Divide And Conquer

Merge Sort, Quick Sort

Sorting

- Arrange an unordered list of elements in some order.
- Some common algorithms
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort

Sorting

- Important primitive
- For today, we’ll pretend all elements are distinct.

6	4	3	8	1	5	2	7

1	2	3	4	5	6	7	8

Length of the list is n

Insertion Sort (Recap)

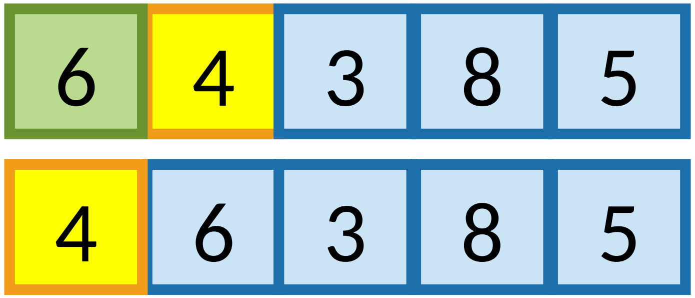
```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Insertion Sort

example


```
Insertion-Sort(A, n)
    for i = 1 to n - 1
        key = A[i]
        j = i - 1
        while j >= 0 and A[j] > key
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key
```

Start by moving A[1] toward the beginning of the list until you find something smaller (or can't go any further):



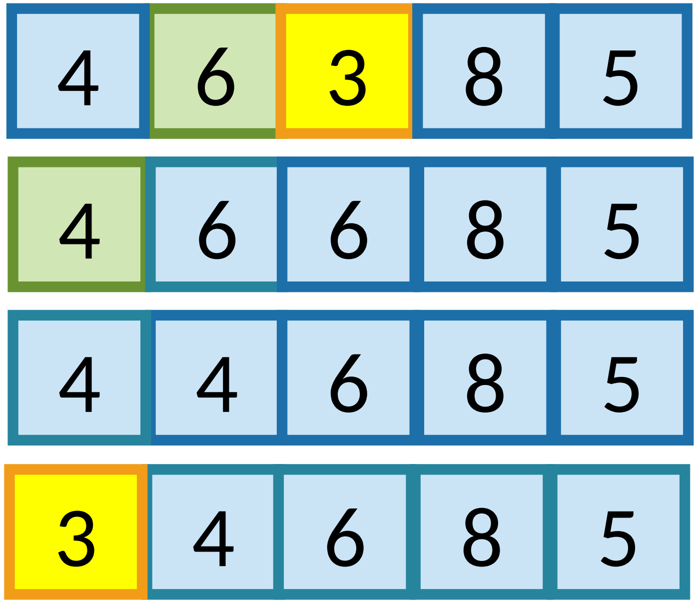
Insertion Sort

example


```
Insertion-Sort(A, n)
    for i = 1 to n - 1
        key = A[i]
        j = i - 1
        while j >= 0 and A[j] > key
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key
```

Then move A[2]:

key = 3



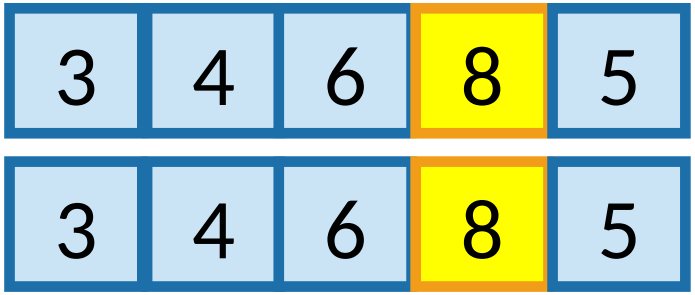
Insertion Sort

example


```
Insertion-Sort(A, n)
    for i = 1 to n - 1
        key = A[i]
        j = i - 1
        while j >= 0 and A[j] > key
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key
```

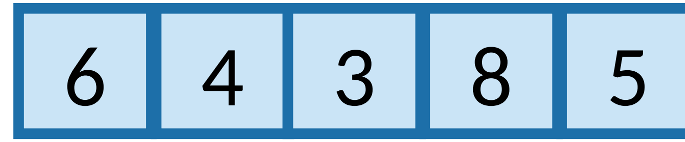
Then move A[3]:

key = 8



Insertion Sort

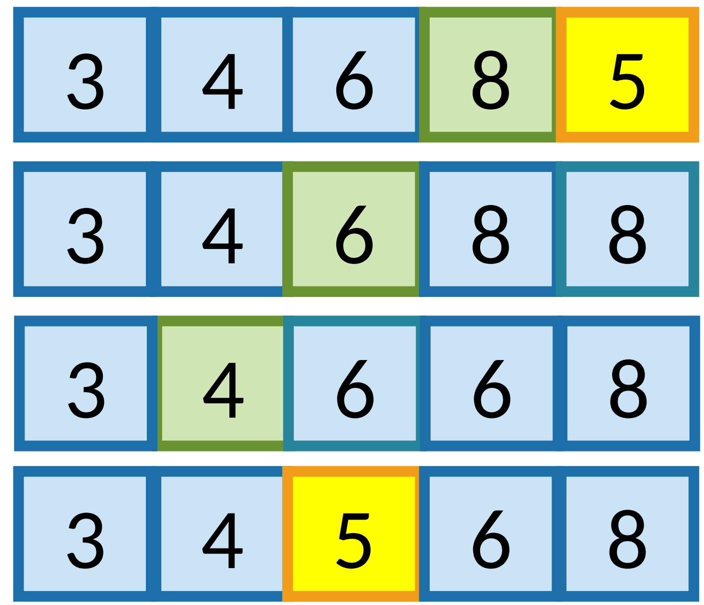
example




```
Insertion-Sort(A, n)
    for i = 1 to n - 1
        key = A[i]
        j = i - 1
        while j >= 0 and A[j] > key
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key
```

Then move A[4]:

key = 5



Insertion Sort

example

6	4	3	8	5

Start by moving A[1] toward the beginning of the list until you find something smaller (or can't go any further):

6	**4**	3	8	5
4	6	3	8	5

Then move A[2]:

4	6	**3**	8	5
3	4	6	8	5

Then move A[3]:

3	4	6	**8**	5
3	4	6	**8**	5

Then move A[4]:

3	4	6	8	**5**
3	4	**5**	6	8

Then we are done!

Why does this work?

- Say you have a sorted list, 3 4 6 8, and another element 5.
- Insert 5 right after the largest thing that's still smaller than 5. (Aka, right after 4).
- Then you get a sorted list: 3 4 5 6 8

This sounds like a job for…

Proof By Induction!

Outline of a proof by induction

Let `A` be a list of length `n`

- Base case:
  - `A[:1]` is sorted at the end of the 0'th iteration. ✓
- Inductive Hypothesis:
  - `A[:i+1]` is sorted at the end of the i^{th} iteration (of the outer loop).
- Inductive step:
  - For any `0 < k < n`, if the inductive hypothesis holds for `i=k-1`, then it holds for `i=k`.
  - Aka, if `A[:k]` is sorted at step `k-1`, then `A[:k+1]` is sorted at step k (previous slide)
- Conclusion:
  - The inductive hypothesis holds for i = 0, 1, …, n-1.
  - In particular, it holds for i=n-1.
  - At the end of the n-1'st iteration (aka, at the end of the algorithm), `A[:n] = A` is sorted.
  - That's what we wanted! ✓

Worst-case Analysis

- In this class we will use worst-case analysis:
  - We assume that a “bad guy” produces a worst-case input for our algorithm, and we measure performance on that worst-case input.

- How many operations are performed by the insertion sort algorithm on the worst-case input?

How fast is InsertionSort?

- Let's count the number of operations!

```
def InsertionSort(A):
   for i in range(1,len(A)):
      current = A[i]
      j = i-1
      while j >= 0 and A[j] > current:
         A[j+1] = A[j]
         j -= 1
      A[j+1] = current
```

By my count*…

- $2n^2 - n \ - 1$ variable assignments
- $2n^2 - n \ - 1$ increments/decrements
- $2n^2 - 4n + 1$ comparisons
- …

*A complete count of the operation will be insignificant from later discussion.

In this class we will use…

- **Big-Oh notation!**
- Gives us a meaningful way to talk about the running time of an algorithm, independent of programming language, computing platform, etc., without having to count all the operations.

Main idea:

Focus on how the runtime **scales** with n (the input size).

Some examples…

(Heuristically: only pay attention to the largest function of n that appears.)

Number of operations	Asymptotic Running Time
$\frac{1}{10} \cdot n^2 + 100$	$O(n^2)$
$0.063 \cdot n^2 - .5\, n + 12.7$	$O(n^2)$
$100 \cdot n^{1.5} - 10^{10000}\sqrt{n}$	$O(n^{1.5})$
$11 \cdot n \log(n) + 1$	$O(n \log(n))$

We say this algorithm is "asymptotically faster" than the others.

Informal definition for O(...)

- Let $T(n)$, $g(n)$ be functions of positive integers.
  - Think of $T(n)$ as a runtime: positive and increasing in n.

- We say "$T(n)$ is $O\big(g(n)\big)$" if:

for all large enough n,

$T(n)$ is at most some constant multiple of $g(n)$.

Here, "constant" means "some number that doesn't depend on n."

Formal definition of O(...)

- Let $T(n)$, $g(n)$ be functions of positive integers.
  - Think of $T(n)$ as a runtime: positive and increasing in n.

- Formally,

$$T(n) = O\big(g(n)\big)$$

"If and only if"

$$\Leftrightarrow$$

"For all"

$$\exists c > 0, n_0 \;\; s.t. \;\; \forall n \geq n_0,$$

$$T(n) \leq c \cdot g(n)$$

"There exists"

$\Omega(\ldots)$ means a lower bound

- We say "$T(n)$ is $\Omega\big(g(n)\big)$" if, for large enough n, $T(n)$ is at least as big as a constant multiple of $g(n)$.

- Formally,

$$T(n) = \Omega\big(g(n)\big)$$
$$\Longleftrightarrow$$
$$\exists c > 0\,, n_0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$c \cdot g(n) \leq T(n)$$

Switched these!!

Θ(…) means both!

- We say "$T(n)$ is $\Theta(g(n))$" iff both:

$$T(n) = O\left(g(n)\right)$$

and

$$T(n) = \Omega\left(g(n)\right)$$

Insertion Sort: running time

```
def InsertionSort(A):
    for i in range(1,len(A)):
        current = A[i]
        j = i-1
        while j >= 0 and A[j] > current:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = current
```

n-1 iterations of the outer loop

In the worst case, about n iterations of this inner loop

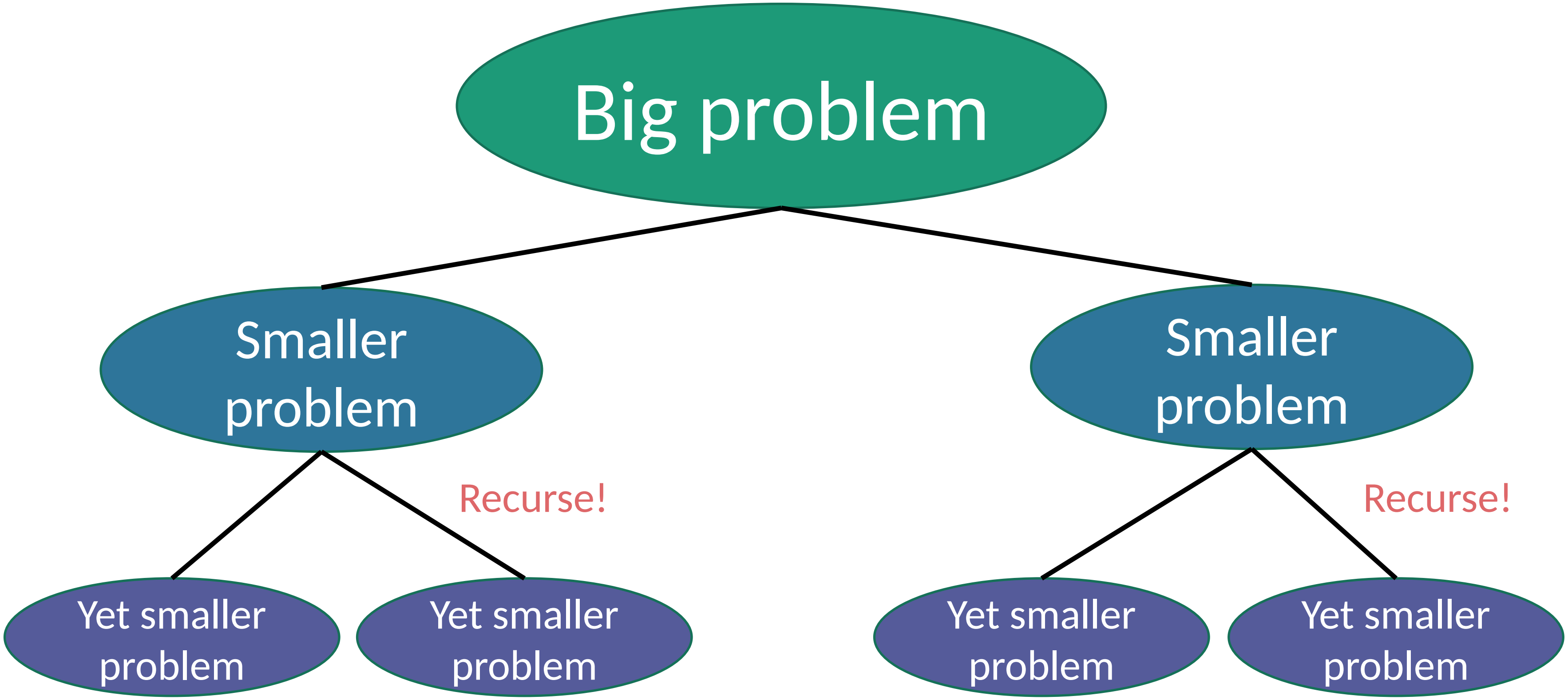
The running time of insertion sort is $O(n^2)$.

InsertionSort is an algorithm that correctly sorts an arbitrary n-element array in time $O(n^2)$.

Can we do better?

Can we do better?

- MergeSort: a divide-and-conquer approach



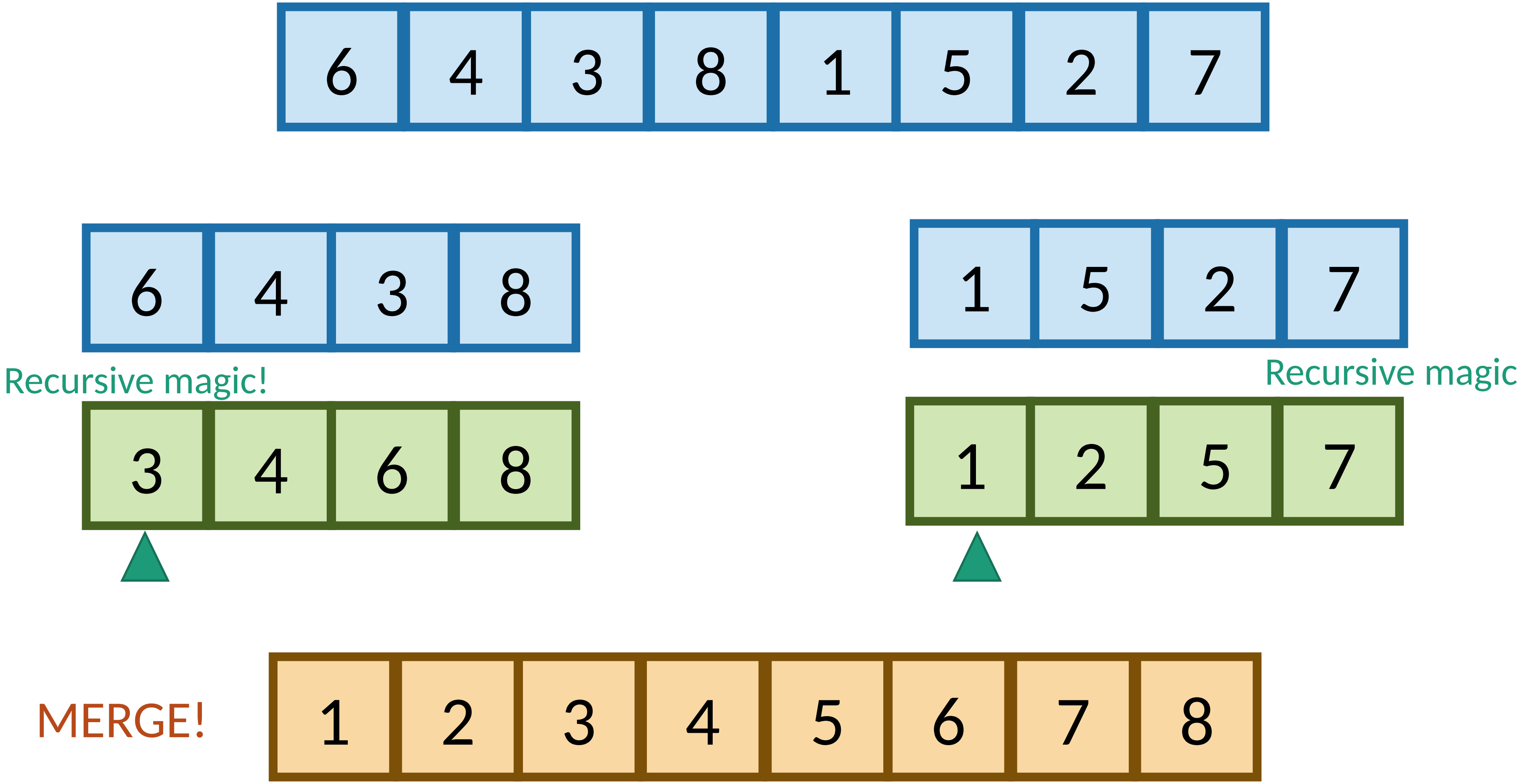
Divide-And-Conquer

- **Divide**
  - Divide the problem into one or more smaller instances of the same problem.
- **Conquer**
  - Solve them smaller problems recursively.
- **Combine**
  - Merge/ combine the solutions to solve the original problem.

Why insertion sort works? (Recap)

- Say you have a sorted list, | 3 | 4 | 6 | 8 | , and another element | 5 | .
- Insert | 5 | right after the largest thing that's still smaller than | 5 | . (Aka, right after | 4 |).
- Then you get a sorted list: | 3 | 4 | 5 | 6 | 8 |
- What if you have two sorted lists? | 3 | 4 | 6 | 8 | | 2 | 5 | 9 | 11 |

MergeSort



6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
3 4 6 8
Recursive magic!
1 2 5 7
MERGE!
1 2 3 4 5 6 7 8

MergeSort Pseudocode

MERGESORT(A):

- n = length(A)
- **if** n ≤ 1:
  - **return** A
- L = MERGESORT(A[0 : n/2])
- R = MERGESORT(A[n/2 : n])
- **return** MERGE(L, R)

If A has length 1,
It is already sorted!

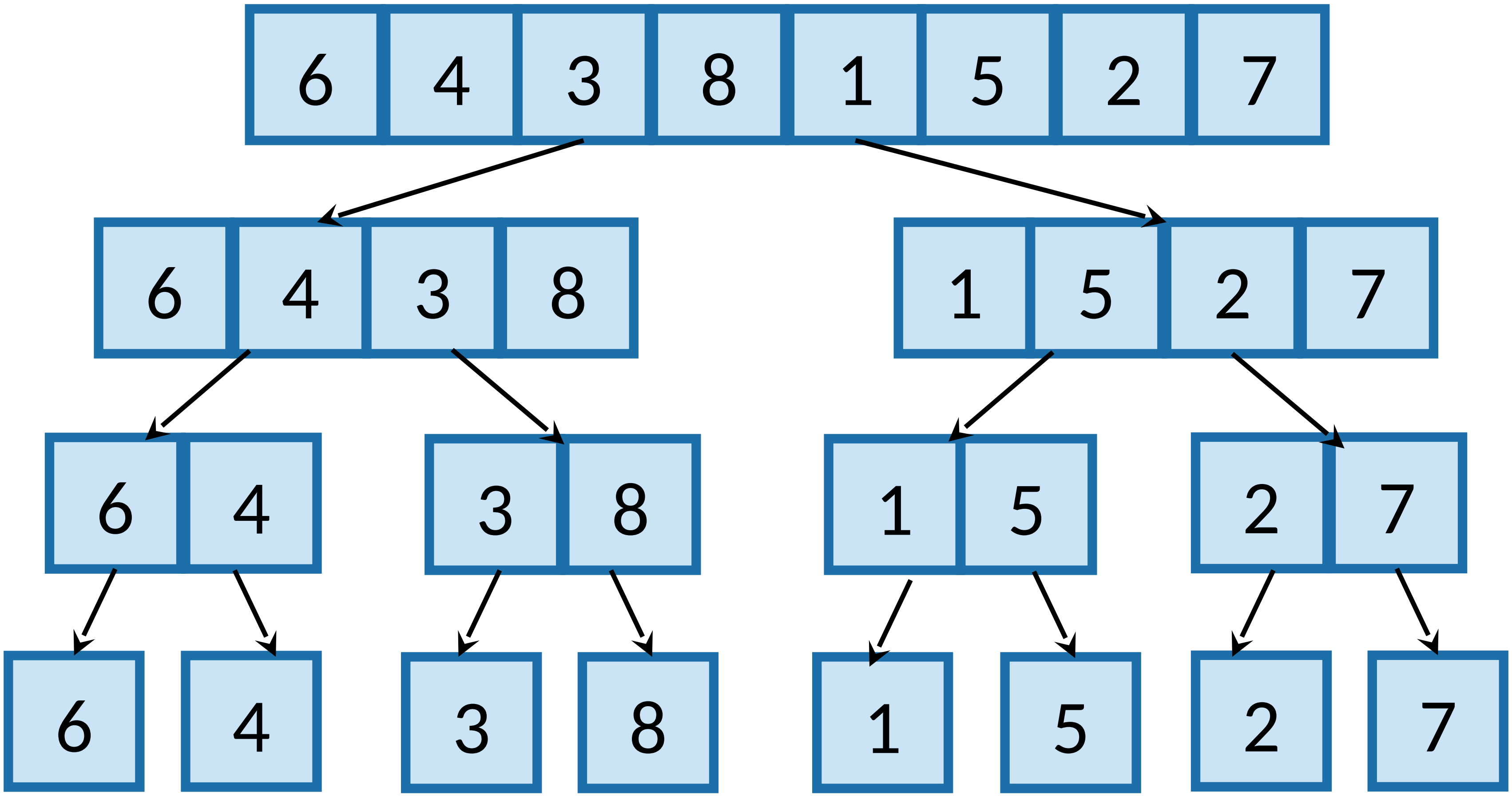
Sort the left half

Sort the right half

Merge the two halves

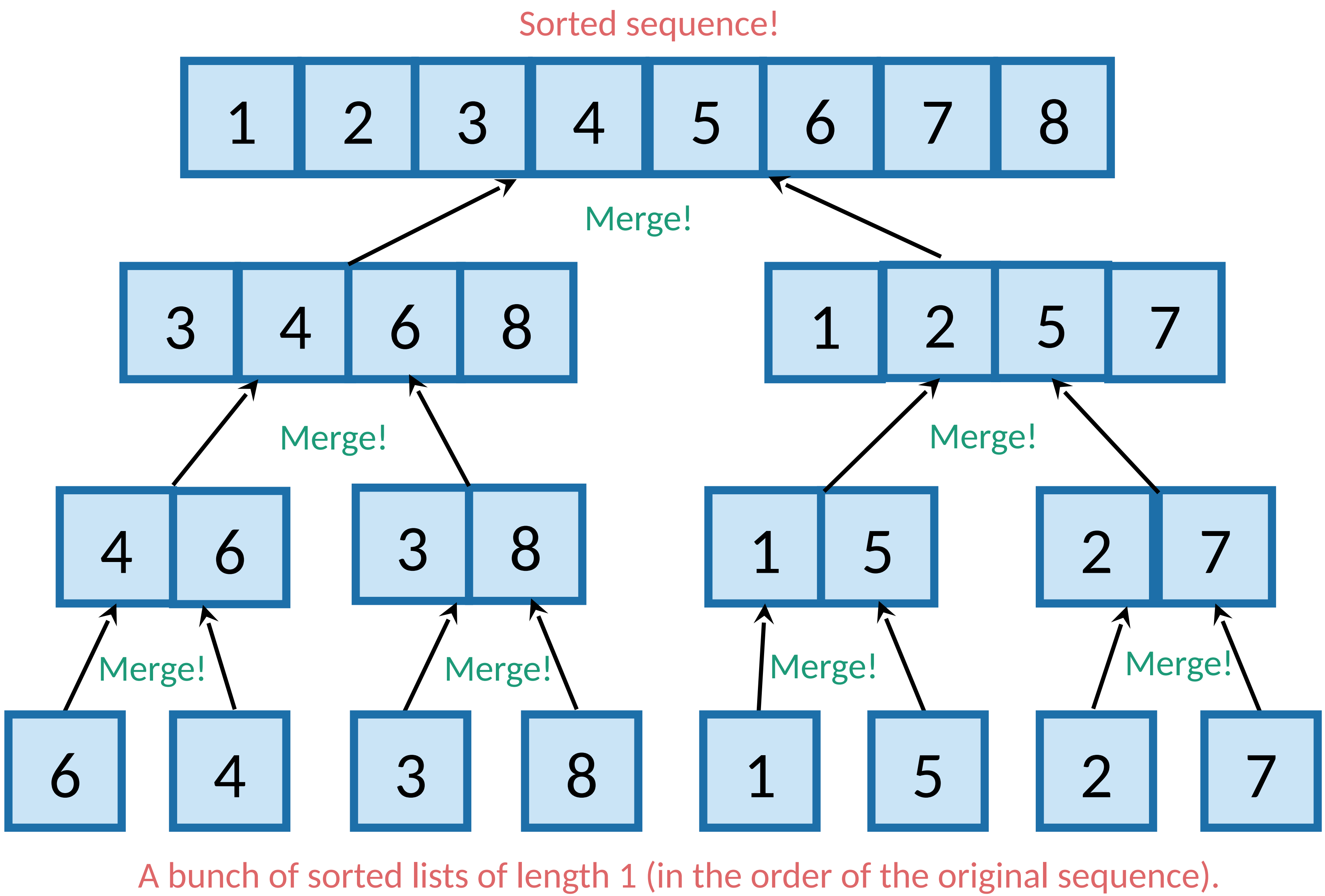
What actually happens?

First, recursively break up the array all the way down to the base cases



This array of length 1 is sorted!

Then, merge them all back up!



Does it work?

- Yet another job for proof by induction!!!
  - Try it yourself.

Assume that n is a power of 2 for convenience.

It's fast

- CLAIM:

  MergeSort runs in time $O(n\log(n))$

- Proof coming soon.
- But first, how does this compare to InsertionSort?
  - Recall InsertionSort ran in time $O(n^2)$.
  - $\log(n)$ grows much more slowly than n
  - $n\log(n)$ grows much more slowly than n^2

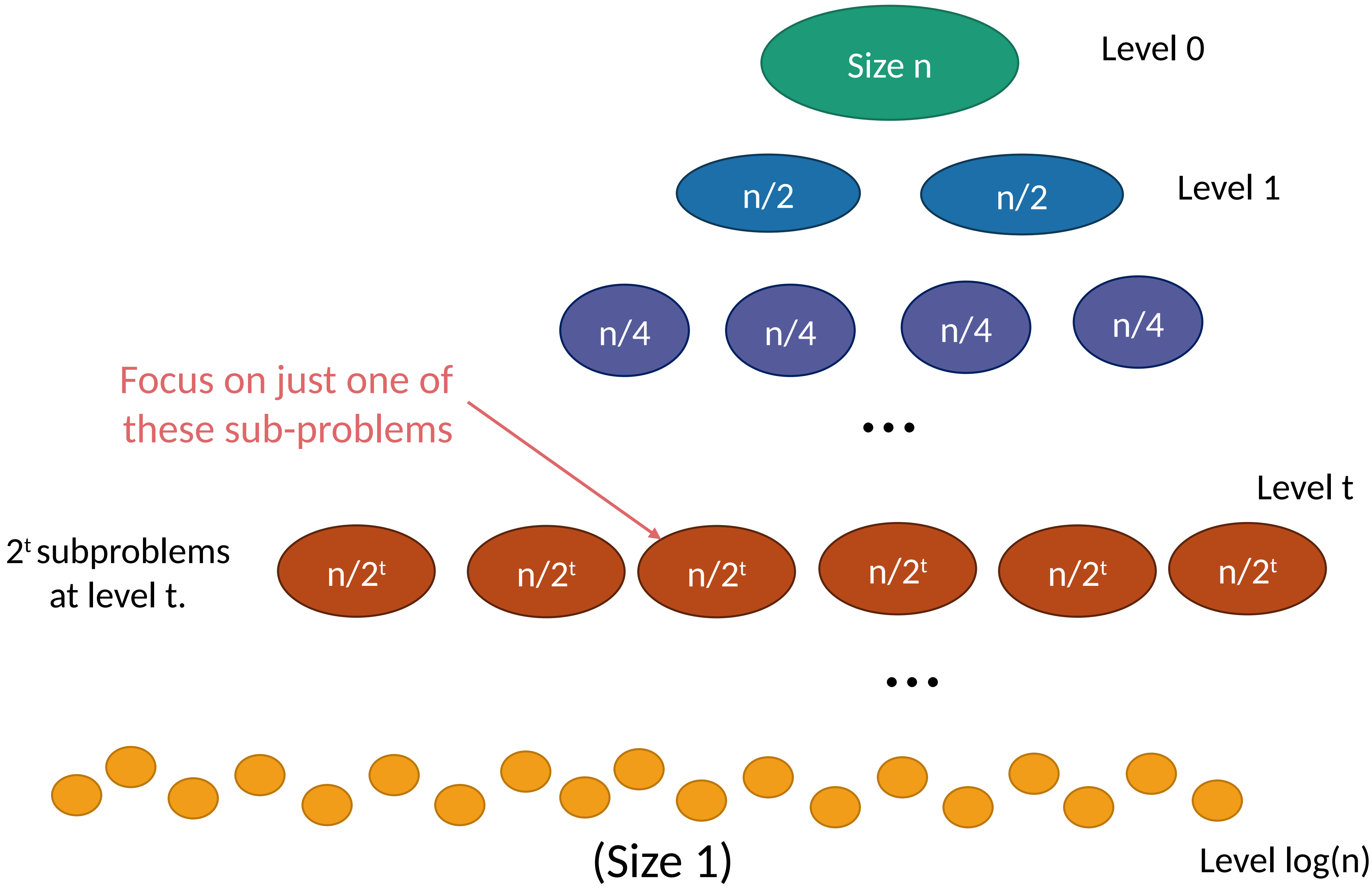
Assume that n is a power of 2 for convenience.

Now let's prove the claim

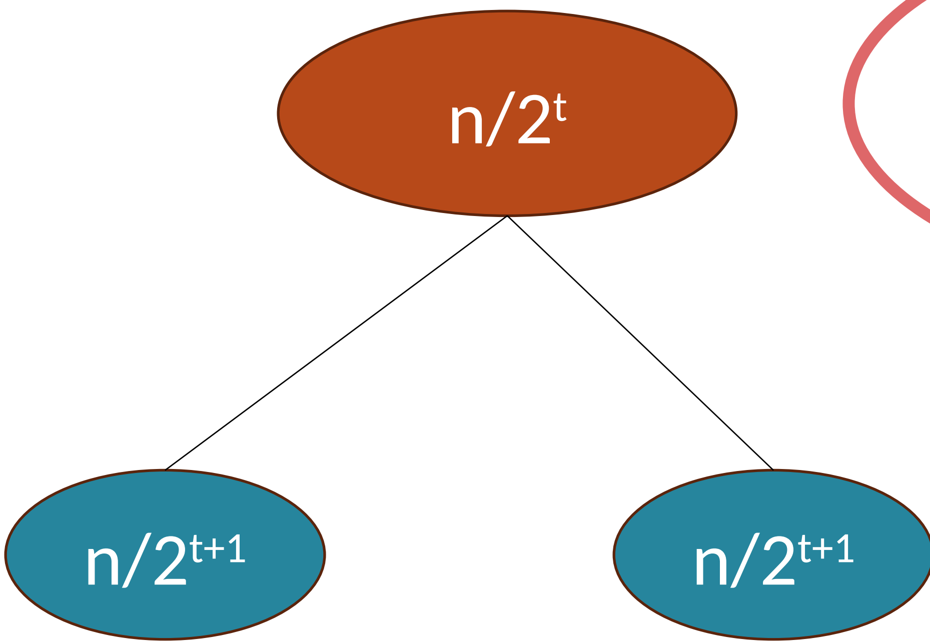
- CLAIM:

MergeSort runs in time $O(n \log(n))$

Let’s prove the claim



How much work in this sub-problem?



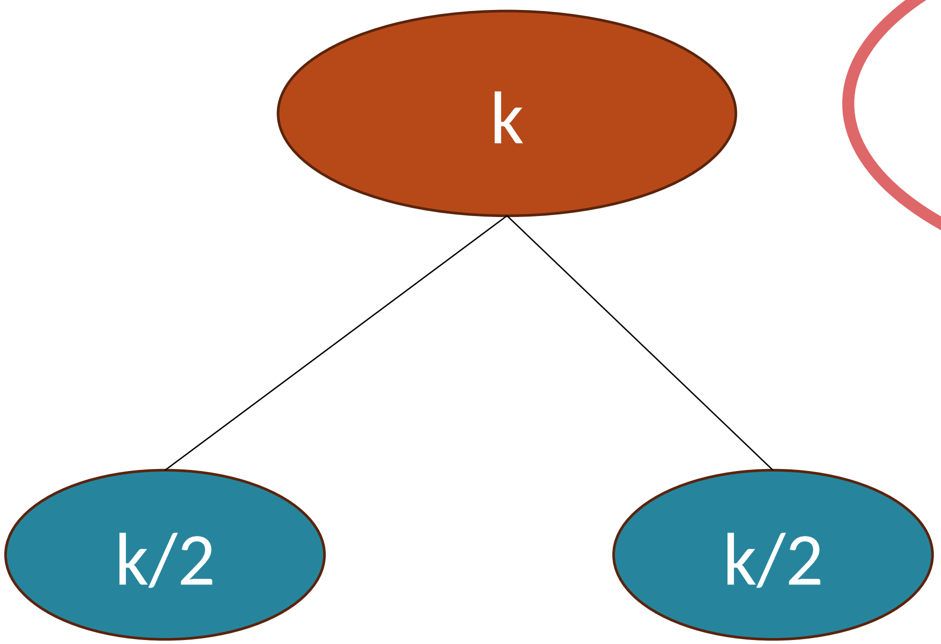
Time spent MERGE-ing the two subproblems

+

Time spent within the two sub-problems

How much work in this sub-problem?

Let $k=n/2^t$...

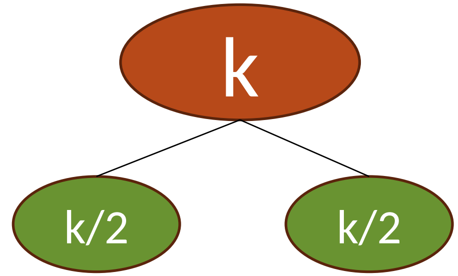


Time spent MERGE-ing the two subproblems

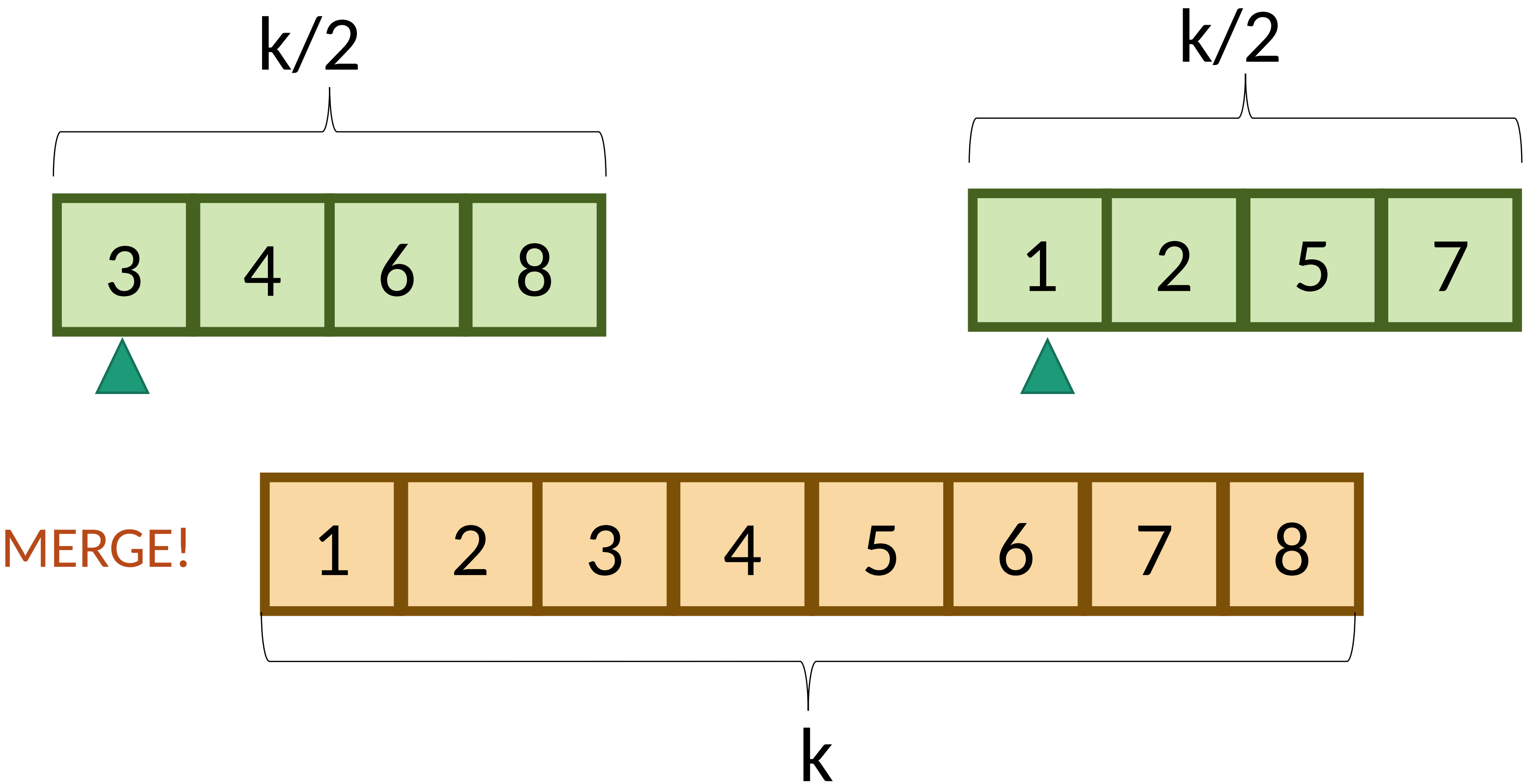
+

Time spent within the two sub-problems

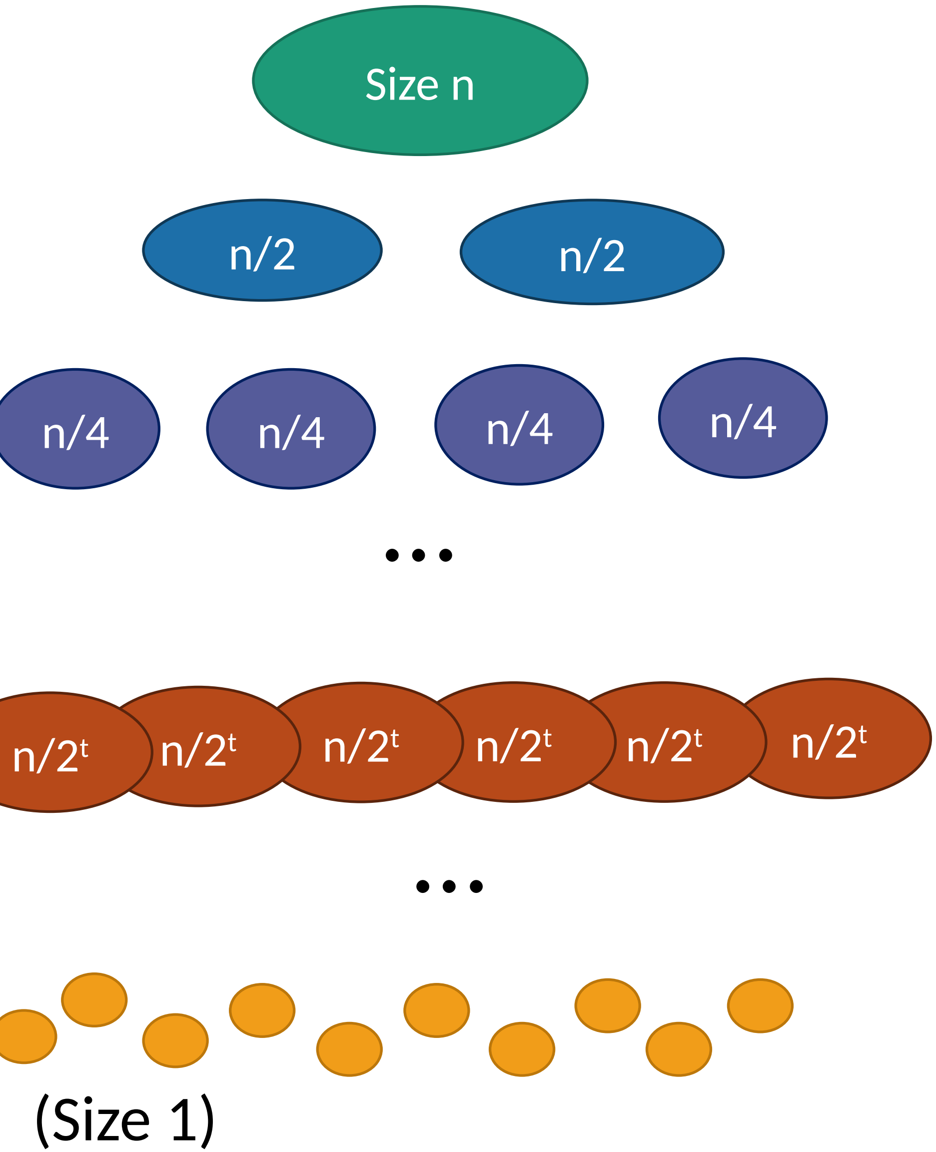
How long does it take to MERGE?



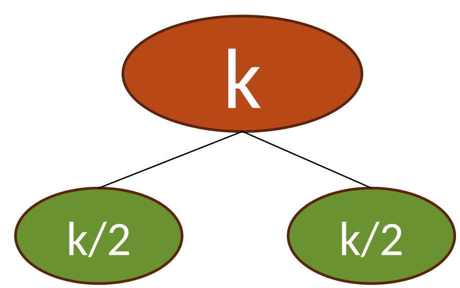
Answer: It takes time O(k), since we just walk across the list once.



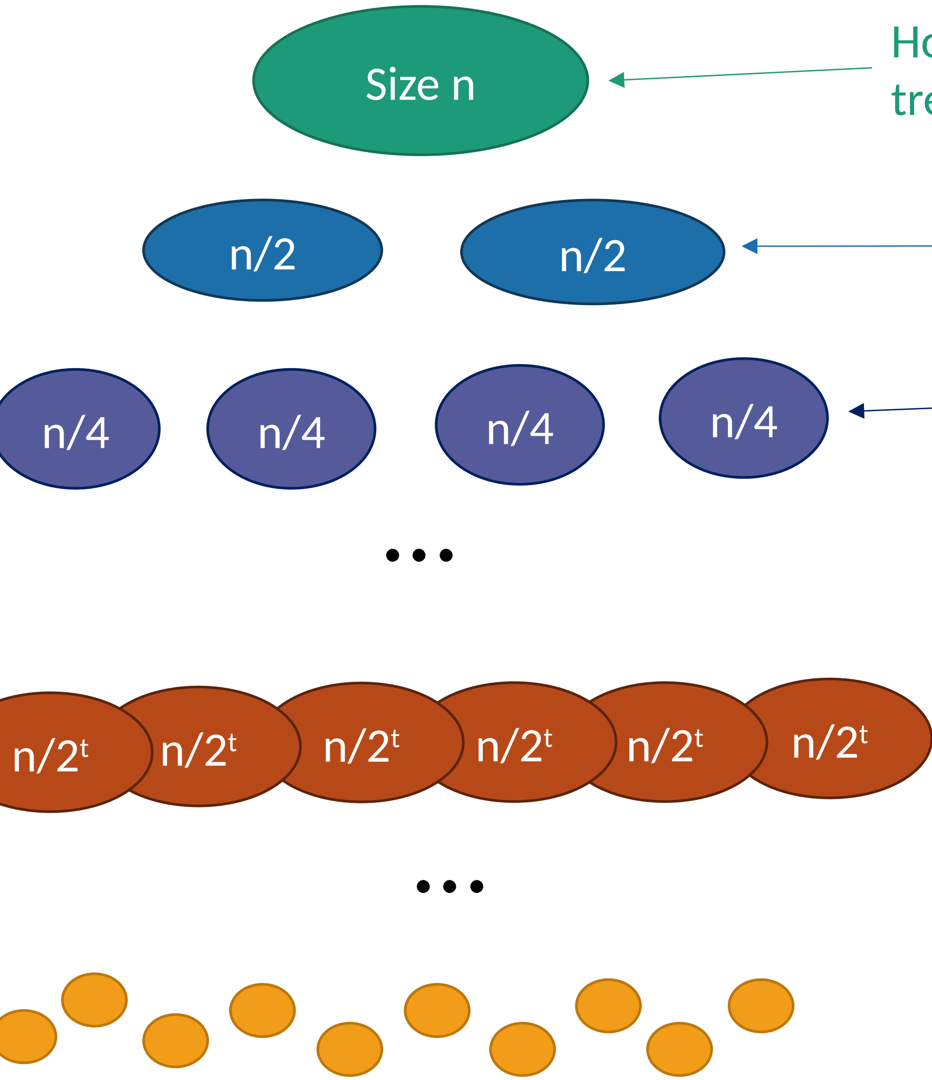
Recursion tree



There are O(k) operations done at this node.



Recursion tree



How many operations are done at this level of the tree? (Just MERGE-ing subproblems).

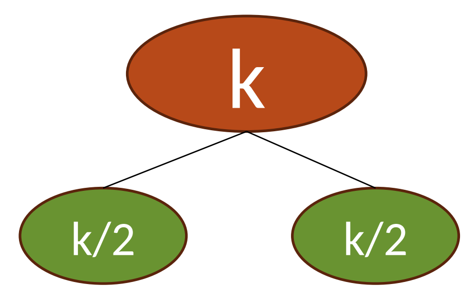
How about at this level of the tree? (between both n/2-sized problems)

This level?

This level?

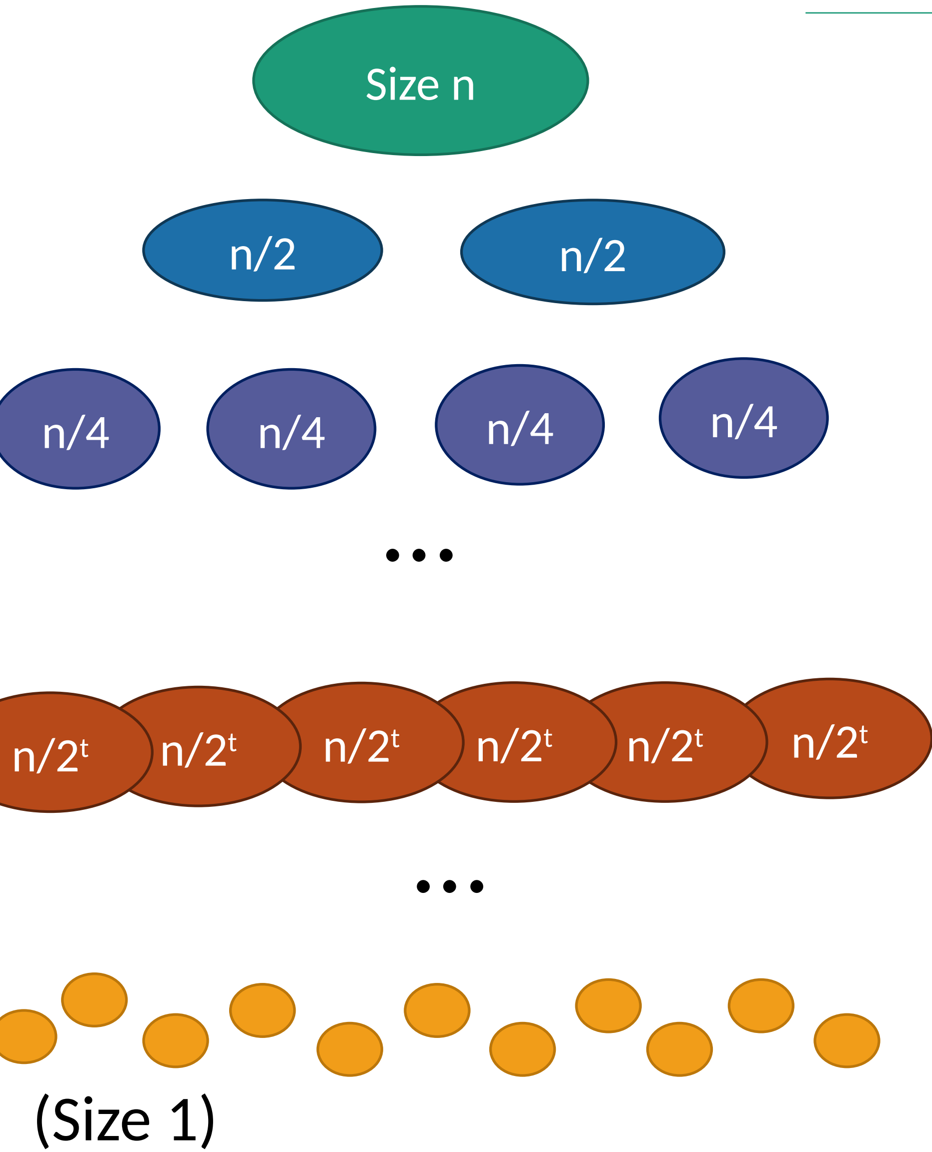
(Size 1)

There are O(k) operations done at this node.



Recursion tree

Work this out yourself!



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	$O(n)$
1	2	$n/2$	$O(n)$
2	4	$n/4$	$O(n)$
…			
t	2^t	$n/2^t$	$O(n)$
…			
$\log(n)$	n	1	$O(n)$

Total runtime…

- O(n) steps per level, at every level
- log(n) + 1 levels
- O(n log(n)) total!

That was the claim!

What have we learned?

- MergeSort correctly sorts a list of n integers in time O(n log(n)).
- That’s (asymptotically) better than InsertionSort!

Can we do better?

- Any deterministic compare-based sorting algorithm must make $\Omega(n \log n)$ compares in the worst-case.
  - How to prove this?

- Is there any other way to sort an array efficiently?

QuickSort

QuickSort: another divide-and-conquer approach

- Divide
  - Partition the array A[1:n] into two (possibly empty) subarrays A[1:q-1] (the low side) and A [q+1:n] (the high side)
  - Each element in the low side of the partition is <=A[q]
    Each element in the high side is of the partition >= A[q].
  - Compute the index q of the pivot as part of this partitioning procedure.
- Conquer
  - Recursively sort the subarrays A[1:q-1] and A[q+1:n]
- Combine
  - Already sorted

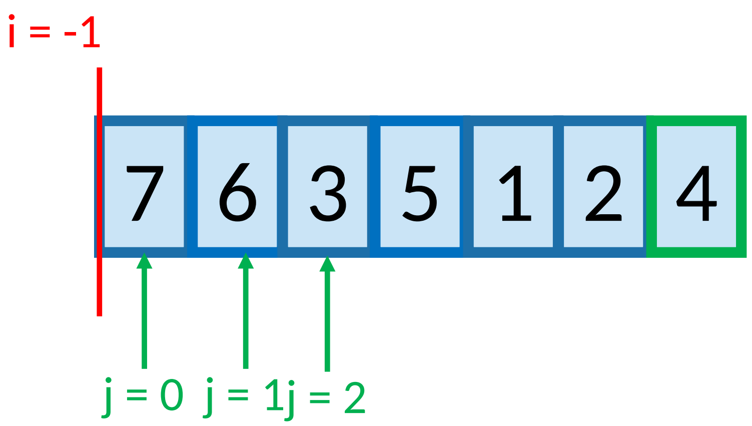
Pseudocode of QuickSort

```
QUICKSORT(A, p, r)
      if p < r
            q = PARTITION(A, p, r)
QUICKSORT(A, p, q – 1)
QUICKSORT(A, q+1, r)
```

PARTITION

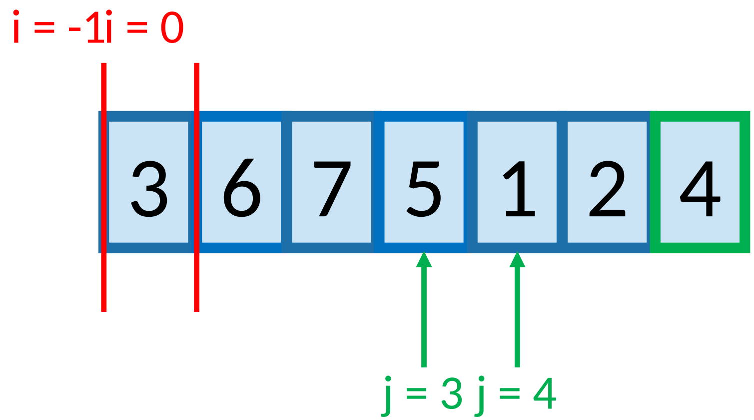
```
PARTITION(A, p, r)
      x = A[r]
      i = p - 1
      for j = p to r - 1
            if A[j] <= x
                  i = i + 1
                  exchange A[i] with A[j]
      exchange A[i + 1] with A[r]
      return i + 1
```

Example of PARTITION



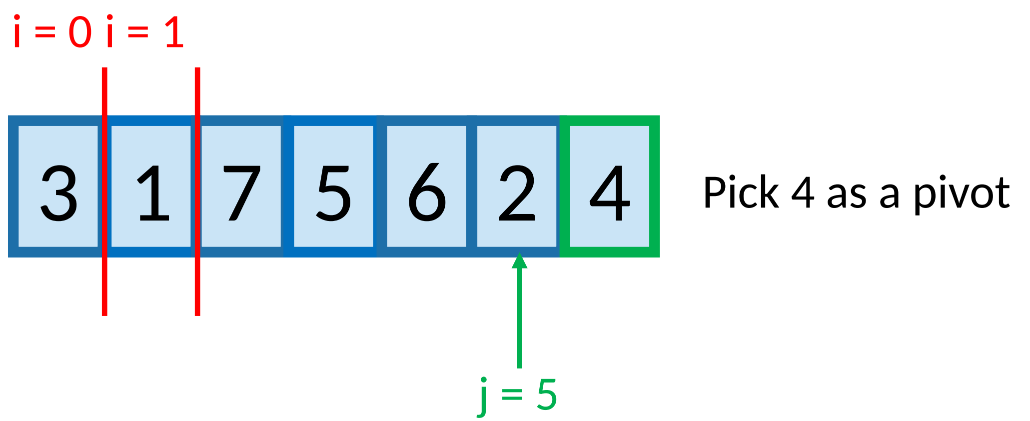
Pick 4 as a pivot

Example of PARTITION

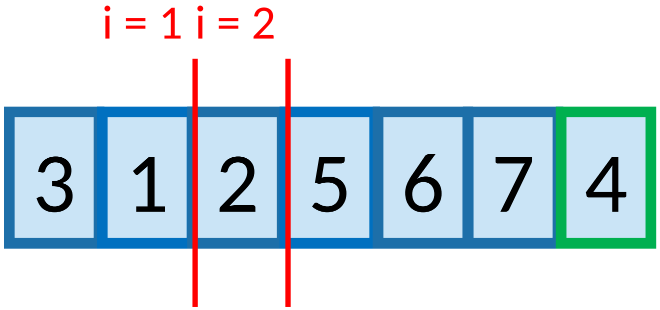


Pick 4 as a pivot

Example of PARTITION

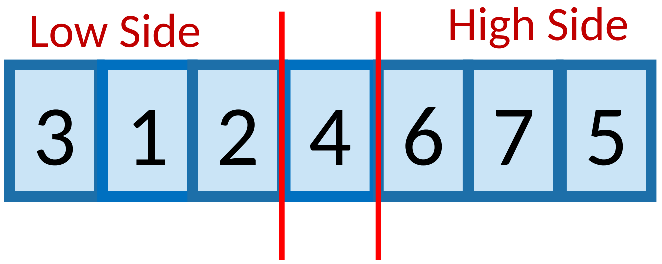


Example of PARTITION



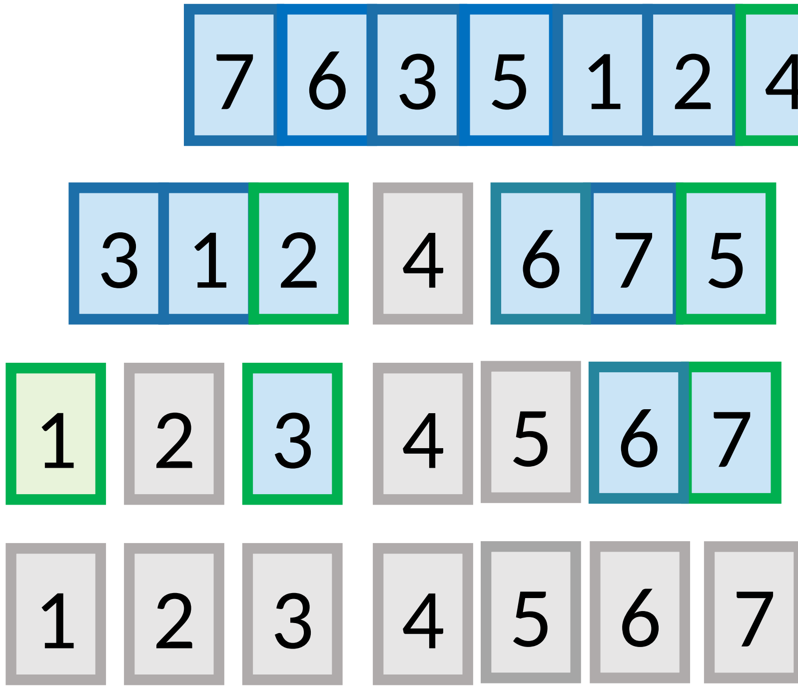
Pick 4 as a pivot

Example of PARTITION



Set the pivot in place

Example of recursive calls



Pick 4 as a pivot

Partition on either side of 4

Recurse on [312] and pick 2 as a pivot.

Recurse on [67] and pick 7 as a pivot.

QuickSort Runtime Analysis

$T(n)$ = The worst-case running time on a problem of size n

- Worst-case partitioning

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

- Best-case partitioning

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

Recurrences

- An equation that describes a function in terms of its value on other, typically smaller, arguments.

  - Recursive Case
    - Involves the recursive invocation of the function on different (usually smaller) inputs
  - Base Case
    - Does not involve a recursive invocation

Algorithmic Recurrences

- A recurrence $T(n)$ is algorithmic if, for every sufficiently large threshold constant $n_0 < n$, the following two properties hold,

1. For all $n < n_0$, we have $T(n) = \Theta(1)$.
2. For all $n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive invocations.

Solving Recurrences

- Substitution Method
  - Guess a solution
  - Use mathematical induction to prove the guess

Substitution Method

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

- Guess
  - $T(n) = O(nlgn)$

- We need to prove,
  - $T(n) \leq cnlgn$ for all, $n_0 \leq n$
  - For specific choice of $c > 0$ and $\mathrm{n}_0 > 0$

Substitution Method

- Inductive Hypothesis
  - $T(n') \leq cn'lgn'$ for all $\mathrm{n}_0 < n' < n$ and $2\mathrm{n}_0 \leq n$

$$T(n) = 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + \Theta(n)$$
$$T(n) \leq 2\mathrm{c}\left\lfloor\frac{n}{2}\right\rfloor \lg\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + \Theta(n)$$
$$T(n) \leq \mathrm{cnlg}\left(\frac{n}{2}\right) + \Theta(n)$$
$$T(n) \leq \mathrm{cnlg}(n) - cnlg2 + \Theta(n)$$
$$T(n) \leq \mathrm{cnlg}(n) - cn + \Theta(n)$$

$$T(n) \leq \mathrm{cnlg}(n)$$

If n_0 is sufficiently large and cn dominates $\Theta(n)$

Substitution Method

- Base Case
  - $n_0 \leq n < 2n_0$

- Assuming
  - $n_0 = 2$
  - $c = \max(T(2), T(3))$

- We get $T(n) \leq cnlgn$

Solving Recurrences

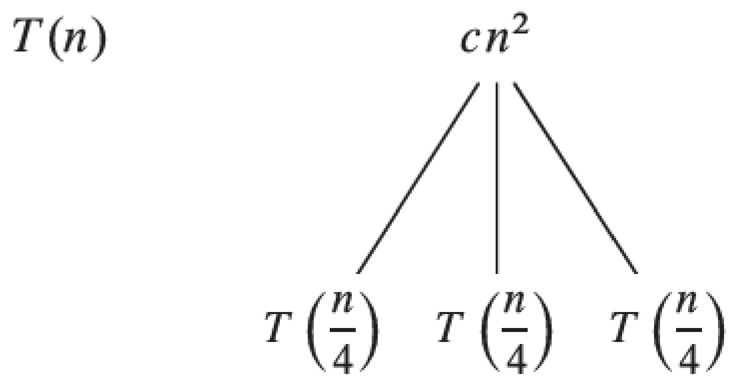
- Substitution Method
  - Guess a solution
  - Use mathematical induction to prove the guess

  - Making a good guess may be difficult
  - Need to be careful about common pitfalls.

Recursion Tree

- Consider the recurrence

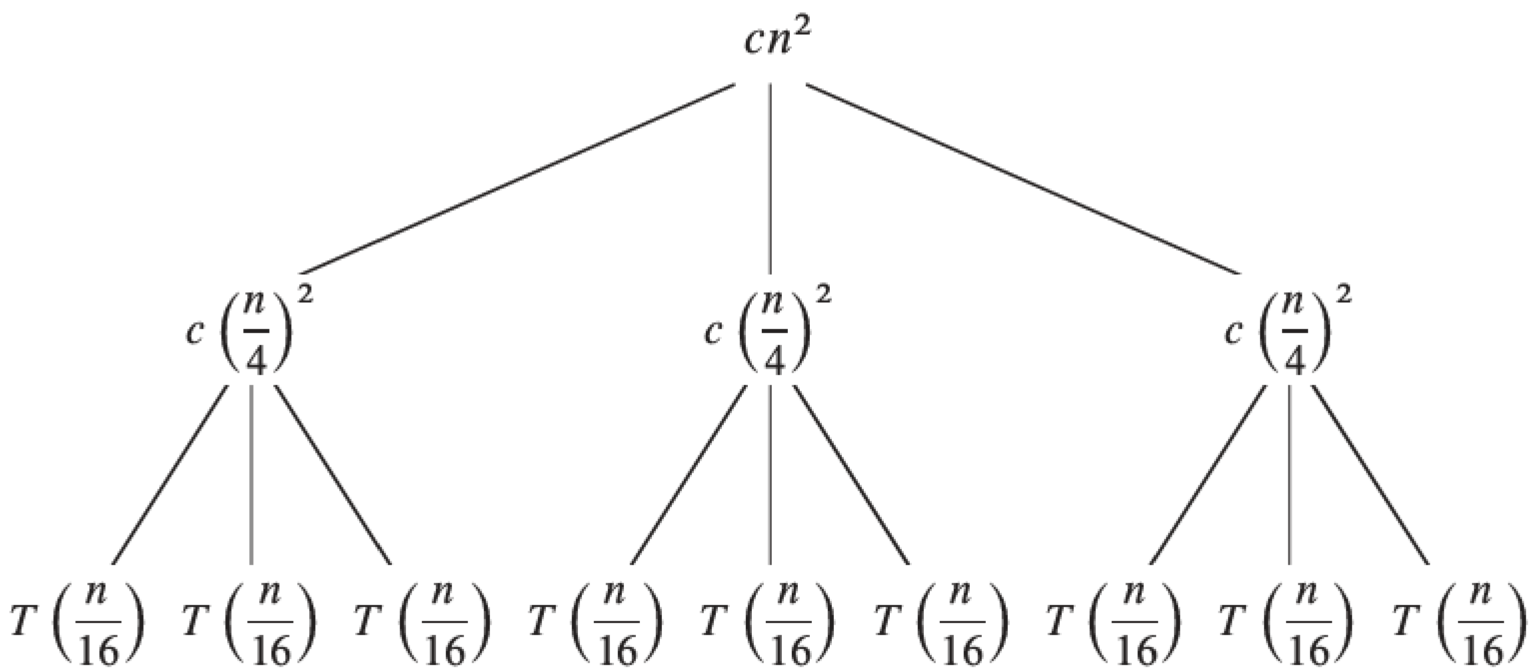
$$T(n) = 3\,T\left(\frac{n}{4}\right) + \Theta(n^2)$$



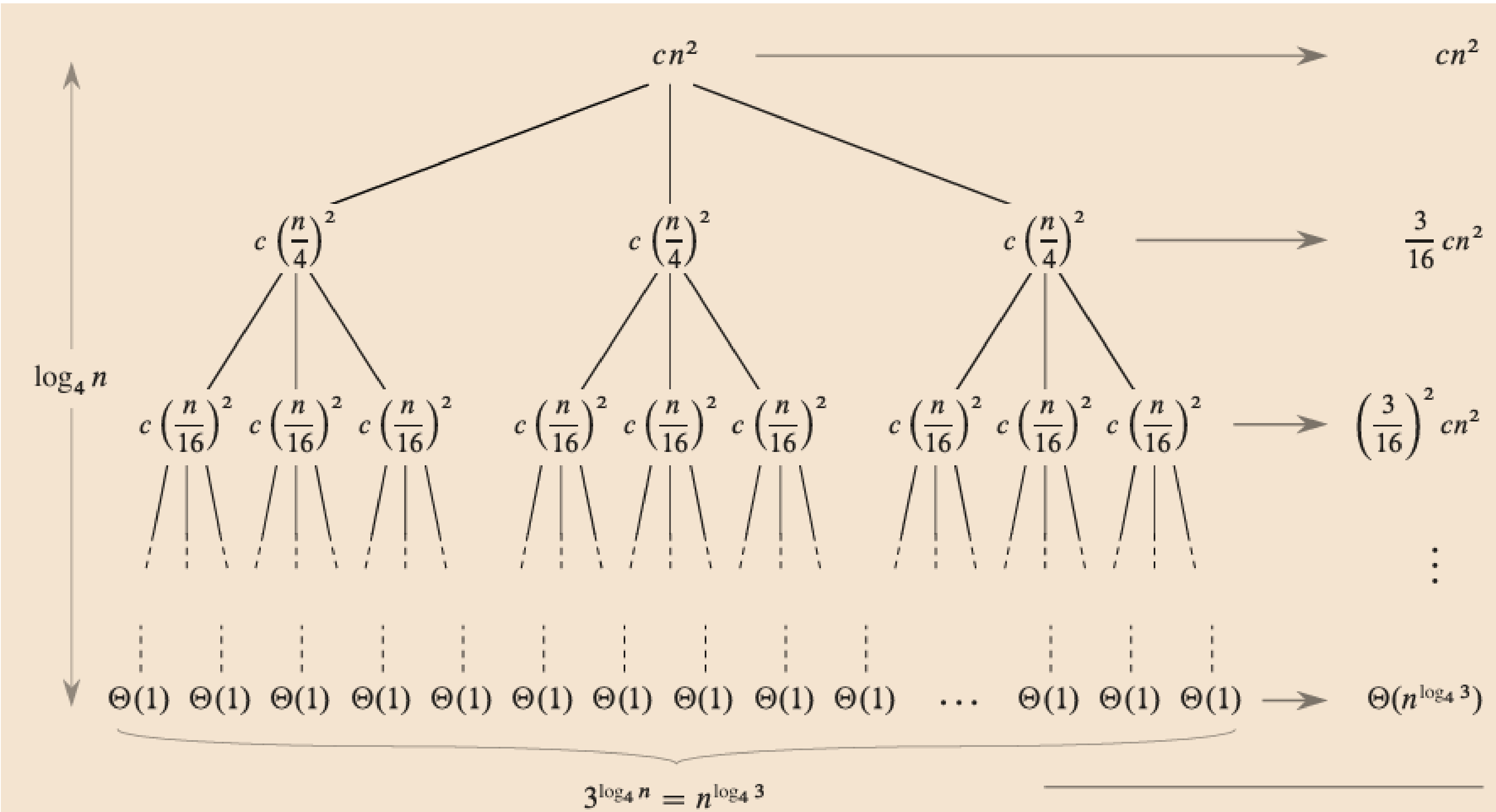
Recursion Tree

- Consider the recurrence

$$T(n) = 3\,T\left(\frac{n}{4}\right) + \Theta(n^2)$$



Recursion Tree



Recursion Tree

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)
\end{aligned}
$$

Recursion Tree

- Unbalanced recursion tree
  - Estimate the height of the tree
  - Estimate the cost from each level
  - Estimate the number of leave nodes of the tree
  - Estimate the cost from all the leaf nodes over all the levels

- Example

$$T(n) = T(n/3) + T(2n/3) + \Theta(n)$$

Master Theorem

- Let's consider the following recurrence relation,

$$T(n) = aT(n/b) + f(n)$$

a > 0 b > 1 Driving Function

1. If $f(n) = \mathcal{O}(n^c)$ and $\log_b a > c$ then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta(n^c)$ and $\log_b a = c$ then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.

2f. (Fancy Version) If $f(n) = \Theta(n^c \lg^k n)$ and $\log_b a = c$ then $T(n) = \Theta\left(n^{\log_b a} \lg^{k+1} n\right)$.

3. If $f(n) = \Omega(n^c)$ and $\log_b a < c$ then $T(n) = \Theta(f(n))$.
   Note: For this case, $f(n)$ must also satisfy a *regularity condition* which states that there is some $C < 1$ and n_0 such that $af(n/b) \leq Cf(n)$ for all $n \geq n_0$. This regularity condition is almost always true and we will not worry about it.

Master Theorem

- Let's consider the following recurrence relation,

$$T(n) = aT(n/b) + f(n)$$

a > 0 b > 1 Driving Function

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Master Theorem

$$T(n) = 9\,T\left(\frac{n}{3}\right) + n$$

- $a = 9,\ b = 3$
- $f(n) = n = O(n^1) = O(n^c)$
- $\log_b a = log_3 9 = 2 > c$

1. If $f(n) = \mathcal{O}(n^c)$ and $\log_b a > c$ then $T(n) = \Theta\left(n^{\log_b a}\right)$.

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

Master Theorem

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

- $a = 1, \;\; b = \frac{2}{3}$
- $f(n) = 1 = O(n^0) = O(n^c)$
- $\log_b a = log_{2/3} 1 = 0 = c$

2. If $f(n) = \Theta(n^c)$ and $\log_b a = c$ then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.

$$T(n) = \Theta(n^0 \; lgn) = \Theta(lgn)$$

Master Theorem

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

- $a = 3, \quad b = 4$
- $f(n) = n\lg n > n = \Omega(n^1) = \Omega(n^c)$
- $\log_b a = log_4 3 < \; c$
- Can we apply case 3?

3. If $f(n) = \Omega(n^c)$ and $\log_b a < c$ then $T(n) = \Theta\left(f(n)\right)$.

- Need to satisfy the regularity condition.

Master Theorem

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

- Regularity condition

Note: For this case, $f(n)$ must also satisfy a *regularity condition* which states that there is some $C < 1$ and n_0 such that $af(n/b) \leq Cf(n)$ for all $n \geq n_0$. This regularity condition is almost always true and we will not worry about it.

- $af\left(\frac{n}{b}\right) = \frac{3n}{4} \lg\left(\frac{3}{4}\right) \leq \frac{3}{4} n \lg n \leq Cf(n)$

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

Master Theorem

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

- Applying master theorem (case 2),

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

MergeSort Runtime Analysis (Revisit)

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

- Applying master theorem (case 2),

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

QuickSort Runtime Analysis (Revisit)

- Best-case partitioning

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

  - Applying master theorem, $T(n) = \Theta(n\,logn)$

- Worst-case partitioning

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

  - Expanding the recurrence (Rec. tree): $T(n) = \Theta(n^2)$

Multiplying Square Matrices

- A and B are square matrices
- Find out $C = A * B$

MATRIX-MULTIPLY(A, B, C, n)

for $i = 1$ **to** n // compute entries in each of n rows
    for $j = 1$ **to** n // compute n entries in row i
        for $k = 1$ **to** n
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ // add in another term of equation (4.1)

- Running Time $\Theta(n^3)$

Can we do better?

Multiplying Square Matrices

- Divide

Assuming $n = 2^x$

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

- Conquer

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}$$

Multiplying Square Matrices

MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)

if $n == 1$
  // Base case.
    $c_{11} = c_{11} + a_{11} \cdot b_{11}$
    return
// Divide.
partition A, B, and C into $n/2 \times n/2$ submatrices
    $A_{11}, A_{12}, A_{21}, A_{22}$; $B_{11}, B_{12}, B_{21}, B_{22}$;
    and $C_{11}, C_{12}, C_{21}, C_{22}$; respectively
// Conquer.

$\Theta(1)$

MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{11}, C_{11}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{12}, C_{12}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{11}, C_{21}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{12}, C_{22}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{21}, C_{11}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{22}, C_{12}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{21}, C_{21}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{22}, C_{22}, n/2)$

$8T(\frac{n}{2})$

Multiplying Square Matrices

- Running Time

$$T(n) = 8\,T\left(\frac{n}{2}\right) + \Theta(1)$$

- Applying master theorem (case 1),

$$T(n) = \Theta(n^3)$$

Can we do better?

Strassen's Algorithm

- Intuitions,
  - Addition is faster than multiplications.
    - $\Theta(n^2)$ vs $\Theta(n^3)$
  - Replace multiplications with additions
    - Remember $x^2 - y^2 = (x + y)(x - y)$

Strassen's Algorithm

- Divide (same as before)
- Conquer
  - Perform 10 additions

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} & \qquad S_6 &= B_{11} + B_{22} \\
S_2 &= A_{11} + A_{12} & \qquad S_7 &= A_{12} - A_{22} \\
S_3 &= A_{21} + A_{22} & \qquad S_8 &= B_{21} + B_{22} \\
S_4 &= B_{21} - B_{11} & \qquad S_9 &= A_{11} - A_{21} \\
S_5 &= A_{11} + A_{22} & \qquad S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

Strassen's Algorithm

- Divide (same as before)
- Conquer
  - Perform 10 additions, $S_1, S_2, \ldots, S_{10}$
  - Perform 7 multiplications

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 \;\; (= A_{11} \cdot B_{12} - A_{11} \cdot B_{22}) \\
P_2 &= S_2 \cdot B_{22} \;\; (= A_{11} \cdot B_{22} + A_{12} \cdot B_{22}) \\
P_3 &= S_3 \cdot B_{11} \;\; (= A_{21} \cdot B_{11} + A_{22} \cdot B_{11}) \\
P_4 &= A_{22} \cdot S_4 \;\; (= A_{22} \cdot B_{21} - A_{22} \cdot B_{11}) \\
P_5 &= S_5 \cdot S_6 \;\; (= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}) \\
P_6 &= S_7 \cdot S_8 \;\; (= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}) \\
P_7 &= S_9 \cdot S_{10} \;\; (= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12})
\end{aligned}
$$

Strassen's Algorithm

- Divide (same as before)
- Conquer
  - Perform 10 additions, $S_1, S_2, \ldots, S_{10}$
  - Perform 7 multiplications $P_1, P_2, \ldots, P_7$
  - Compute C_{ij} with S and P matrices

$$C_{11} = C_{11} + P_5 + P_4 - P_2 + P_6 \qquad C_{12} = C_{12} + P_1 + P_2$$

$$C_{21} = C_{21} + P_3 + P_4 \qquad C_{22} = C_{22} + P_5 + P_1 - P_3 - P_7$$

Strassen’s Algorithm Runtime Analysis

- Running Time

$$T(n) = 7\,T\left(\frac{n}{2}\right) + \Theta(n^2)$$

- Applying master theorem (case 1),

$$T(n) = \Theta\left(n^{\log_2 7}\right) = \Theta(n^{2.80755\ldots})$$

Reference

- Introduction to Algorithms, CLRS, 4th edition.
  - Chapter 2 (Getting Started)
    - Section 2.1, 2.3
  - Chapter 4 (Divide-and-Conquer)
    - Sections 4.1 – 4.5
  - Chapter 7 (Quick Sort)
    - Sections 7.1 and 7.2

- Additional Resource:
  - mastertheorem.pdf