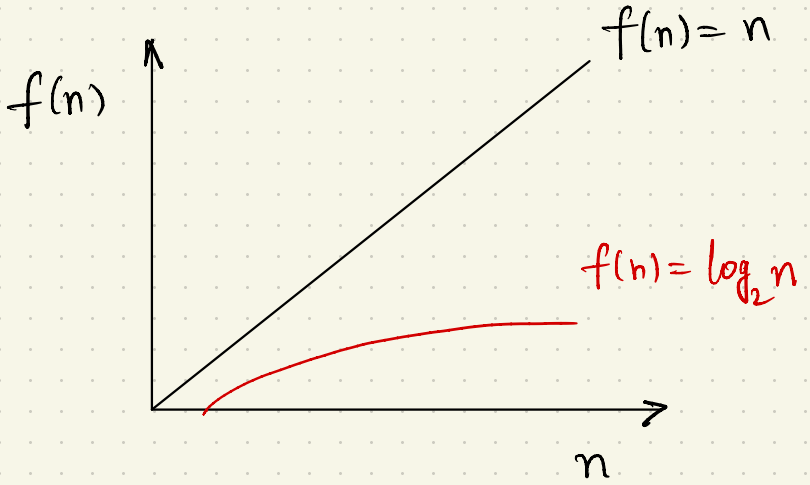
MERGE SORT RUNNING TIME

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

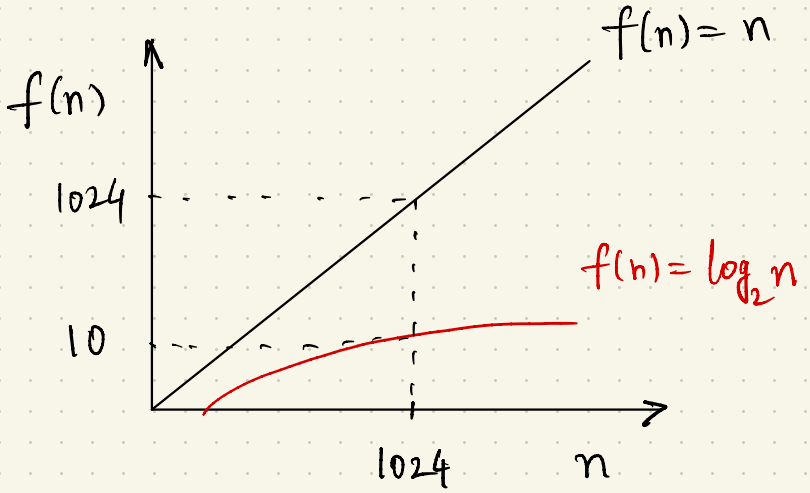
Theorem: For every input array of length $n \geq 1$, Merge Sort performs at most $6n \log_2 n + 6n$ operations.



$\log_2$: 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)

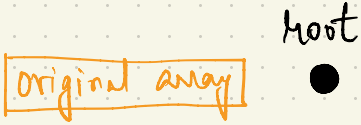
MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)

level 0 outermost call

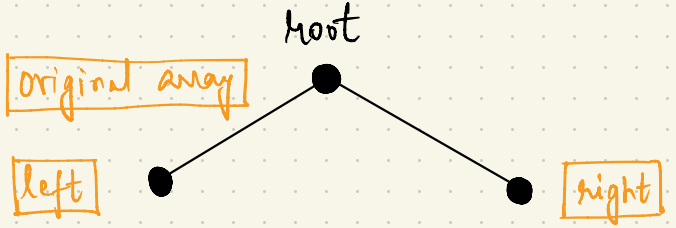
MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)

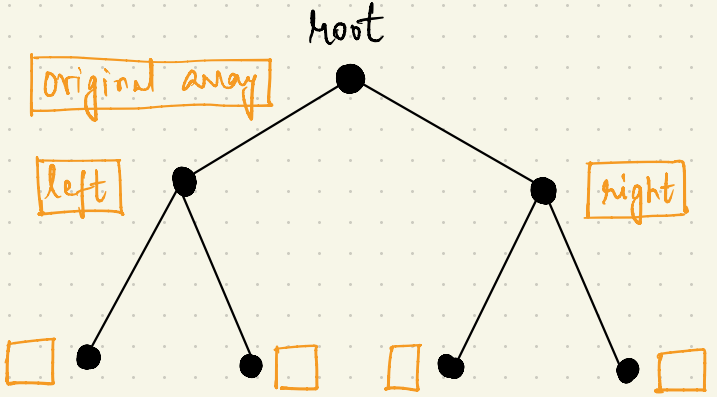


level 0 outermost call

level 1 first recursive calls

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



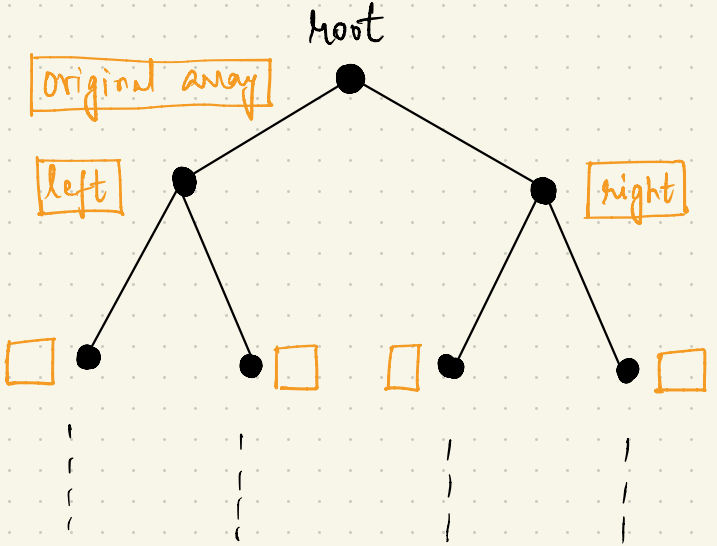
level 0 outermost call

level 1 first recursive calls

level 2

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



level 0 outermost call

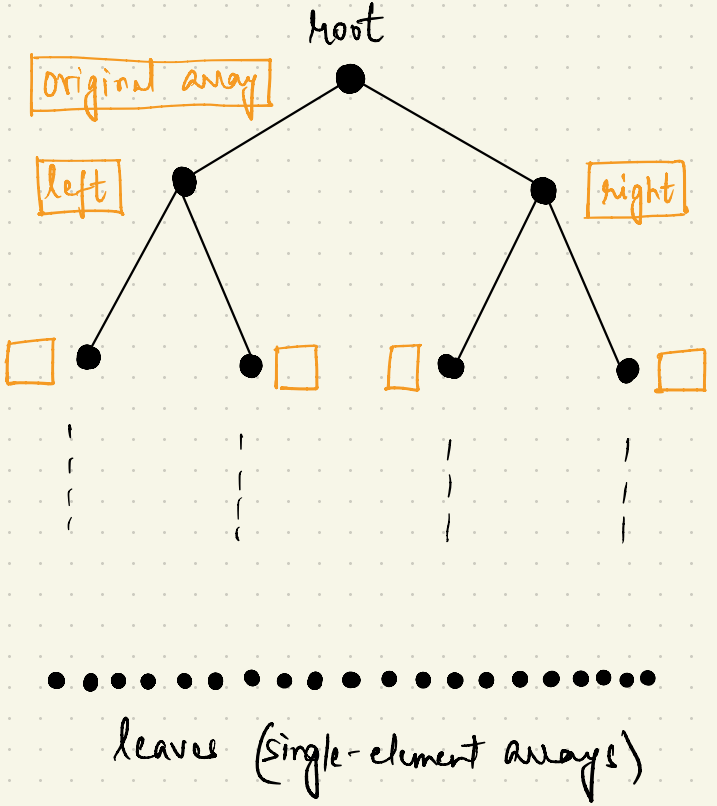
level 1 first recursive calls

level 2

leaves (single-element arrays)

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



level 0 outermost call

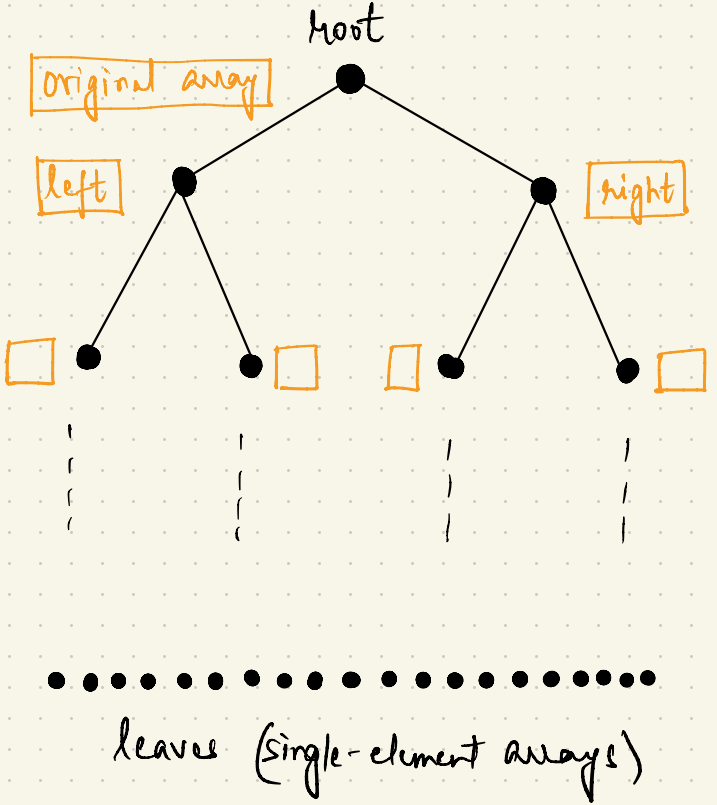
level 1 first recursive calls

level 2

level ??

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



level 0 outermost call

level 1 first recursive calls

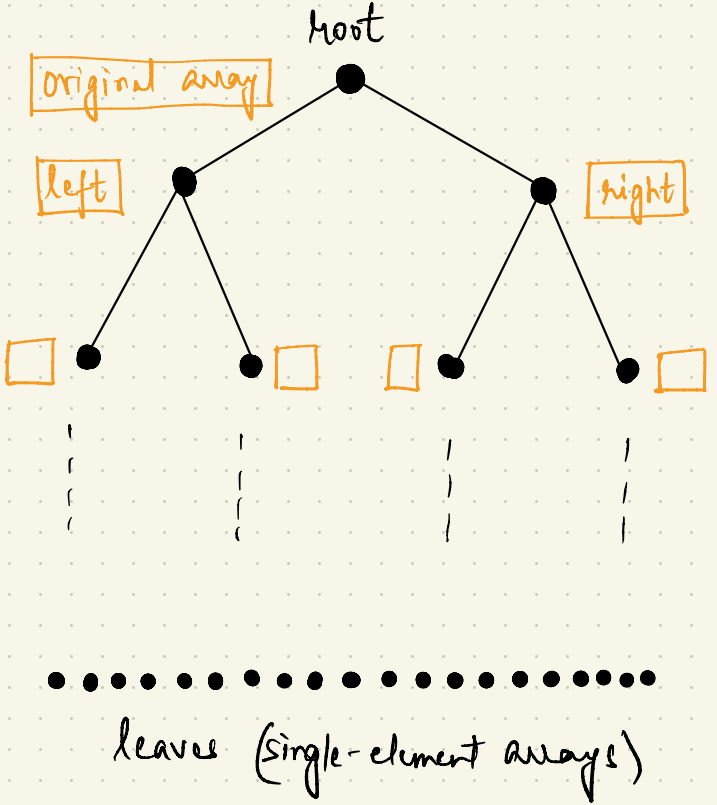
level 2

level $\log_2 n$

MERGE SORT RUNNING TIME

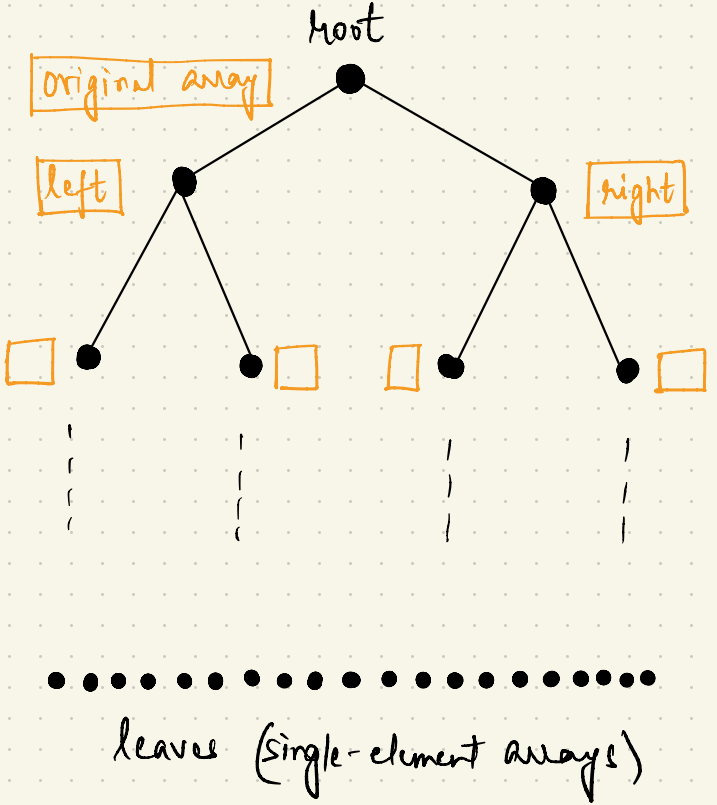
Proof: (assuming n is power of 2)

$\log_2 n + 1$ levels in total



MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



$\log_2 n + 1$ levels in total

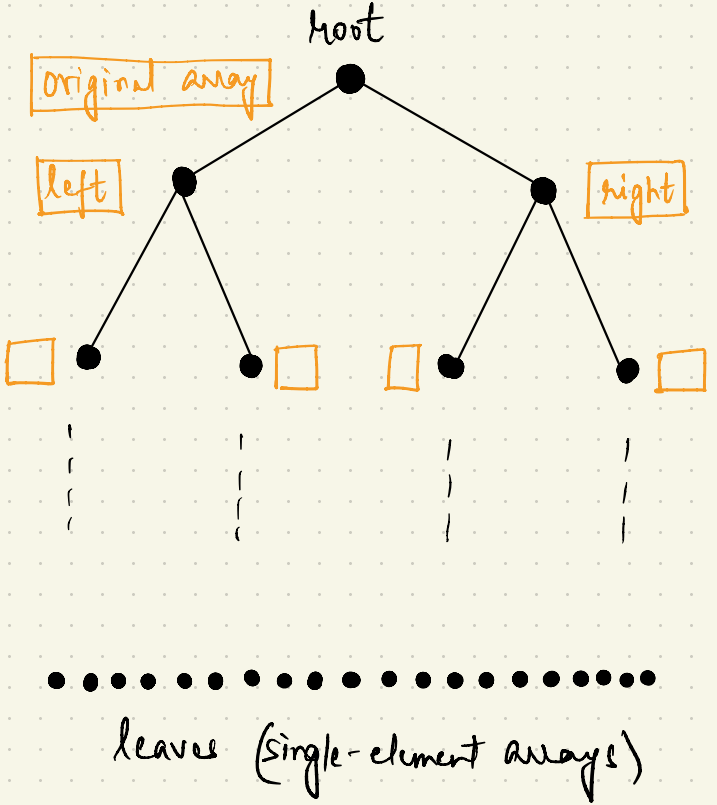
At level $j \in \{0, 1, \ldots, \log_2 n\}$

?? subproblems

each of size ??

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



$\log_2 n + 1$ levels in total

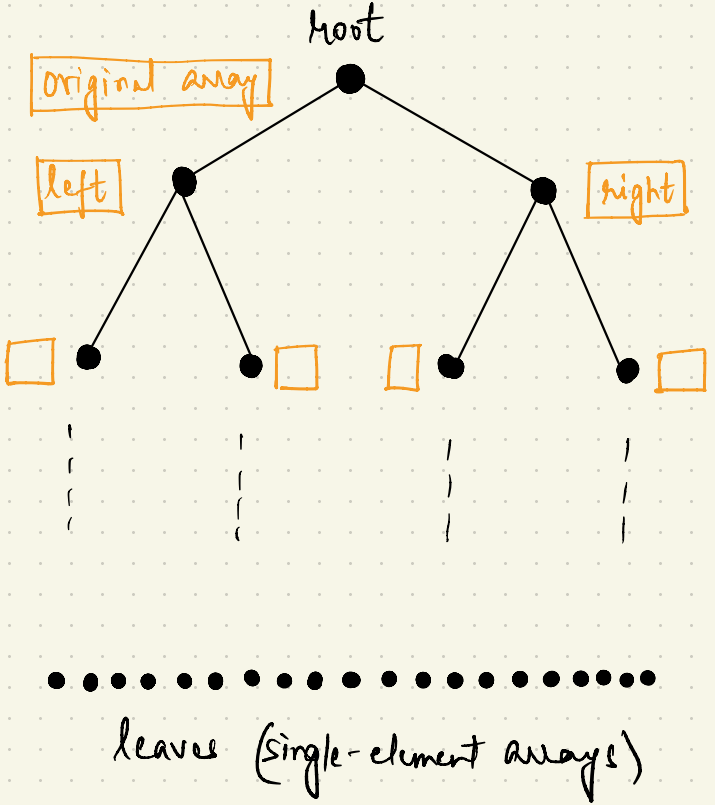
At level $j \in \{0, 1, \ldots, \log_2 n\}$

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?

MERGE SORT : PSEUDOCODE

```
if n ≤ 1
        return A
else
        L := left half of A sorted recursively
        R := right    "        "        "

        return MERGE(L, R)
```

MERGE SORT : PSEUDOCODE

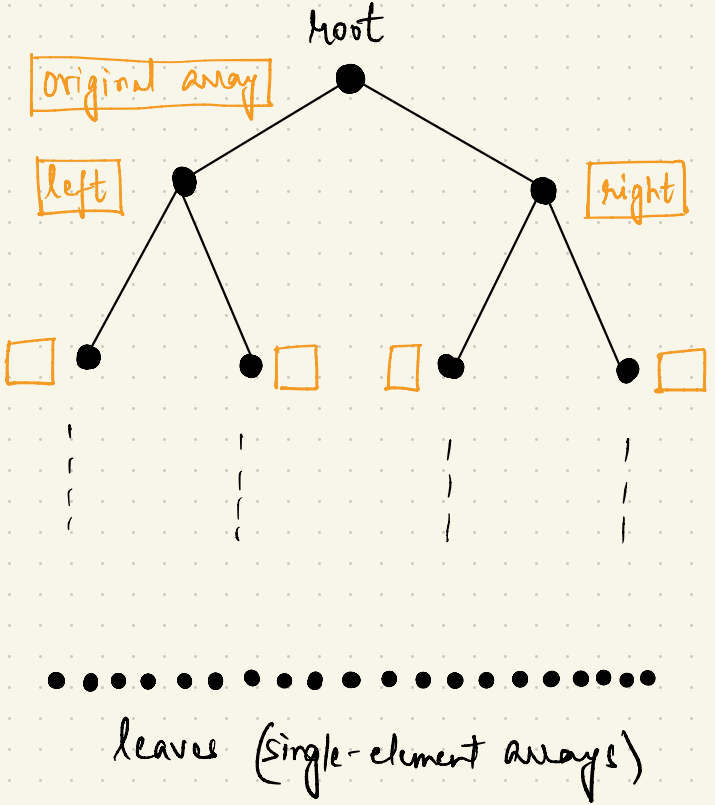
```
if n ≤ 1
        return A
else
        L := left half of A sorted recursively
        R := right    "        "        "

        return MERGE(L, R)
```

only need to count this

MERGE SORT RUNNING TIME

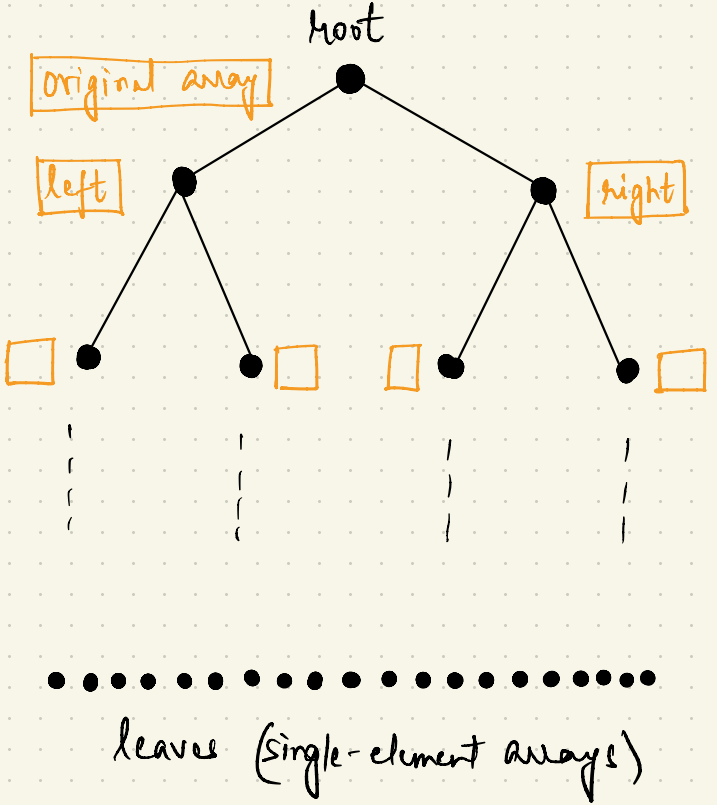
Proof: (assuming n is power of 2)



How much work does Merge Sort do at level j?

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



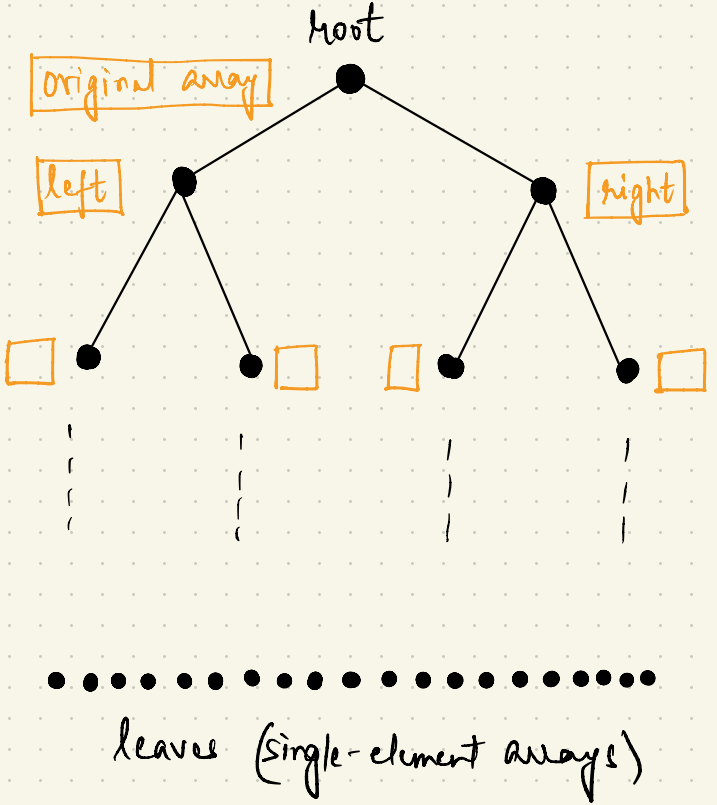
How much work does Merge Sort do at level j?

=

Work done by merge at level j

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

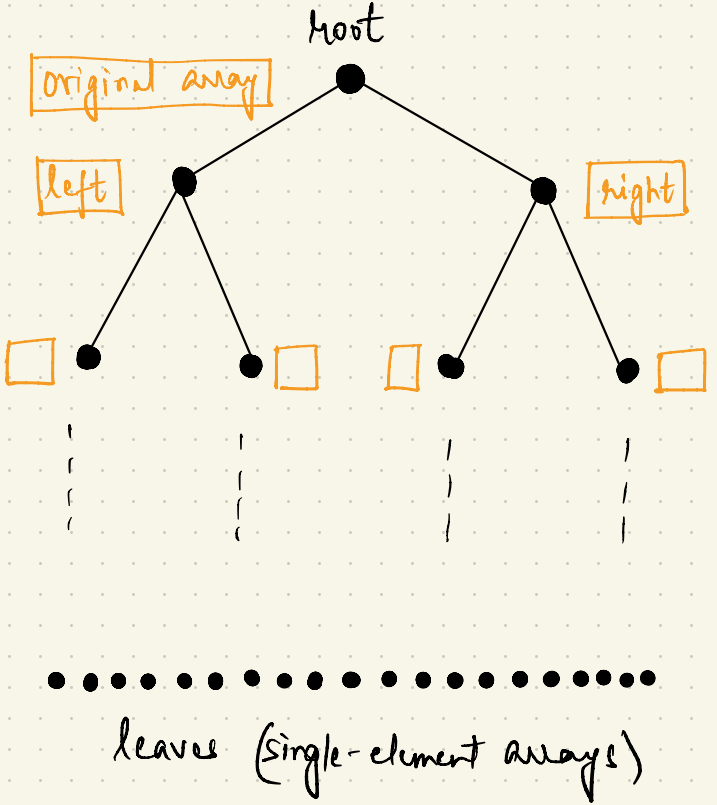
$\leq 2^j$

/

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

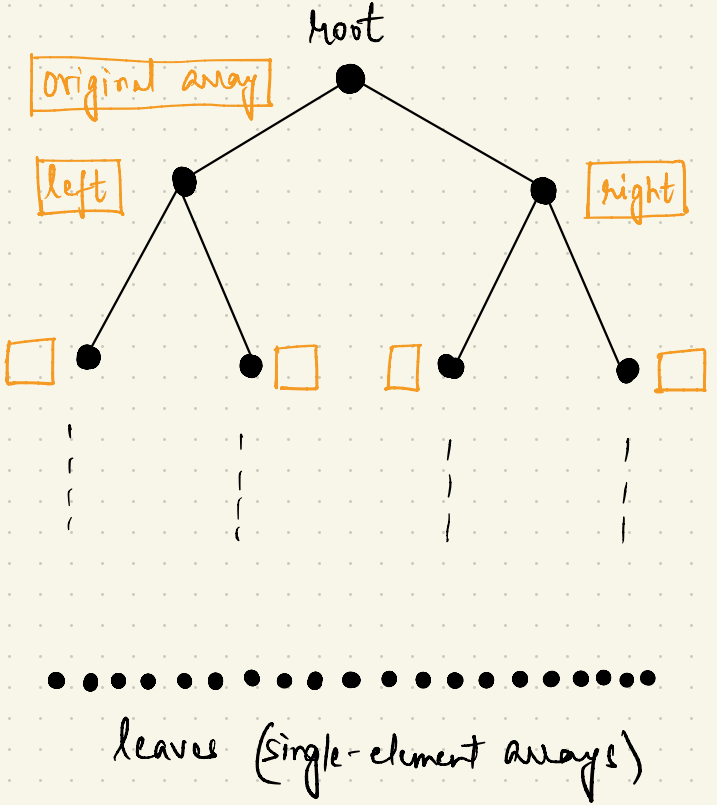
$\leq 2^j \times 6\left(n/2^j\right)$

2^j: # subproblems

$6\left(n/2^j\right)$: size $\ell = n/2^j$ (from Lemma)

MERGE SORT RUNNING TIME

Proof: (assuming n is power of 2)



How much work does Merge Sort do at level j?

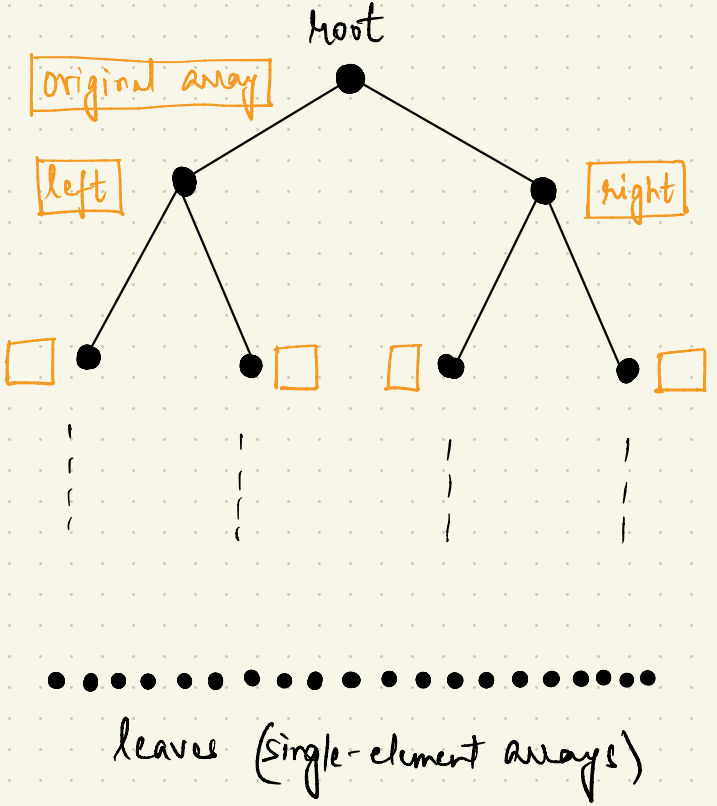
=

Work done by merge at level j

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

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

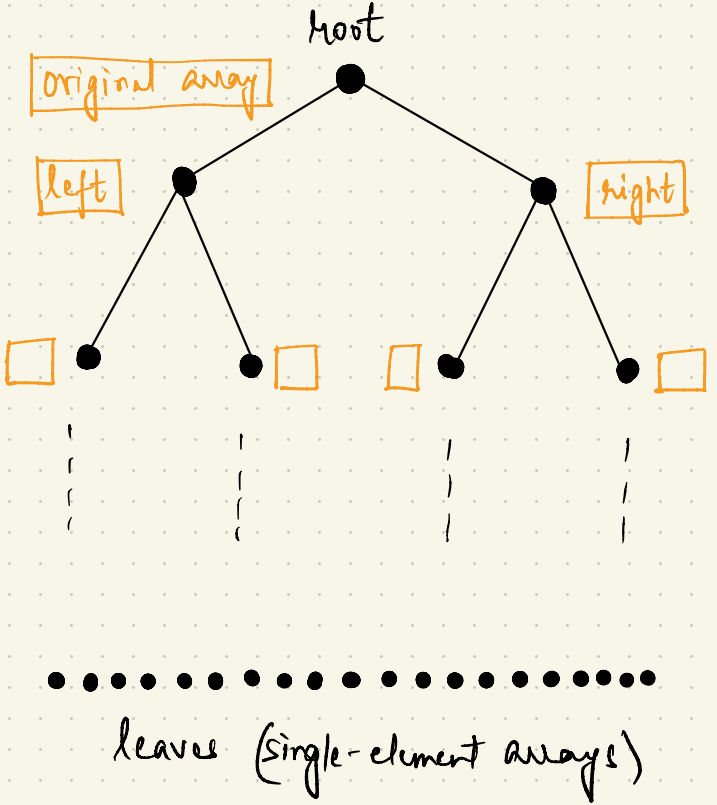
$$\leq 2^j \times 6\left(n/2^j\right)$$

$$= 6n$$

independent of 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

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

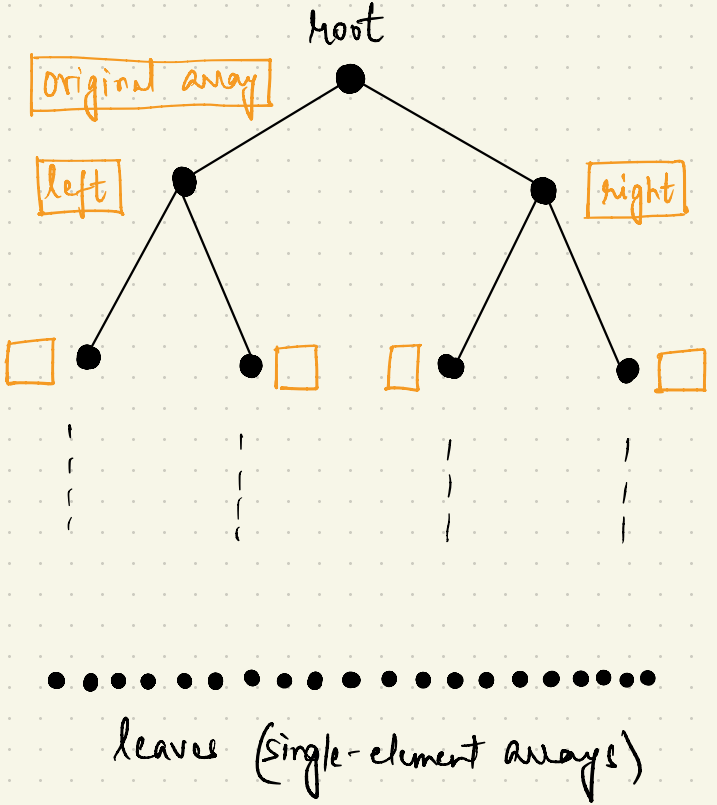
$= 6n$

subproblems ↑ by 2X

size / subproblem ↓ by 2X

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

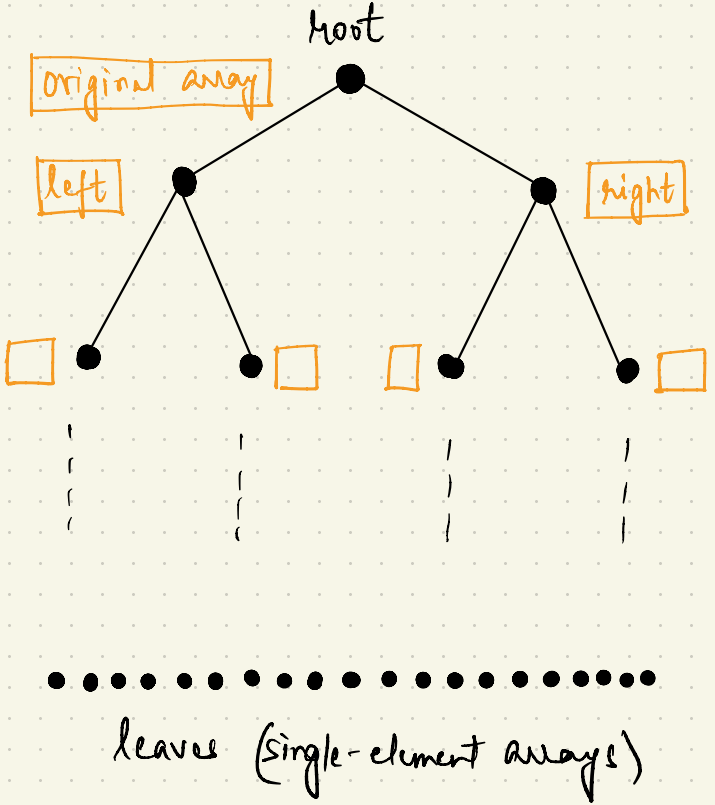
$\leq 2^j \times 6\left(n/2^j\right)$

$= 6n$

Total work done =

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

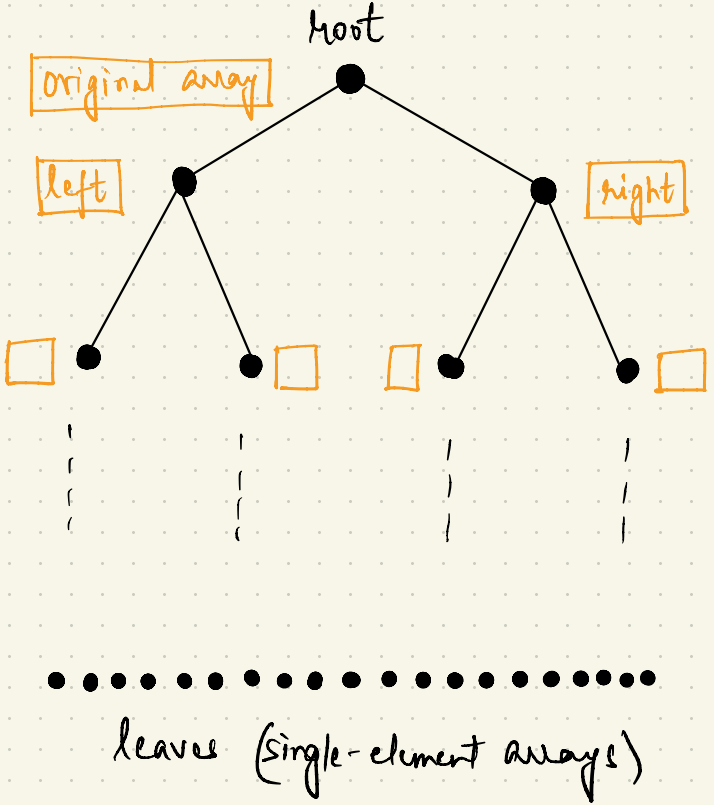
$\leq 2^j \times 6\left(n/2^j\right)$

$= 6n$

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

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

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

$$= 6n$$

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 "$4\ell + 2$" 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

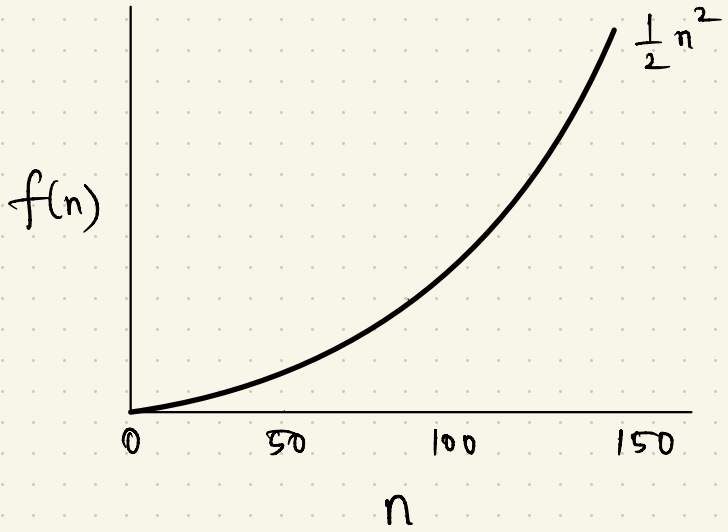
Asymptotic Analysis

ASYMPTOTIC ANALYSIS

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

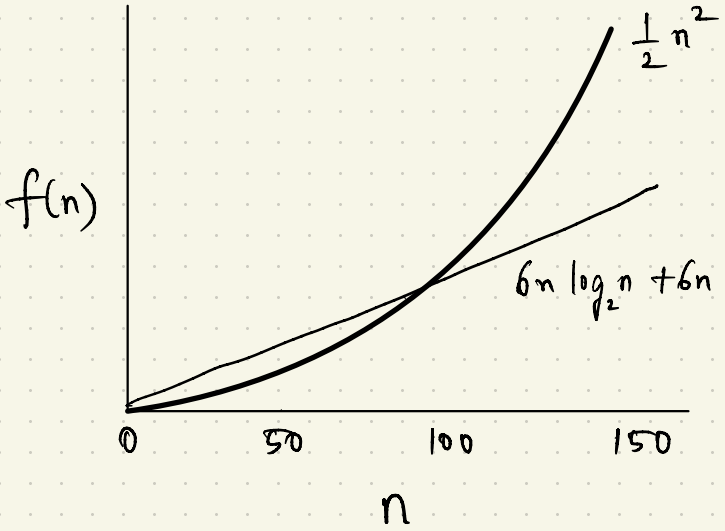
Asymptotic Analysis

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



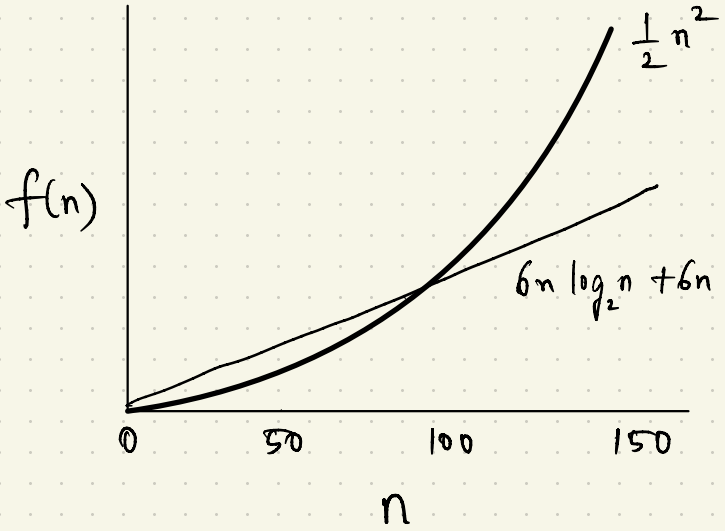
ASYMPTOTIC ANALYSIS

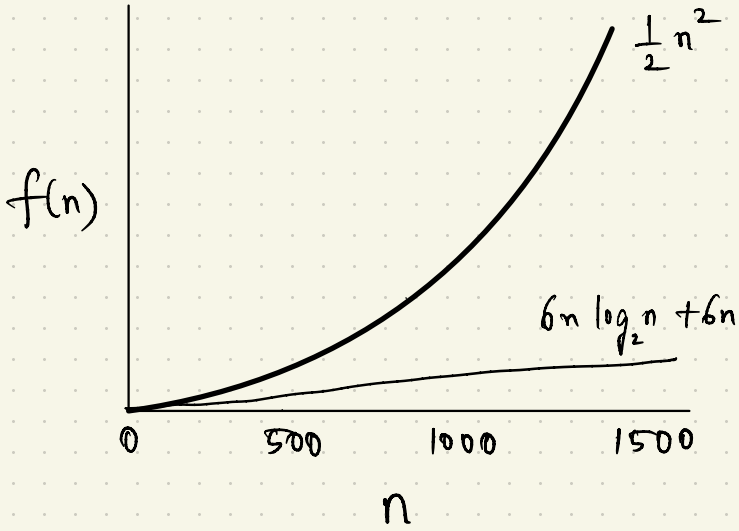
Which running time is better : $6n \log_2 n + 6n$ or $\frac{1}{2} n^2$?



Asymptotic Analysis

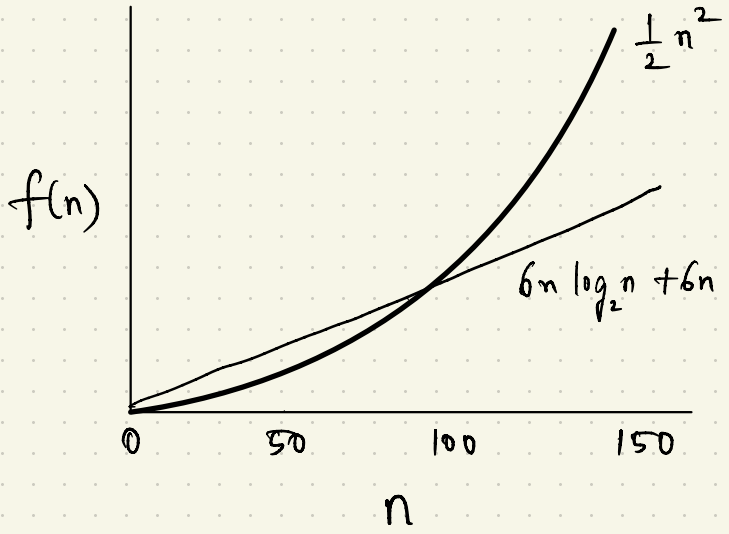
Which running time is better : $6n \log_2 n + 6n$ or $\frac{1}{2} n^2$?

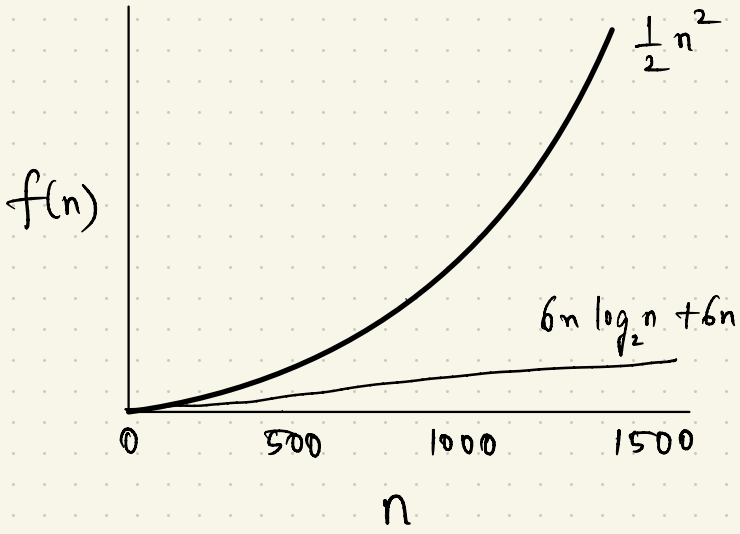




Asymptotic Analysis

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 $\Longrightarrow$ 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 $O\left(n^{10^{100}}\right)$; the running time of the algorithm of [RT12] is similar.

Theory Comput Syst (2014) 54:531–550
DOI 10.1007/s00224-013-9501-0

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.

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!