CSE 417 Autumn 2025

Lecture 7: Computing More

Nathan Brunelle

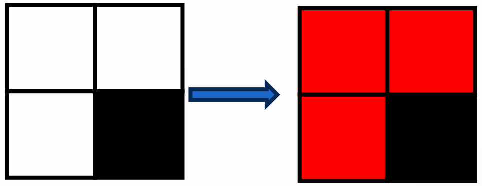
Homeworks

HW 1 feedback expected Thursday (tomorrow)

HW 2 out, due Friday 11:59pm.

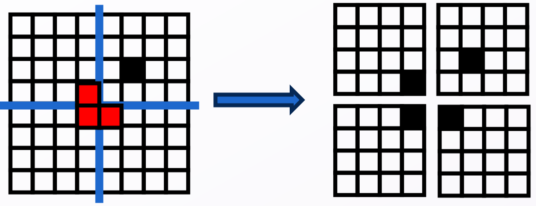
Divide and Conquer Review

Divide and Conquer (Trominoes)



Base Case:

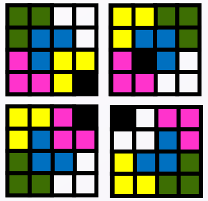
For a 2×2 board, the empty cells will be exactly a tromino



Divide:

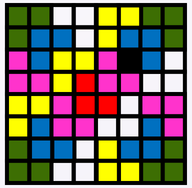
Break of the board into quadrants of size $2^{n-1} \times 2^{n-1}$ each

Put a tromino at the intersection such that all quadrants have one occupied cell



Conquer:

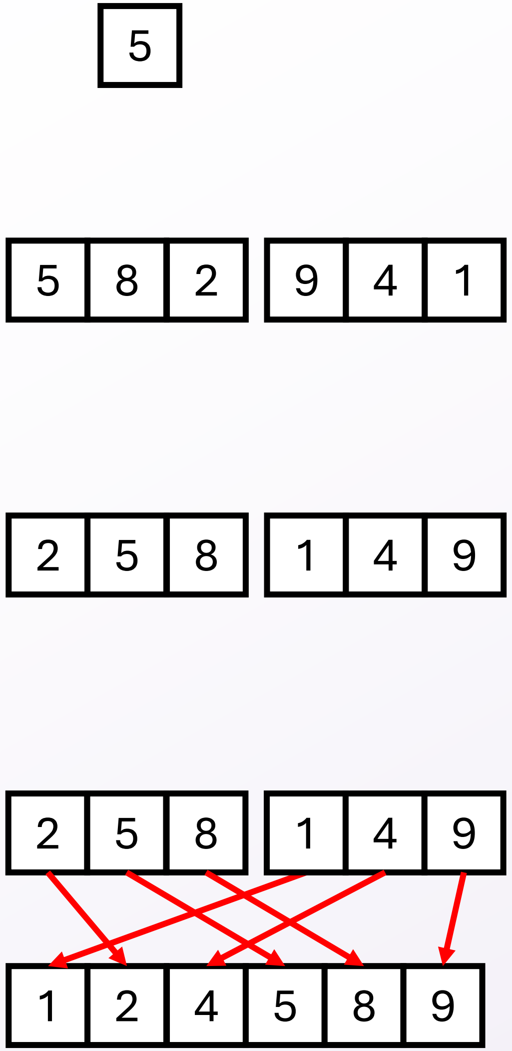
Cover each quadrant



Combine:

Reconnect quadrants

Divide and Conquer (Merge Sort)



Base Case:

If the list is of length 1 or 0, it’s already sorted, so just return it

(Alternative: when length is ≤ 15, use insertion sort)

Divide:

Split the list into two “sublists” of (roughly) equal length

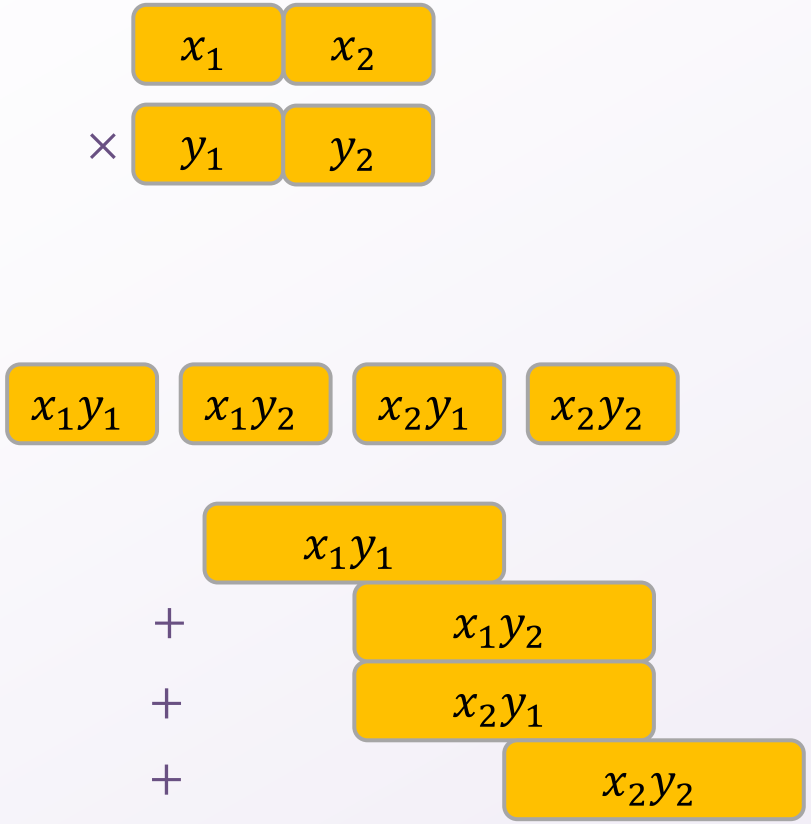
Conquer:

Sort both lists recursively

Combine:

Merge sorted sublists into one sorted list

Divide and Conquer (Integer Multiplication)



Base Case:

If there is only 1 place value, just multiply them

Divide:

Break the operands into 4 values:

- x_1 is the most significant $\frac{n}{2}$ digits of x
- x_2 is the least significant $\frac{n}{2}$ digits of x
- y_1 is the most significant $\frac{n}{2}$ digits of y
- y_2 is the most significant $\frac{n}{2}$ digits of y

Conquer:

Compute each of x_1y_1, x_1y_2, x_2y_1, and x_2y_2

Combine:

Return $2^n(x_1y_1) + 2^{\frac{n}{2}}(x_1y_2 + x_2y_1) + (x_2y_2)$

Divide and Conquer (Running Time)

$T(c) = k$

$a = number\ of\ subproblems$

$\frac{n}{b} = size\ of\ each\ subproblem$

$f_d(n)$ = time to divide

$a \cdot T\left(\frac{n}{b}\right)$

$f_c(n)$ =time to combine

Base Case:

When the problem size is small ($\leq c$), solve non-recursively

Divide:

When problem size is large, identify 1 or more smaller versions of exactly the same problem

Conquer:

Recursively solve each smaller subproblem

Combine:

Use the subproblems' solutions to solve to the original

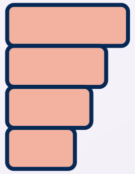
Overall: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ **where** $f(n) = f_d(n) + f_c(n)$

Master Theorem

Master Theorem: Suppose that $T(n) = a \cdot T(n/b) + O(n^k)$ for $n > b$.

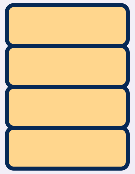
If $a < b^k$ then $T(n)$ is $O(n^k)$

- Cost is dominated by work at top level of recursion



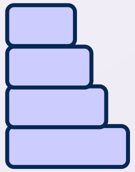
If $a = b^k$ then $T(n)$ is $O(n^k \log n)$

- Total cost is the same for all $\log_b n$ levels of recursion



If $a > b^k$ then $T(n)$ is $O(n^{\log_b a})$

- Note that $\log_b a > k$ in this case
- Cost is dominated by total work at lowest level of recursion



Binary search: $a = 1$, $b = 2$, $k = 0$ so $a = b^k$: Solution: $O\left(n^0 \log n\right) = O(\log n)$

Mergesort: $a = 2$, $b = 2$, $k = 1$ so $a = b^k$: Solution: $O(n^1 \log n) = O(n \log n)$

Integer Multiplication

Schoolbook Integer Multiplication

```
      695273
    × 123412
    --------
     1390546
     695273
    2781092
   2085819
  1390546
  695273
 -----------
 85805031476
```

Decimal

```
      110110
    × 101110
    --------
      000000
     110110
    110110
   110110
  000000
 110110
 -----------
 100110110100
```

Binary

Elementary school algorithm

$O(n^2)$ time for n-bit integers

Divide and Conquer method

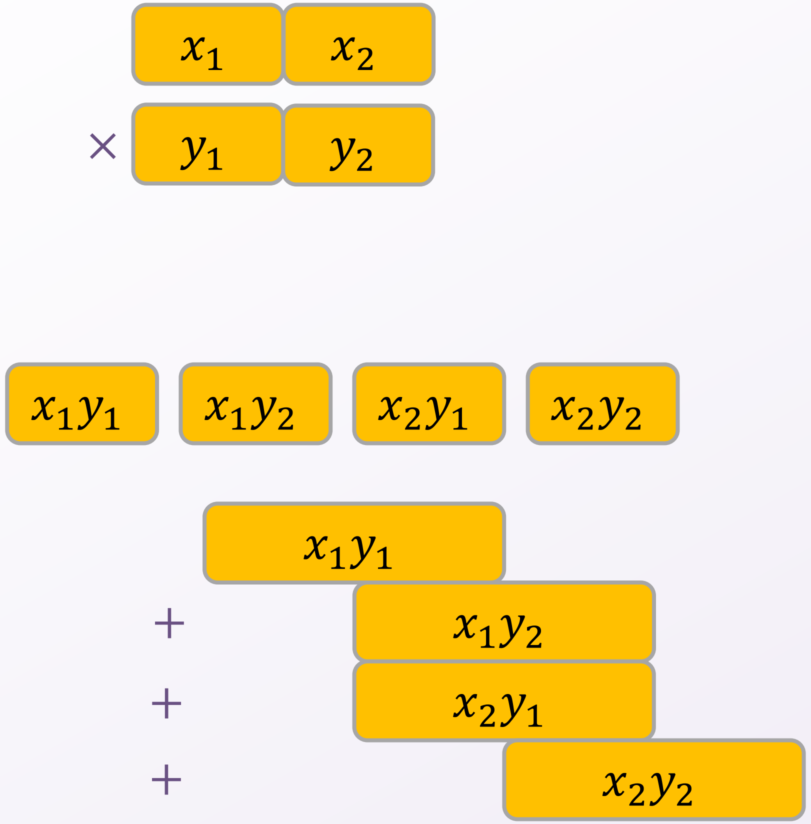
$$\begin{array}{r} x_1 \; x_2 \\ \times \; y_1 \; y_2 \end{array} \quad \begin{array}{l} = 2^{\frac{n}{2}} x_1 + x_2 \\ = 2^{\frac{n}{2}} y_1 + y_2 \end{array}$$

$$2^n \; (x_1 \times y_1 \quad +$$

$$2^{\frac{n}{2}} \; (x_1 \times y_2 + x_2 \times y_1 \quad +$$

$$(x_2 \times y_2$$

Divide and Conquer (Integer Multiplication)



Base Case:

If there is only 1 place value, just multiply them

Divide:

Break the operands into 4 values:

- x_1 is the most significant $\frac{n}{2}$ digits of x
- x_2 is the least significant $\frac{n}{2}$ digits of x
- y_1 is the most significant $\frac{n}{2}$ digits of y
- y_2 is the most significant $\frac{n}{2}$ digits of y

Conquer:

Compute each of x_1y_1, x_1y_2, x_2y_1, and x_2y_2

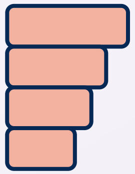
Combine:

Return $2^n(x_1y_1) + 2^{\frac{n}{2}}(x_1y_2 + x_2y_1) + (x_2y_2)$

Integer Multiplication Recurrence Solution

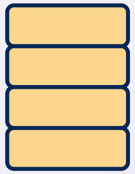
Master Theorem: Suppose that $T(n) = a \cdot T(n/b) + O(n^k)$ for $n > b$.

If $a < b^k$ then $T(n)$ is $O(n^k)$



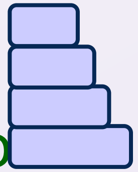
- Cost is dominated by work at top level of recursion

If $a = b^k$ then $T(n)$ is $O(n^k \log n)$



- Total cost is the same for all $\log_b n$ levels of recursion

If $a > b^k$ then $T(n)$ is $O(n^{\log_b a})$



- Note that $\log_b a > k$ in this case
- Cost is dominated by total work at lowest level of recursion

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

$a = 4$, $b = 2$, $k = 1$, so $a > b^k$: Solution: $O(n^{log_b a}) = O(n^2)$

Improving the algorithm

Ways to reduce running time

Master Theorem: Suppose that $T(n) = a \cdot T(n/b) + O(n^k)$ for $n > b$.

If $a < b^k$ then $T(n)$ is $O(n^k)$

If $a = b^k$ then $T(n)$ is $O(n^k \log n)$

If $a > b^k$ then $T(n)$ is $O(n^{\log_b a})$

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

$a = 4$, $b = 2$, $k = 1$, so $a > b^k$: Solution: $O(n^{\log_b a}) = O(n^2)$

What changes to the recurrence improve running time?

Karatsuba Method

Can't avoid these

$$2^n(x_1y_1) + 2^{\frac{n}{2}}(x_1y_2 + x_2y_1) + x_2y_2$$

Can we do this with one multiplication?

$$(x_1 + x_2)(y_1 + y_2) =$$

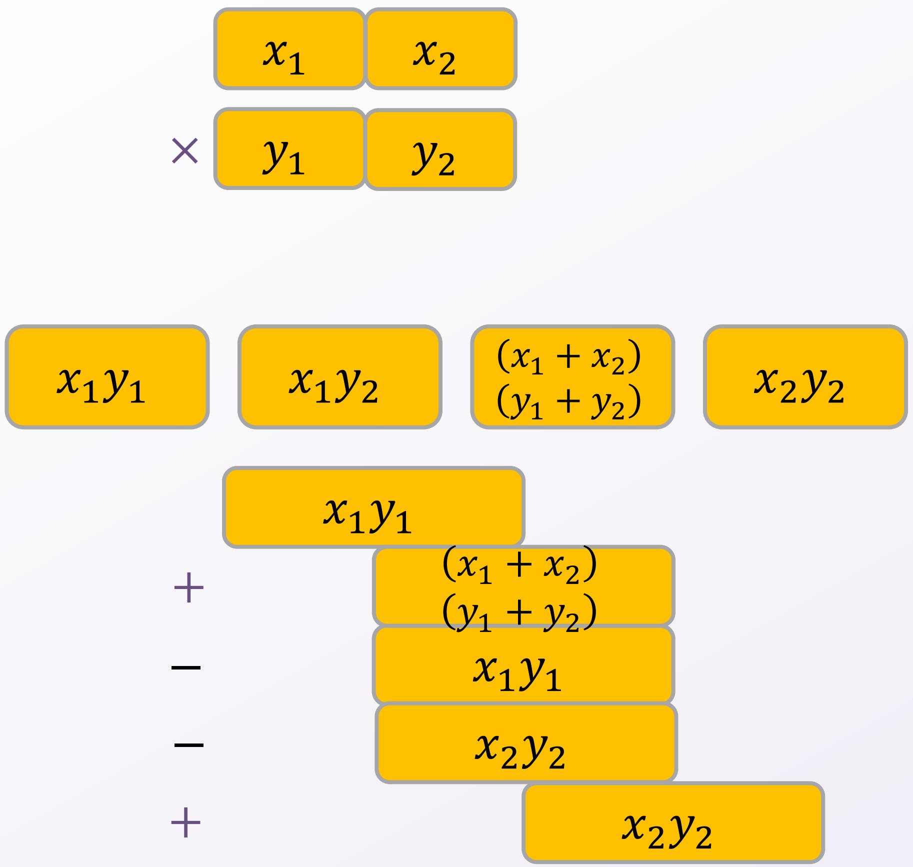
$$x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2$$

$$x_1y_2 + x_2y_1 = (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2$$

Two multiplications

One multiplication

Divide and Conquer (Karatsuba Method)



Base Case:

If there is only 1 place value, just multiply them

Divide:

Break the operands into 4 values:

- x_1 is the most significant $\frac{n}{2}$ digits of x
- x_2 is the least significant $\frac{n}{2}$ digits of x
- y_1 is the most significant $\frac{n}{2}$ digits of y
- y_2 is the most significant $\frac{n}{2}$ digits of y

Conquer:

Compute each of x_1y_1, $(x_1+x_2)(y_1+y_2)$, and x_2y_2

Combine:

Return

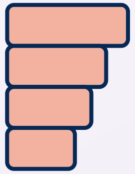
$$2^n(x_1y_1) + 2^{\frac{n}{2}}\big((x_1+x_2)(y_1+y_2) - x_1y_1 - x_2y_2\big) + (x_2y_2)$$

Karatsuba Method Recurrence Solution

Master Theorem: Suppose that $T(n) = a \cdot T(n/b) + O(n^k)$ for $n > b$.

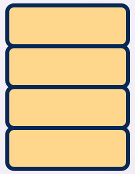
If $a < b^k$ then $T(n)$ is $O(n^k)$

- Cost is dominated by work at top level of recursion



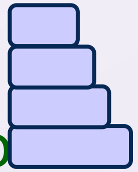
If $a = b^k$ then $T(n)$ is $O(n^k \log n)$

- Total cost is the same for all $\log_b n$ levels of recursion



If $a > b^k$ then $T(n)$ is $O(n^{\log_b a})$

- Note that $\log_b a > k$ in this case
- Cost is dominated by total work at lowest level of recursio



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

$a = 3$, $b = 2$, $k = 1$, so $a > b^k$: Solution: $O(n^{log_b a}) = O\left(n^{log_2 3}\right) = O(n^{1.585})$

The “Technique of Computing More”

Sometimes, it’s helpful to perform more tasks in your combine and conquer algorithm. We’ll see 2 examples:

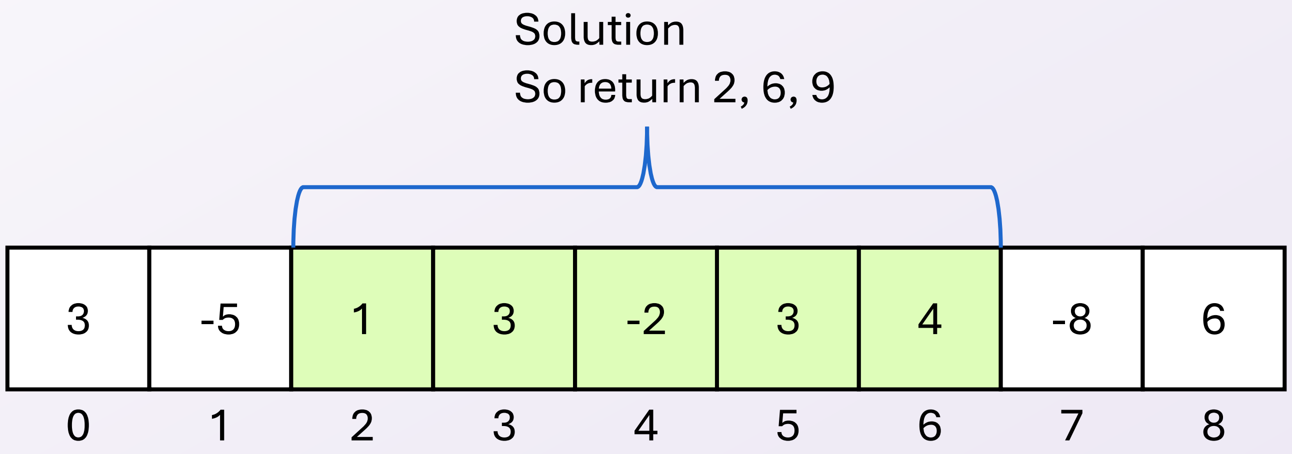
1) More tasks give better running time
2) More tasks enable correctness

Maximum Sum Subarray

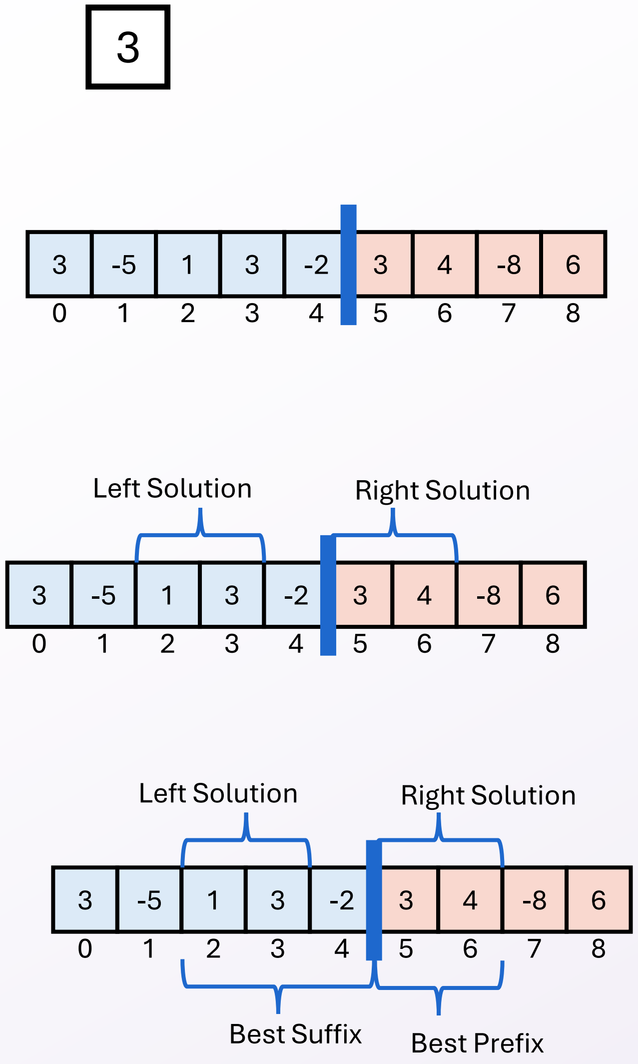
Maximum Sum Subarray - Problem

Given an array of integers, find the contiguous subarray with the maximum sum, then return:

1. The **start** index of that subarray
2. The **end** index of that subarray
3. The **sum** of the elements in that subarray



Maximum Sum Subarray (D&C from reading)



Base Case:

If $i = j$ then return $i, i, arr[i]$ as the start, end, sum respectively

Divide:

Split the list into two "sublists" of (roughly) equal length. So the left is i to $\frac{i+j}{2}$ and the right is $\frac{i+j}{2} + 1$ to j

Conquer:

Find the start, end and sum of each subarray. Call these $leftStart, leftEnd, leftSum, rightStart, rightEnd, rightSum$

Combine:

Find the best suffix of the left subarray and best prefix of the right subarray. Return depending on which of $leftSum$, $rightSum$, and $middleSum$ is largest

Running Time (D&C from reading)

2 subproblems

Each size $\frac{n}{2}$

$O(1)$ time to divide

$O(n)$ time to combine (from finding prefix and suffix)

Base Case:

If $i = j$ then return $i, i, arr[i]$ as the start, end, sum respectively

Divide:

Split the list into two "sublists" of (roughly) equal length. So the left is i to $\frac{i+j}{2}$ and the right is $\frac{i+j}{2} + 1$ to j

Conquer:

Find the start, end and sum of each subarray. Call these $leftStart$, $leftEnd$, $leftSum$, $rightStart$, $rightEnd$, $rightSum$

Combine:

Find the best suffix of the left subarray and best prefix of the right subarray. Return depending on which of $leftSum$, $rightSum$, and $middleSum$ is largest

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

Reducing the Maximum Sum Subarray Running Time

Master Theorem: Suppose that $T(n) = a \cdot T(n/b) + O(n^k)$ for $n > b$.

If $a < b^k$ then $T(n)$ is $O(n^k)$

If $a = b^k$ then $T(n)$ is $O(n^k \log n)$

If $a > b^k$ then $T(n)$ is $O(n^{\log_b a})$

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

$a = 2$, $b = 2$, $k = 1$, so $a = b^k$: Solution: $O(n^1 \log n) = O(n \log n)$

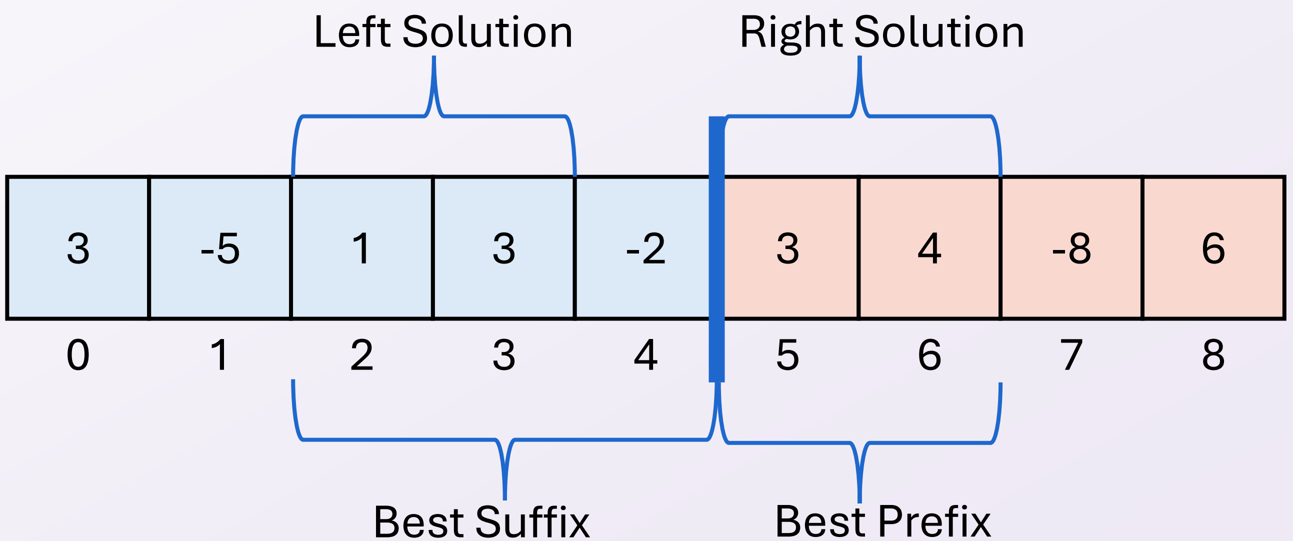
What changes to the recurrence improve running time?

The Technique of Computing More

In general: it's worthwhile to offload tasks onto subproblems if it can asymptotically reduce the time spent in dividing + combining

All the information needed to find the best suffix to the left and best prefix to the right are already present in the subproblem.

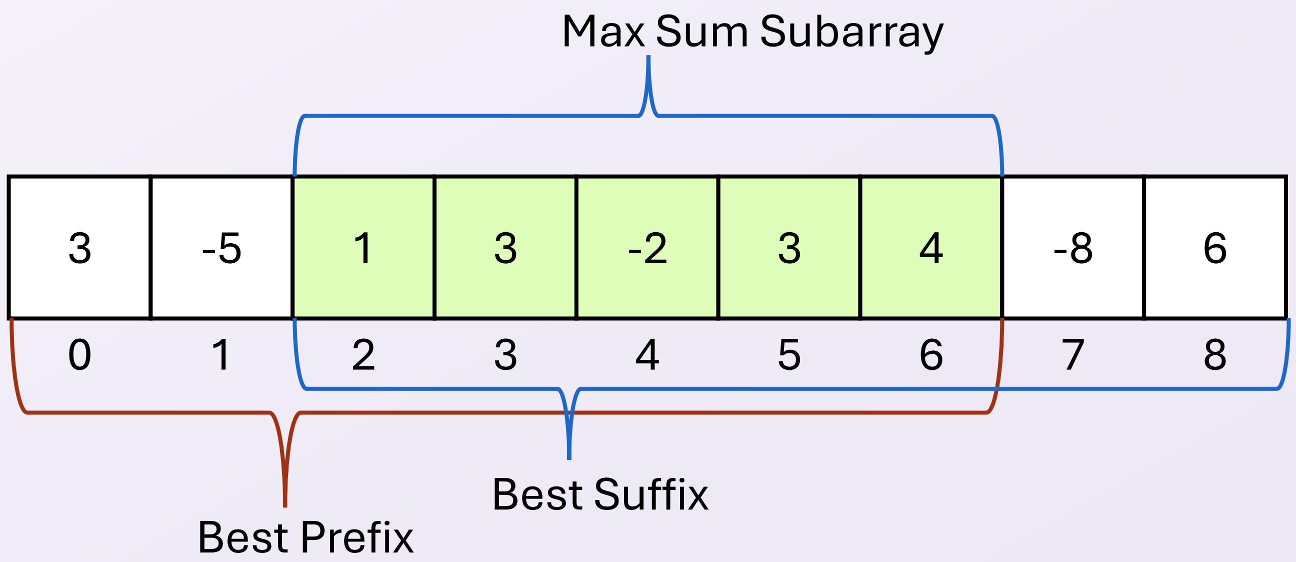
Can we save time if we have the subproblems return those values?



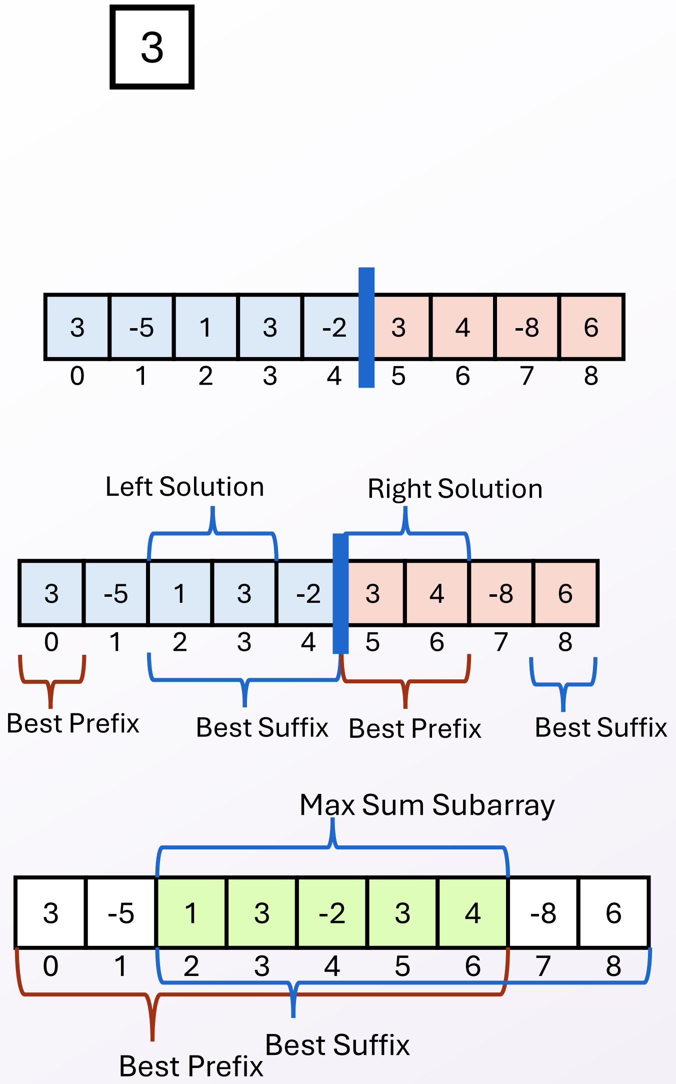
Maximum Sum Subarray – New Problem

Given an array of integers, find the contiguous subarray with the maximum sum, then return:

1. The **start** index of that subarray
2. The **end** index of that subarray
3. The **sum** of the elements in that subarray
4. The **start** index of the best **suffix** of that subarray
5. The **sum** of the elements in that **suffix**
6. The **end** index of the best **prefix** of that subarray
7. The **sum** of the elements in that **prefix**
8. The **sum** of the **entire** subarray



Maximum Sum Subarray (Improved D&C)



Base Case:

If $i = j$ then: start=i, end =i, max sum=$arr[i]$, suffix start =i, suffix sum=$arr[i]$, prefix start =i, prefix sum=$arr[i]$, and total sum=$arr[i]$

Divide:

Split the list into two “sublists” of (roughly) equal length. So the left is i to $\frac{i+j}{2}$ and the right is $\frac{i+j}{2} + 1$ to j

Conquer:

Find all 8 return values for each half, we’ll have a $left$ and $right$ version of each

Combine:

Use the 16 return values from the conquer step to identify the 8 return values for this step (details on the next slide)

Improve D&C Combine Step

Finding the Max Sum Subarray (range and sum):

Use the same process as before!

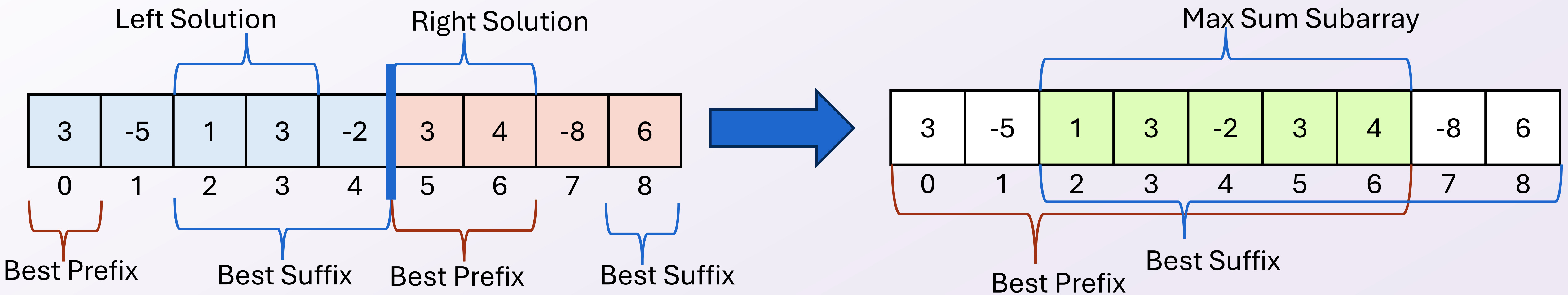
It will be one of: Left Solution, Right Solution, or Suffix+Prefix

Finding the Best Prefix (end and sum):

It will be one of: Left prefix, entire left + right prefix

Finding the Best Suffix (start and sum):

It will be one of: right suffix, entire right + left suffix



Finding the best prefix - Justification

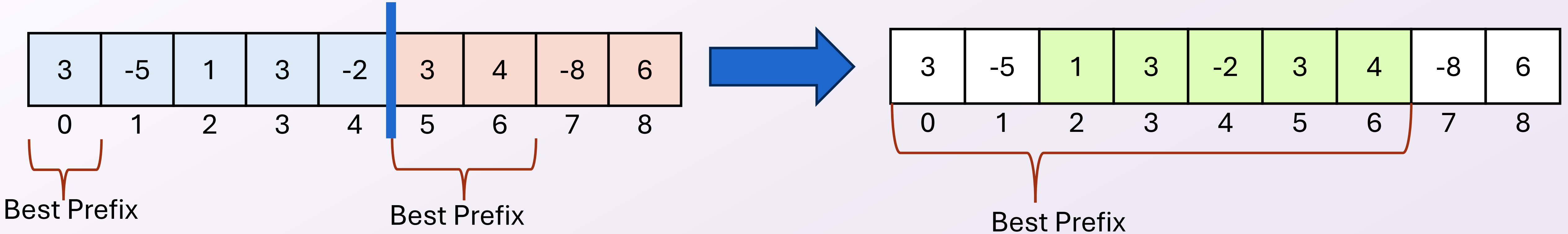
Finding the Best Prefix (end and sum):

It will be one of: Left prefix, entire left + right prefix

Proof:

Case 1: The best prefix is entirely on the left. In this case the answer will match the best prefix of the left subproblem

Case 2: The best prefix has at least one element from the right. In this case we must take the entire left half, then add in the best prefix from the right



Finding the best suffix - Justification

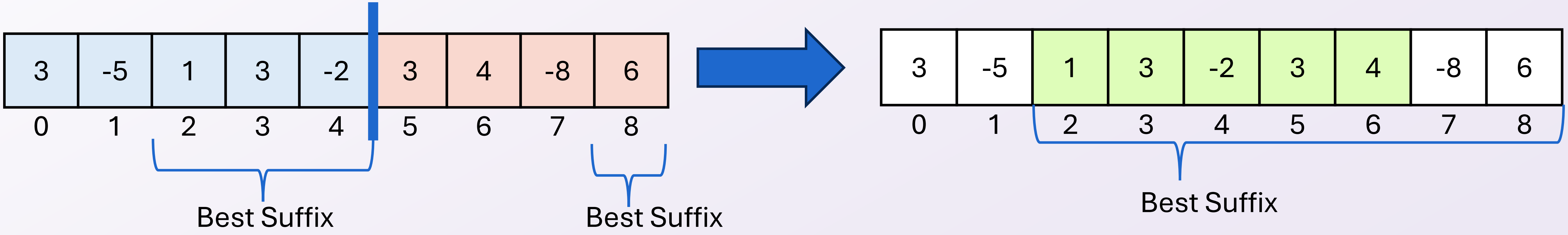
Finding the Best Suffix (start and sum):

It will be one of: right suffix, entire right + left suffix

Proof:

Case 1: The best suffix is entirely on the right. In this case the answer will match the best suffix of the right subproblem

Case 2: The best suffix has at least one element from the left. In this case we must take the entire right half, then add in the best suffix from the left



Running Time (Improved D&C)

2 subproblems

Each size $\frac{n}{2}$

$O(1)$ time to divide

$O(1)$ time to combine

Base Case:

If $i = j$ then: start=i, end =i, max sum=$arr[i]$, suffix start =i, suffix sum=$arr[i]$, prefix start =i, prefix sum=$arr[i]$, and total sum=$arr[i]$

Divide:

Split the list into two "sublists" of (roughly) equal length. So the left is i to $\frac{i+j}{2}$ and the right is $\frac{i+j}{2} + 1$ to j

Conquer:

Find all 8 return values for each half, we'll have a $left$ and $right$ version of each

Combine:

Use the 16 return values from the conquer step to identify the 8 return values for this step (details on the next slide)

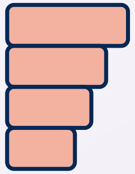
Overall: $T(n) = 2T\left(\frac{n}{2}\right) + 1$

Improved D&C Recurrence Solution

Master Theorem: Suppose that $T(n) = a \cdot T(n/b) + O(n^k)$ for $n > b$.

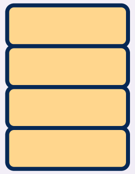
If $a < b^k$ then $T(n)$ is $O(n^k)$

- Cost is dominated by work at top level of recursion



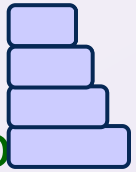
If $a = b^k$ then $T(n)$ is $O(n^k \log n)$

- Total cost is the same for all $\log_b n$ levels of recursion



If $a > b^k$ then $T(n)$ is $O(n^{\log_b a})$

- Note that $\log_b a > k$ in this case
- Cost is dominated by total work at lowest level of recursio



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

$a = 2, b = 2, k = 0$, so $a > b^k$: Solution: $O(n^{log_b a}) = O\left(n^{log_2 2}\right) = O(n)$

Binary Tree Diameter

Binary Trees – Vocab Review

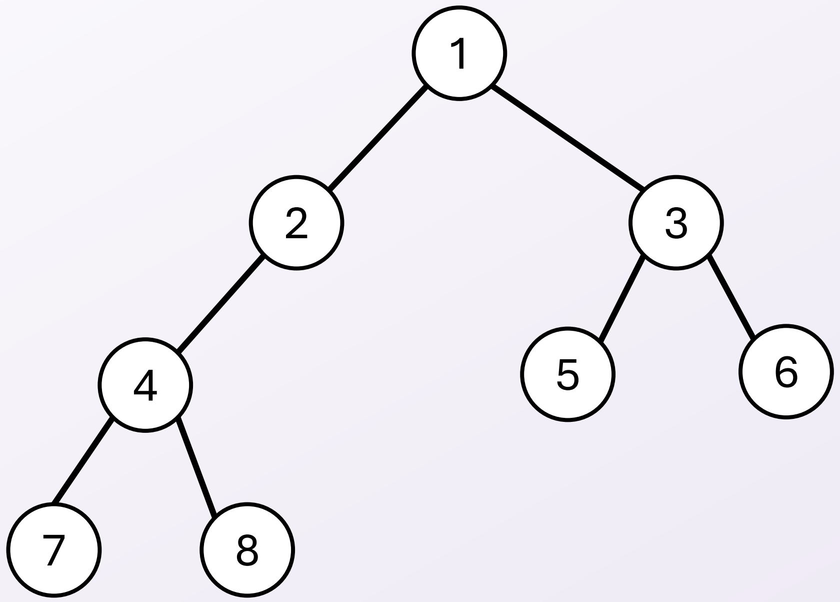
Nodes: Objects in the tree (labelled 1-8 here). They contain a value and may have a link to up to two other nodes

Child Node: a node linked to by some other node, that node is called its “parent”. E.g. 4 is the child of 2

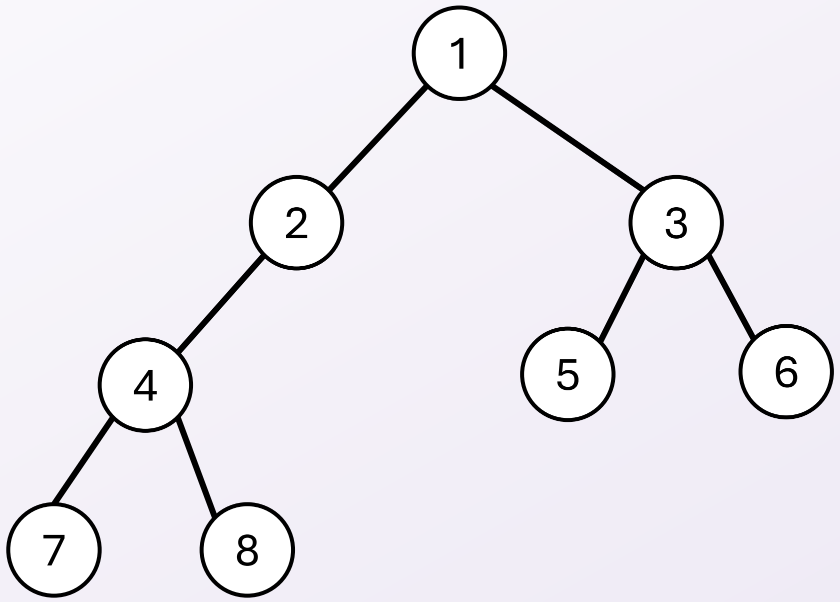
Sibling Nodes: two nodes that share a parent. E.g. 2 and 3 are siblings

Root Node: The unique node which has no parent. Node 1 is the root

Leaf Nodes: Nodes that have no children. 5,6,7, and 8 here



Binary Tree Height - Definition

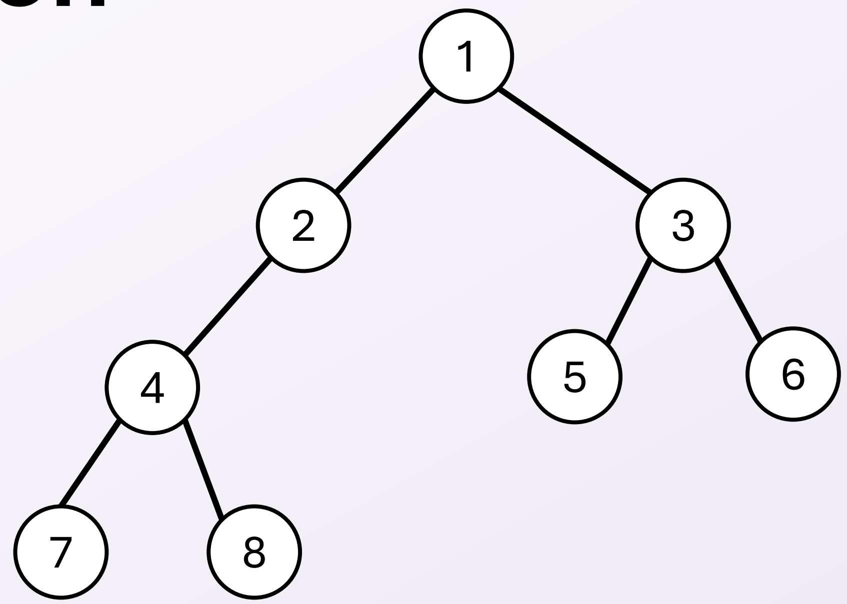


Distance: The distance between two nodes is the number of links you must follow to get from one to the other. E.g. the distance from 2 to 8 is 2, the distance from 2 to 6 is 3.

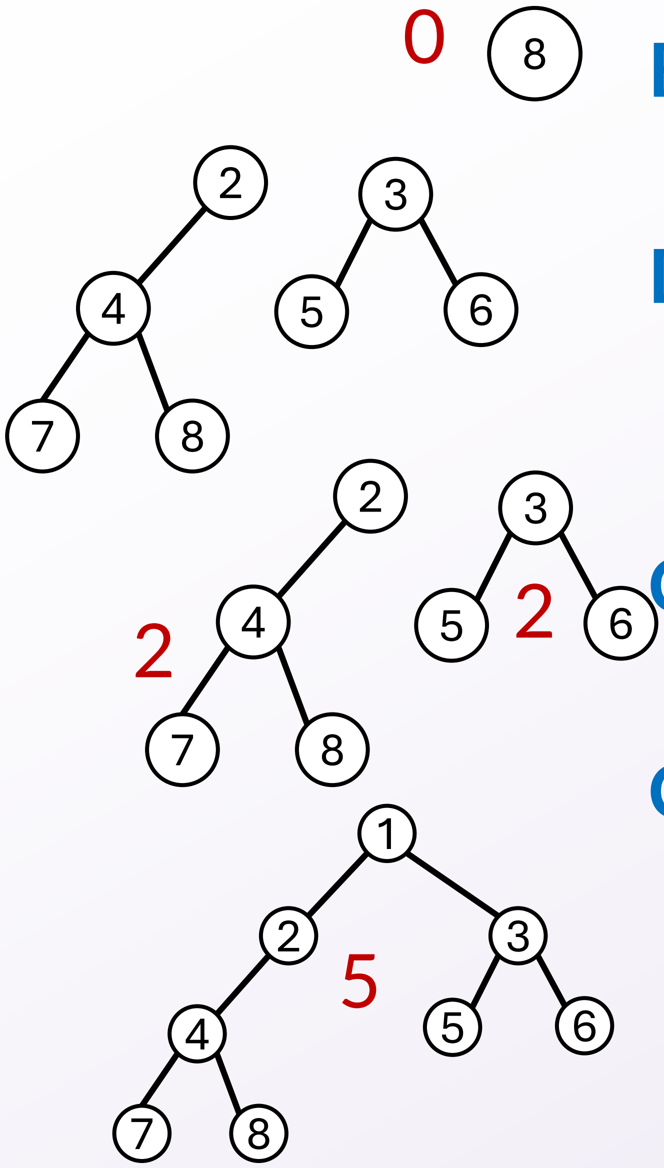
Height: The height of a binary tree is the largest distance from the root to some leaf. The height of this tree is 3 (1 is 3 away from 7)

Binary Tree Diameter - Definition

Diameter: The maximum distance between two nodes in a binary tree. The diameter of this tree is 5, because 7 is distance 5 from node 6.



Binary Tree Diameter – Incorrect Algorithm



Base Case:

If the current node is a leaf, the diameter is 0

Divide:

Split the tree into the left subtree and the right subtree

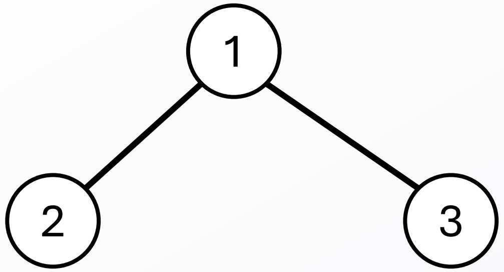
Conquer:

Find the diameter of each subtree

Combine:

Return the diameter of the left subtree + the diameter of the right subtree + 1

Incorrect Algorithm - Counterexample



Diameter of the left subtree: 0

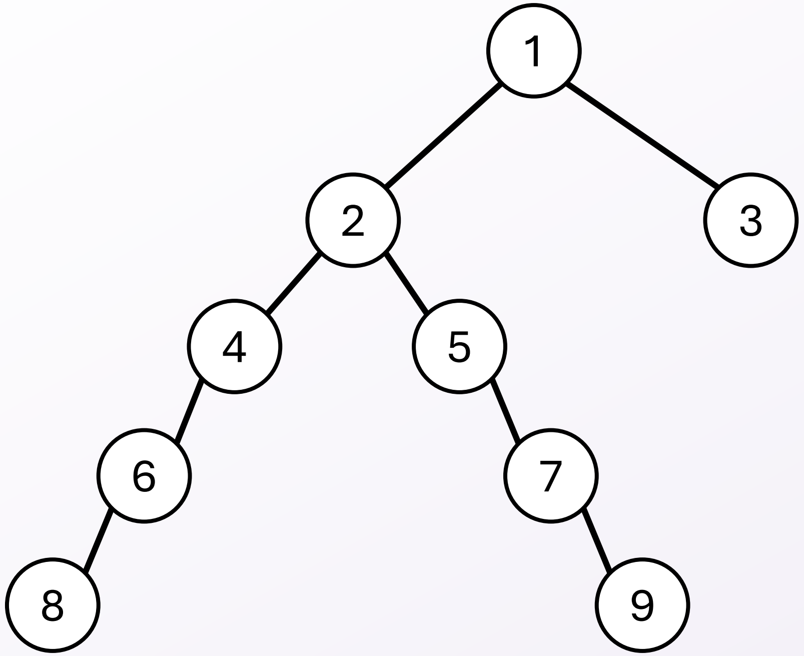
Diameter of the right subtree: 0

Diameter of the whole tree: 2

Diameter ended up being:

the distance to a left leaf + distance to a right leaf

Incorrect Algorithm - Counterexample



Diameter of the left subtree: 6

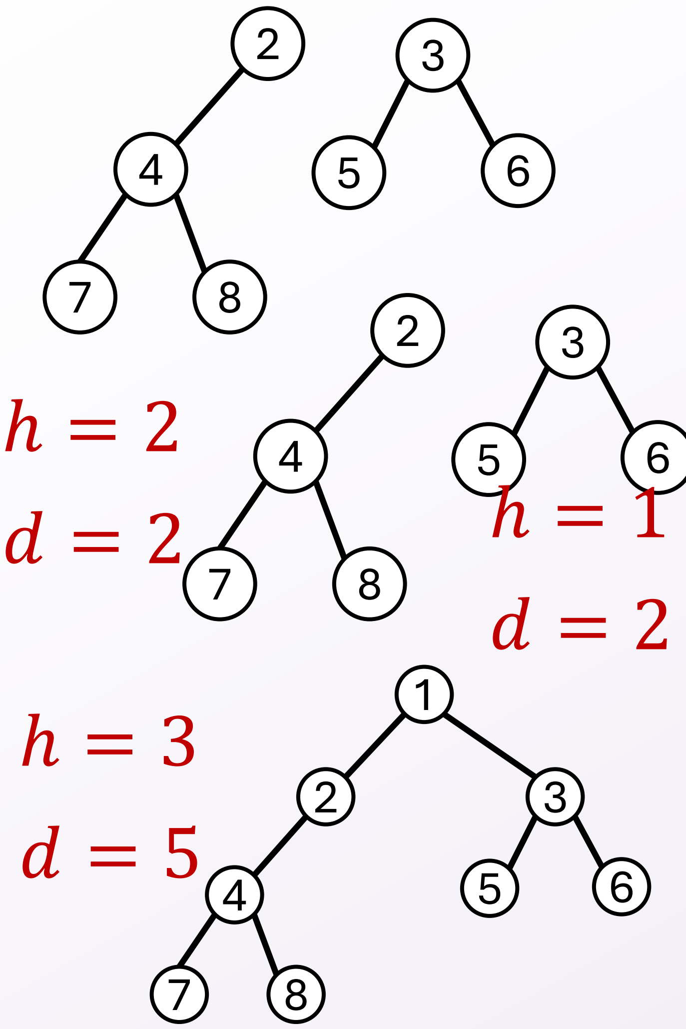
Diameter of the right subtree: 0

Diameter of the whole tree: 6

Diameter ended up being:

The diameter of a subtree

Binary Tree Diameter – Correct Algorithm



Base Case:

If the node is null the diameter and height are -1.

Divide:

Split the tree into the left subtree and the right subtree

Conquer:

Find the diameter and height of each subtree

Combine:

Height = 1 + max(left height, right height)

Diameter = max(left diameter,
right diameter,
left height + right height +2)