Sorting Algorithms Foundations - New

Annotated Slides

Sorting problem definition

INPUT

sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7

OUTPUT

a permutation of the sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

2 4 5 7 10

$b_1 \le b_2 \le b_3 \ldots \le b_n$

i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

(Column heading over 0–10: a[])

entries in black are examined to find the minimum

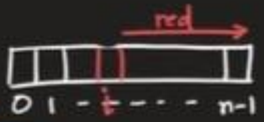
entries in red are a[min]

entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

int array of length n

```
function selectionsort(A):
    for i in 0 to n-1:
        minvalue = A[i]
        minindex = i
        for red in i+1 to n-1:
            if A[red] < minvalue:
                minvalue = A[red]
                minindex = red
        swap A[i], A[minindex]
    return A
```



How do we analyse this algorithm?

- Argue that it is CORRECT.
- Quantify its EFFICIENCY
    - a) Time : How long the algorithm takes to run ("running time")
    - b) Space : How much memory does the algorithm use to run?

Input size clearly matters, and is system-independent

A larger array will take a longer time to sort.

So why not define running time as a function of input size ?

T(n): where n is the input size. $n = 1, 2, 3, \ldots$

How do we compute an expression for T(n) in terms of n?

How do we measure running time?

Number of basic operations in a high-level language.
Examples of basic operations, with a fixed execution time, but different constant values for each:

hours_until_dinner = 2 c_1 //Variable assignment

class_end_time = t + 2.5 c_2 //Simple arithmetic operation on a variable

s = “Hope you are not already feeling sleepy!” //Variable assignment c_3

if (sleeping_time < class_end_time)
    then print “Watch recording later” c_4 //Slightly longer basic operation

Two basic operations in sorting algorithms

These would dominate the running time.

Comparison: `if a[inner] < a[min]:` C_5

Exchange/Swap: `(a[i],a[min]) = (a[min],a[i])` C_6

Two basic operations in sorting algorithms

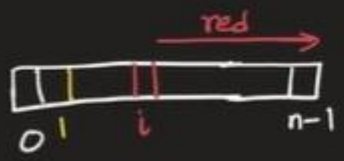
These would dominate the running time.

Comparison: `if a[inner] < a[min]:` C_5

Exchange/Swap: `(a[i],a[min]) = (a[min],a[i])` C_6

int array

function SelectionSort(A):




```
function SelectionSort(A):
    for i in 0 to n-1:                          -> c8 n
    {
        minvalue = A[i]                         <- c1      c1 n
        minindex = i                            <- c2      c2 n
        for red in i+1 to n-1:
        {
            if A[red] < minvalue:               <- c3
                minvalue = A[red]               <- c4
                minindex = red                  <- c5
        }                                       n(n-1)/2 * c9
        Swap A[i], A[minindex]                  <- c6      c6 n
    }
    return A                                    <- c7      c7
```

As $n \to \infty$

$$\frac{bn+c}{an^2} = \underset{\downarrow\, 0}{\frac{b}{an}} + \underset{\downarrow\, 0}{\frac{c}{an^2}}$$

$$T(n) = c_8 n + c_1 n + c_2 n + \frac{n(n-1)}{2} c_9 + c_6 n + c_7$$

$$S = 1 + 2 + \dots + n-1$$

$$S = n-1 + n-2 + \dots + 1$$

$$2S = \underbrace{n + n + \dots + n}_{n-1 \text{ times}}$$

$$= n(n-1)$$

$$S = n(n-1)/2$$

$$T(n) = \underbrace{an^2}_{\text{Dominant}} + \underbrace{bn + c}_{\text{Lower-order terms}}$$

$$\sim an^2$$

$\overline{an}$ $\overline{an^2}$

$\downarrow$ $\downarrow$

0 0

$T(n) = c_8 n + c_1 n + c_2 n + \frac{n(n-1)}{2} c_9 + c_8 n + c_7$

$S = 1 + 2 + \ldots + n-1$

$S = n-1 + n-2 + \ldots + 1$

$2S = \underbrace{n + n + \ldots + n}_{n-1 \text{ times}}$

$= n(n-1)$

$T(n) = \underbrace{an^2}_{\text{Dominant}} + \underbrace{bn + c}_{\text{Lower-order terms}}$

$\sim an^2$

- Drop the lower order terms
- To make the analysis system-independent, we also ignore the constant factors.

$T(n) = \boxed{\Theta(n^2)}$ $\qquad$ $T(n) = \Theta(n)$

$\sim cn^2$ $\qquad$ $\sim cn$

Asymptotic Time Complexity

As $n \to \infty$, $T(n)$ grows in a quadratic fashion

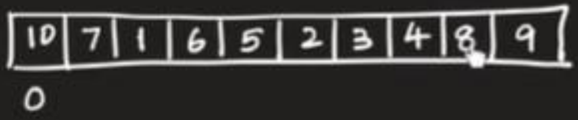
$b_0 \leq b_1 \leq b_2 \ldots \leq b_{n-1}$

A brute force algorithm follows directly from the problem description.

To get the minimum elements one by one, selection sort scans the array from left to right and keeps track of the next minimum.

Instead, we could have scanned the array from right to left, and <u>bubbled</u> up the minimum to the left by repeated exchanges.

Instead, we could have scanned the array from right to left, and bubbled up the minimum to the left by repeated exchanges.



Instead, we could have scanned the array from right to left, and <u>bubbled</u> up the minimum to the left by repeated exchanges.

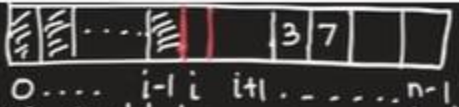
10	7	1	6	2	5	3	4	8	9

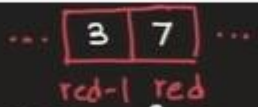
0

Instead, we could have scanned the array from right to left, and <u>bubbled</u> up the minimum to the left by repeated exchanges.

1	10	7	2	6	5	3	4	8	9

0





Instead, we could have scanned the array from right to left, and bubbled up the minimum to the left by repeated exchanges.

1	2	10	7	3	6	5	4	8	9
0	1								

BUBBLE SORT

and keeps track of the next minimum.

Instead, we could have scanned the array from right to left, and <u>bubbled</u> up the minimum to the left by repeated exchanges.

$T(n) = \Theta(n^2)$

```
function BubbleSort(A):        ← int array

    for i in 0 to n-1:         ← c3        c3*n
    {
        for red in n-1 down to i+1:     ← c2     n(n-1)/2 * c2
            if A[red-1] > A[red]:                 n(n-1)/2 * c1
                swap A[red-1], A[red]     c1
    }
    return A     ← c4
```

BUBBLE SORT

#comparison operations

$= (n-1) + (n-2) + \dots 0$

$= n(n-1)/2$

DESIGN STRATEGY # 1 : BRUTE FORCE

- Directly based off problem statement
- Subjective
- Try to be 'exhaustive'.

DESIGN STRATEGY #2 : DECREASE-AND-CONQUER

- Decrease the given problem of size n to size n-1

SELECTION & BUBBLE SORT

Upfront work to nibble away one element

$n \rightarrow n-1$

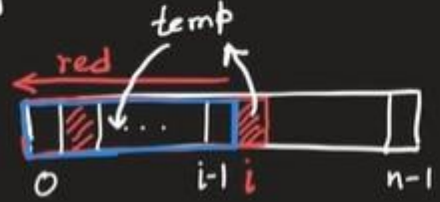
Work at the end to extend the $(n-1)^{st}$ solution

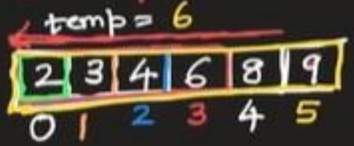
$n-1 \longrightarrow n$

INSERTION SORT

int array

```
function InsertionSort(A):
    for i in 0 to n-1:
        temp = A[i]
        red = i-1
        while red ≥ 0 and A[red] > temp:
            A[red+1] = A[red]
            red --
        A[red+1] = temp
    return A
```





$$T(n) = \sum_{i=0}^{n-1} (\text{Time spent by manager } i)$$

#right shift ops $* \, c_1 + c_2$

as high as i

as low as 0

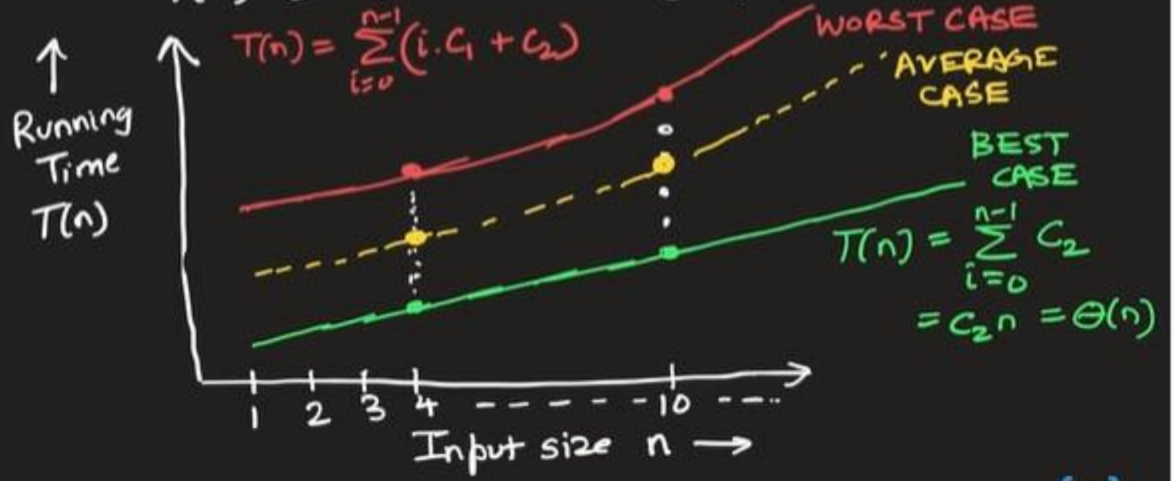
$$T(n) = \sum_{i=0}^{n-1} (\text{Time spent by manager } i)$$

Time spent by manager i = #rightshift ops $* \, C_1 + C_2$

#rightshift ops: as high as i, as low as 0

For the SAME input size n,

T(n) can take a variety of values!



1 2 3 4

Input size $n \longrightarrow$

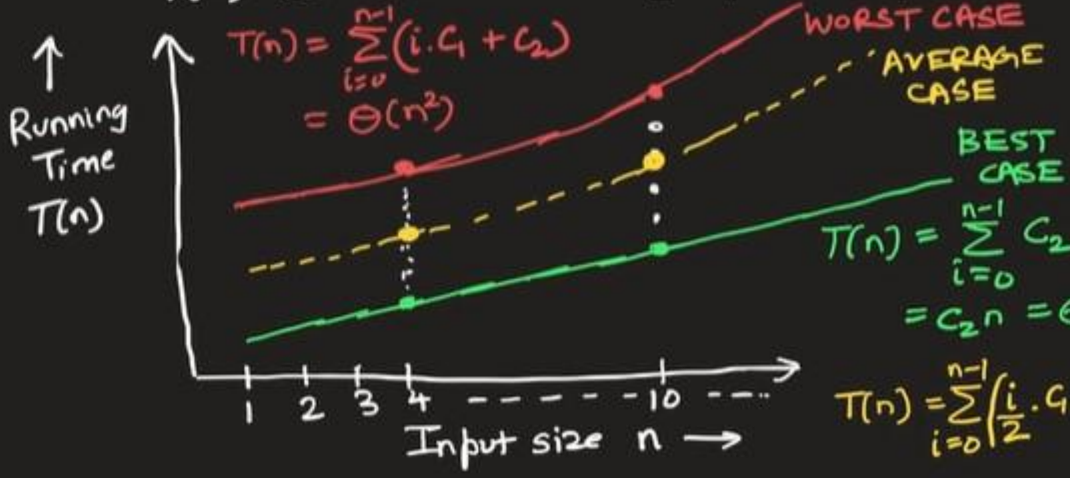
$$
\begin{aligned}
T(n) &= \sum_{i=0}^{n-1} (i \cdot c_1 + c_2) \\
&= \sum_{i=0}^{n-1} i c_1 + \underbrace{\boxed{\sum_{i=0}^{n-1} c_2}}_{c_2 n} \\
&= c_1 \sum_{i=0}^{n-1} i + c_2 n \\
&= c_1 (0 + 1 + \dots + n-1) + c_2 n \\
&= c_1 \cdot \frac{n(n-1)}{2} + c_2 n \\
&= \Theta(n^2)
\end{aligned}
$$

$$T(n) = \sum_{i=0}^{n-1} (\text{Time spent by manager } i)$$

\#right shift ops $* \, C_1 + C_2$

as high as i
as low as 0
on average $i/2$

For the SAME input size n,
$T(n)$ can take a variety of values!

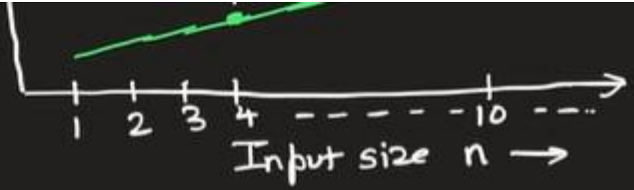


$$T(n) = \sum_{i=0}^{n-1} (i \cdot C_1 + C_2) = \Theta(n^2)$$

$$T(n) = \sum_{i=0}^{n-1} C_2 = C_2 n = \Theta(n)$$

$$T(n) = \sum_{i=0}^{n-1} \left(\frac{i}{2} \cdot C_1 + C_2\right)$$

$$T(n) = \sum (i \cdot C_1 + C_2)$$



$$= c_2 n = \Theta(n)$$

$$T(n) = \sum_{i=0}^{n-1}\left(\frac{i}{2}\cdot c_1 + c_2\right) = \Theta(n^2)$$

$$T(n) = \sum_{i=0}^{n-1}\left(\frac{i}{2}\cdot c_1 + c_2\right)$$

$$T(n)$$

$$= \sum_{i=0}^{n-1}\frac{i}{2}c_1 + \boxed{\sum_{i=0}^{n-1} c_2} \rightarrow c_2 n$$

$$= c_1 \sum_{i=0}^{n-1}\frac{i}{2} + c_2 n$$

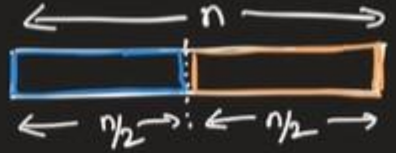
$$= c_1\left(\frac{0+1+\dots+n-1}{2}\right) + c_2 n$$

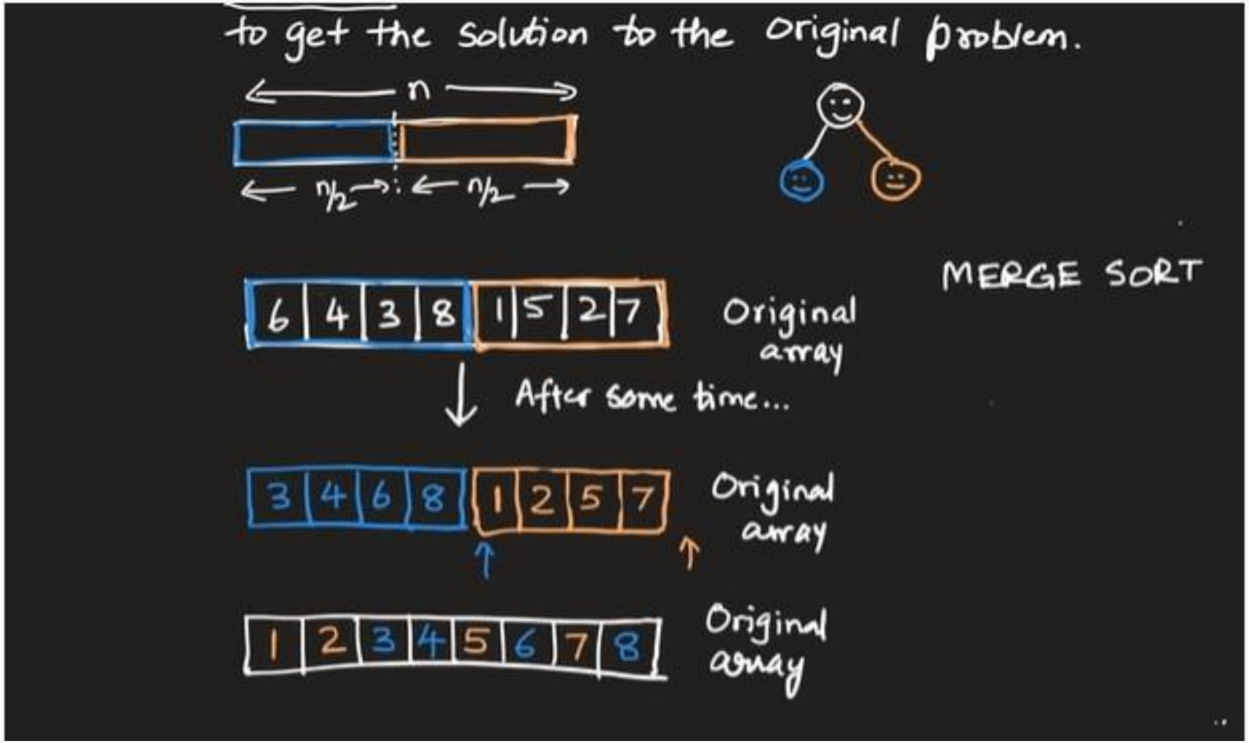
$$= c_1 \cdot \frac{n(n-1)}{2\times 2} + c_2 n$$

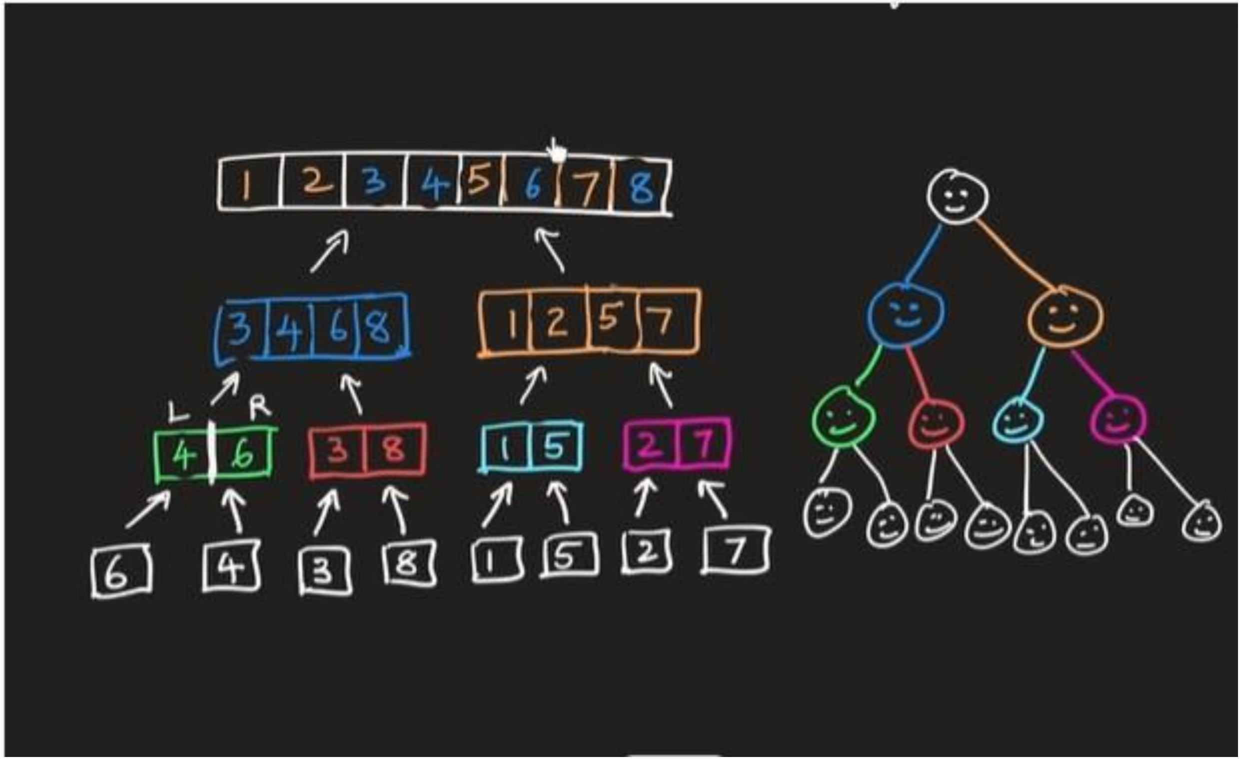
$$= \Theta(n^2) \quad \Theta(n^2)$$

DESIGN STRATEGY #3 : DIVIDE & CONQUER

- Divide the problem into multiple smaller instances (most often 2), generally of the same size.
- Solve the smaller instances (typically using recursion)
- Combine the solutions to the smaller instances to get the solution to the original problem.

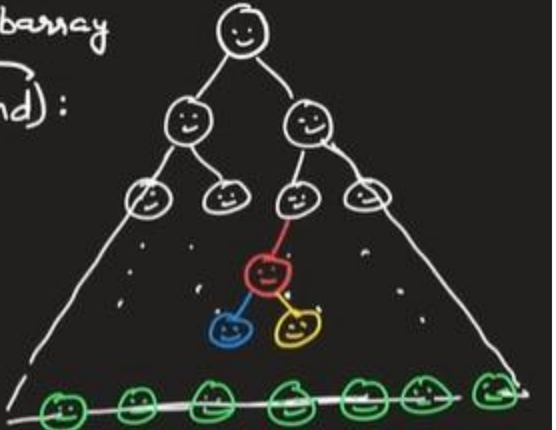


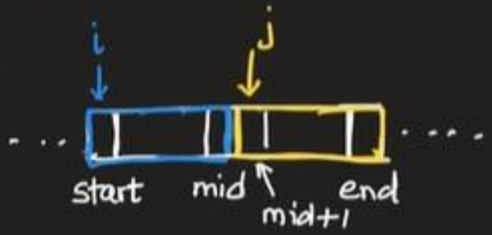

to get the solution to the original problem.
n
n/2
n/2
MERGE SORT
6 4 3 8 1 5 2 7
Original array
After some time...
3 4 6 8 1 2 5 7
Original array
1 2 3 4 5 6 7 8
Original array



1 2 3 4 5 6 7 8
3 4 6 8
1 2 5 7
L
R
4 6
3 8
1 5
2 7
6
4
3
8
1
5
2
7

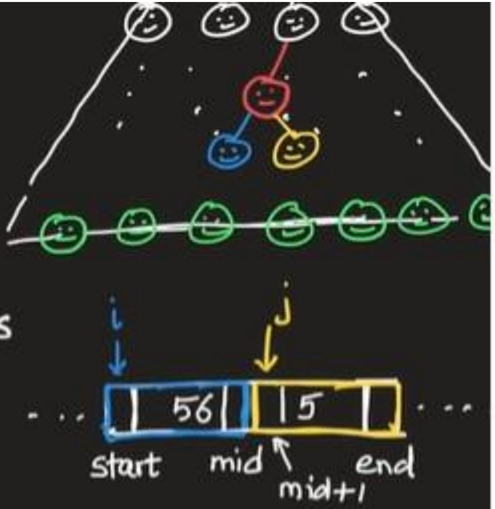
```
                     int subarray
function mergesort ( A, start,end):
  # Leaf worker
  if start == end :
     return
  # Internal node worker
  mid = (start + end)/2
  mergesort( A , start, mid)
  mergesort (A, mid+1 , end)
  //Merge the two sorted halves
  i = start, j = mid+1
  aux = an empty array of
            size end-start+1
  while i ≤ mid and j ≤ end :
```



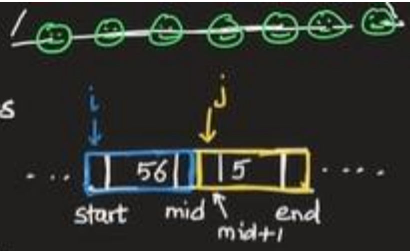


end -

```
if start == end:
    return
# Internal node worker
mid = (start + end)/2
mergesort(A, start, mid)
mergesort(A, mid+1, end)
//Merge the two sorted halves
i = start, j = mid+1
aux = an empty array of
        size end-start+1
while i ≤ mid and j ≤ end:
    if A[i] ≤ A[j]:
        aux.append(A[i])
        i++
    else: // A[i] > A[j]
        aux.append(A[j])
        j++
```

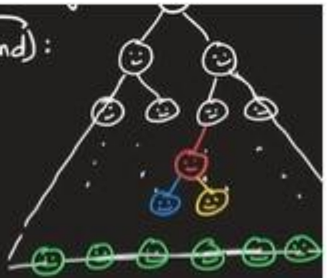


```
mid = (start + end) / 2
mergesort(A, start, mid)
mergesort(A, mid+1, end)
//Merge the two sorted halves
i = start, j = mid+1
aux = an empty array of
        size end-start+1
while i ≤ mid and j ≤ end:
{   if A[i] ≤ A[j]:
        aux.append(A[i])
        i++
    else: // A[i] > A[j]
        aux.append(A[j])
        j++
}
// Gather phase
while i ≤ mid:
{   aux.append(A[i])
    i++
}
while j ≤ end:
{   aux.append(A[j])
    j++
}
A[start...end] ← aux
return
```



end - start + 1
[start, end]
[2, 5]   5-2+1 = 4

```
function mergesort (A, start, end):
  # Leaf worker
  if start == end:
     return
  # Internal node worker
  mid = (start + end)/2
  mergesort(A, start, mid)
  mergesort(A, mid+1, end)
  // Merge the two sorted halves
  i = start, j = mid+1
  aux = an empty array of
         size end-start+1
  while i ≤ mid and j ≤ end:
  {    if A[i] ≤ A[j]:
          aux.append (A[i])
          i++
       else: // A[i] > A[j]
          aux.append (A[j])
          j++
  }
   // Gather phase
   while i ≤ mid:
   { aux.append (A[i])
     i++ }
   while j ≤ end:
   { aux.append (A[j])
     j++
```



int array

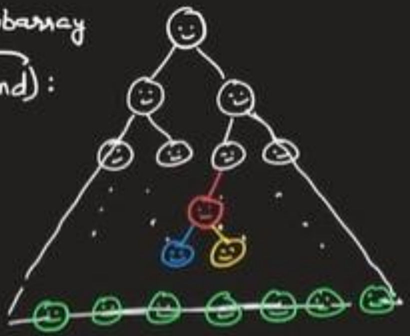
```
function mergesort (A):
  → helper (A, 0, length(A) - 1)
  → return A
```

int subarray

```
function helper (A, start, end):
  # Leaf worker
  if start == end:
     return
  # Internal node worker
  mid = (start + end)/2
  helper (A, start, mid)
  helper (A, mid+1, end)
  // Merge the two sorted halves
  i = start, j = mid+1
  aux = an empty array of
         size end-start+1
  while i ≤ mid and j ≤ end:
  {    if A[i] ≤ A[j]:
          aux.append (A[i])
           i++
      else: // A[i] > A[j]
          aux.append (A[j])
          j++
  }
  // Gather phase
  while i ≤ mid:
  {  aux.append (A[i])
     i++
```

$$\text{function helper } (\underbrace{A, \text{start}, \text{end}}^{\text{int subarray}}):$$

```
function helper (A, start, end):
  # Leaf worker
  if start == end :
     return
  # Internal node worker
  mid = (start + end)/2
  helper (A, start, mid)
  helper (A, mid+1, end)
  //Merge the two sorted halves
  i = start, j = mid+1
  aux = an empty array of
          size end-start+1
  while i ≤ mid and j ≤ end :
  {   if A[i] ≤ A[j] :
         aux.append (A[i])
          i++
      else: // A[i] > A[j]
         aux.append (A[j])
         j++
  }
  // Gather phase
  while i ≤ mid:
  ( aux.append (A[i])
```



end - start + 1

[start, end]

[2, 5] $5-2+1 = 4$

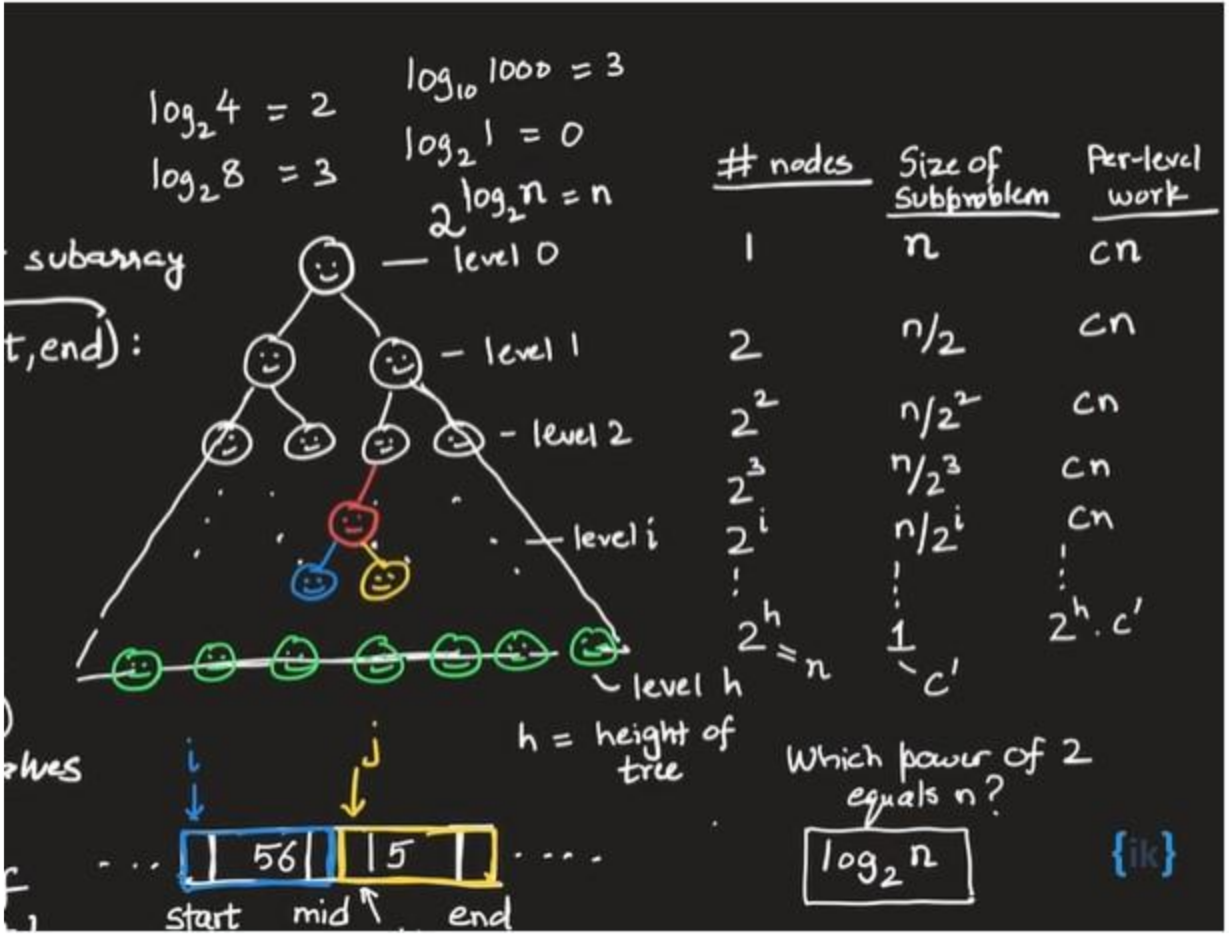
Internal node worker

```
mid = (start + end)/2
helper (A, start, mid)
helper (A, mid+1, end)
// Merge the two sorted halves
i = start, j = mid+1
aux = an empty array of
        size end-start+1
while i ≤ mid and j ≤ end :
{    if A[i] ≤ A[j] :
        aux.append (A[i])
        i++
     else : // A[i] > A[j]
        aux.append (A[j])
        j++
}
// Gather phase
while i ≤ mid :
{  aux.append (A[i])
   i++ }
while j ≤ end :
{  aux.append (A[j])
   j++ }
A[start ... end] ← aux
return
```

end − start + 1
[start, end]
[2, 5] 5−2+1 = 4

Running time for internal node worker = c (size of subproblem)

→ For leaf worker = c′



log2 4 = 2
log2 8 = 3
log10 1000 = 3
log2 1 = 0
2^(log2 n) = n
subarray
t, end):
level 0
level 1
level 2
level i
level h
h = height of tree
# nodes
Size of subproblem
Per-level work
1
n
cn
2
n/2
cn
2^2
n/2^2
cn
2^3
n/2^3
cn
2^i
n/2^i
cn
2^h = n
1
c'
2^h . c'
Which power of 2 equals n?
log2 n
i
j
56
15
start
mid
end
lves
{ik}

$\log_{10} 1000 = 3 \qquad \log_2 1024 = 10$

$\log_2 1 = 0 \qquad \log_2 n \ll n$

$2^{\log_2 n} = n$

# nodes	Size of subproblem	Per-level work
1	n	cn
2	$n/2$	cn
2^2	$n/2^2$	cn
2^3	$n/2^3$	cn
2^i	$n/2^i$	cn
$\vdots$	$\vdots$	$\vdots$
$2^h = n$	$1 \sim c'$	$\underbrace{2^h}_{n} \cdot c'$

$T(n) = cn * \#\text{ levels in tree}$

$= cn(\log_2 n + 1)$

$= cn \log_2 n + \cancel{cn}$

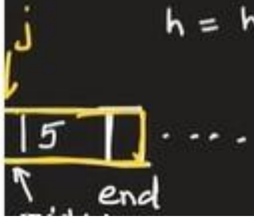
$= \Theta(n \log_2 n)$

$= \Theta(n \log n)$

h = height of tree

Which power of 2 equals n?

$h = \boxed{\log_2 n}$



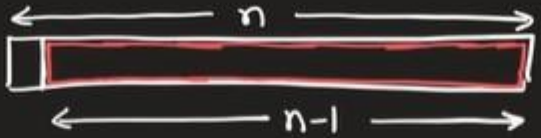
return
if the array is already sorted

int array

```
function mergesort (A):
    helper (A, 0, length(A) - 1)
    return A
```

int su

```
function helper (A, start, e
  # Leaf worker
  if start == end :
      return
  # Internal node worker
  mid = (start + end)/2
  helper (A, start, mid)
  helper (A, mid+1, end)
  //Merge the two sorted halve
  i = start, j = mid+1
  aux = an empty array of
          size end-start+1
  while i ≤ mid and j ≤ end
      if A[i] ≤ A[j] :
          aux.append (A[i])
```



$$T(n) = cn + T(n-1)$$

Recurrence equation

$$T(n-1) = c(n-1) + T(n-2)$$

$$\begin{aligned}
T(n) &= cn + c(n-1) + T(n-2) \\
&= cn + c(n-1) + c(n-2) + T(n-3) \\
&= cn + c(n-1) + c(n-2) + c(n-3) + \ldots\ldots + T(0) \\
&= c[\underbrace{n + n-1 + n-2 + n-3 + \ldots\ldots 1}] + \\
&= \Theta(n^2)
\end{aligned}$$

$$1 + 2 + \ldots n-1 + n = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2}$$

$$T(n) = T(n-1) + cn$$

$$T(n) = T(n-1) + c$$
$$= T(n-2) + c + c$$
$$= T(0) + \underbrace{c + c + c + \dots c}_{n \text{ times}}$$
$$= cn$$
$$= \Theta(n)$$

$$. + T(0)$$

$$n-1 + n$$
$$= \frac{n(n-1)}{2} + n$$
$$\frac{n(n+1)}{2}$$

$$T(n) = \boxed{c} + T(n/2) + T(n/2) + c'n$$

$$\boxed{T(n) = 2T(n/2) + c'n}$$

$$= c'n + 2\left[c'\left(\frac{n}{2}\right) + 2T(n/4)\right]$$

$$T(n/2) = 2T(n/4) + c'\left(\frac{n}{2}\right)$$

$$= c'n + c'n + 4T(n/4)$$
$$= c'n + c'n + c'n + c'n + \dots + n * T(1)$$
$$= c'n \log n$$
$$= \Theta(n \log n)$$

$T(n) = T(n-1) + cn$

$T(n) = T(n-1) + c$

$T(n) = 2T(n/2) + cn$

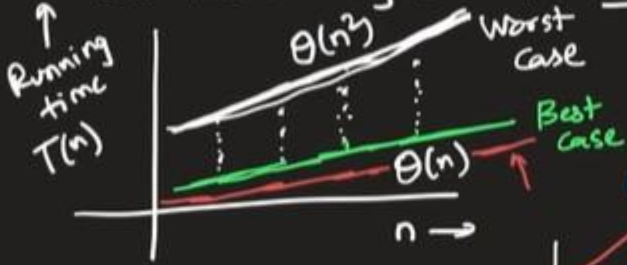
$T(n) = T(n-1) + \Theta(n)$

$T(n) = T(n-1) + \Theta(1)$

$T(n) = 2T(n/2) + \Theta(n)$

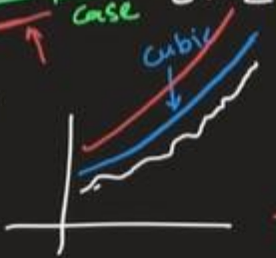
best case   a lower

The worst case running time is an upper bound on the running time for any input of size n



$x = 2$

$x \le 2$



$c'n \le \boxed{T(n) \le cn^2}$

$T(n) = O(n^2)$

↑ Big-Oh notation

$T(n) = \Omega(n)$

↑ BIG-OMEGA notation

How important is the distinction between n log n and n^2?

US population = 320 million

Suppose we had to sort census data based on names.

Assume that a CPU can perform 100 million basic operations per second (10^8).

Executing n log n operations would take ~90 seconds or 1.5 minutes.

Executing $\boxed{n^2}$ operations would take 32.5 years!

$$\boxed{\frac{320M \times \log(320M)}{10^8}}$$

$$\frac{(320M)^2}{10^8}$$

growth rate	problem size solvable in minutes			

{ik}

$$\frac{(320M)^2}{10^8}$$

$10^{\sim}$

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
$\Theta(1)$	any	any	any	any
$\Theta(\log N)$	any	any	any	any
$\Theta(N)$	millions	tens of millions	hundreds of millions	billions
$\Theta(N \log N)$	hundreds of thousands	millions	millions	hundreds of millions
$\Theta(N^2)$	hundreds	thousand	thousands	tens of thousands
$\Theta(N^3)$	hundred	hundreds	thousand	thousands
$\Theta(2^N)$	20	20s	20s	30

{ik}

Merge sort requires extra memory

An algorithm is said to be **in place** if it does not require extra memory, except a constant amount of memory units. — $\Theta(1)$

Is merge sort in place? No

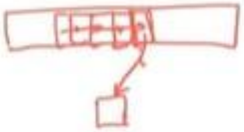
$\Theta(n \log n)$

$\Theta(n)$ aux space

Insertion sort? Selection sort? Bubble sort?

Yes

$\Theta(1)$ aux space



Merge sort running time

Insertion Sort running time

$c\, n \log n > c' n^2$ (worst-case comparison)

for small n, since $c \gg c'$

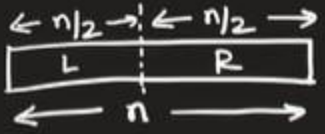
TimSort

Combines merge-sort and insertion-sort for worst-case $\boxed{\Theta(n \log n)}$ time and best-case $\boxed{\Theta(n) \text{ time}}$ ← when array is sorted or almost sorted

Timsort has been Python's standard sorting algorithm since version 2.3

Tim Peters

Divide-and-conquer → Merge Sort



DIVIDE - Trivial
SOLVE - Trivial
COMBINE - cn work,
used extra space

Decrease-and-conquer → Insertion Sort

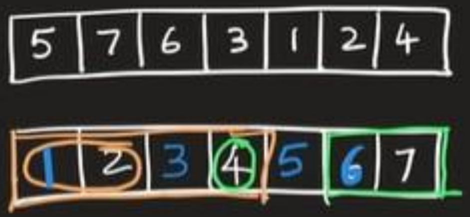
→ Selection Sort
Bubble Sort

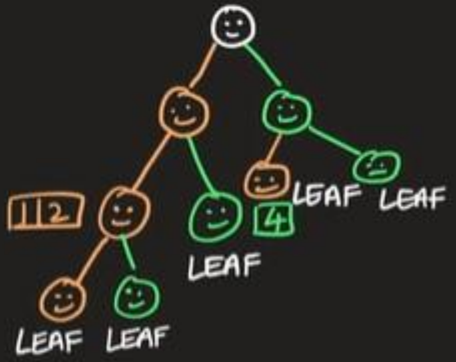
$$T(n) = 2T(n/2) + cn$$
$$= \Theta(n \log n)$$

- Pick an arbitrary value, say the value at index 0, and use it as the PIVOT.

{ik}

and use it as the PIVOT.





DIVIDE - cn

SOLVE - Trivial

COMBINE - Trivial

$$T(n) = cn + T(\text{left partition}) + T(\text{right partition})$$

If pivot is median, the split will be even

LUCKY CASE

BEST CASE

$$T(n) = cn + T(n/2) + T(n/2)$$

$$T(n) = 2T(n/2) + cn$$

$$T(n) = \Theta(n \log n)$$

partition)

If pivot is median, the split will be even

$$T(n) = cn + T(n/2) + T(n/2)$$
$$T(n) = 2T(n/2) + cn$$
$$T(n) = \Theta(n \log n)$$

LUCKY CASE
BEST CASE

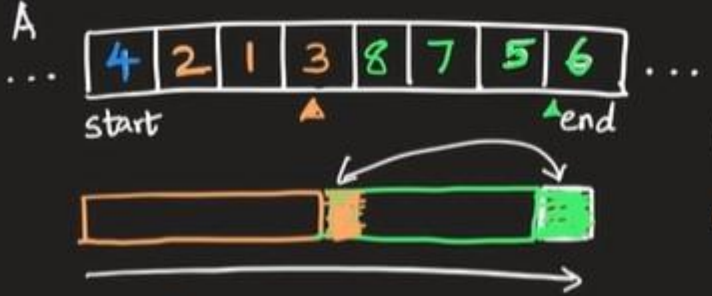
If pivot is the smallest / largest element, the split will be extremely skewed.

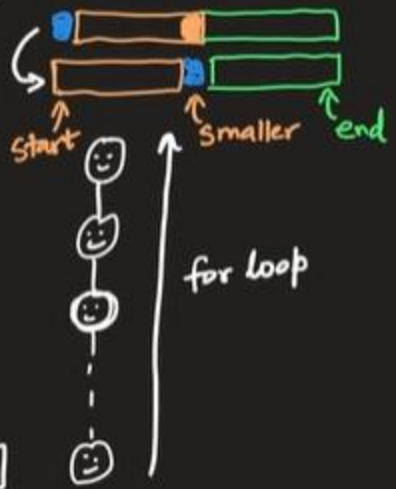
$$T(n) = cn + T(n-1)$$
$$T(n) = \Theta(n^2)$$

WORST CASE

AVERAGE CASE = ? $\Theta(n \log n)$

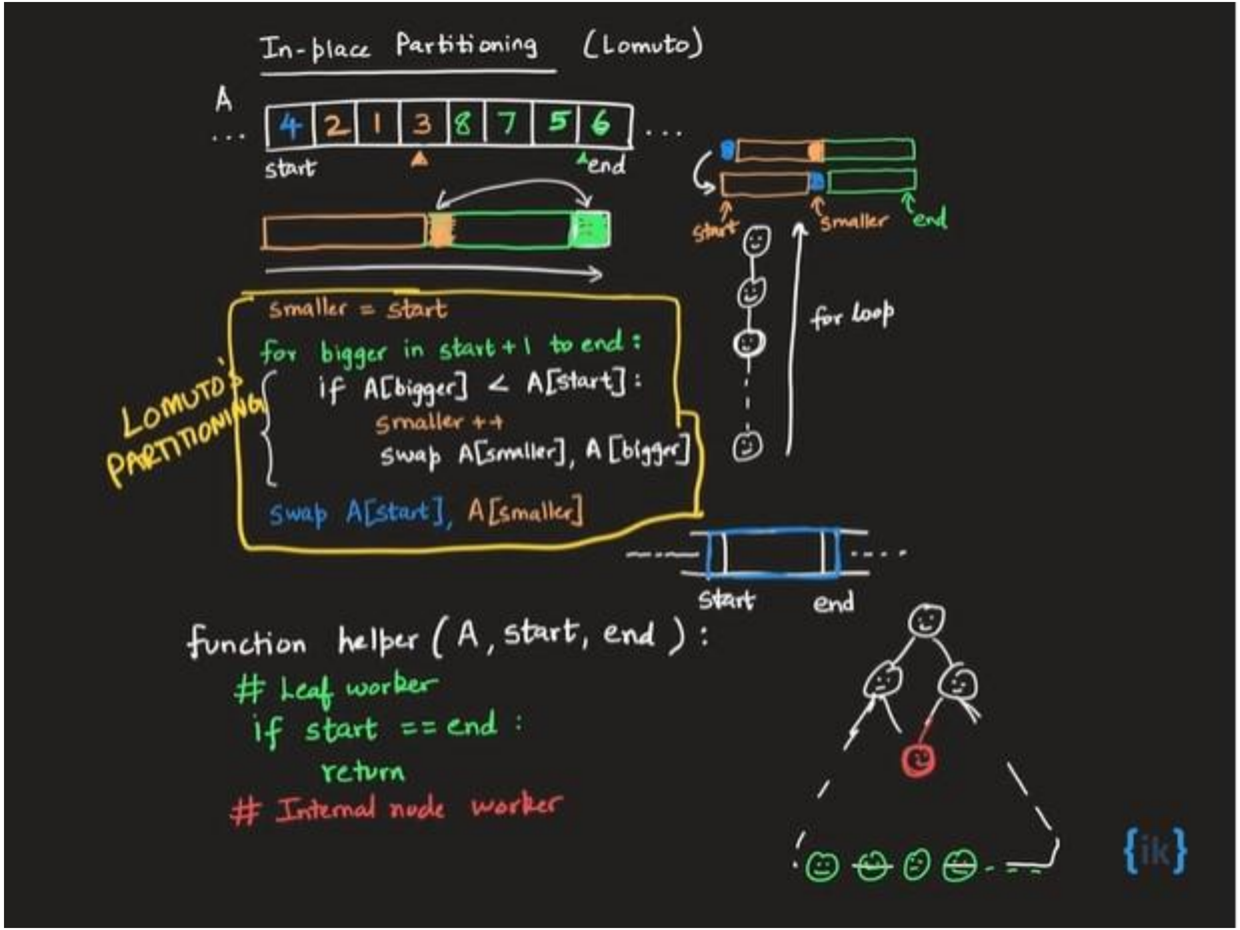
In-place Partitioning (Lomuto)

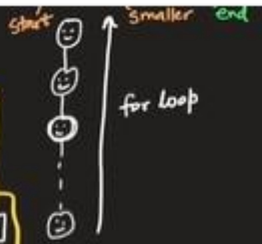





```
smaller = start
for bigger in start+1 to end:
{   if A[bigger] < A[start]:
        smaller++
        swap A[smaller], A[bigger]
}
swap A[start], A[smaller]
```

{ik}

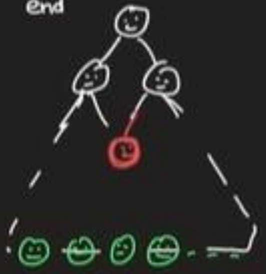

In-place Partitioning (Lomuto)
A
... 4 2 1 3 8 7 5 6 ...
start
end
start
smaller
end
for loop
LOMUTO'S PARTITIONING
smaller = start
for bigger in start+1 to end:
if A[bigger] < A[start]:
smaller ++
swap A[smaller], A[bigger]
swap A[start], A[smaller]
start
end
function helper (A, start, end):
Leaf worker
if start == end :
return
Internal node worker
{ik}



LOMUTO'S PARTITIONING

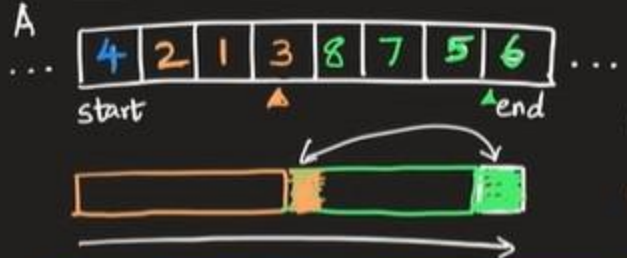
```
smaller = start
for bigger in start+1 to end:
    if A[bigger] < A[start]:
        smaller ++
        swap A[smaller], A[bigger]
swap A[start], A[smaller]
```

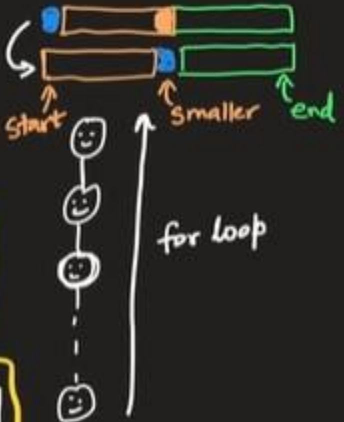

```
function helper(A, start, end):
    # Leaf worker
    if start == end:
        return
    # Internal node worker
```



$\boxed{\text{LOMUTO's partitioning}}$

In-place Partitioning (Lomuto)

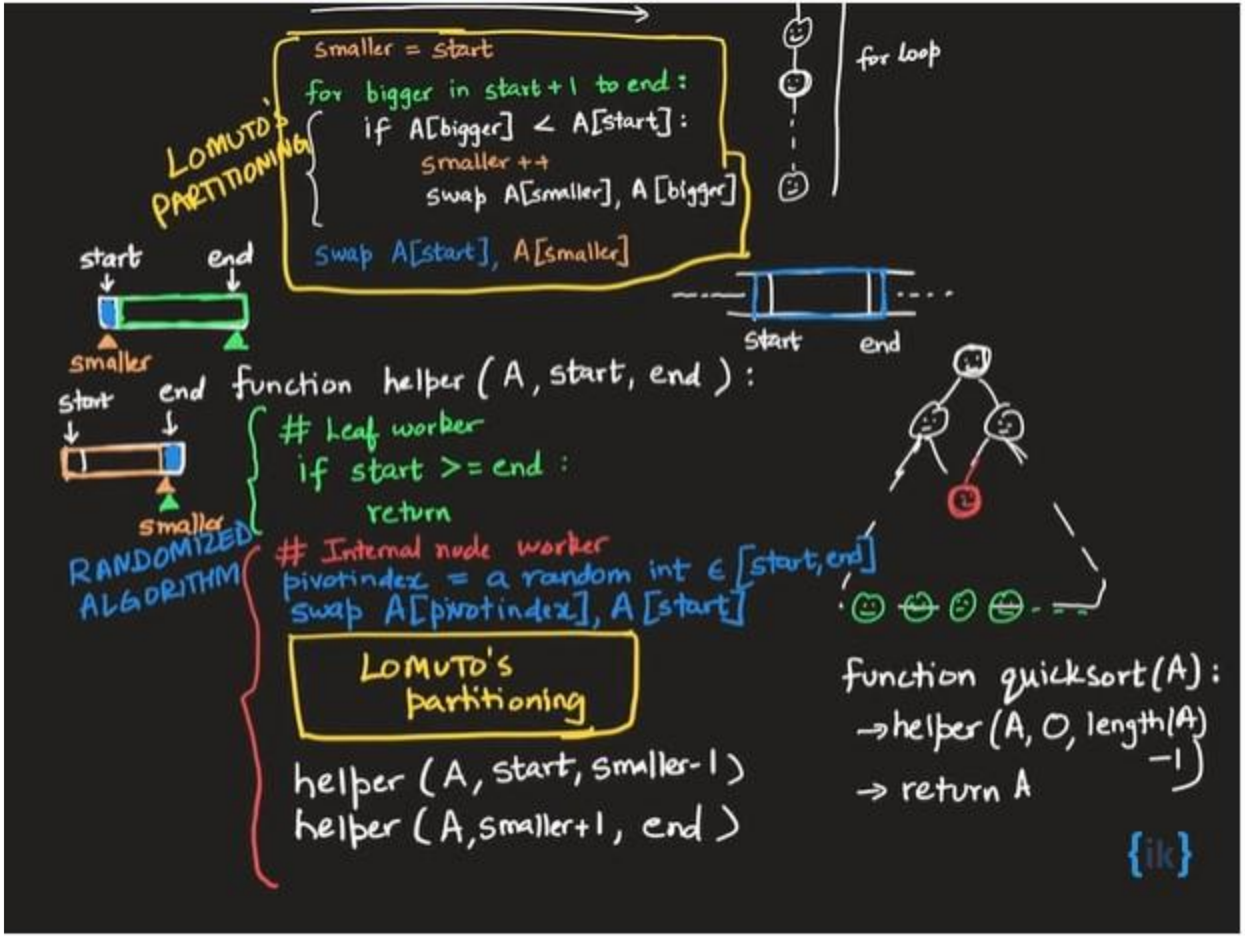




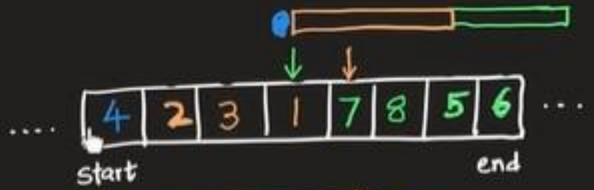
OMUTO'S
RTITIONING

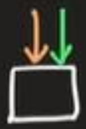
```
smaller = start
for bigger in start+1 to end:
    if A[bigger] < A[start]:
        smaller ++
        swap A[smaller], A[bigger]
swap A[start], A[smaller]
```

function helper (A, start, end):



smaller = start
for bigger in start+1 to end :
if A[bigger] < A[start] :
smaller ++
swap A[smaller], A[bigger]
swap A[start], A[smaller]
LOMUTO'S PARTITIONING
for loop
start
end
smaller
start
end
start
end
smaller
function helper (A, start, end) :
Leaf worker
if start >= end :
return
# Internal node worker
pivotindex = a random int ∈ [start, end]
swap A[pivotindex], A[start]
RANDOMIZED ALGORITHM
LOMUTO'S partitioning
helper (A, start, smaller-1)
helper (A, smaller+1, end)
function quicksort(A) :
→ helper (A, 0, length(A) -1)
→ return A
{ik}





```
smaller = start+1
bigger = end
while smaller ≤ bigger:
    if A[smaller] < A[start]:
        smaller ++
    else if A[bigger] > A[start]:
        bigger --
    else: // both pointers stuck
        swap A[smaller], A[bigger]
        smaller ++, bigger --
swap A[start], A[bigger]
```

HOARE's Partitioning

$T(n) = \Theta(n)$

If partitioning is completely skewed

$$T(n) = cn + T(0) + T(n-1)$$

If partitioning is even (not skewed at all)

$$T(n) = cn + T(n/2) + T(n/2)$$

If partitioning is "half-skewed"

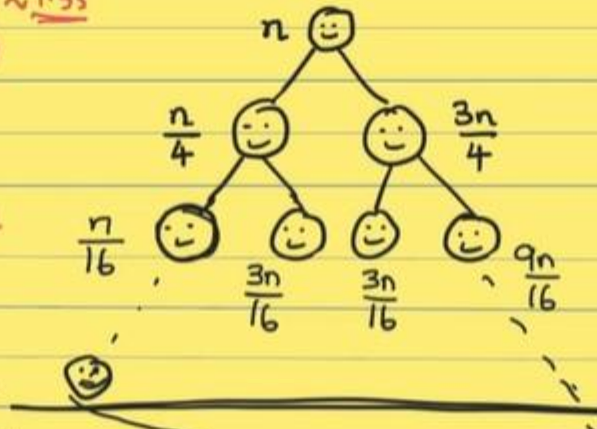
$$T(n) = cn + T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right)$$

If partitioning is "half-skewed"

$$T(n) = cn + T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right)$$

$\sqrt{2} \sim 1.4$

$4/3 \sim 1.33$



0
1
2
⋮
L
L+1
L+2
⋮
h

Per-level work
cn
cn
cn
⋮
cn

Total work is At least $cn(1+L)$
and
at most $cn(1+h)$

$$cn(1+L) \le T(n) \le cn(1+h)$$

$$T(n) = \Theta(n \log n)$$

$$L = \log_4 n = \frac{1}{2}\log_2 n$$

$$h = \log_{4/3} n = \frac{\log_2 n}{\log_2 (4/3)}$$

$$px = \boxed{\log_2 n} = p \log_c n = \boxed{\log_2 c \cdot \log_c n}$$

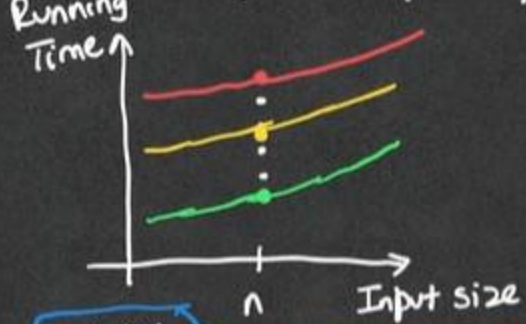
$\log_c n = x$

$\log_2 n$ $\quad c^x = n$

$c = 2^p$ $\quad 2^{px}$

{fk}

$T(n)$ = Average case running time of quicksort on an input of size n



$$\underline{T(n) = \Omega(n \log n)} \quad - (1)$$

$$\underline{T(n) = O(n^2)}$$

CLAIM:

$$\boxed{T(n) = O(n \log n)} \quad - (2)$$

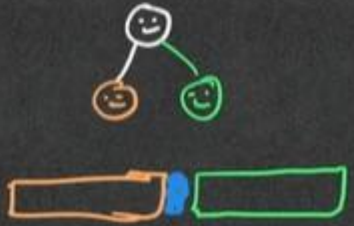
$$\underline{T(n) = \Theta(n \log n)}$$

ze n

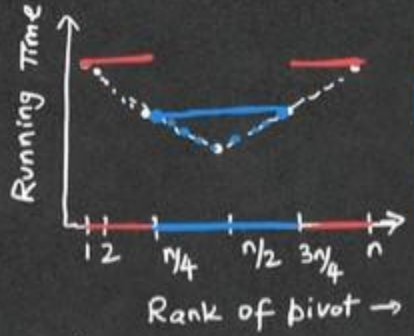
$\underline{T(n) = \Omega(n \log n)}$ — (1)

$\underline{T(n) = O(n^2)}$

(2)



$$T(n) = \begin{cases} cn + T(0) + T(n-1) \\ cn + T(1) + T(n-2) \\ cn + T(2) + T(n-3) \\ \vdots \\ cn + T(n/4) + T(3n/4) \\ \vdots \\ cn + T(n/2) + T(n/2) \\ \vdots \\ cn + T(3n/4) + T(n/4) \\ \vdots \\ cn + T(n-2) + T(1) \\ cn + T(n-1) + T(0) \end{cases}$$



$$T(n) = cn + \frac{1}{n}\Big[\, T(0) + T(n-1) + T(1) + T(n-2) + \cdots + T(n/4) + T(3n/4) + \cdots + T(n/2) + T(n/2) + \cdots + T(3n/4) + T(n/4) + \cdots + T(n-1) + T(0) \,\Big]$$

T(n)

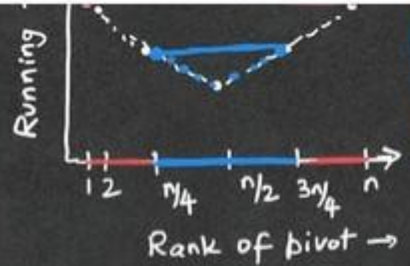
$$cn + T(n-1) + T(0)$$

$$T(n) \leq cn + \frac{1}{n}\left[\frac{n}{2}\cdot T(n) + \frac{n}{2}\cdot\big[T(n/4) + T(3n/4)\big]\right]$$

$$cn + \frac{1}{\cancel{n}}\cdot\frac{\cancel{n}}{2}\big[\, T(n) + T(n/4) + T(3n/4)\big]$$

$$T(n) \leq cn + \underline{\frac{1}{2}T(n)} + \frac{1}{2}\underbrace{\big[T(n/4) + T(3n/4)\big]}_{\text{half-skewed}}$$

$$\frac{1}{2}T(n) \leq cn + \frac{1}{2}\big[T(n/4) + T(3n/4)\big]$$



$$
\begin{aligned}
&+\ T(n/4) + T(3n/4) \\
&\ \vdots \\
&+\ T(n/2) + T(n/2) \\
&\ \vdots \\
&+\ T(3n/4) + T(n/4) \\
&\ \vdots \\
&\left.+\ T(n-1) + T(0)\right\} \to T(n)
\end{aligned}
$$

$$T(n) \leq cn + \frac{1}{n}\left[\frac{n}{2}\cdot T(n) + \frac{n}{2}\cdot\left[T(n/4) + T(3n/4)\right]\right]$$

$$cn + \frac{1}{\cancel{n}}\cdot\frac{\cancel{n}}{2}\left[T(n) + T(n/4) + T(3n/4)\right]$$

$$T(n) \leq cn + \underline{\frac{1}{2}T(n)} + \frac{1}{2}\underbrace{\left[T(n/4) + T(3n/4)\right]}_{\text{half-skewed}}$$

$$\frac{1}{2}T(n) \leq cn + \frac{1}{2}\left[T(n/4) + T(3n/4)\right]$$

$$T(n) \leq c'n + T(n/4) + T(3n/4)$$

$$T(n) = O(n \log n)$$

Merge sort vs Quick sort

Merge Sort is Θ(n log n) for best, average and worst case

Quick Sort is Θ(n log n) for best & average case, but $\Theta(n^2)$ worst-case

But Quick Sort is also in-place, which Merge Sort is not. (although call stack space in Quicksort should also be noted)

{ik}

Running time estimates:

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

If Randomized Quicksort is implemented well, it is more likely that your computer will be struck by a lightning bolt than the Quicksort running in $O(n^2)$ time.

{ik}

Merge sort vs Quick sort

Merge Sort is O(n log n) for best, average and worst case

Quick Sort is O(n log n) for best & average case, but O(n^2) worst-case

Quick Sort runs faster in empirical analysis.

Quick Sort is also in-place, which Merge Sort is not.

But Merge Sort does have one advantage: **stability**

A typical application. First, sort by name; then sort by section.

`Selection.sort(a, Student.BY_NAME);`

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

1st sorting

`Selection.sort(a, Student.BY_SECTION);`

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

2nd sorting

sorted by time sorted by location (not stable) sorted by location (stable)

Rohde	2	A	232-343-5555	343 Forbes

Battle	4	C	874-088-1212	121 Whitman

1st sorting

2nd sorting

sorted by time		sorted by location (not stable)		sorted by location (stable)	
Chicago	09:00:00	Chicago	09:25:52	Chicago	09:00:00
Phoenix	09:00:03	Chicago	09:03:13	Chicago	09:00:59
Houston	09:00:13	Chicago	09:21:05	Chicago	09:03:13
Chicago	09:00:59	Chicago	09:19:46	Chicago	09:19:32
Houston	09:01:10	Chicago	09:19:32	Chicago	09:19:46
Chicago	09:03:13	Chicago	09:00:00	Chicago	09:21:05
Seattle	09:10:11	Chicago	09:35:21	Chicago	09:25:52
Seattle	09:10:25	Chicago	09:00:59	Chicago	09:35:21
Phoenix	09:14:25	Houston	09:01:10	Houston	09:00:13
Chicago	09:19:32	Houston	09:00:13	Houston	09:01:10
Chicago	09:19:46	Phoenix	09:37:44	Phoenix	09:00:03
Chicago	09:21:05	Phoenix	09:00:03	Phoenix	09:14:25
Seattle	09:22:43	Phoenix	09:14:25	Phoenix	09:37:44
Seattle	09:22:54	Seattle	09:10:25	Seattle	09:10:11
Chicago	09:25:52	Seattle	09:36:14	Seattle	09:10:25
Chicago	09:35:21	Seattle	09:22:43	Seattle	09:22:43
Seattle	09:36:14	Seattle	09:10:11	Seattle	09:22:54
Phoenix	09:37:44	Seattle	09:22:54	Seattle	09:36:14

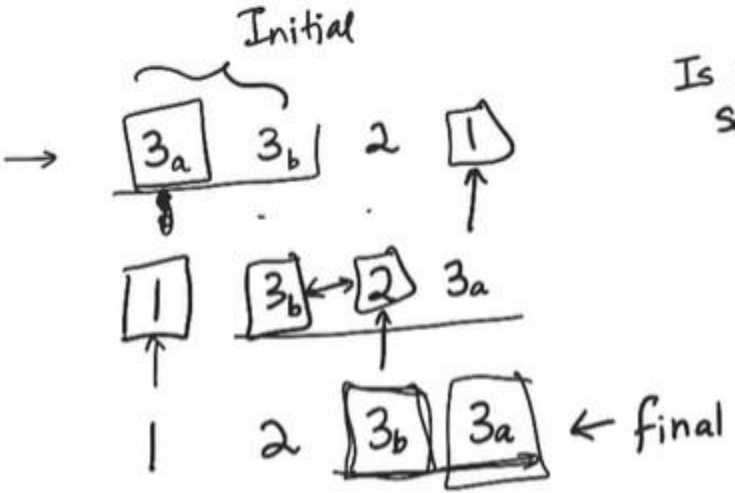
no longer sorted by time

still sorted by time

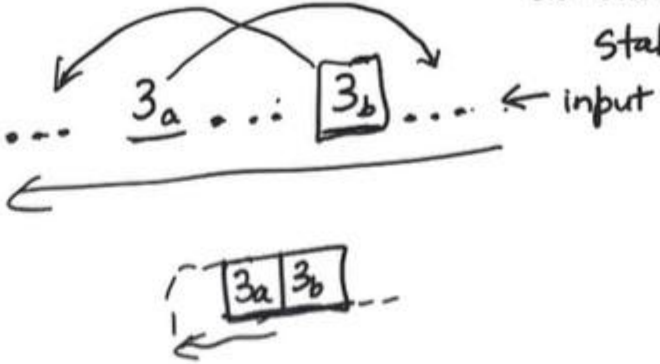
A stable sort preserves the relative order of items with equal keys.

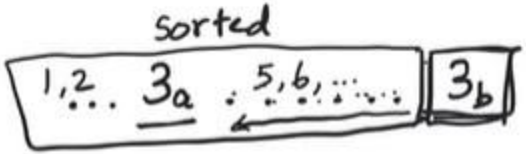
{ik}

Is selection sort stable?

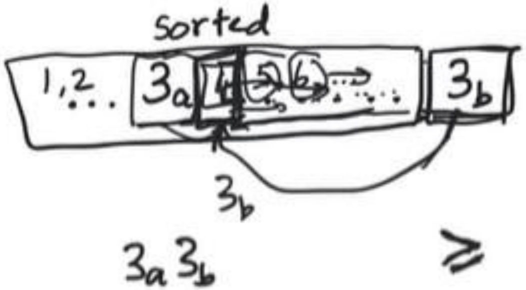


Is bubble sort stable? Yes



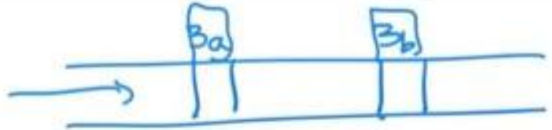


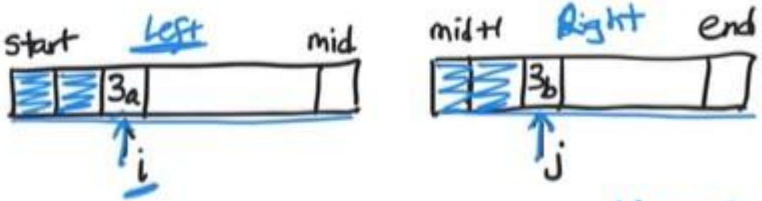
Is insertion sort stable?



Is insertion sort <u>stable</u>?

Yes





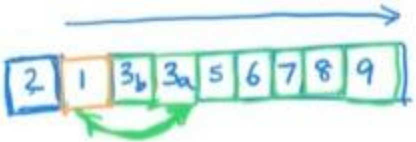
Merge Sort

```
if A[i] ≤ A[j]
    aux.append(A[i])
else:
    aux.append(A[j])
```

B1 B2 aux

```
    aux.append(A[i])
else:
    aux.append(A[j])
```

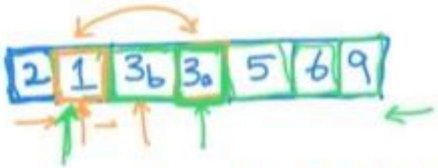
Lomuto's



Is Quicksort stable? No

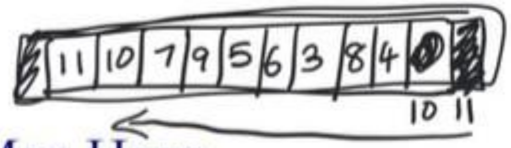
9 10

F G

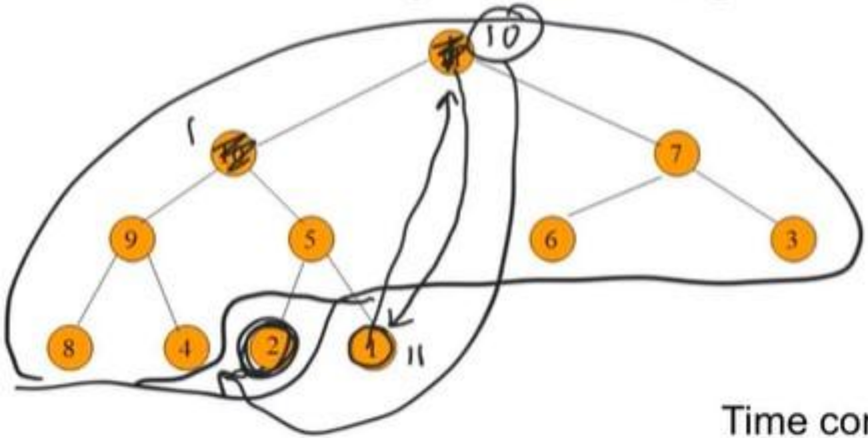


Hoare's partitioning

Find home for 1.



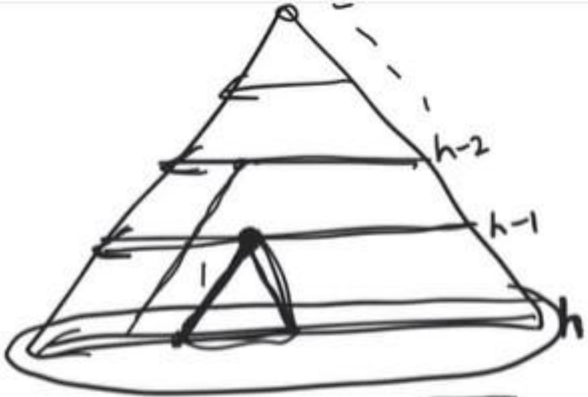
Initializing A Max Heap



Time complexity = ?

Done.

{ik}



Buildheap

# nodes		Time for each node		
2^h	$\times$	0	$=$	
2^{h-1}	$\times$	1	$=$	
2^{h-2}	$\times$	2	$=$	
$\vdots$				
1	$\times$	$\log n$ (h)	$=$	

$$\boxed{T(n)} = 2^h \times 0 + 2^{h-1} \times 1 + 2^{h-2} \times 2 + \dots \dots 1 \times \dots$$

$$\leq 2^h \cdot 0 + 2^{h-1} \cdot 1 + 2^{h-2} \cdot 2 + \dots \dots \infty$$

$$\leq \quad 2^h \cdot 0 + 2 \cdot 1$$

$$1 \times \log n = \quad + 2^{h-2} \cdot 2 + \dots \infty$$

h

$$\underline{T(n)} \leq \underline{2^h \cdot 0 + 2^{h-1} \cdot 1 + 2^{h-2} \cdot 2} + \dots 2^{h-i} \cdot i + \dots \underline{+ \dots \infty}$$

$$\sum_{i=0}^{\infty} 2^{h-i} \cdot i = \sum_{i=0}^{\infty} \frac{2^h}{2^i} \cdot i$$

$$= 2^h \boxed{\sum_{i=0}^{\infty} \frac{i}{2^i}} \qquad \frac{+1}{\times 2}$$

Arithmetico-Geometric series

{ik}

$$T(n) \le 2^{h}\cdot 0 + 2^{h-1}\cdot 1 + 2^{h-2}\cdot 2 + \dots 2^{h-i}\cdot i + \dots + \dots \infty$$

$$\sum_{i=0}^{\infty} 2^{h-i}\cdot i = \sum_{i=0}^{\infty} \frac{2^{h}}{2^{i}}\cdot i$$

= ?

$$= 2^{h} \boxed{\sum_{i=0}^{\infty} \frac{i}{2^{i}}}$$

$$\frac{+1}{\times 2}$$

Arithmetico-Geometric series

$$S = \frac{0}{2^0} + \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \dots \infty$$

?

$$= 2\left|\boxed{\sum_{i=0} \frac{i}{2^i}}\right. \qquad \times 2$$

Arithmetico-Geometric series

$$\frac{S}{2} = \frac{0}{2^0} + \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots \infty$$

$$\underbrace{S - \frac{S}{2}}_{\frac{S}{2}} = \left[\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots\right.$$

$$S = 1 + \frac{1}{2} + \frac{1}{2^2} + \cdots$$

?

$$= \frac{2}{\quad}\boxed{\sum_{i=0} \frac{i}{2^i}}$$

$\times 2$

Arithmetico-Geometric series

$$\frac{S}{2} = \frac{0}{2^1} + \frac{1}{2^2} + \frac{2}{2^3} + \dots \infty$$

$$S = \frac{0}{2^0} + \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \dots \infty$$

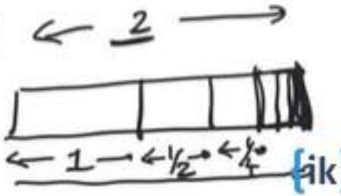
$$\underbrace{S - \frac{S}{2}}_{\frac{S}{2}} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \dots$$

$$\boxed{S = 1 + \frac{1}{2} + \frac{1}{2^2} + \dots \infty}$$

$$\frac{S}{2} = \frac{1}{2} + \frac{1}{2^2} + \dots \infty$$

$$S - \frac{S}{2} = 1 = \frac{S}{2} \Rightarrow \boxed{S = 2}$$

$$\boxed{1} + \boxed{\frac{1}{2}} + \boxed{\frac{1}{4}} + \boxed{\frac{1}{8}} + \dots$$



{ik}

nodes   time for each node   T(n)

$2^h \times 0 \Rightarrow 2^h \times 0 + 2^{h-1} \times 1 + 2^{h-2} \times 2 + \dots$

$2^{h-1} \quad T(n) \leq 2^h \cdot 0 + 2^{h-1} + 2^{h-2} \times 2 + \dots + \infty$

$2^{h-2} \times 2 \quad + \dots 2^{h-i} \cdot i + \dots$

$\vdots \qquad \leq 2^h \cdot 0 + 2^{h-1} \cdot 1$

$1 \times \log n \qquad \sum_{i=0}^{\infty} 2^{h-i} \cdot i + \sum_{i=0}^{\infty} \frac{2^h}{2^i} \cdot i \dots \infty$

h

$T(n) \leq 2^h \boxed{\sum_{i=0}^{\infty} \frac{i}{2^i}} = 2$ +1   ×2

Arithmetico-Geometric series

$T(n) \leq 2^h \cdot 2$

$S = \frac{0}{2^0} + \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \dots \infty$

: ?

{ik}

$$T(n) \le 2^h \cdot 2 = \boxed{2^{h+1}} \sim n$$

$$T(n) \le \underline{n}$$

$$\boxed{T(n) = O(n)}$$

$$2^{h+1} - 1 = n$$

Binary Heap (MaxHeap)

Insert → $O(\log n)$

Delete → $O(\log n)$

$\Big\}\ O(n \log n)$

Increase

Change priority → $O(\log n)$

Decrease

Buildheap → $O(n)$

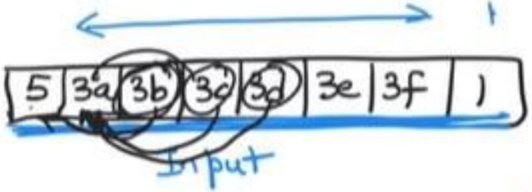
Heapsort

$\boxed{O(n)} + O(n \log n)$

$= O(n \log n)$

In-place? Yes

Stable? No



{ik}

	Worst-case
Selection Sort	$O(n^2)$
Bubble Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Merge Sort	$O(n \log n)$ ✓
Quicksort	$O(n^2)$ ✓
Heapsort	$O(n \log n)$ ✓

Lower bounds on Sorting

Comparison-based sorts $\geq n \log n$

All the algorithms we have seen use $\boxed{\text{comparison operations}}$.

If we can get a lower bound on just the number of comparison operations, that would also be a lower bound on the entire algorithm.

{ik}

a	b	c

DECISION TREE

for visualizing the execution of a sorting algorithm.

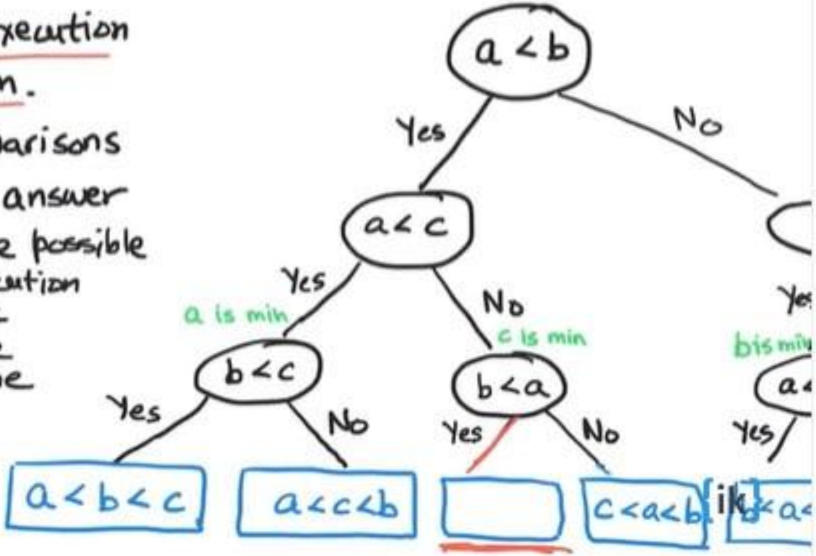
Internal nodes = comparisons

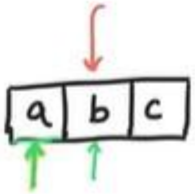
Leaf nodes = Found answer

Root-to-leaf path = One possible execution

Path length = Running time

Height of tree = worst-case running time

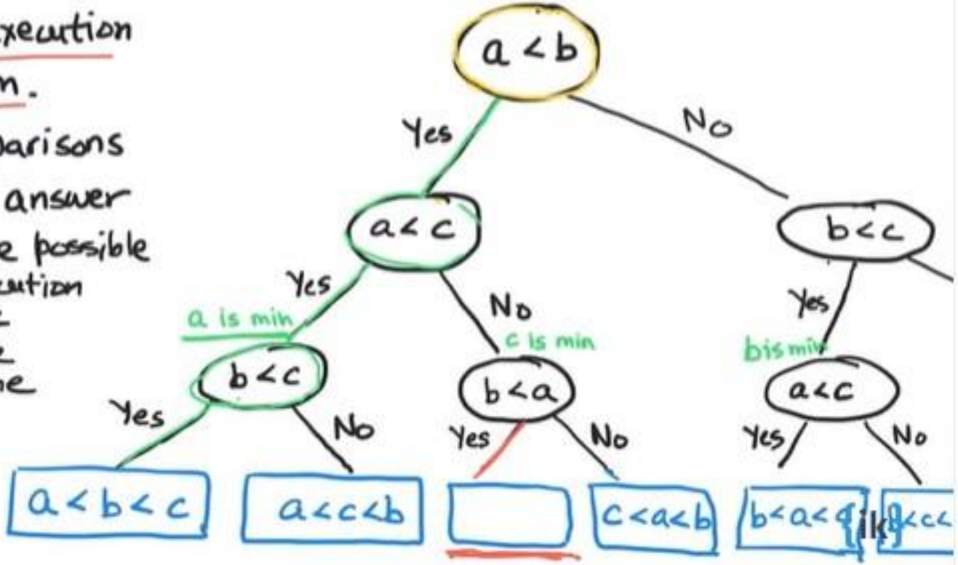


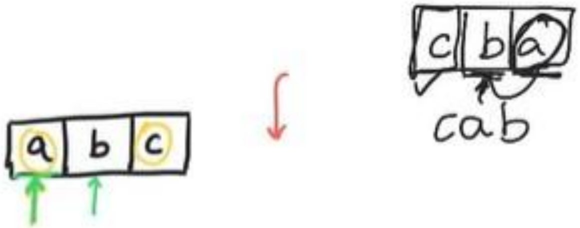


ION TREE

isualizing the execution
sorting algorithm.

al nodes = comparisons
odes = Found answer
-leaf path = One possible execution
th = Running time
tree = Worst-case running time





CISION TREE

visualizing the execution

a sorting algorithm.

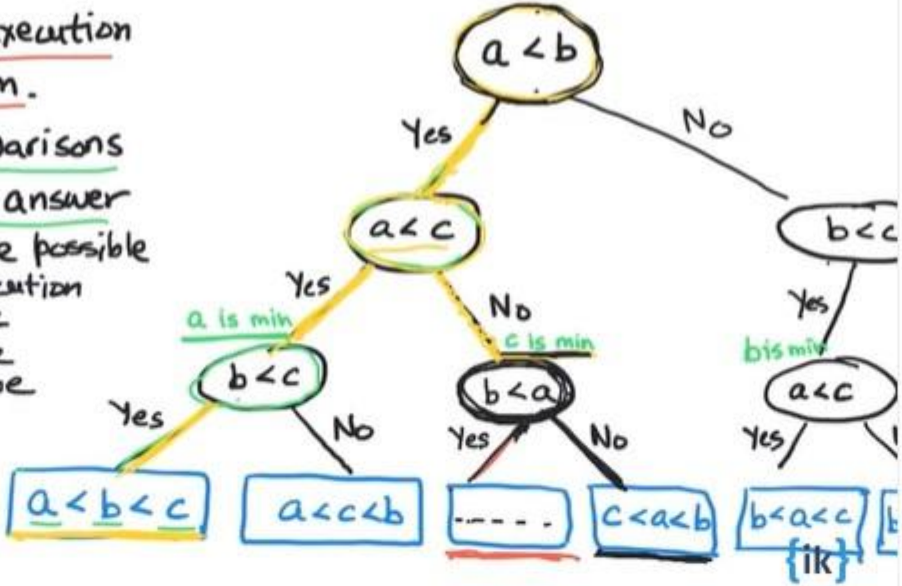
ernal nodes = comparisons

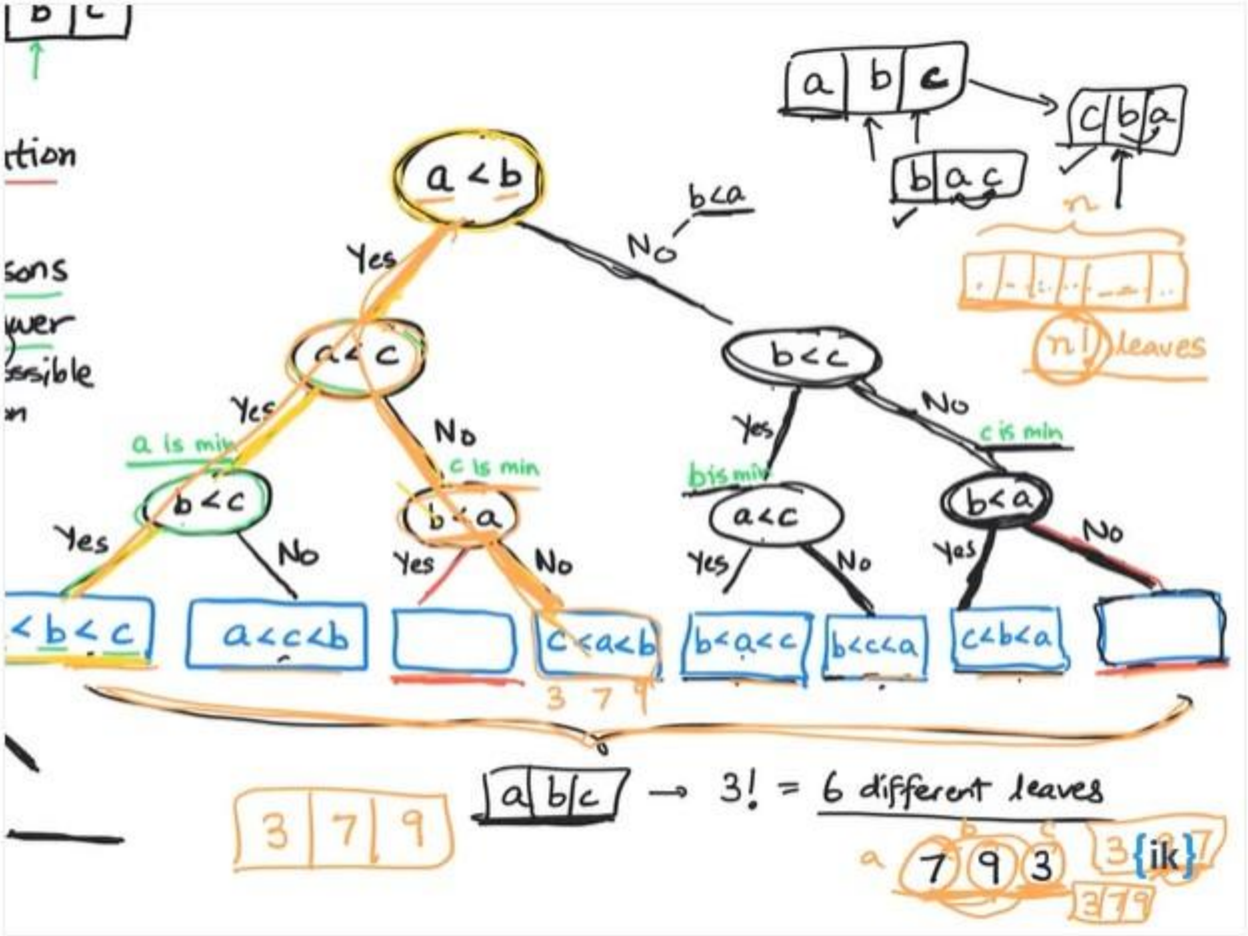
f nodes = Found answer

t-to-leaf path = One possible execution

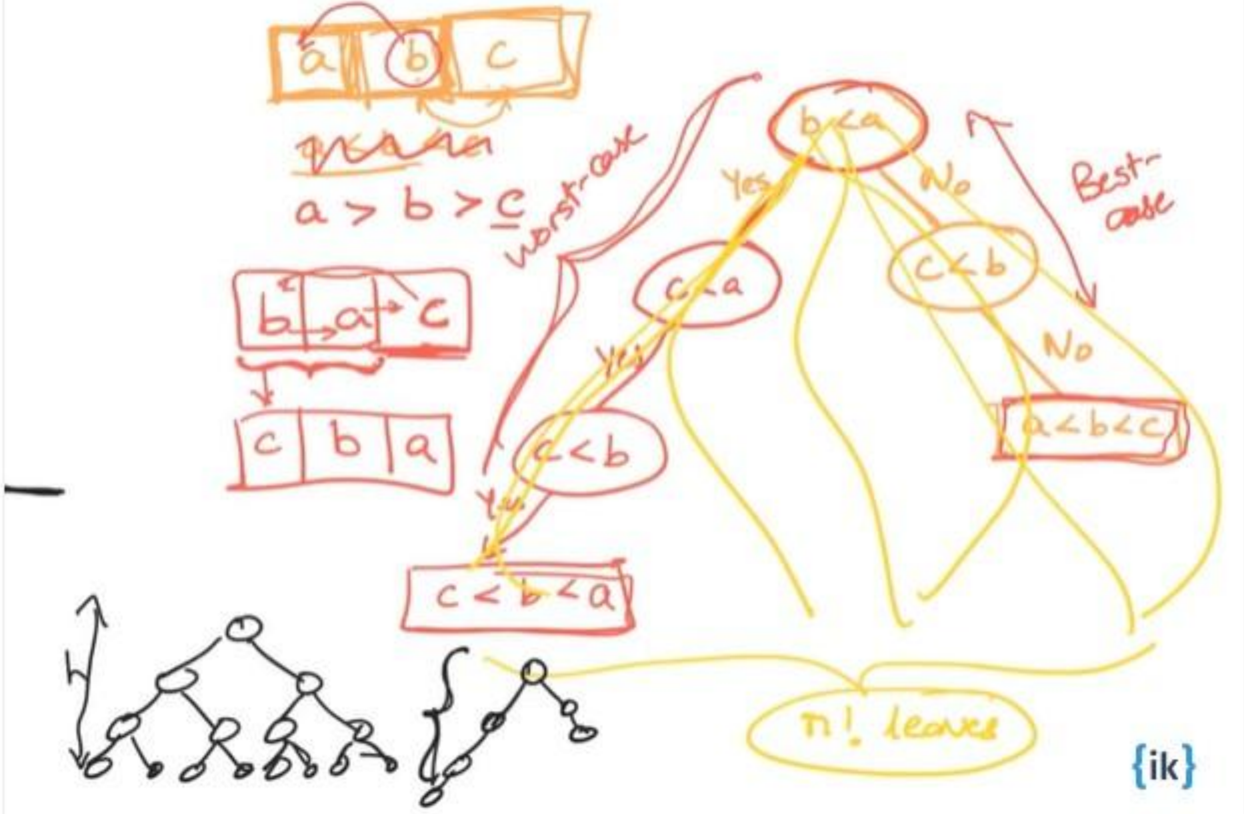
length = Running time

t of tree = Worst-case running time





a < b
Yes
No
b<a
a<c
b<c
a is min
c is min
b is min
b<c
b<a
a<c
b<a
<b<c
a<c<b
c<a<b
b<a<c
b<c<a
c<b<a
n! leaves
a b c → 3! = 6 different leaves
3 7 9
7 9 3


a b c
a > b > c
worst-case
b a c
c b a
b<a
Yes
No
Best-case
c<a
c<b
Yes
No
a<b<c
c<b
Yes
c<b<a
h
n! leaves

$$h = \boxed{\log(n!)}$$

$$n \cdot (n-1) \cdot (n-2) \cdots 1$$

$$n \cdot n \cdots n$$

$$n! < n^n$$

$$\log n! < n \log n$$

$$n \cdot (n-1) \cdot (n-2) \cdots (n/2) \cdots$$

$$\cancel{1 \times 2 \times 3 \times \cdots (n/2)} \times (n/2+1) \times \cdots n$$

$$n! > \left(\frac{n}{2}+1\right)\left(\frac{n}{2}+2\right) \cdots n$$

$$n/2 \quad n/2 \quad n/2$$

$$n! > n/2 \cdot n/2 \cdot n/2 \cdots = (n/2)^{n/2}$$

$$\log n! > \frac{n}{2} \log\left(\frac{n}{2}\right)$$

$$= \frac{n}{2}\left[\log n - 1\right]$$

$$> \approx \boxed{\frac{1}{2} n \log n}$$

Height of decision tree $> \boxed{\frac{1}{2} n \log n}$

$$T(n) > \boxed{c\, n \log n}$$

$$= \Omega(n \log n)$$

O(k) auxiliary space

O(k) time + O(n)

k

1	1
2	2
3	1
4	0
5	1
6	1
7	0
8	0
9	1

Age count array

1, 2, 2, 3, 5, 6, 9

O(n)

O(k)

Sorting an array with many entries of 'small' values

You are given an array of student objects. Each student has an integer-valued age field that is to be treated as a key. Rearrange the elements of the array so that the ages appear in sorted order.

O(k) auxiliary space

$O(n + k)$ → n^2

9	6	3	5	2	1	2

O(n)

Counting Sort

n^2

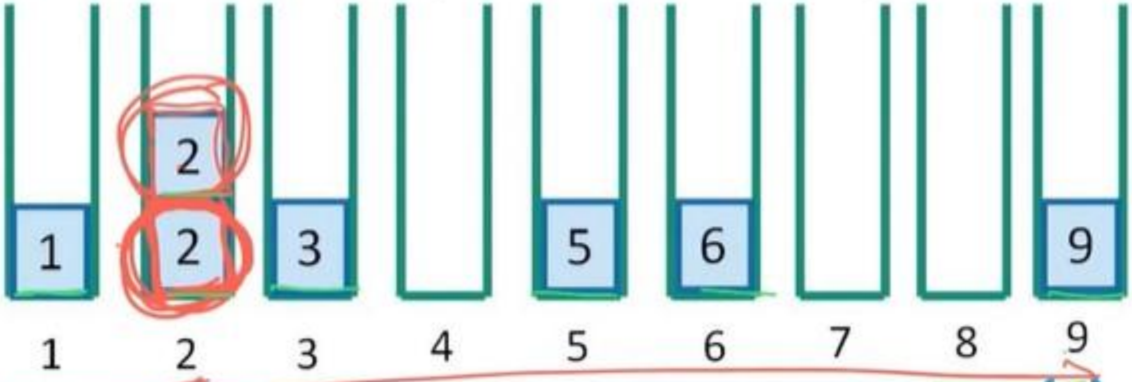
Counting Sort

$O(n)$

$\boxed{O(k+n)}$

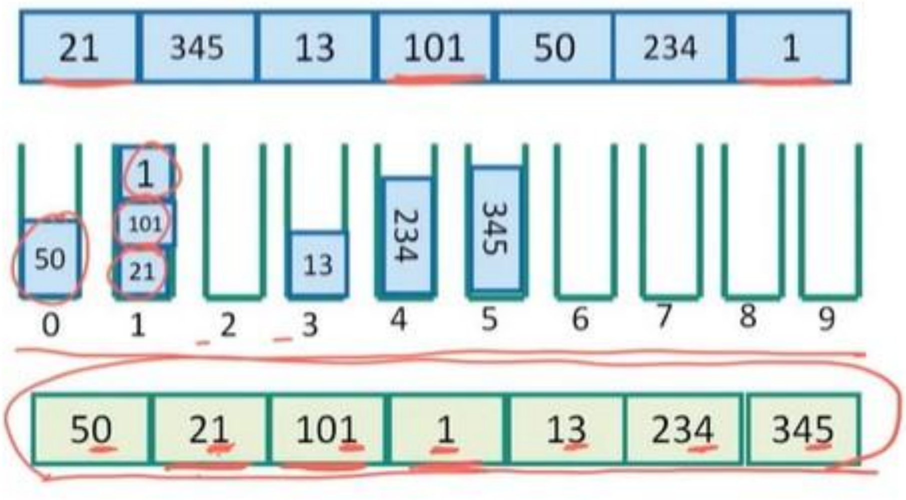
$O(n+k)$

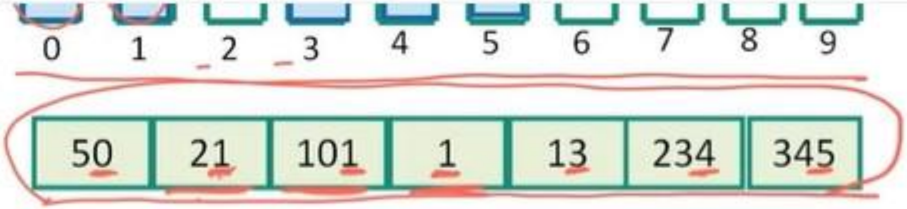
Just count how many students of each age there are!



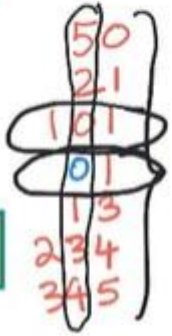
21
345
13
101
50
234
1

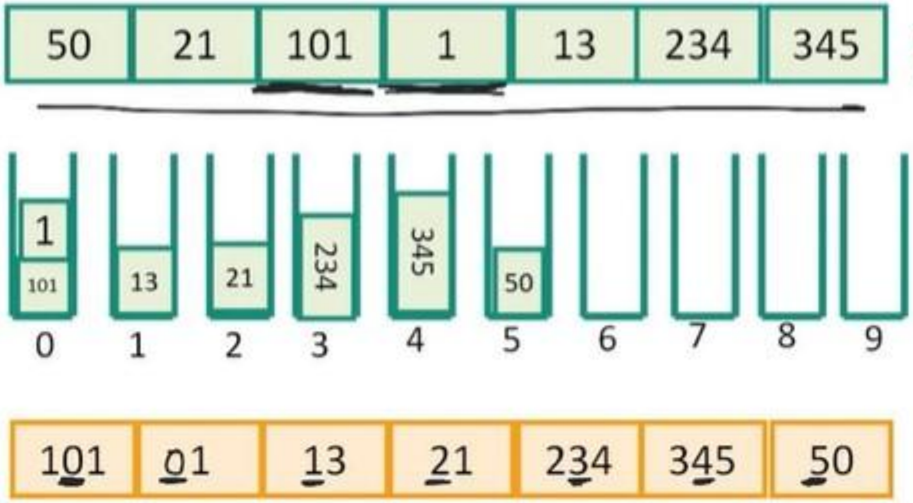
Step 1: CountingSort on least significant digit





Step 2: CountingSort on the 2^nd least sig. digit

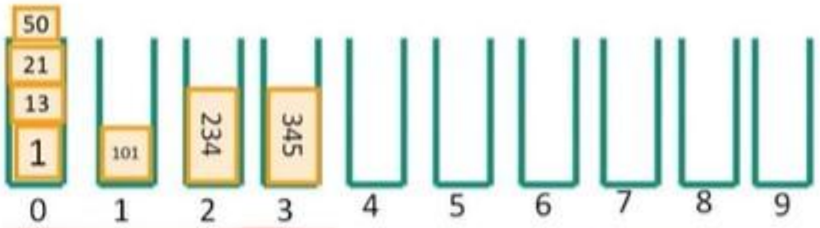




101	01	13	21	234	345	50

Step 3: CountingSort on the 3rd least sig. digit

101	1	13	21	234	345	50



101
001
013
021
234
345
050

1	13	21	50	101	234	345

It worked!!

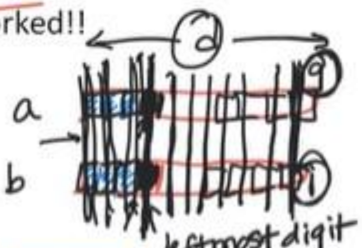
{ik}

1	13	21	50	101	234	345

It worked!!

Why does this work?

$a < b$



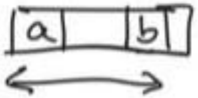
leftmost digit where they differ

Original array:

21	345	13	101	50	234	1

Next array is sorted by the first digit.

50	21	101	1	13	234	345



Next array is sorted by the first two digits.

101	01	13	21	234	345	50

$O(n)$

$k = 10$

Next array is sorted by all three digits.

001	013	021	050	101	234	345

Sorted array

d phases $\Rightarrow O(dn) = \overset{d=3}{O(n)}$

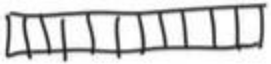
$O(n+k) = O(n)$

base

Radix Sort

n integers, d digits, in some base (base-10 in the example)

21	345	13	101	50	234	1



21
345
13
101

- It matters how we pick the base.

$n\,\boxed{\log n}$ ≥ 20 $\sim 20n$

$n = (10^3) \times (10^3) \approx 2^{10} \times 2^{10} = \boxed{2^{20}}$

n = 1 million

32 bits

$\boxed{32n}$

base-256

“I think the bubble sort is the wrong way to go”

0 – 255

$4n$