CS 483 - Data Structures and Algorithm Analysis

Lecture IV: Chapter 4

R. Paul Wiegand

George Mason University, Department of Computer Science

February 15, 2006

Outline

A General Plan for Divide-And-Conquer

1. Decompose a problem instance
2. Solve component problem instances
3. Combine components into composite solution

Example: $\sum_{i=0}^{n} a_i$

A General Plan for Divide-And-Conquer

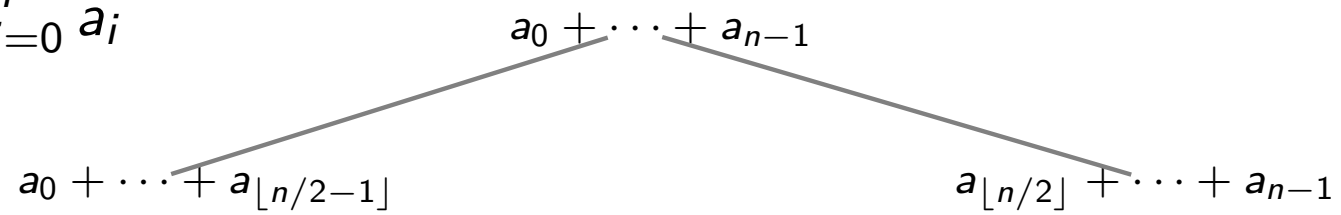
1. Decompose a problem instance
2. Solve component problem instances
3. Combine components into composite solution

Example: $\sum_{i=0}^{n} a_i$ $\qquad\qquad$ $a_0 + \cdots + a_{n-1}$

A General Plan for Divide-And-Conquer

1. Decompose a problem instance
2. Solve component problem instances
3. Combine components into composite solution

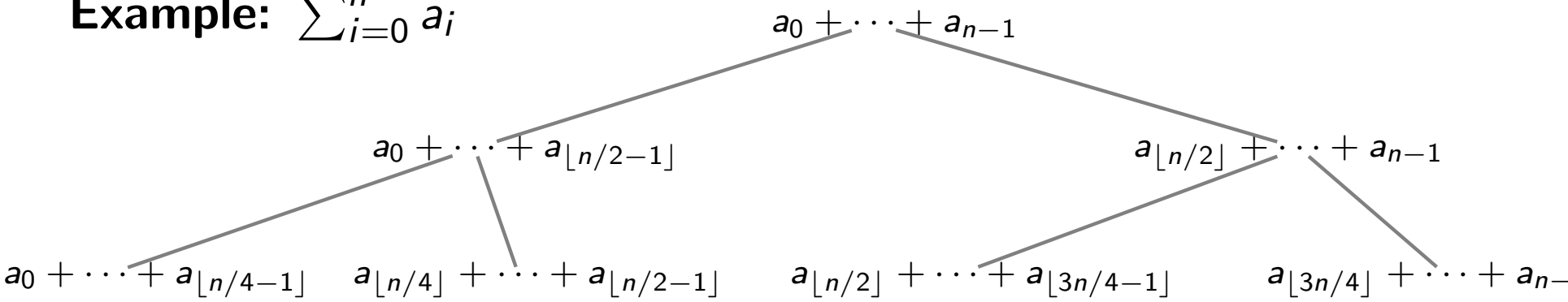
Example: $\sum_{i=0}^{n} a_i$



A General Plan for Divide-And-Conquer

1. Decompose a problem instance
2. Solve component problem instances
3. Combine components into composite solution

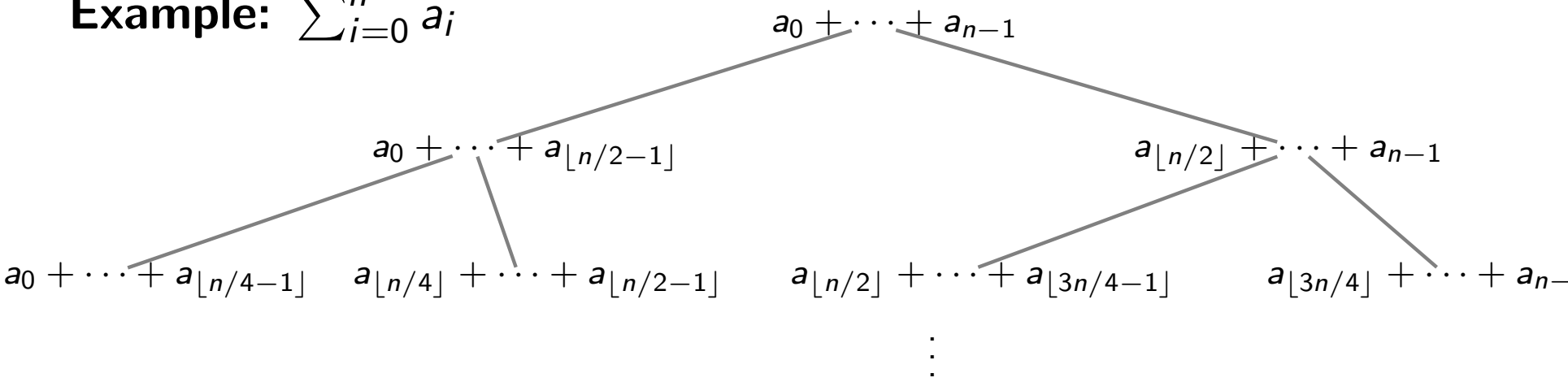
Example: $\sum_{i=0}^{n} a_i$



A General Plan for Divide-And-Conquer

1. Decompose a problem instance
2. Solve component problem instances
3. Combine components into composite solution

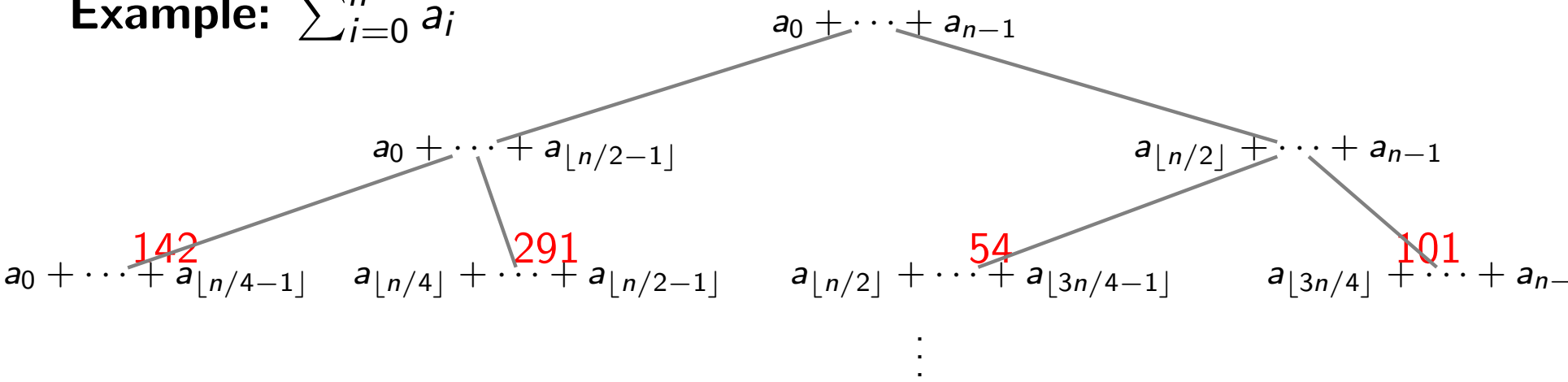
Example: $\sum_{i=0}^{n} a_i$



A General Plan for Divide-And-Conquer

1. Decompose a problem instance
2. Solve component problem instances
3. Combine components into composite solution

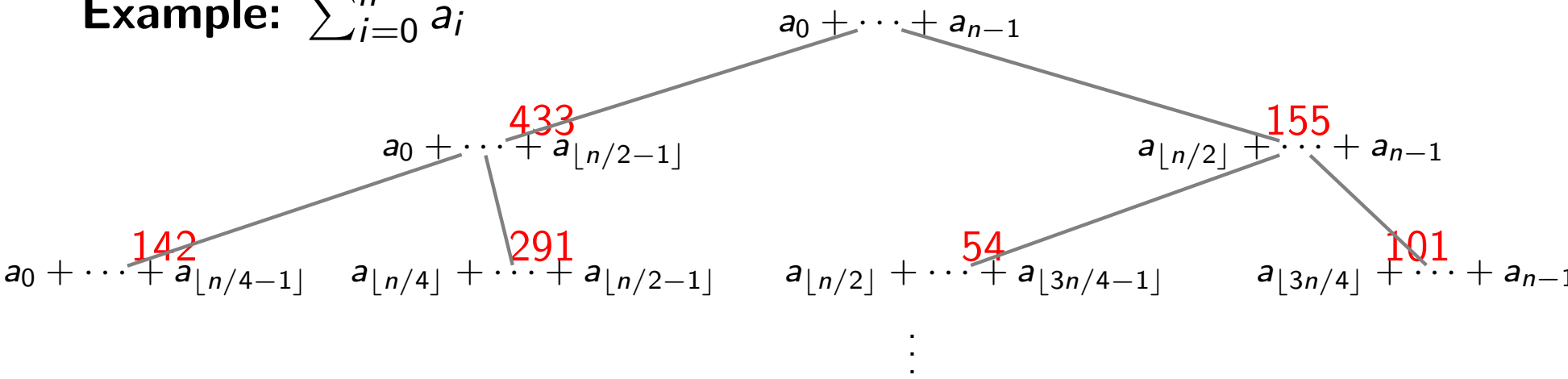
Example: $\sum_{i=0}^{n} a_i$



A General Plan for Divide-And-Conquer

1. Decompose a problem instance
2. Solve component problem instances
3. Combine components into composite solution

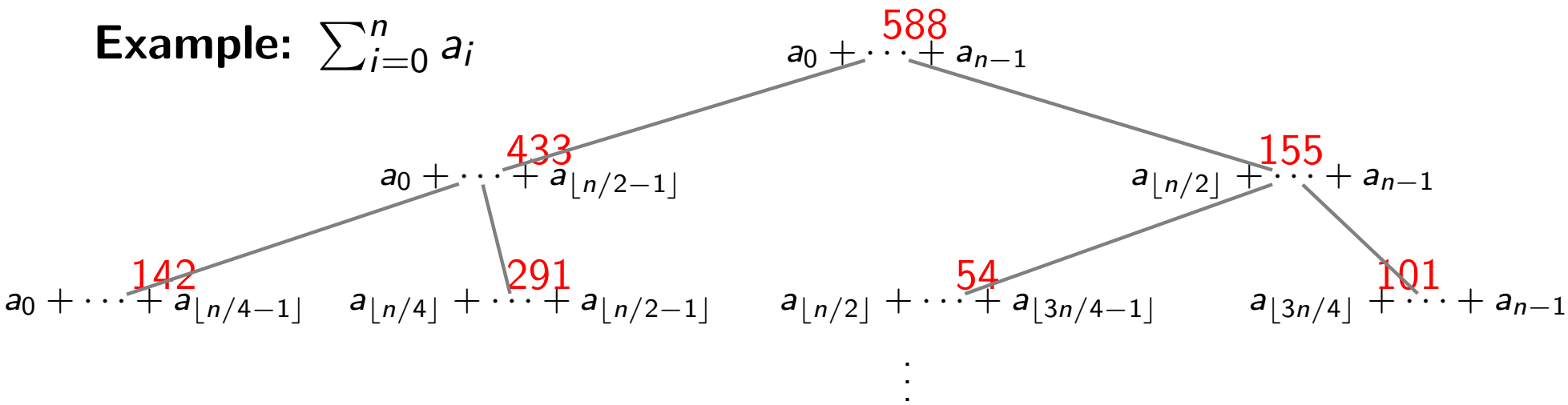
Example: $\sum_{i=0}^{n} a_i$



A General Plan for Divide-And-Conquer

1. Decompose a problem instance
2. Solve component problem instances
3. Combine components into composite solution

Example: $\sum_{i=0}^{n} a_i$



Some Comments about Divide-and-Conquer

- Is this D&C example more efficient than brute force?

Some Comments about Divide-and-Conquer

- Is this D&C example more efficient than brute force?

  No ... it is $\Theta(n)$

- Divide-and-conquer is not necessarily superior ...

Some Comments about Divide-and-Conquer

- Is this D&C example more efficient than brute force?

  No ... it is $\Theta(n)$
- Divide-and-conquer is not necessarily superior ...
- But many times it is, and many of the most efficient algorithms in CS are D&C
- D&C typically involves recursion (at least conceptually)
- D&C is well-suited for parallelization

The Master Theorem

- More generally, a problem of size n can be partitioned into a instances of non-overlapping components of size $\frac{n}{b}$ such that $a \geq 1$ and $b > 1$ (* we assume n is a power of b for simplicity)
- Given this, the *general divide-and-conquer recurrence* can be defined as: $T(n) := aT\left(\frac{n}{b}\right) + f(n)$
- This generalization allows us an analysis short-cut:

Master Theorem

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in the general divide-and-conquer recurrence then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

The Master Theorem

- More generally, a problem of size n can be partitioned into a instances of non-overlapping components of size $\frac{n}{b}$ such that $a \geq 1$ and $b > 1$ (* we assume n is a power of b for simplicity)
- Given this, the *general divide-and-conquer recurrence* can be defined as: $T(n) := aT\left(\frac{n}{b}\right) + f(n)$
- This generalization allows us an analysis short-cut:

Master Theorem

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in the general divide-and-conquer recurrence then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

For example:

- Recurrence for addition: $A(n) = 2A(n/2) + 1$

The Master Theorem

- More generally, a problem of size n can be partitioned into a instances of non-overlapping components of size $\frac{n}{b}$ such that $a \geq 1$ and $b > 1$ (* we assume n is a power of b for simplicity)
- Given this, the *general divide-and-conquer recurrence* can be defined as: $T(n) := aT\left(\frac{n}{b}\right) + f(n)$
- This generalization allows us an analysis short-cut:

Master Theorem

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in the general divide-and-conquer recurrence then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

For example:

- Recurrence for addition: $A(n) = 2A(n/2) + 1$
- Since $f(n) = 1$, it is in $\Theta(n^0)$
- So $a = 2$, $b = 2$ and $d = 0$

The Master Theorem

- More generally, a problem of size n can be partitioned into a instances of non-overlapping components of size $\frac{n}{b}$ such that $a \geq 1$ and $b > 1$ (* we assume n is a power of b for simplicity)
- Given this, the *general divide-and-conquer recurrence* can be defined as: $T(n) := aT\left(\frac{n}{b}\right) + f(n)$
- This generalization allows us an analysis short-cut:

Master Theorem

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in the general divide-and-conquer recurrence then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

For example:

- Recurrence for addition: $A(n) = 2A(n/2) + 1$
- Since $f(n) = 1$, it is in $\Theta(n^0)$
- So $a = 2$, $b = 2$ and $d = 0$
- Since $a > b^d$, $A(n) \in$?

The Master Theorem

- More generally, a problem of size n can be partitioned into a instances of non-overlapping components of size $\frac{n}{b}$ such that $a \geq 1$ and $b > 1$ (* we assume n is a power of b for simplicity)
- Given this, the *general divide-and-conquer recurrence* can be defined as: $T(n) := aT\left(\frac{n}{b}\right) + f(n)$
- This generalization allows us an analysis short-cut:

Master Theorem

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in the general divide-and-conquer recurrence then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

For example:

- Recurrence for addition: $A(n) = 2A(n/2) + 1$
- Since $f(n) = 1$, it is in $\Theta(n^0)$
- So $a = 2$, $b = 2$ and $d = 0$
- Since $a > b^d$, $A(n) \in \Theta(n^{\lg 2}) = \Theta(n)$

Specifying MergeSort

MergeSort($A[0 \ldots n-1]$)

```
if n > 1
   B[0 ... ⌊n/2⌋ − 1] ⟵c A[0 ... ⌊n/2⌋ − 1]
   C[0 ... ⌈n/2⌉ − 1] ⟵c A[⌊n/2⌋ ... n − 1]
   MergeSort(B)
   MergeSort(C)
   Merge(B,C,A)
```

Specifying MergeSort

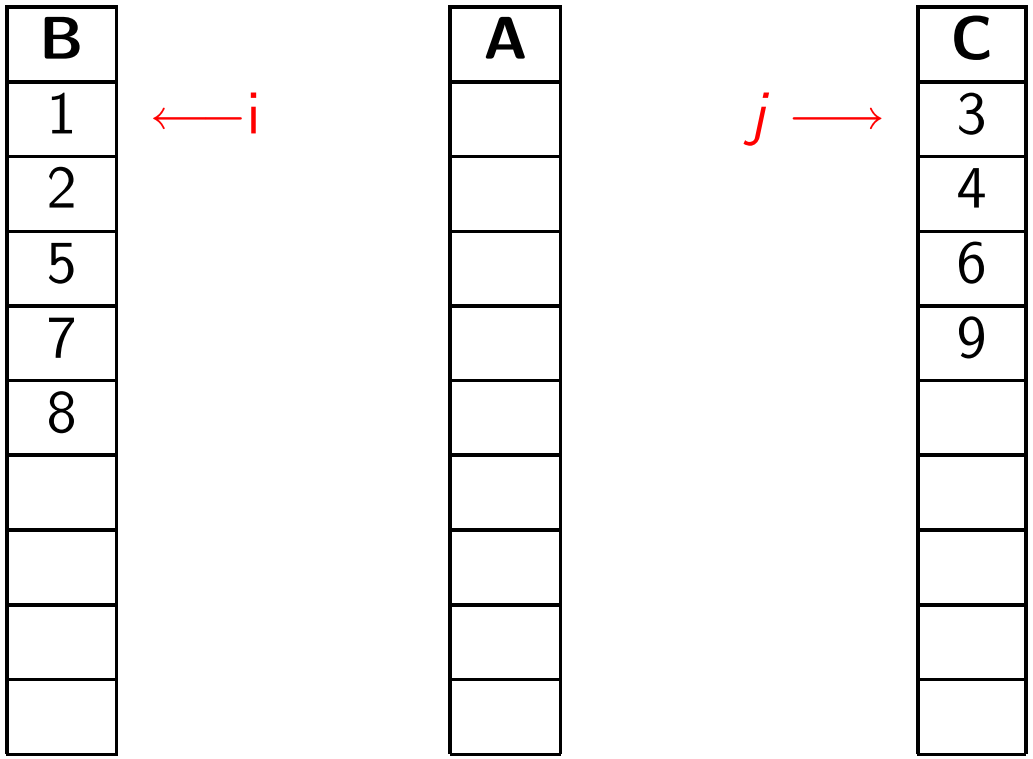
MergeSort($A[0 \ldots n-1]$)

```
if n > 1
  B[0 ... ⌊n/2⌋ − 1] ⟵_c A[0 ... ⌊n/2⌋ − 1]
  C[0 ... ⌈n/2⌉ − 1] ⟵_c A[⌊n/2⌋ ... n − 1]
  MergeSort(B)
  MergeSort(C)
  Merge(B,C,A)
```

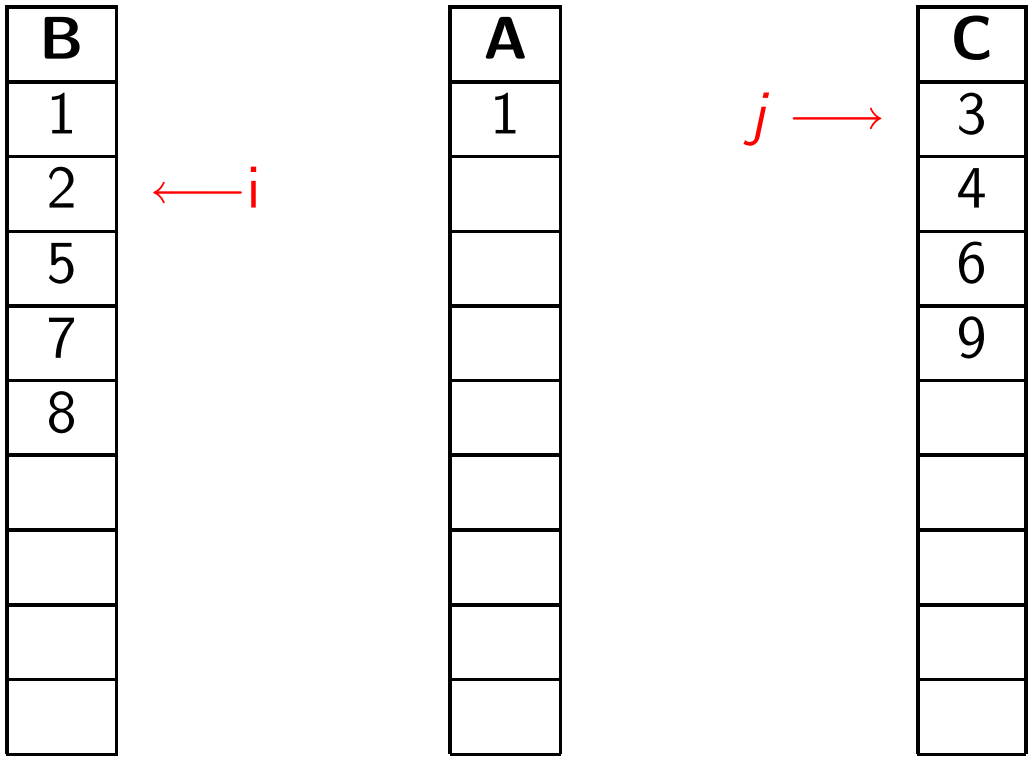
Merge($B[0 \ldots p-1], C[0 \ldots q-1], A[0 \ldots p+q-1]$)

```
i, j, k ⟵ 0
while i < p and j < q do
  if B[i] ≤ C[j] A[k] ⟵ B[i]; i++
  else A[k] ⟵ C[j]; j++
  k++
if i = p A[k ... p + q − 1] ⟵_c C[j ... q − 1]
else A[k ... p + q − 1] ⟵_c B[i ... p − 1]
```

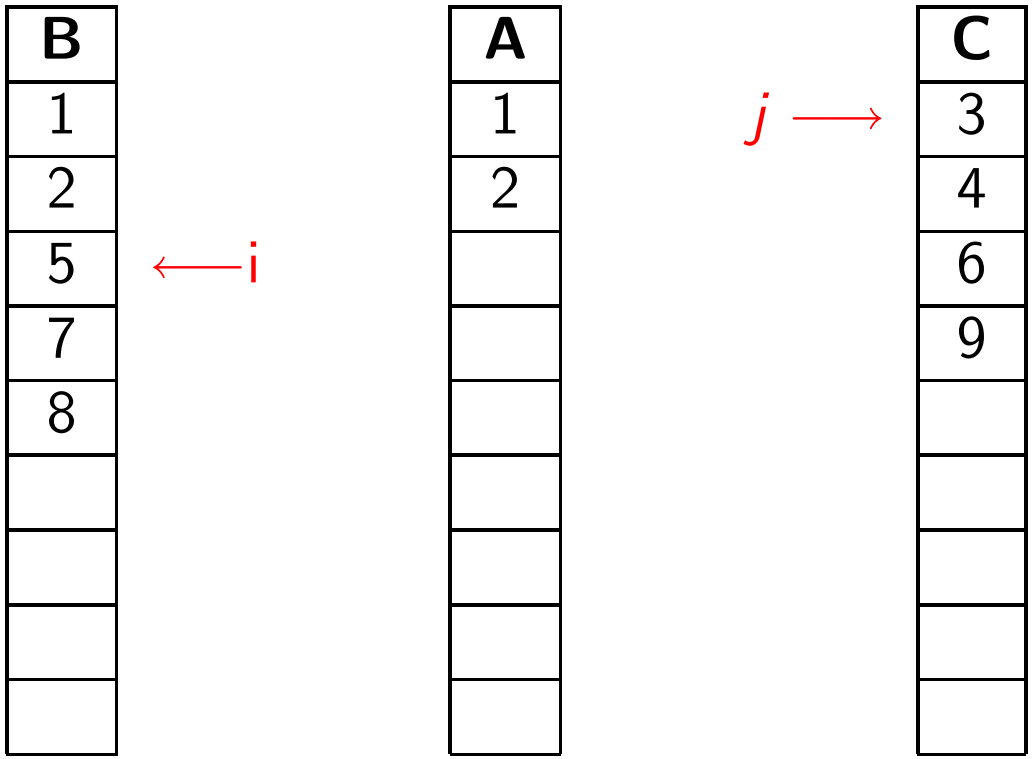
Merging Two Sorted Lists



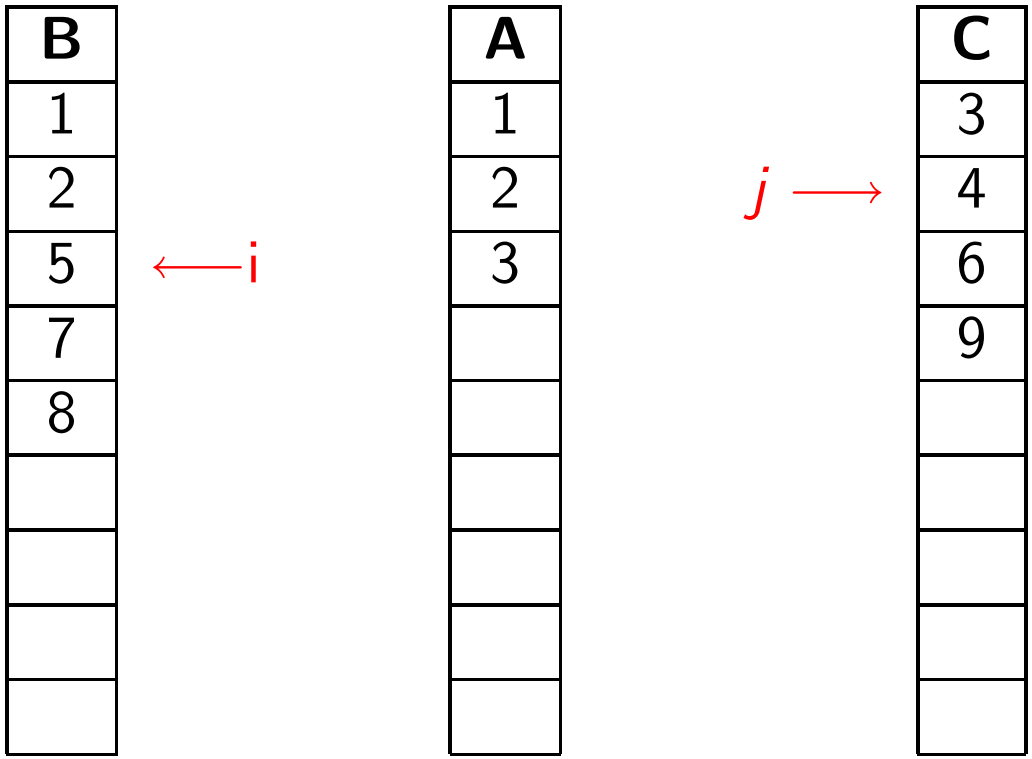
Merging Two Sorted Lists



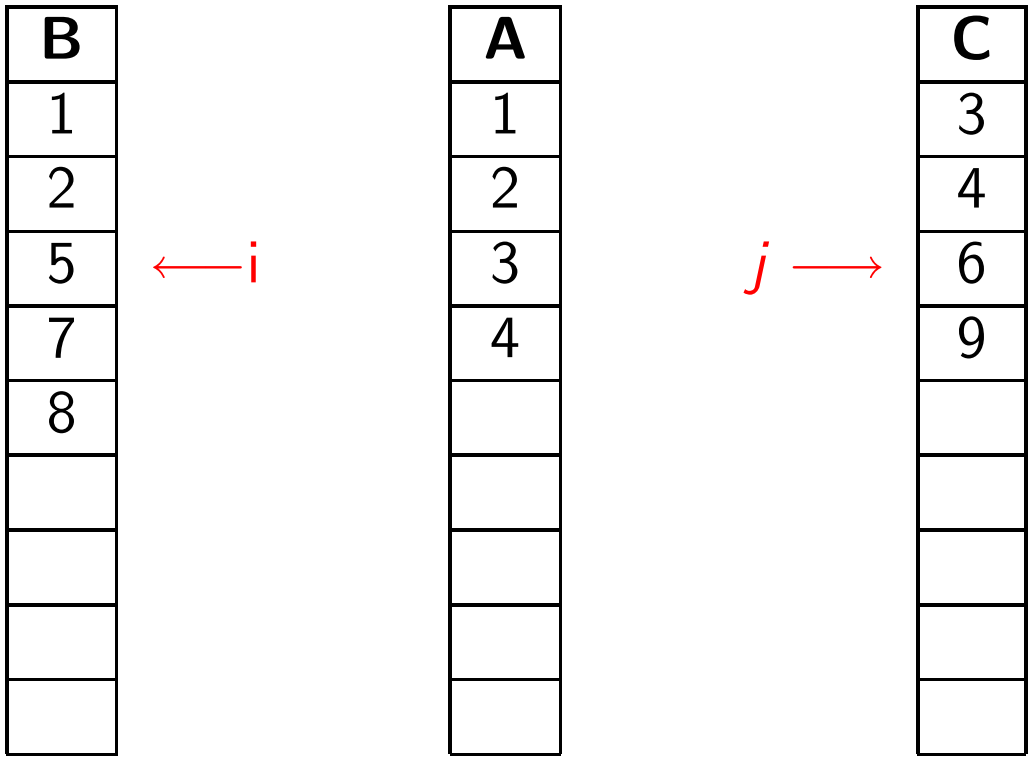
Merging Two Sorted Lists



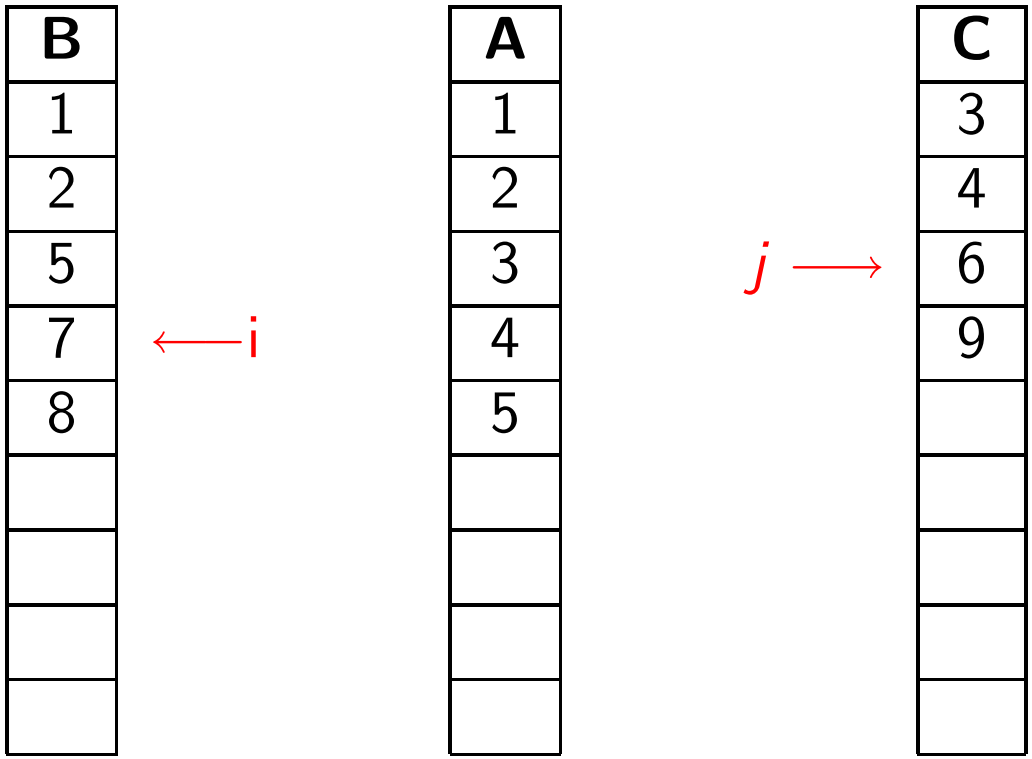
Merging Two Sorted Lists



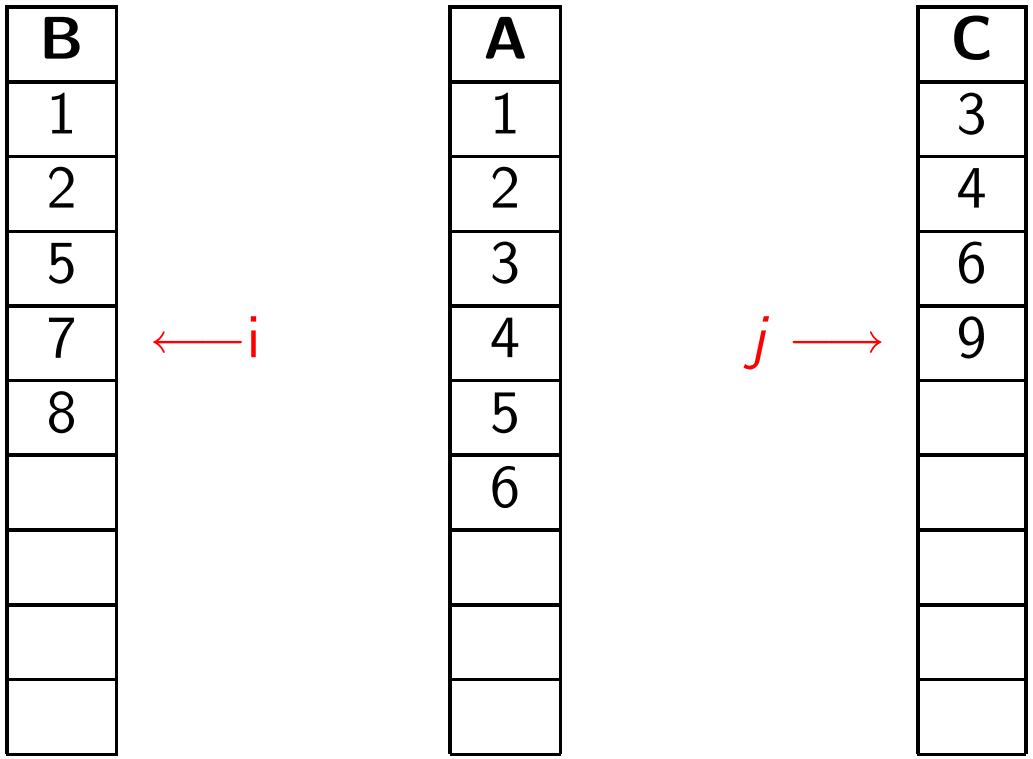
Merging Two Sorted Lists



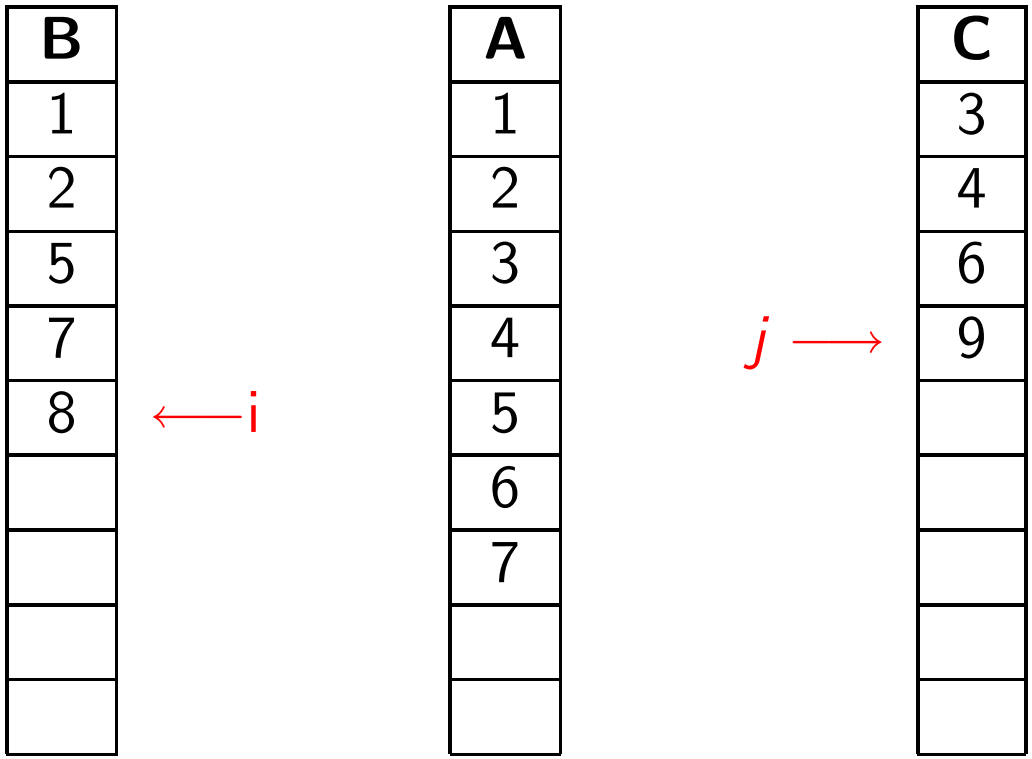
Merging Two Sorted Lists



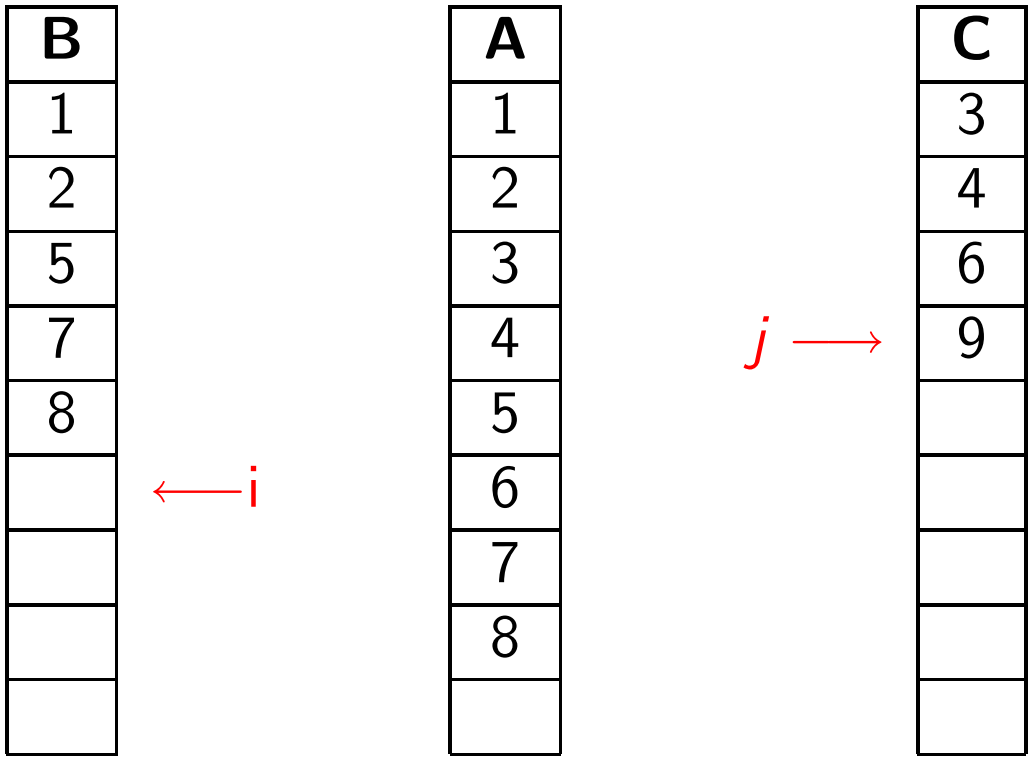
Merging Two Sorted Lists



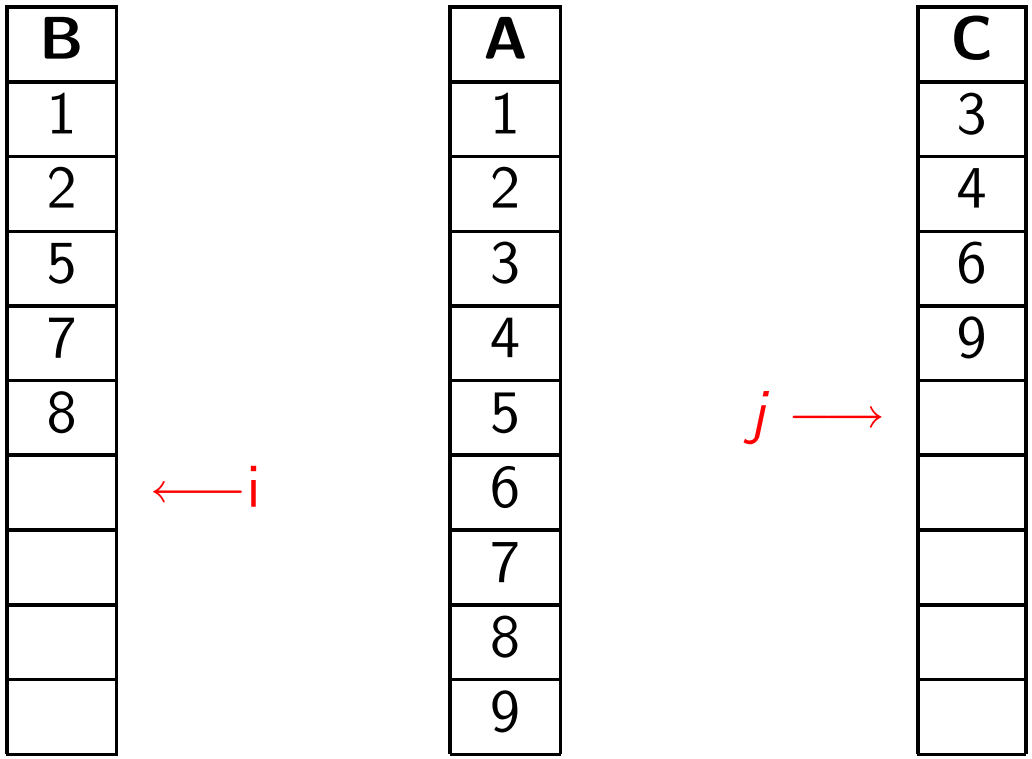
Merging Two Sorted Lists



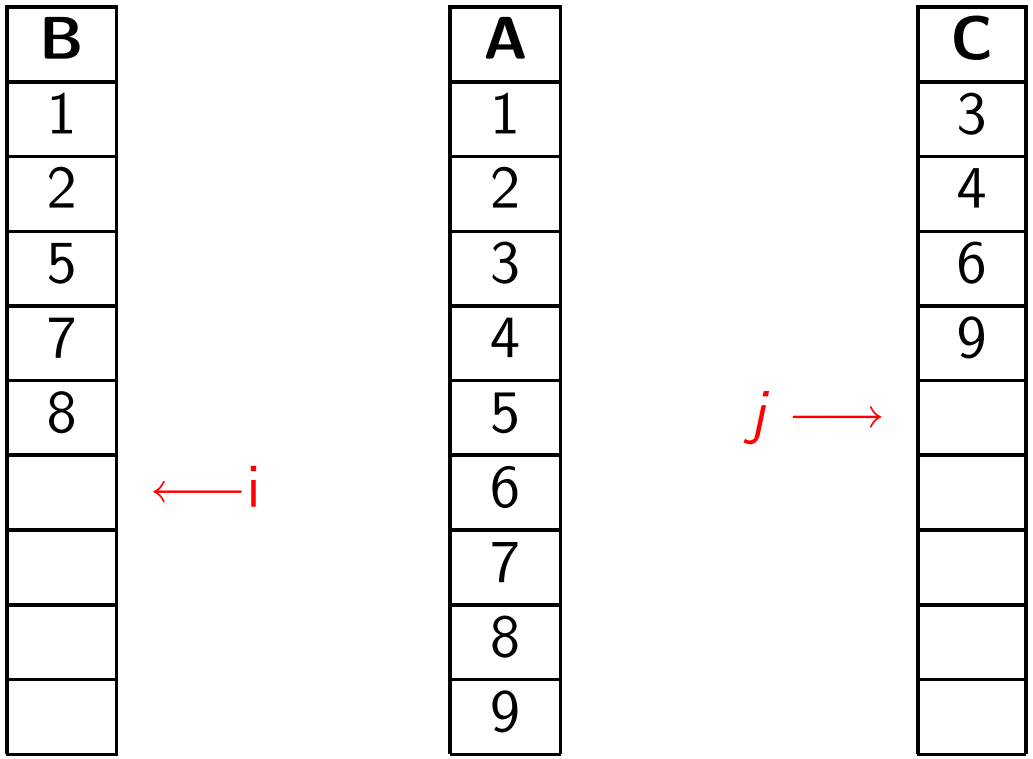
Merging Two Sorted Lists



Merging Two Sorted Lists

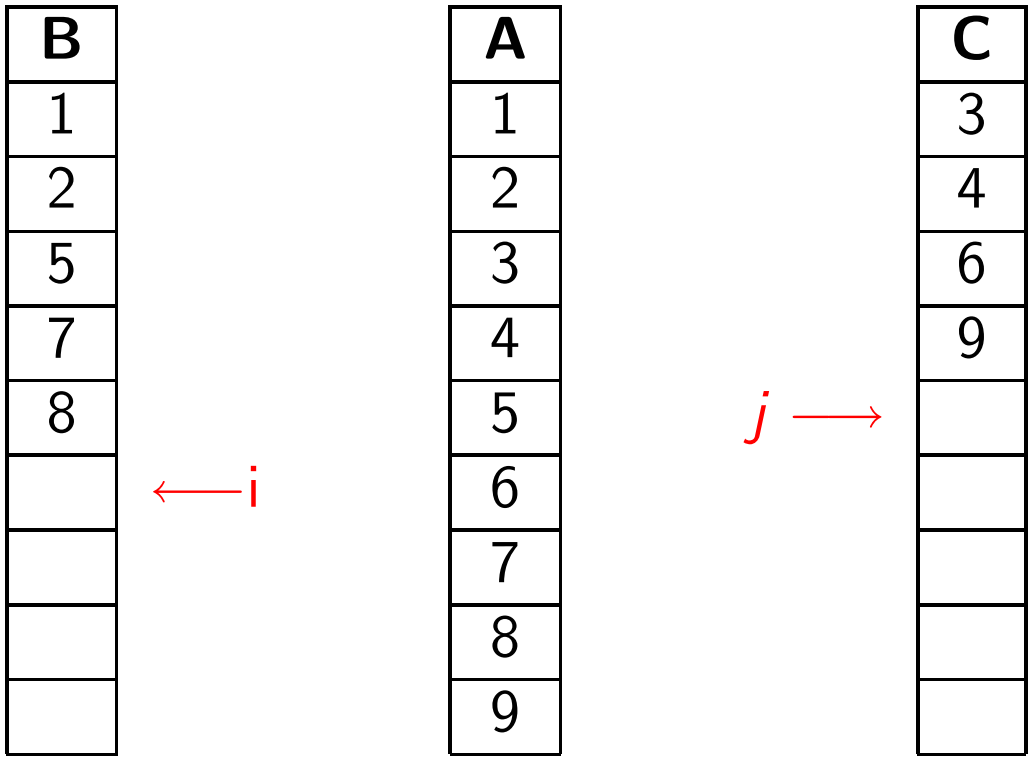


Merging Two Sorted Lists



How many comparisons?

Merging Two Sorted Lists



How many comparisons?

$n - 1$

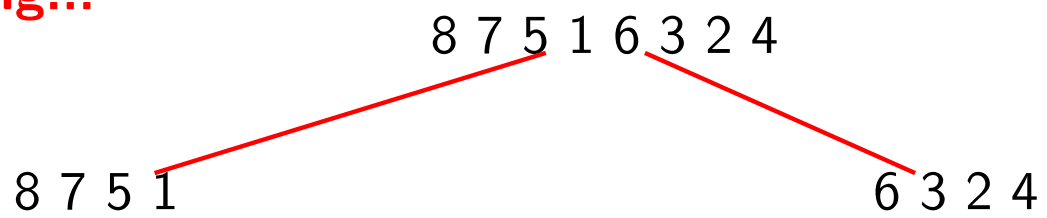
Decomposing & Combining

Decomposing...

8 7 5 1 6 3 2 4

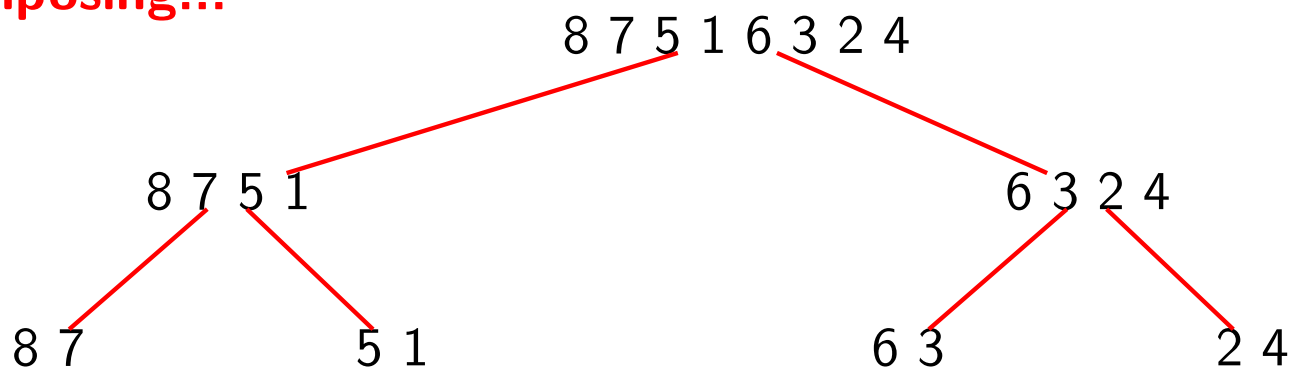
Decomposing & Combining

Decomposing...



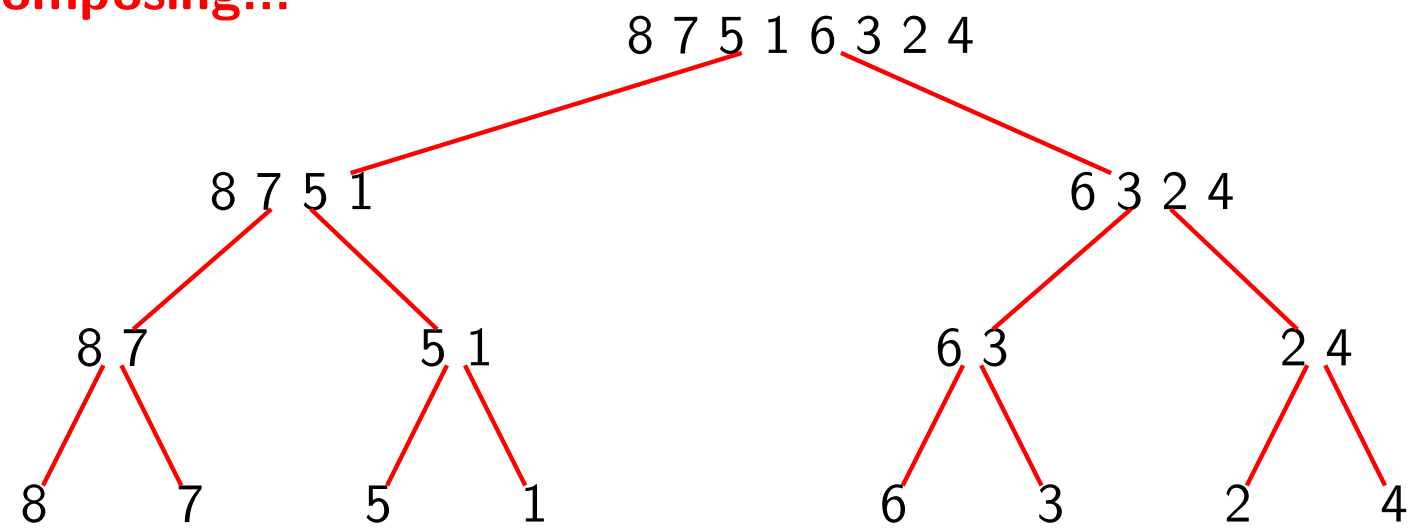
Decomposing & Combining

Decomposing...

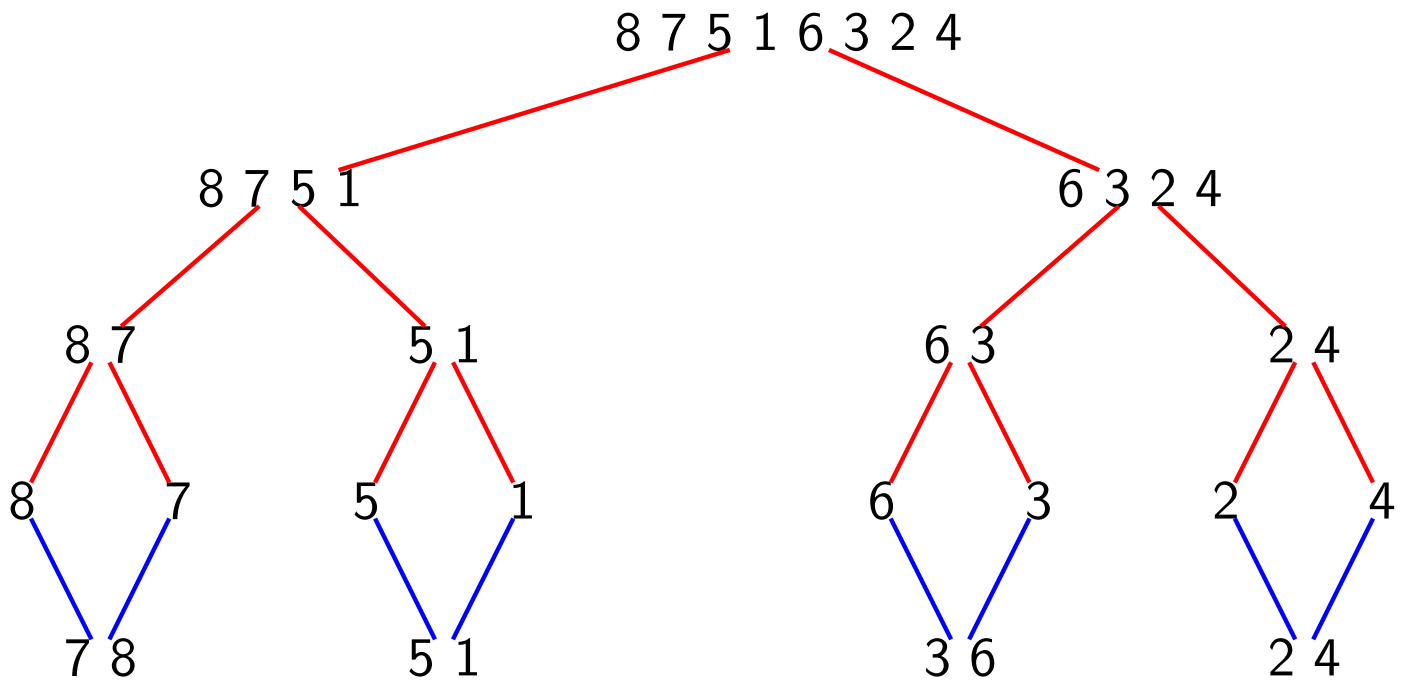


Decomposing & Combining

Decomposing...

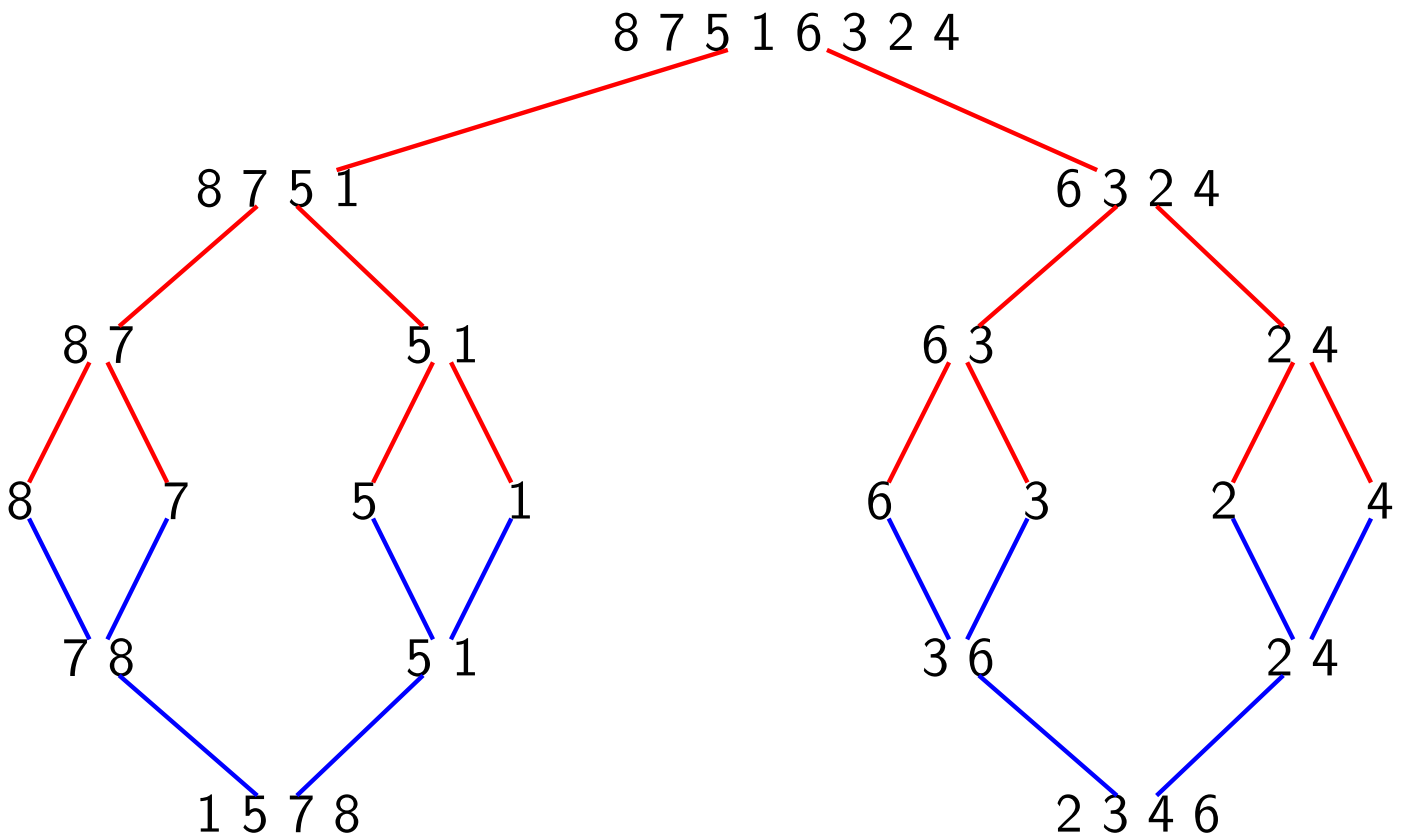


Decomposing & Combining



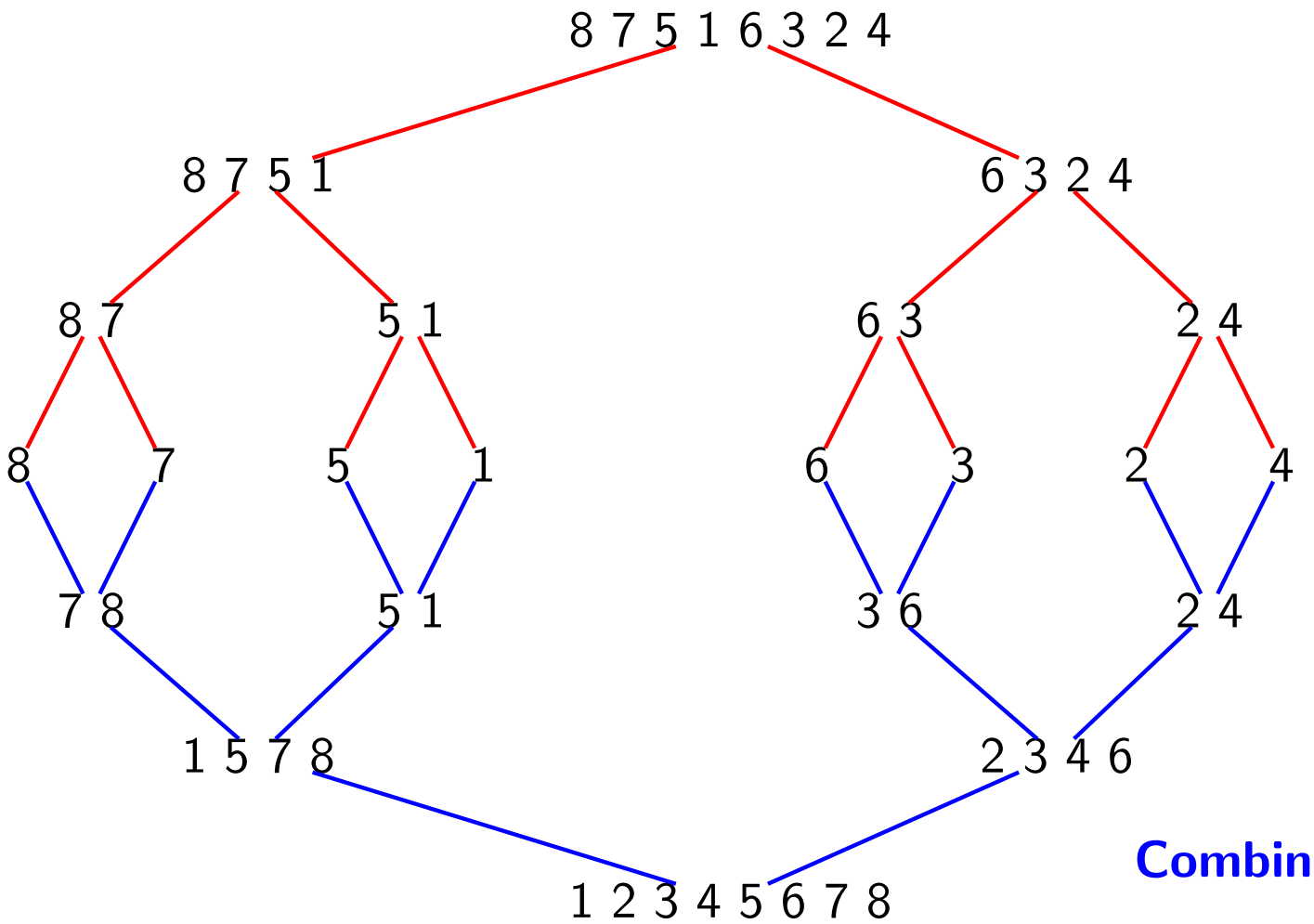
Combining...

Decomposing & Combining



Combining...

Decomposing & Combining



Combining...

Analyzing MergeSort

- We count key comparisons
- Assume `wlog` that n is a power of 2
- $C(n) = 2C(n/2) + M(n),\ n > 1, C(1) = 0$
- Applying the Mater Theorem:

Analyzing MergeSort

- We count key comparisons
- Assume `wlog` that n is a power of 2
- $C(n) = 2C(n/2) + M(n),\ n > 1, C(1) = 0$
- Applying the Mater Theorem:
  - $a = 2,\ b = 2$

Analyzing MergeSort

- We count key comparisons
- Assume `wlog` that n is a power of 2
- $C(n) = 2C(n/2) + M(n),\ n > 1, C(1) = 0$
- Applying the Mater Theorem:
  - $a = 2,\ b = 2$
  - In worst case: $M(n) \in \Theta(n^1)$, so $d = 1$

Analyzing MergeSort

- We count key comparisons
- Assume `wlog` that n is a power of 2
- $C(n) = 2C(n/2) + M(n),\ n > 1, C(1) = 0$
- Applying the Mater Theorem:
  - $a = 2,\ b = 2$
  - In worst case: $M(n) \in \Theta(n^1)$, so $d = 1$
  - $a = b^d$, so...?

Analyzing MergeSort

- We count key comparisons
- Assume `wlog` that n is a power of 2
- $C(n) = 2C(n/2) + M(n),\ n > 1, C(1) = 0$
- Applying the Mater Theorem:
  - $a = 2,\ b = 2$
  - In worst case: $M(n) \in \Theta(n^1)$, so $d = 1$
  - $a = b^d$, so...?
  - $C(n) \in \Theta(n \lg n)$

Specifying QuickSort

QuickSort($A[l \ldots r]$)

```
if l > r
  s ⟵ Partition(A[l ... r])
  QuickSort(A[l ... s − 1)
  QuickSort(A[s + 1 ... r)
```

Specifying QuickSort

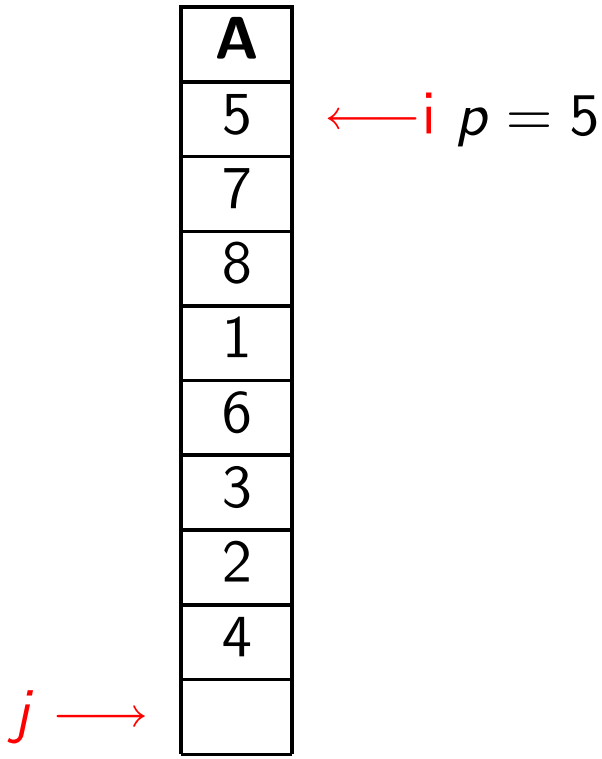
QuickSort($A[l \ldots r]$)

```
if l > r
  s ⟵ Partition(A[l ... r])
  QuickSort(A[l ... s − 1)
  QuickSort(A[s + 1 ... r)
```

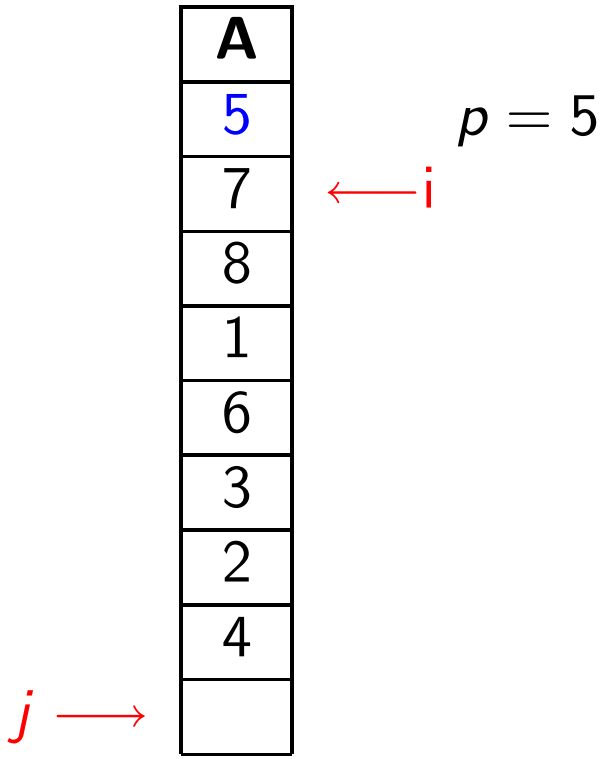
Partition($A[l \ldots r]$)

```
p ⟵ A[l]
i ⟵ l
j ⟵ r + 1
repeat
  repeat i++ until p ≥ A[i]
  repeat j−− until p ≤ A[j]
  Swap(A[i], A[j])
until i ≥ j
Swap(A[i], A[j])
Swap(A[l], A[j])
```

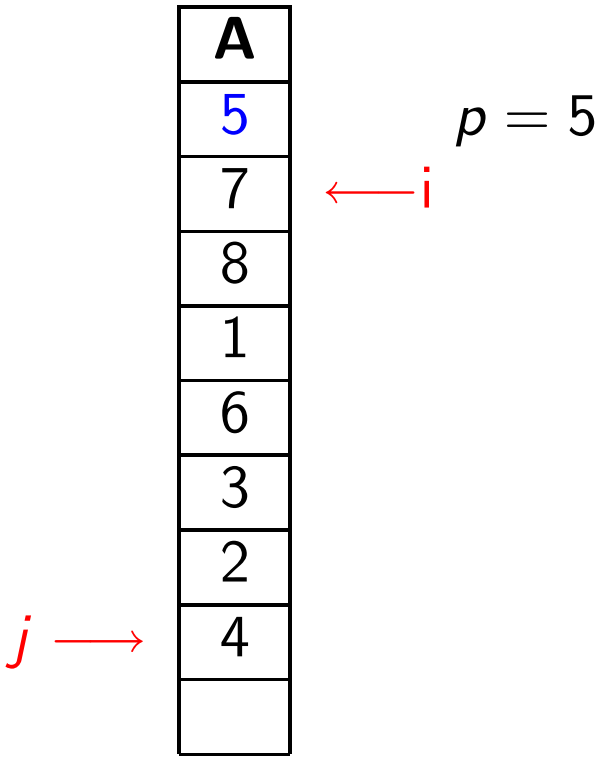
Partition



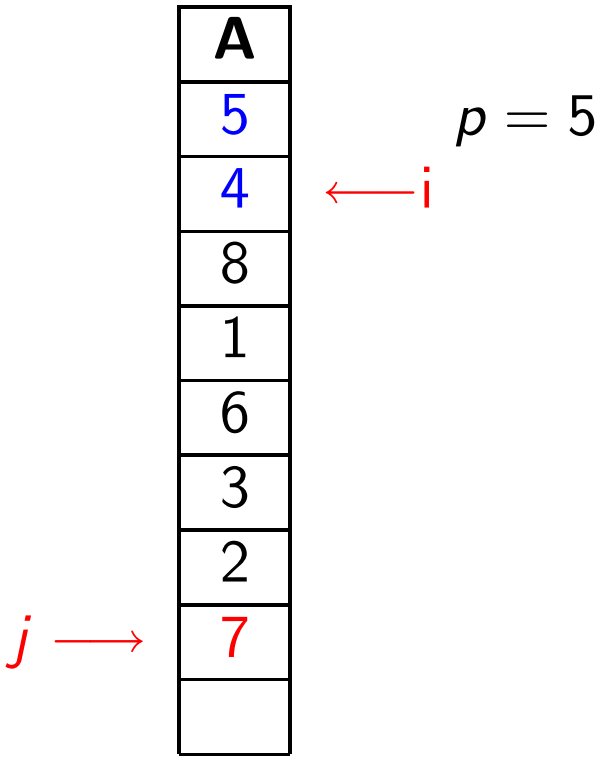
Partition



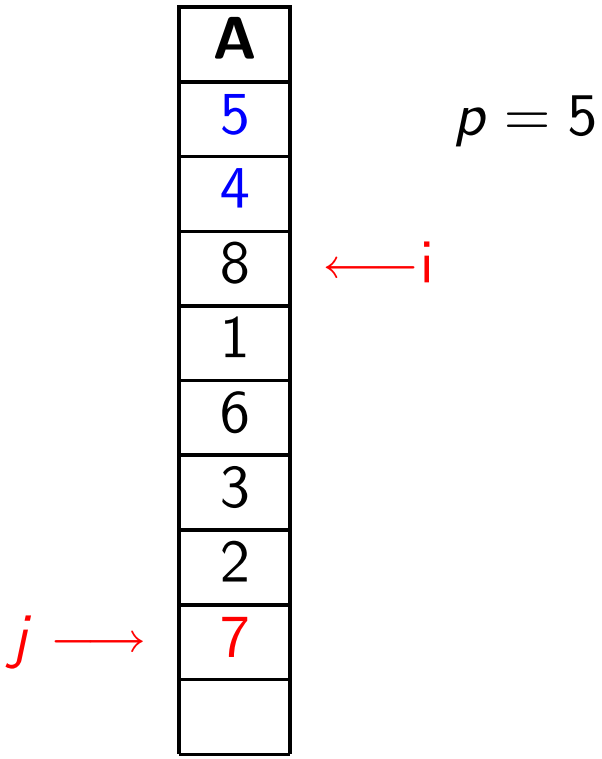
Partition



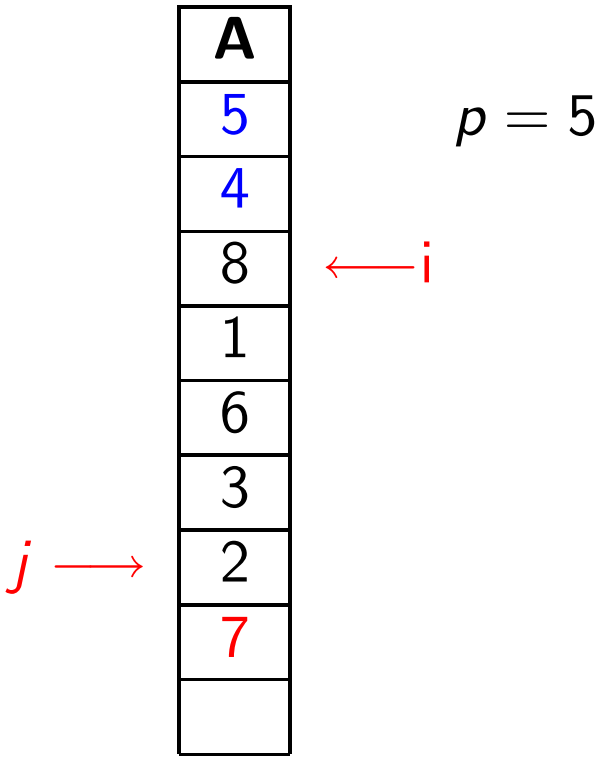
Partition



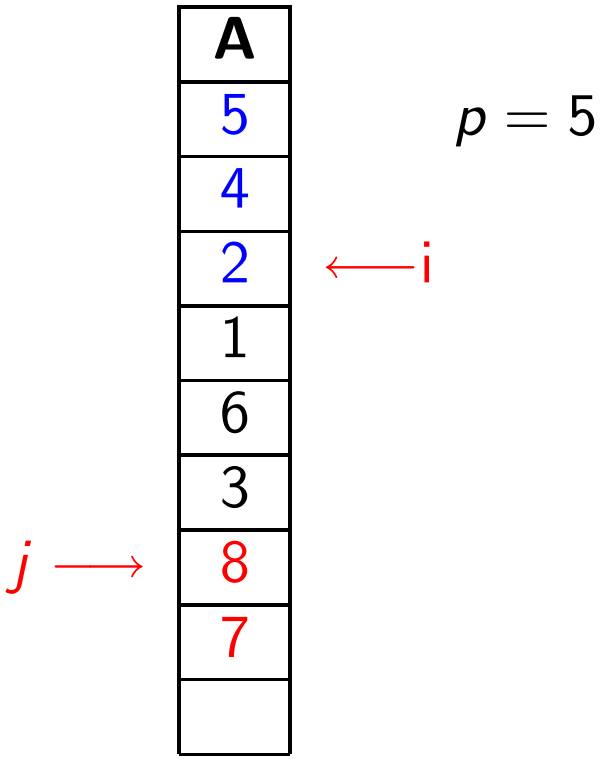
Partition



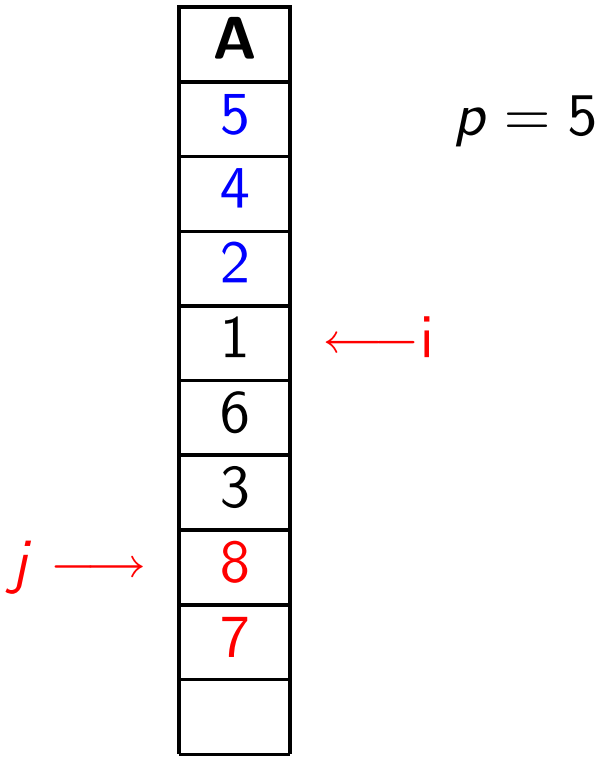
Partition



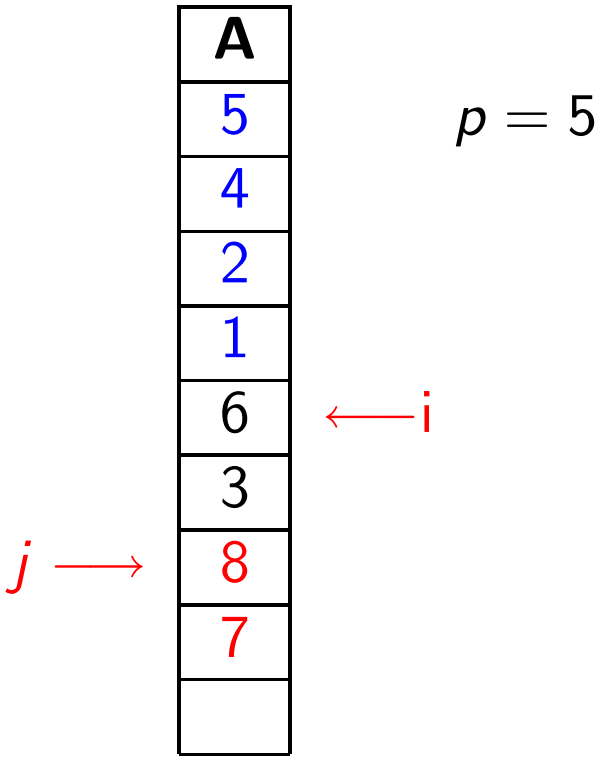
Partition



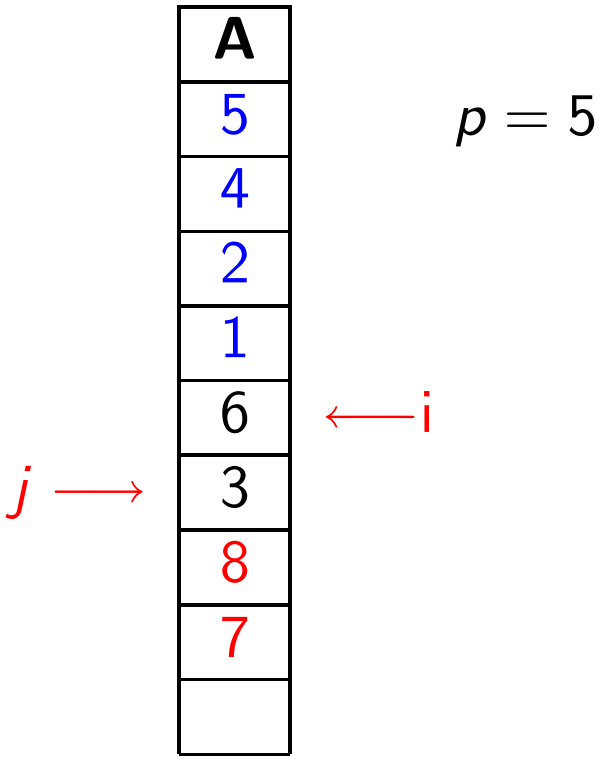
Partition



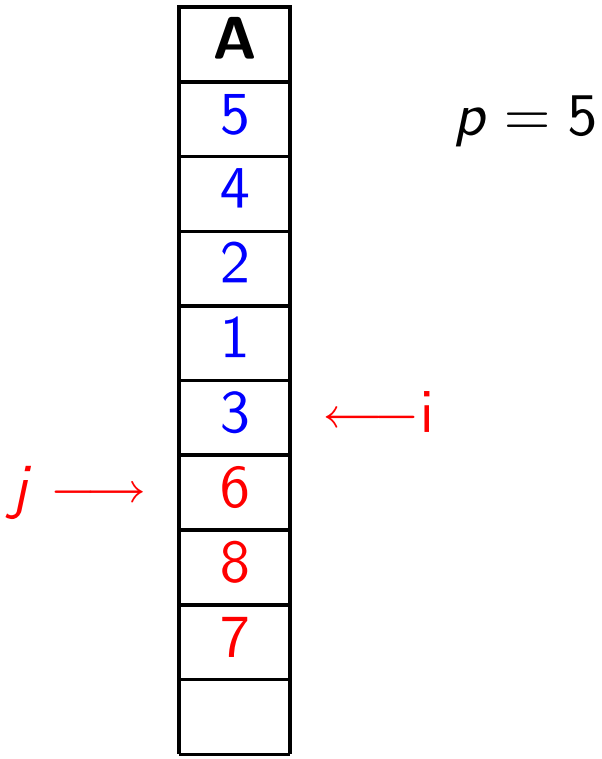
Partition



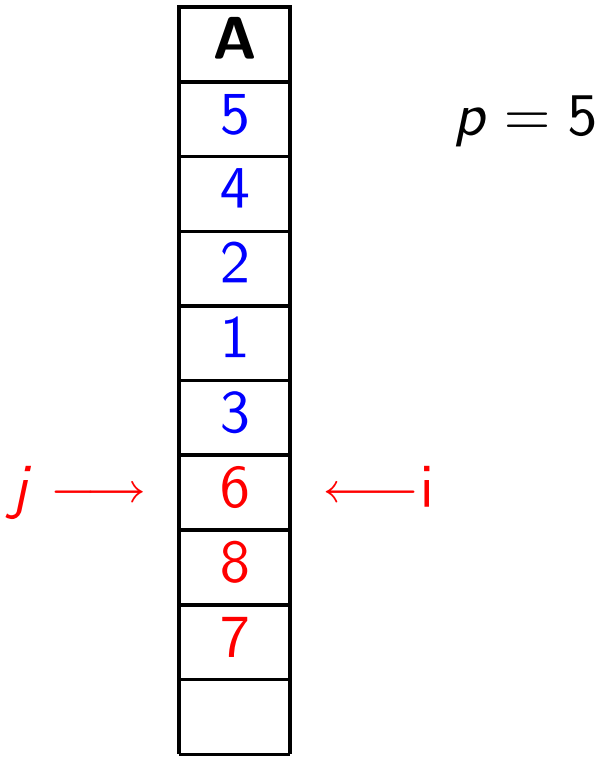
Partition



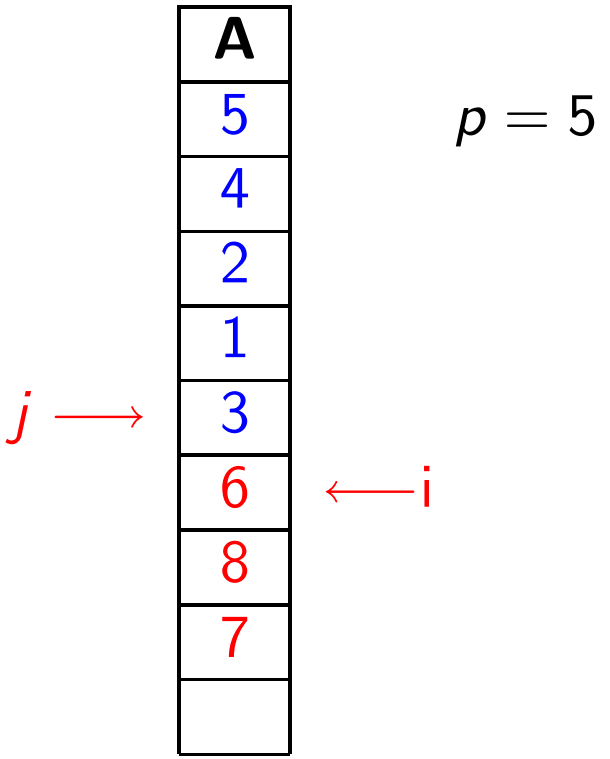
Partition



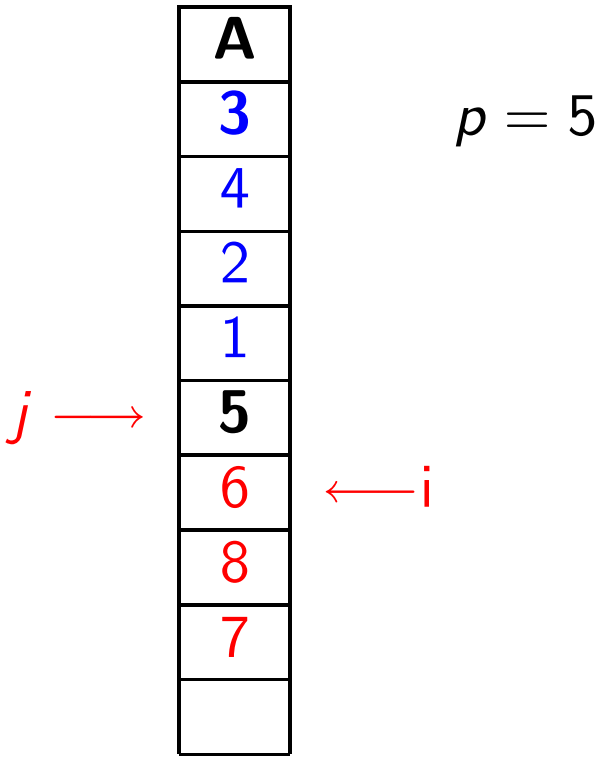
Partition



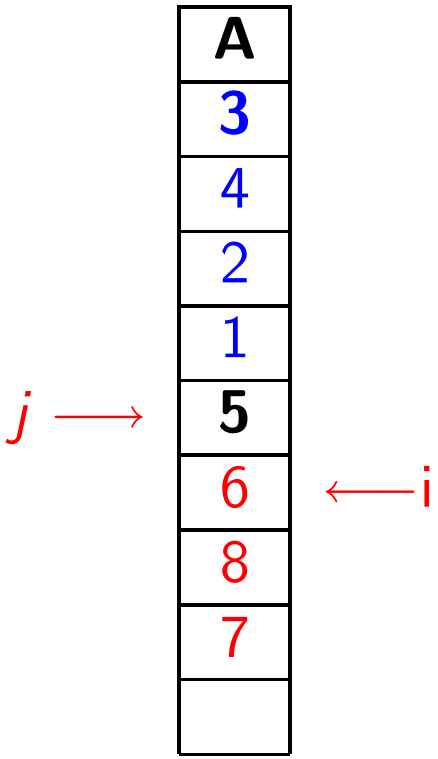
Partition



Partition



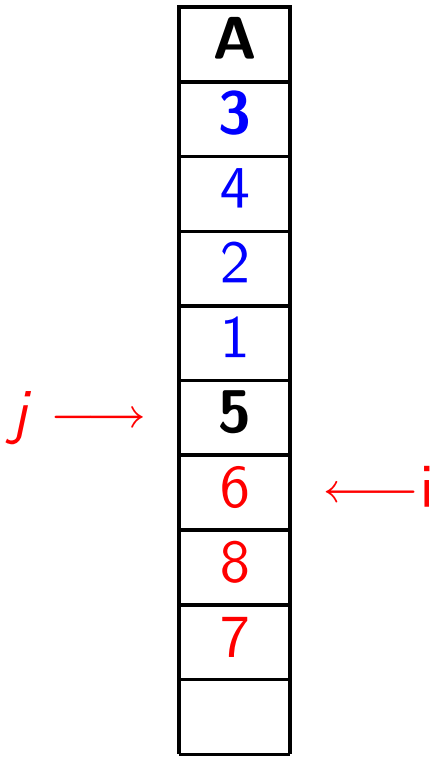
Partition



$p = 5$

How many comparisons?

Partition



$p = 5$

How many comparisons?

n

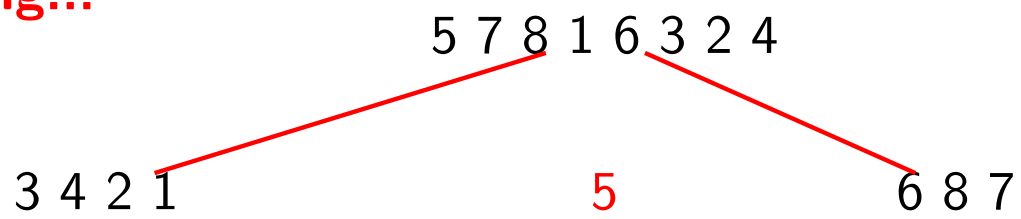
Decomposing & Combining

Decomposing...

5 7 8 1 6 3 2 4

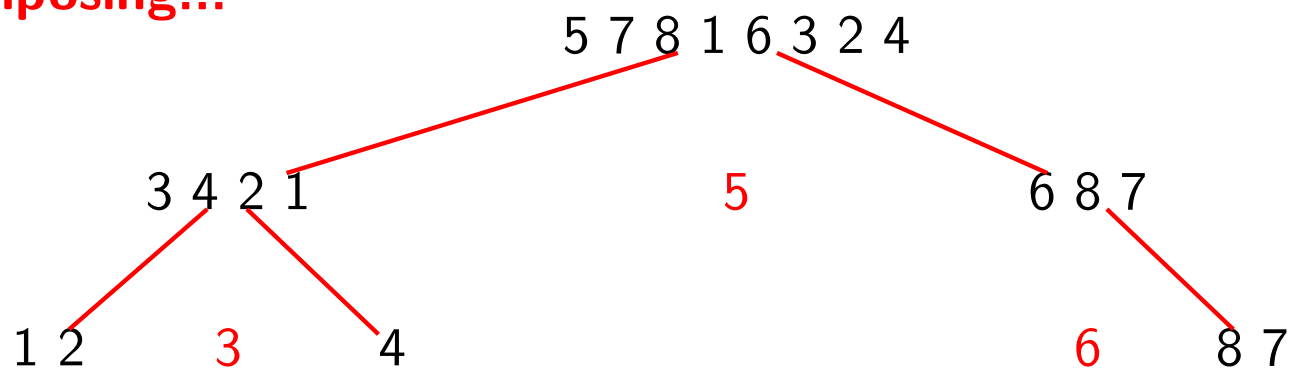
Decomposing & Combining

Decomposing...



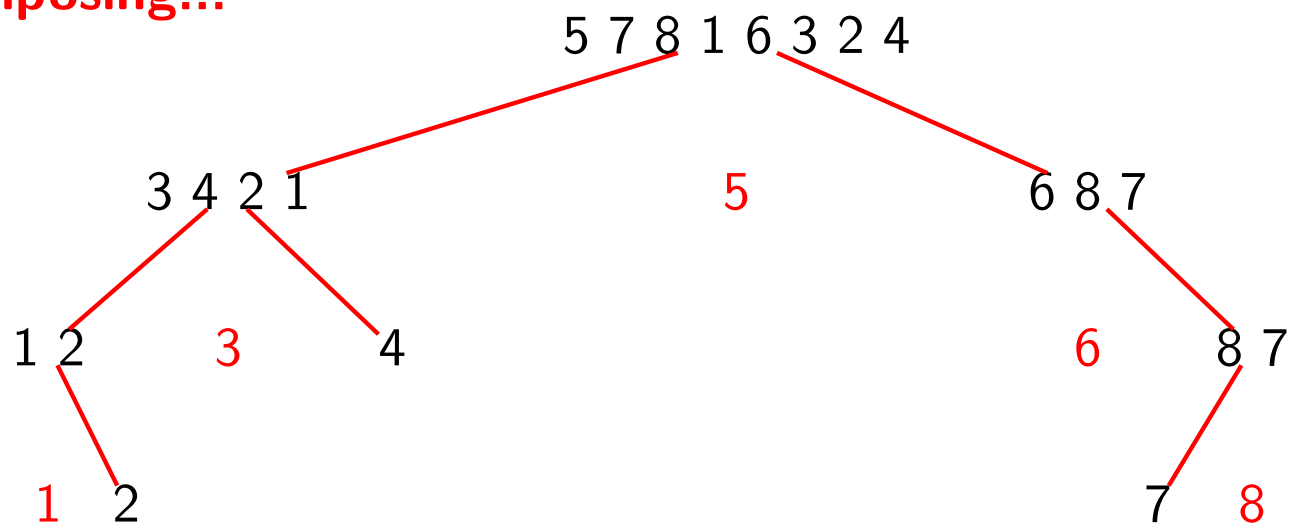
Decomposing & Combining

Decomposing...

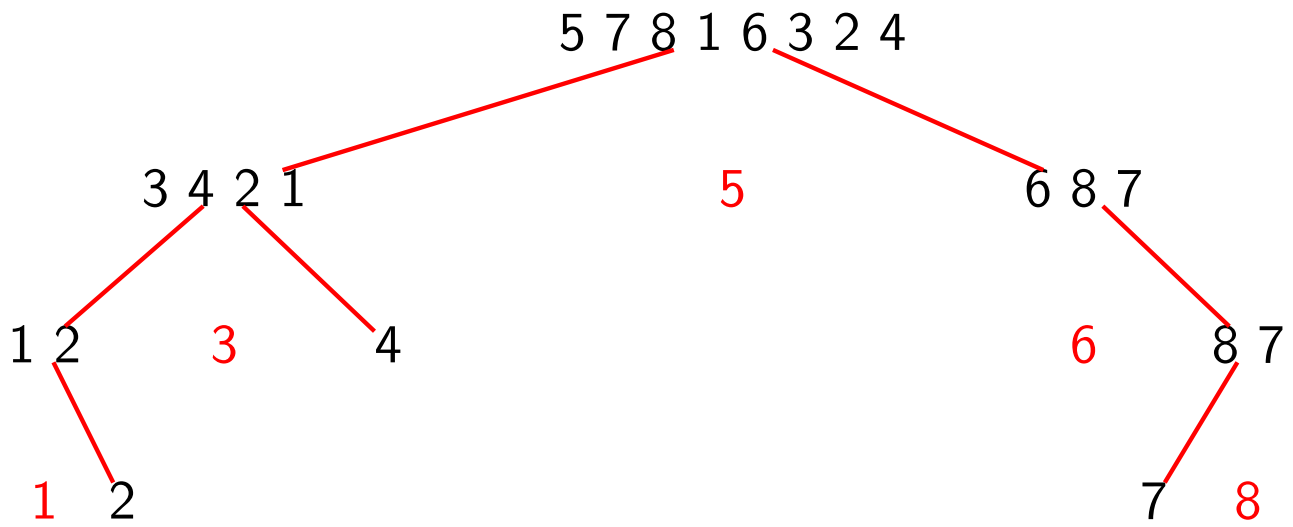


Decomposing & Combining

Decomposing...



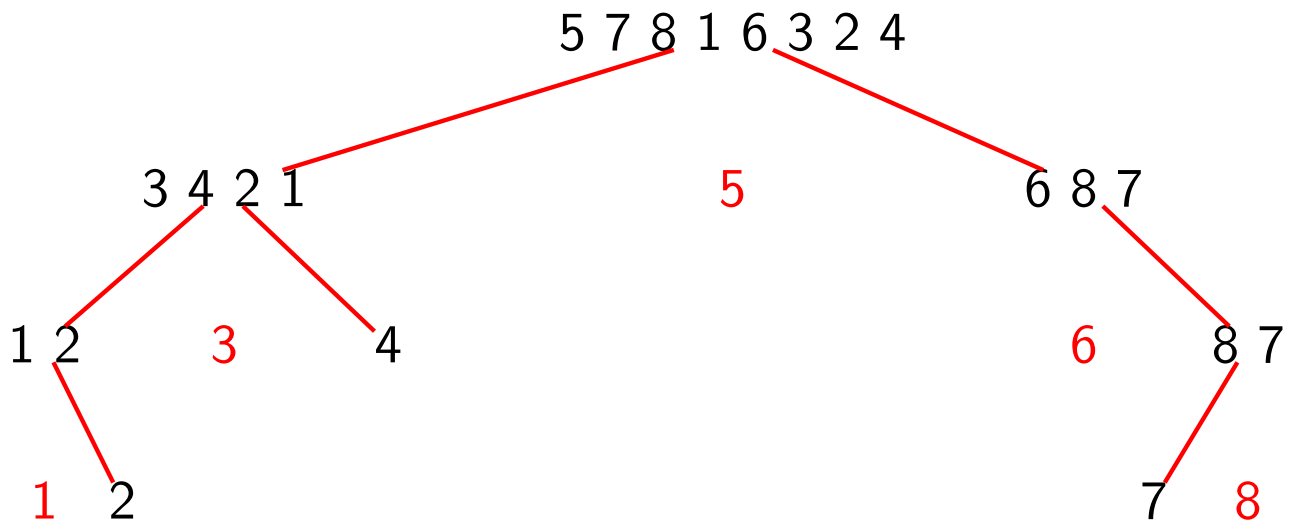
Decomposing & Combining



1 2 3 4 5 6 7 8

Combining...

Decomposing & Combining



1 2 3 4 5 6 7 8 **Combining...**

(It's already combined!)

Analyzing QuickSort

In QuickSort, the size of the split depends on the result of the Partition function...

- Best case:
- Worst case:
- Average case:

Analyzing QuickSort

In QuickSort, the size of the split depends on the result of the Partition function...

- Best case:
  - Ideally, the partition splits the sublist in half
  - $C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$
- Worst case:
- Average case:

Analyzing QuickSort

In QuickSort, the size of the split depends on the result of the Partition function...

- Best case:
  - Ideally, the partition splits the sublist in half
  - $C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$
  - By the Master Theorem: $C_{best}(n) \in \Theta(n \lg n)$
- Worst case:
- Average case:

Analyzing QuickSort

In QuickSort, the size of the split depends on the result of the Partition function...

- Best case:
  - Ideally, the partition splits the sublist in half
  - $C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$
  - By the Master Theorem: $C_{best}(n) \in \Theta(n \lg n)$
- Worst case:
  - But the partition might split only one item in the sublist...
  - This happens when the sublist is already in increasing order
  - This degenerates the tree into a list, pulling one item at a time and calling Partition on the remaining $n - 1$ items
- Average case:

Analyzing QuickSort

In QuickSort, the size of the split depends on the result of the Partition function...

- Best case:
  - Ideally, the partition splits the sublist in half
  - $C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$
  - By the Master Theorem: $C_{best}(n) \in \Theta(n \lg n)$
- Worst case:
  - But the partition might split only one item in the sublist...
  - This happens when the sublist is already in increasing order
  - This degenerates the tree into a list, pulling one item at a time and calling Partition on the remaining $n - 1$ items
  - $C_{worst}(n) = (n + 1) + n + (n - 1) + \cdots 3 \in \Theta(n^2)$
- Average case:

Analyzing QuickSort

In QuickSort, the size of the split depends on the result of the Partition function...

- Best case:
  - Ideally, the partition splits the sublist in half
  - $C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$
  - By the Master Theorem: $C_{best}(n) \in \Theta(n \lg n)$
- Worst case:
  - But the partition might split only one item in the sublist...
  - This happens when the sublist is already in increasing order
  - This degenerates the tree into a list, pulling one item at a time and calling Partition on the remaining $n-1$ items
  - $C_{worst}(n) = (n+1) + n + (n-1) + \cdots 3 \in \Theta(n^2)$
- Average case:
  - Assume the partition is unbiased `wrt` position
  - $s \in [0, n-1], Pr\{s\} = \frac{1}{n} \forall s$
  - $C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} \left[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)\right]$, $C_{avg}(0) = 0$, $C_{avg}(1) = 0$

Analyzing QuickSort

In QuickSort, the size of the split depends on the result of the Partition function...

- Best case:
  - Ideally, the partition splits the sublist in half
  - $C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$
  - By the Master Theorem: $C_{best}(n) \in \Theta(n \lg n)$
- Worst case:
  - But the partition might split only one item in the sublist...
  - This happens when the sublist is already in increasing order
  - This degenerates the tree into a list, pulling one item at a time and calling Partition on the remaining $n-1$ items
  - $C_{worst}(n) = (n+1) + n + (n-1) + \cdots 3 \in \Theta(n^2)$
- Average case:
  - Assume the partition is unbiased `wrt` position
  - $s \in [0, n-1], Pr\{s\} = \frac{1}{n} \forall s$
  - $C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} \left[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)\right]$, $C_{avg}(0) = 0$, $C_{avg}(1) = 0$
  - $C_{avg} \in O(n \ln n)$

Analyzing QuickSort

In QuickSort, the size of the split depends on the result of the Partition function...

- Best case:
  - Ideally, the partition splits the sublist in half
  - $C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$
  - By the Master Theorem: $C_{best}(n) \in \Theta(n \lg n)$
- Worst case:
  - But the partition might split only one item in the sublist...
  - This happens when the sublist is already in increasing order
  - This degenerates the tree into a list, pulling one item at a time and calling Partition on the remaining $n-1$ items
  - $C_{worst}(n) = (n+1) + n + (n-1) + \cdots 3 \in \Theta(n^2)$
- Average case:
  - Assume the partition is unbiased `wrt` position
  - $s \in [0, n-1], Pr\{s\} = \frac{1}{n} \forall s$
  - $C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} \left[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)\right]$, $C_{avg}(0) = 0$, $C_{avg}(1) = 0$
  - $C_{avg} \in O(n \ln n)$

Some fixes include:
Randomizing input order
median-of-three partitioning

Specifying BinarySearch

BinarySearch($A[0 \ldots n-1, K]$)

```
l ⟵ 0
r ⟵ n − 1
while l ≤ r do
    m ⟵ ⌊(l+r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ⟵ m − 1
    else l ⟵ m + 1
return −1
```

$K = 51$

	A	
l ⟶	0	17
	1	22
	2	30
	3	35
	4	37
⊢m	5	43
	6	51
	7	64
	8	86
⟵r	9	100

Specifying BinarySearch

BinarySearch($A[0 \ldots n-1, K]$)

```
l ⟵ 0
r ⟵ n − 1
while l ≤ r do
    m ⟵ ⌊(l+r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ⟵ m − 1
    else l ⟵ m + 1
return −1
```

$K = 51$

	A	
	0	17
	1	22
	2	30
	3	35
	4	37
l ⟶	5	43
	6	51
m ⊣	7	64
	8	86
	9	100

⟵r

Specifying BinarySearch

BinarySearch($A[0 \ldots n-1, K]$)

```
l ⟵ 0
r ⟵ n − 1
while l ≤ r do
    m ⟵ ⌊(l+r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ⟵ m − 1
    else l ⟵ m + 1
return −1
```

$K = 51$

	A	
0	17	
1	22	
2	30	
3	35	
4	37	
l ⟶ 5	43	⊢m
6	51	⟵r
7	64	
8	86	
9	100	

Specifying BinarySearch

BinarySearch($A[0 \ldots n-1, K]$)

```
l ⟵ 0
r ⟵ n − 1
while l ≤ r do
    m ⟵ ⌊(l+r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ⟵ m − 1
    else l ⟵ m + 1
return −1
```

$K = 51$

	A
0	17
1	22
2	30
3	35
4	37
5	43
l ⟶ 6	51 ⟵ r
7	64
8	86
9	100

Specifying BinarySearch

BinarySearch($A[0 \ldots n-1, K]$)

```
l ⟵ 0
r ⟵ n − 1
while l ≤ r do
    m ⟵ ⌊(l+r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ⟵ m − 1
    else l ⟵ m + 1
return −1
```

$K = 51$

	A
0	17
1	22
2	30
3	35
4	37
5	43
l ⟶ 6	51 ⟵ r
7	64
8	86
9	100

Analyzing Binary Search

- Best case:
  - Key is at the midpoint in the list
  - $C_{best} \in \Theta(1)$, constant time
  - *Very* unlikely...
- Worst case:
- Average case:

Analyzing Binary Search

- Best case:
  - Key is at the midpoint in the list
  - $C_{best} \in \Theta(1)$, constant time
  - *Very* unlikely...
- Worst case:
  - If the key is not in the list ...
  - $C_{worst}(n) = C_{worst}(\lfloor \frac{n}{2} \rfloor) + 1$ for $n > 1$, $C_{worst}(1) = 1$
- Average case:

Analyzing Binary Search

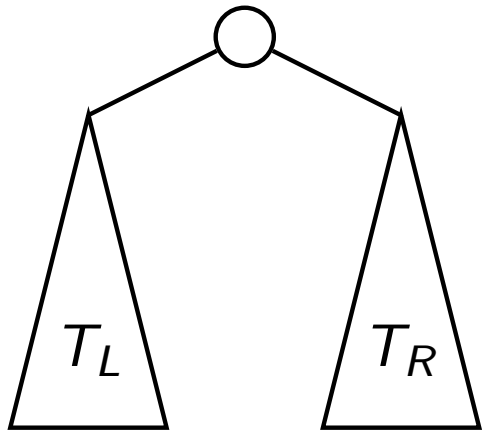
- Best case:
  - Key is at the midpoint in the list
  - $C_{best} \in \Theta(1)$, constant time
  - *Very* unlikely...
- Worst case:
  - If the key is not in the list ...
  - $C_{worst}(n) = C_{worst}(\lfloor \frac{n}{2} \rfloor) + 1$ for $n > 1$, $C_{worst}(1) = 1$
  - By the Master Theorem: $C_{worst}(n) \in \Theta(\lg n)$
- Average case:

Analyzing Binary Search

- Best case:
  - Key is at the midpoint in the list
  - $C_{best} \in \Theta(1)$, constant time
  - *Very* unlikely...
- Worst case:
  - If the key is not in the list ...
  - $C_{worst}(n) = C_{worst}(\lfloor \frac{n}{2} \rfloor) + 1$ for $n > 1$, $C_{worst}(1) = 1$
  - By the Master Theorem: $C_{worst}(n) \in \Theta(\lg n)$
- Average case:
  - Not substantially worse than the worst case, actually
  - $C_{avg} \in \Theta(\lg n)$

Binary Trees

- *Binary tree*, T — a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R, called the *left* and *right* subtree, respectively
- Note that the very definition recursively divides the tree into smaller, similar structures
- Many tree-related problems are solved by applying D&C methods
- In particular, many tree-related problems require an algorithm to *traverse* a tree



The Height Algorithm

Height(T)

```
if T = ∅ return -1
else return max{Height(T_L), Height(T_R)} + 1
```

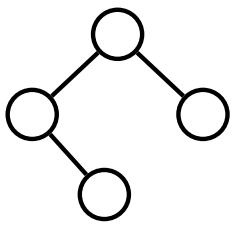
- Measure problem size by the number of nodes in a given tree, $n(T)$
- The counts for Maximum and addition operations will be the same
- So, $A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1$, for $n(T) > 0$, $A(0) = 0$

The Height Algorithm

Height(T)

```
if T = ∅ return -1
else return max{Height(T_L), Height(T_R)} + 1
```

- Measure problem size by the number of nodes in a given tree, $n(T)$
- The counts for Maximum and addition operations will be the same
- So, $A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1$, for $n(T) > 0$, $A(0) = 0$

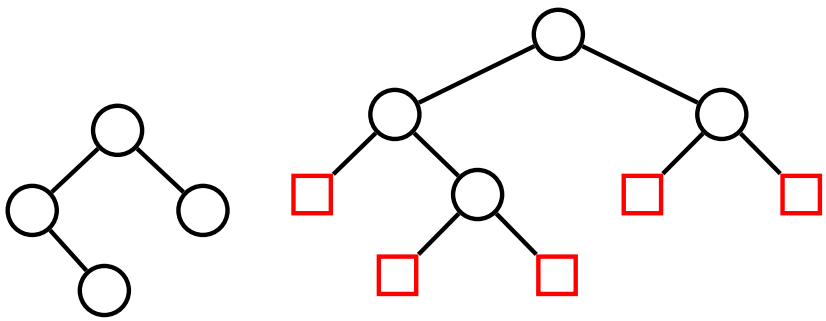


The Height Algorithm

Height(T)

```
if T = ∅ return -1
else return max{Height(T_L), Height(T_R)} + 1
```

- Measure problem size by the number of nodes in a given tree, $n(T)$
- The counts for Maximum and addition operations will be the same
- So, $A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1$, for $n(T) > 0$, $A(0) = 0$



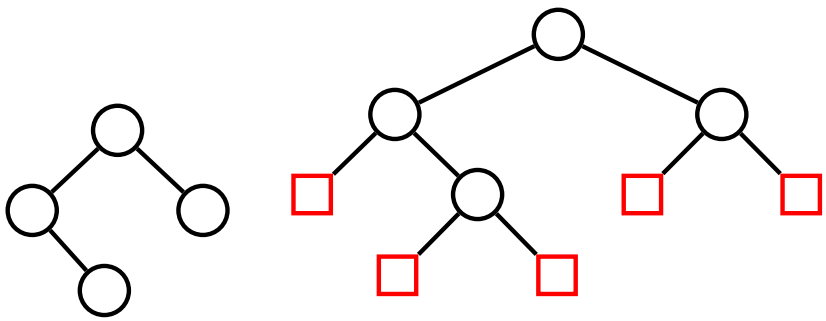
- Can draw tree's *extension* by replacing empty subtrees with special nodes

The Height Algorithm

Height(T)

```
if T = ∅ return -1
else return max{Height(T_L), Height(T_R)} + 1
```

- Measure problem size by the number of nodes in a given tree, $n(T)$
- The counts for Maximum and addition operations will be the same
- So, $A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1$, for $n(T) > 0$, $A(0) = 0$



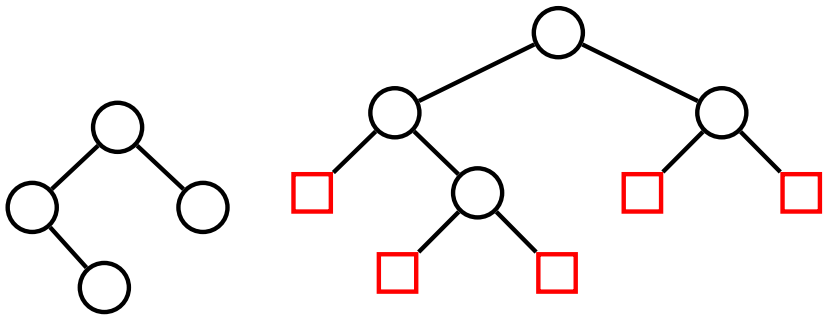
- Can draw tree's *extension* by replacing empty subtrees with special nodes
- Special nodes are *external* nodes
- Original nodes are *internal* nodes

The Height Algorithm

Height(T)

```
if T = ∅ return -1
else return max{Height(T_L), Height(T_R)} + 1
```

- Measure problem size by the number of nodes in a given tree, $n(T)$
- The counts for Maximum and addition operations will be the same
- So, $A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1$, for $n(T) > 0$, $A(0) = 0$



How many external nodes does a tree with n internal nodes have?

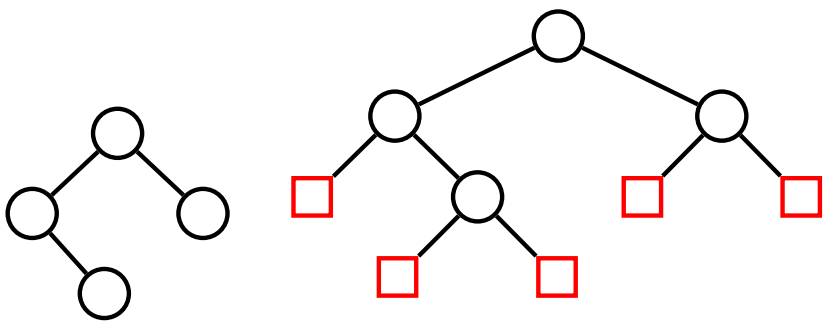
- Can draw tree's *extension* by replacing empty subtrees with special nodes
- Special nodes are *external* nodes
- Original nodes are *internal* nodes
- Height makes one addition per internal node, one comparison per internal *and* external node

The Height Algorithm

Height(T)

```
if T = ∅ return -1
else return max{Height(T_L), Height(T_R)} + 1
```

- Measure problem size by the number of nodes in a given tree, $n(T)$
- The counts for Maximum and addition operations will be the same
- So, $A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1$, for $n(T) > 0$, $A(0) = 0$

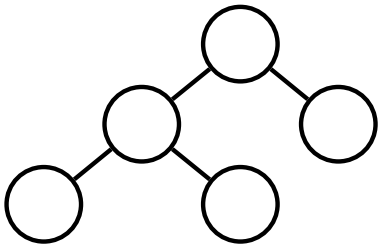


How many external nodes does a tree with n internal nodes have? $x = n + 1$

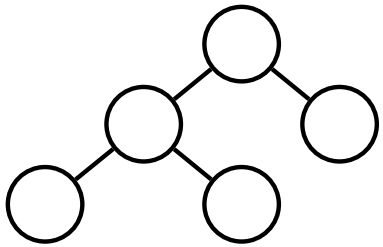
- Can draw tree's *extension* by replacing empty subtrees with special nodes
- Special nodes are *external* nodes
- Original nodes are *internal* nodes
- Height makes one addition per internal node, one comparison per internal *and* external node

Different Kinds of Tree Traversal

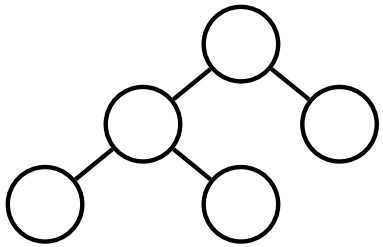
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

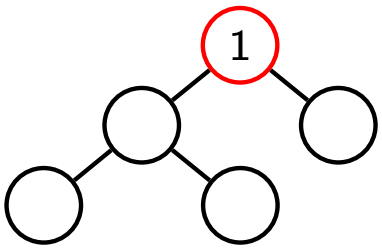


Preorder traversal:
Visit left subtree, then right subtree, then root

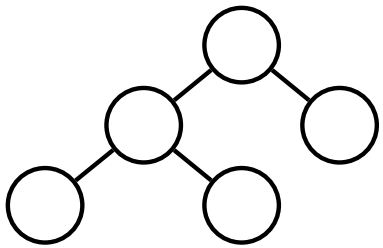


Different Kinds of Tree Traversal

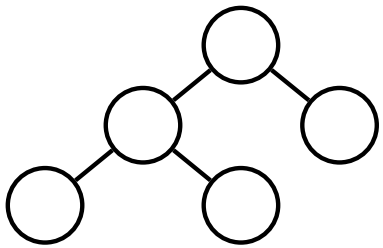
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

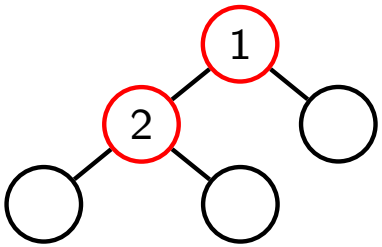


Preorder traversal:
Visit left subtree, then right subtree, then root

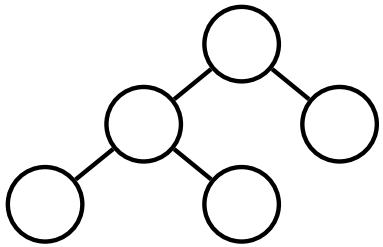


Different Kinds of Tree Traversal

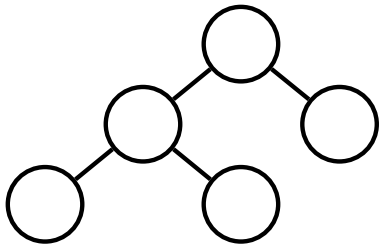
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

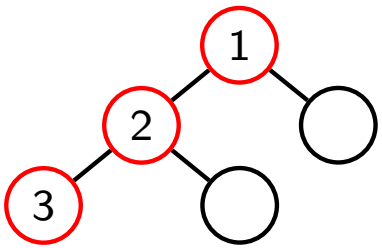


Preorder traversal:
Visit left subtree, then right subtree, then root

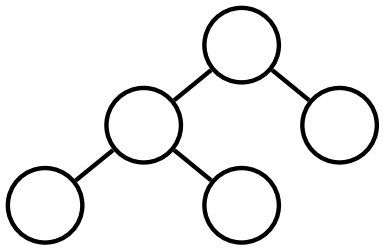


Different Kinds of Tree Traversal

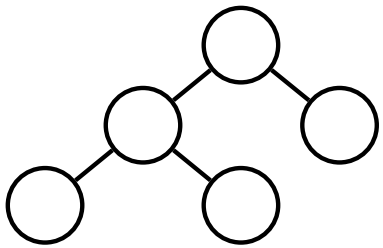
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

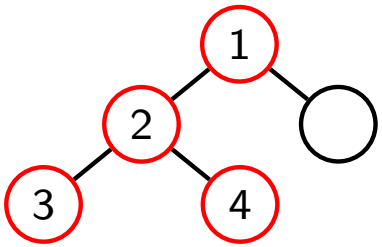


Preorder traversal:
Visit left subtree, then right subtree, then root

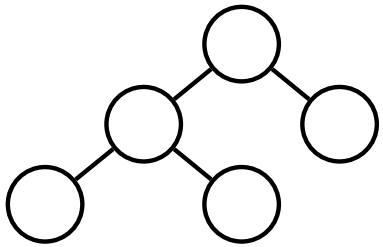


Different Kinds of Tree Traversal

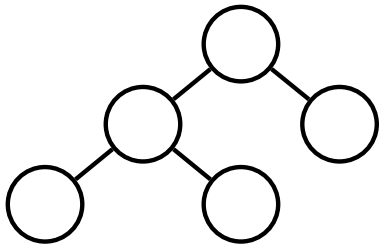
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

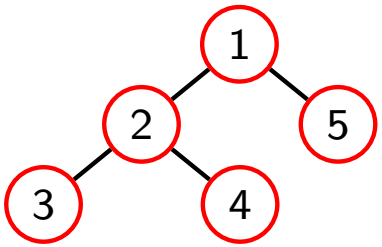


Preorder traversal:
Visit left subtree, then right subtree, then root

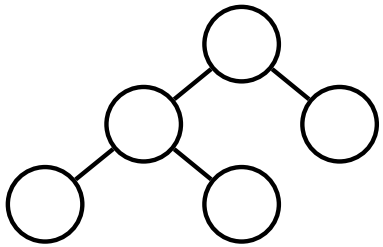


Different Kinds of Tree Traversal

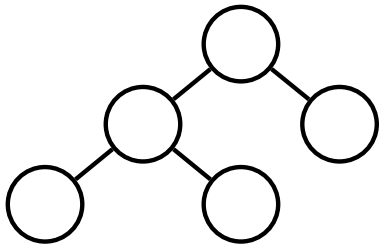
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

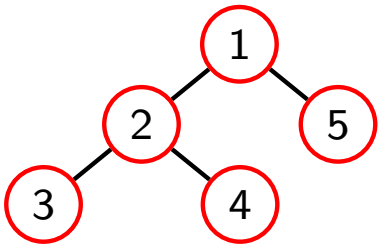


Preorder traversal:
Visit left subtree, then right subtree, then root

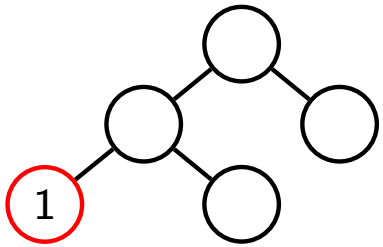


Different Kinds of Tree Traversal

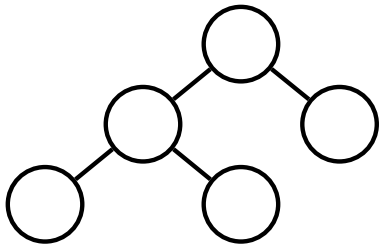
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

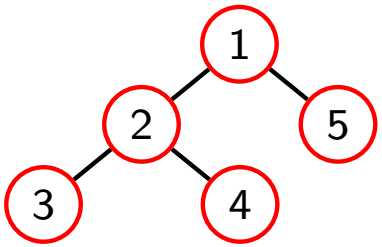


Preorder traversal:
Visit left subtree, then right subtree, then root

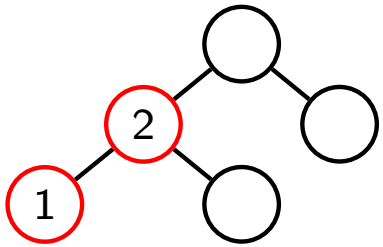


Different Kinds of Tree Traversal

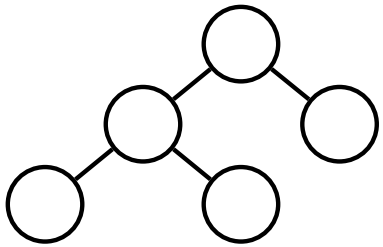
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

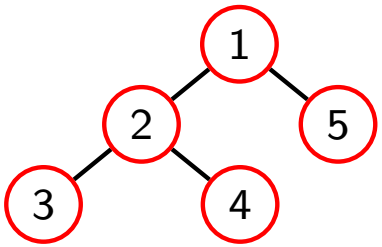


Preorder traversal:
Visit left subtree, then right subtree, then root

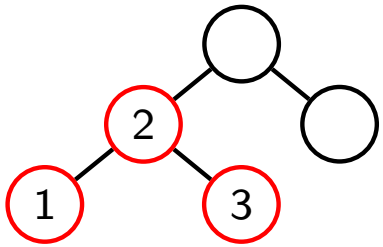


Different Kinds of Tree Traversal

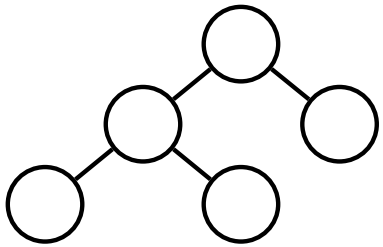
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

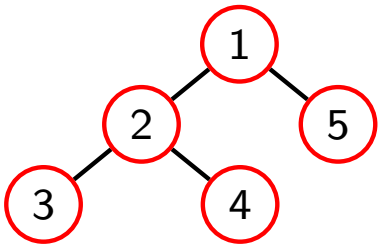


Preorder traversal:
Visit left subtree, then right subtree, then root

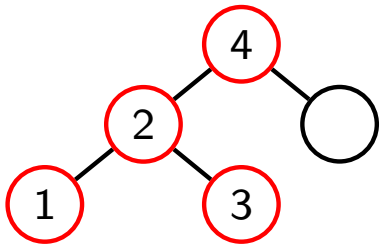


Different Kinds of Tree Traversal

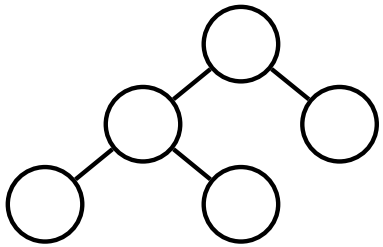
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

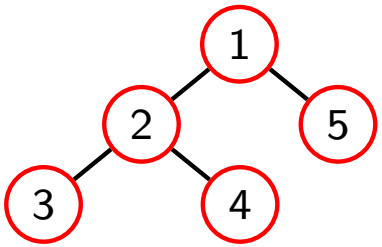


Preorder traversal:
Visit left subtree, then right subtree, then root

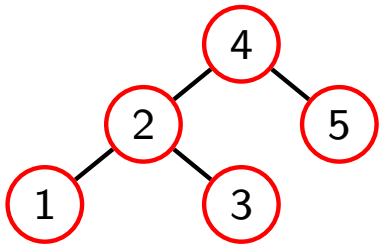


Different Kinds of Tree Traversal

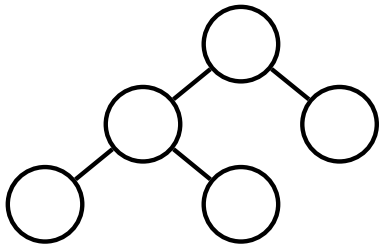
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

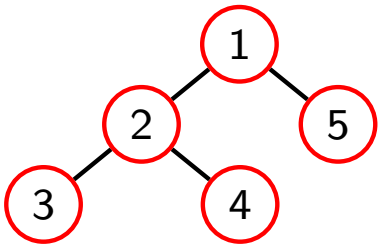


Preorder traversal:
Visit left subtree, then right subtree, then root

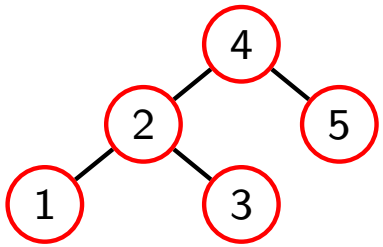


Different Kinds of Tree Traversal

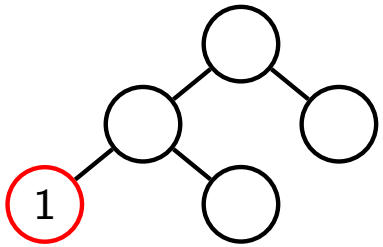
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

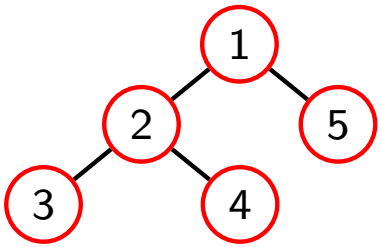


Preorder traversal:
Visit left subtree, then right subtree, then root

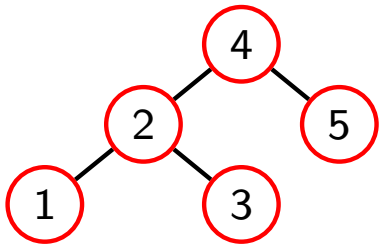


Different Kinds of Tree Traversal

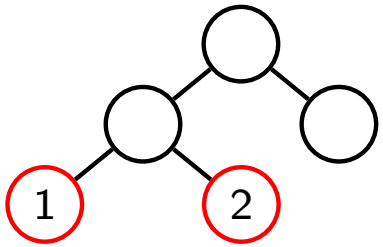
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

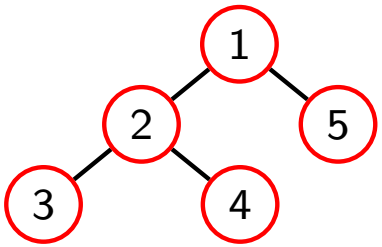


Preorder traversal:
Visit left subtree, then right subtree, then root

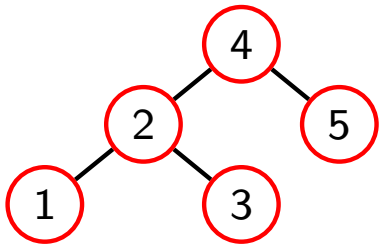


Different Kinds of Tree Traversal

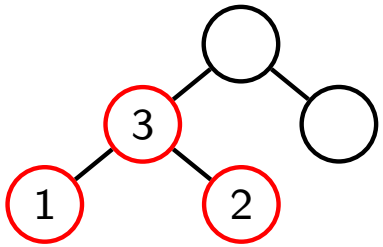
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

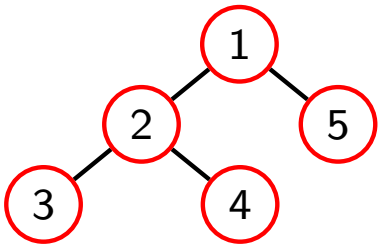


Preorder traversal:
Visit left subtree, then right subtree, then root

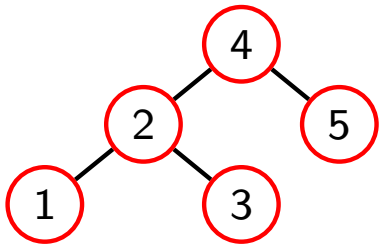


Different Kinds of Tree Traversal

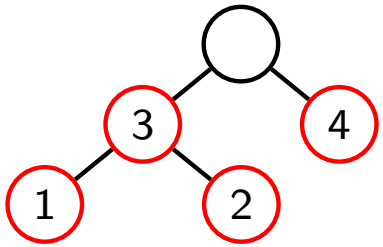
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

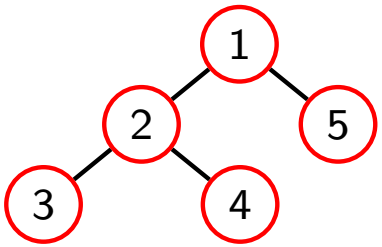


Preorder traversal:
Visit left subtree, then right subtree, then root

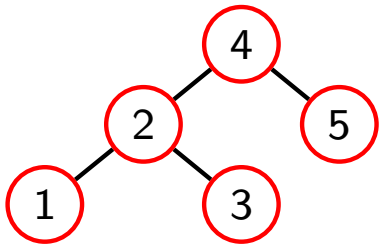


Different Kinds of Tree Traversal

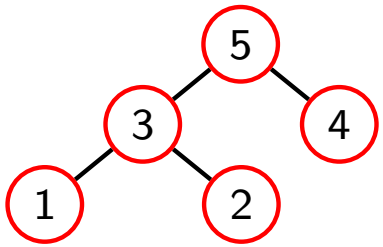
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree

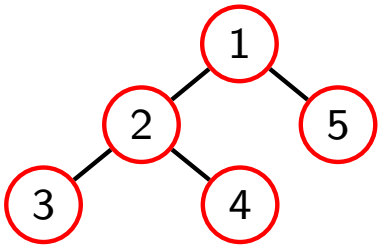


Preorder traversal:
Visit left subtree, then right subtree, then root

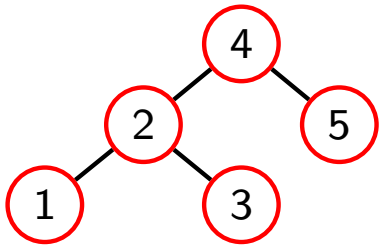


Different Kinds of Tree Traversal

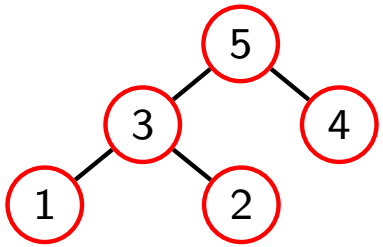
Preorder traversal:
Visit root, then left subtree, then right subtree



Inorder traversal:
Visit left subtree, then root, then right subtree



Preorder traversal:
Visit left subtree, then right subtree, then root



In general, traversals are $\Theta(n)$, but not all binary tree operations require full traversal of the tree (e.g., Find, Insert, etc.)

Multiplication of Two-Digit Integers

- Pen-and-paper multiplication:
  - If there are n digits in first integer, m in the second, this requires nm digit multiplications
  - In the worst case, $n = m$, so $M(n) \in O(n^2)$

Multiplication of Two-Digit Integers

- Pen-and-paper multiplication:
  - If there are n digits in first integer, m in the second, this requires nm digit multiplications
  - In the worst case, $n = m$, so $M(n) \in O(n^2)$
- But consider a simple 2-digit example 23×14:
  - We can break these down by digit
  - When multiplied: $\left(2 \cdot 10^1 + 3 \cdot 10^0\right) \cdot \left(1 \cdot 10^1 + 4 \cdot 10^0\right)$

Multiplication of Two-Digit Integers

- Pen-and-paper multiplication:
  - If there are n digits in first integer, m in the second, this requires nm digit multiplications
  - In the worst case, $n = m$, so $M(n) \in O(n^2)$
- But consider a simple 2-digit example 23×14:
  - We can break these down by digit
  - When multiplied: $\left(2 \cdot 10^1 + 3 \cdot 10^0\right) \cdot \left(1 \cdot 10^1 + 4 \cdot 10^0\right)$
  - Rearranged: $(2 \cdot 1)10^2 + (3 \cdot 1 + 2 \cdot 4)10^1 + (3 \cdot 4)10^0$

Multiplication of Two-Digit Integers

- Pen-and-paper multiplication:
  - If there are n digits in first integer, m in the second, this requires nm digit multiplications
  - In the worst case, $n = m$, so $M(n) \in O(n^2)$
- But consider a simple 2-digit example 23×14:
  - We can break these down by digit
  - When multiplied: $\left(2 \cdot 10^1 + 3 \cdot 10^0\right) \cdot \left(1 \cdot 10^1 + 4 \cdot 10^0\right)$
  - Rearranged: $(2 \cdot 1)10^2 + (3 \cdot 1 + 2 \cdot 4)10^1 + (3 \cdot 4)10^0$
  - Notice: $(3 \cdot 1 + 2 \cdot 4) = (2 + 3) \cdot (1 + 4) - (2 \cdot 1) - (3 \cdot 4)$

Multiplication of Two-Digit Integers

- Pen-and-paper multiplication:
  - If there are n digits in first integer, m in the second, this requires nm digit multiplications
  - In the worst case, $n = m$, so $M(n) \in O(n^2)$
- But consider a simple 2-digit example 23×14:
  - We can break these down by digit
  - When multiplied: $(2 \cdot 10^1 + 3 \cdot 10^0) \cdot (1 \cdot 10^1 + 4 \cdot 10^0)$
  - Rearranged: $(2 \cdot 1)10^2 + (3 \cdot 1 + 2 \cdot 4)10^1 + (3 \cdot 4)10^0$
  - Notice: $(3 \cdot 1 + 2 \cdot 4) = (2 + 3) \cdot (1 + 4) - (2 \cdot 1) - (3 \cdot 4)$
- For any pair of two-digit numbers ($a = a_1 a_0$, $b = b_1 b_0$):

$$\begin{aligned} c &= a \cdot b = c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0, \text{ where} \\ c_2 &= a_1 \cdot b_1 \\ c_0 &= a_0 \cdot b_0 \\ c_1 &= (a_1 + a_0) \cdot (b_1 + b_0) - (c_2 + c_0) \end{aligned}$$

Multiplication of Two-Digit Integers

- Pen-and-paper multiplication:
  - If there are n digits in first integer, m in the second, this requires nm digit multiplications
  - In the worst case, $n = m$, so $M(n) \in O(n^2)$
- But consider a simple 2-digit example 23×14:
  - We can break these down by digit
  - When multiplied: $(2 \cdot 10^1 + 3 \cdot 10^0) \cdot (1 \cdot 10^1 + 4 \cdot 10^0)$
  - Rearranged: $(2 \cdot 1)10^2 + (3 \cdot 1 + 2 \cdot 4)10^1 + (3 \cdot 4)10^0$
  - Notice: $(3 \cdot 1 + 2 \cdot 4) = (2 + 3) \cdot (1 + 4) - (2 \cdot 1) - (3 \cdot 4)$
- For any pair of two-digit numbers ($a = a_1 a_0$, $b = b_1 b_0$):

$$
\begin{aligned}
c &= a \cdot b = c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0, \text{ where} \\
c_2 &= a_1 \cdot b_1 \\
c_0 &= a_0 \cdot b_0 \\
c_1 &= (a_1 + a_0) \cdot (b_1 + b_0) - (c_2 + c_0)
\end{aligned}
$$

We saved one multiplication!

Multiplication of Large Integers

- We can generalize this method using D&C
- Let the digits of a and b be partitioned `s.t.`:
  - $a = a_1 a_0 \Longrightarrow a = a_1 \cdot 10^{n/2} + a_0$
  - $b = b_1 b_0 \Longrightarrow b = b_1 \cdot 10^{n/2} + b_0$
- Using the same trick:

$$\begin{aligned} c &= a \cdot b = \left(a_1 \cdot 10^{n/2} + a_0\right) \cdot \left(b_1 \cdot 10^{n/2} + b_0\right) \\ &= (a_1 \cdot b_1) \cdot 10^n + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot 10^{n/2} + (a_0 \cdot b_0) \\ &= c_2 \cdot 10^n + c_1 \cdot 10^{n/2} + c_0, \text{ where} \\ c_2 &= a_1 \cdot b_1 \\ c_0 &= a_0 \cdot b_0 \\ c_1 &= (a_1 + a_0) \cdot (b_1 + b_0) - (c_2 + c_0) \end{aligned}$$

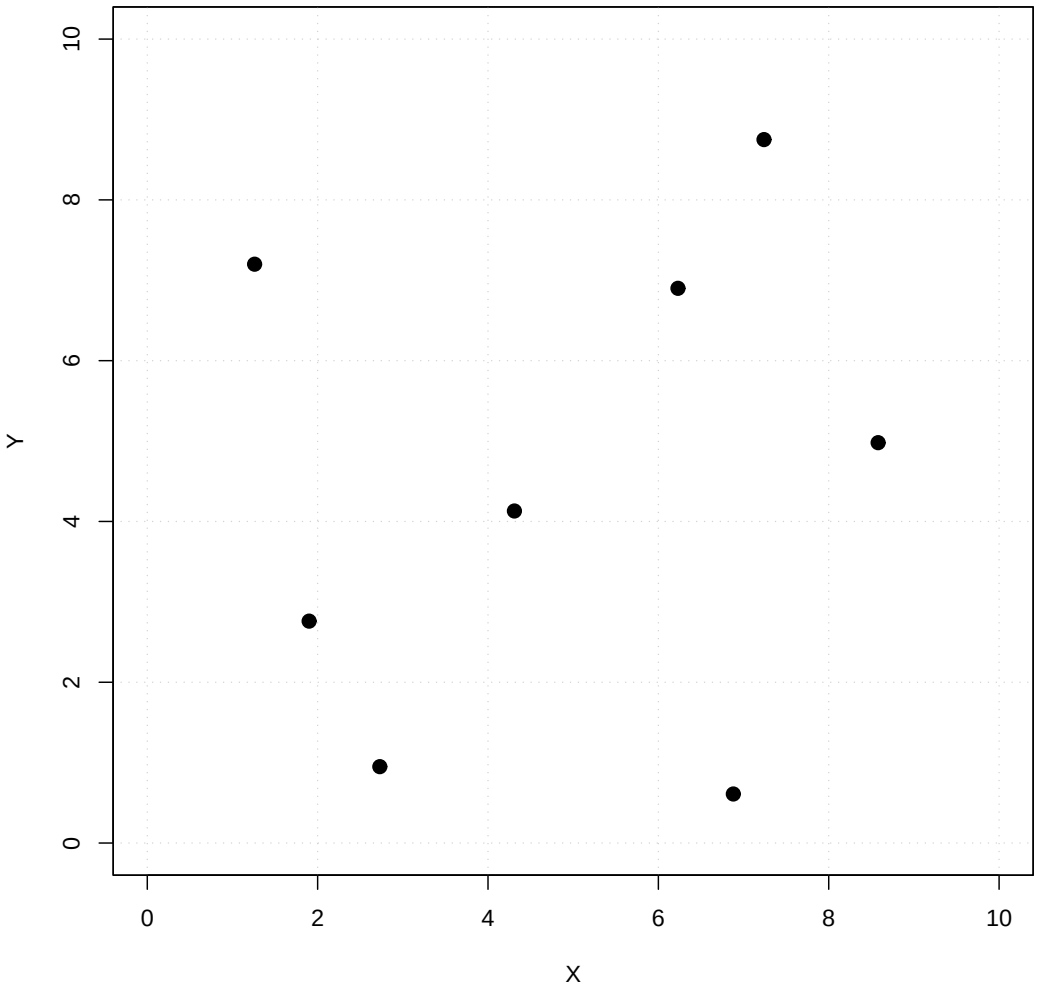
Divide-and-Conquer Multiplication

- Analysis of this is straightforward:
  - We count multiplications
  - We divide digits in half each time, apply multiplication three times when combining
  - Recurrence: $M(n) = 3M\left(\frac{n}{2}\right)$ for $n > 1,\ M(1) = 1$
  - $M(n) \in \Theta\left(n^{\lg 3} \approx n^{1.585}\right)$
- For small to moderate sized input, the standard multiplication method is faster

Divide-and-Conquer Multiplication

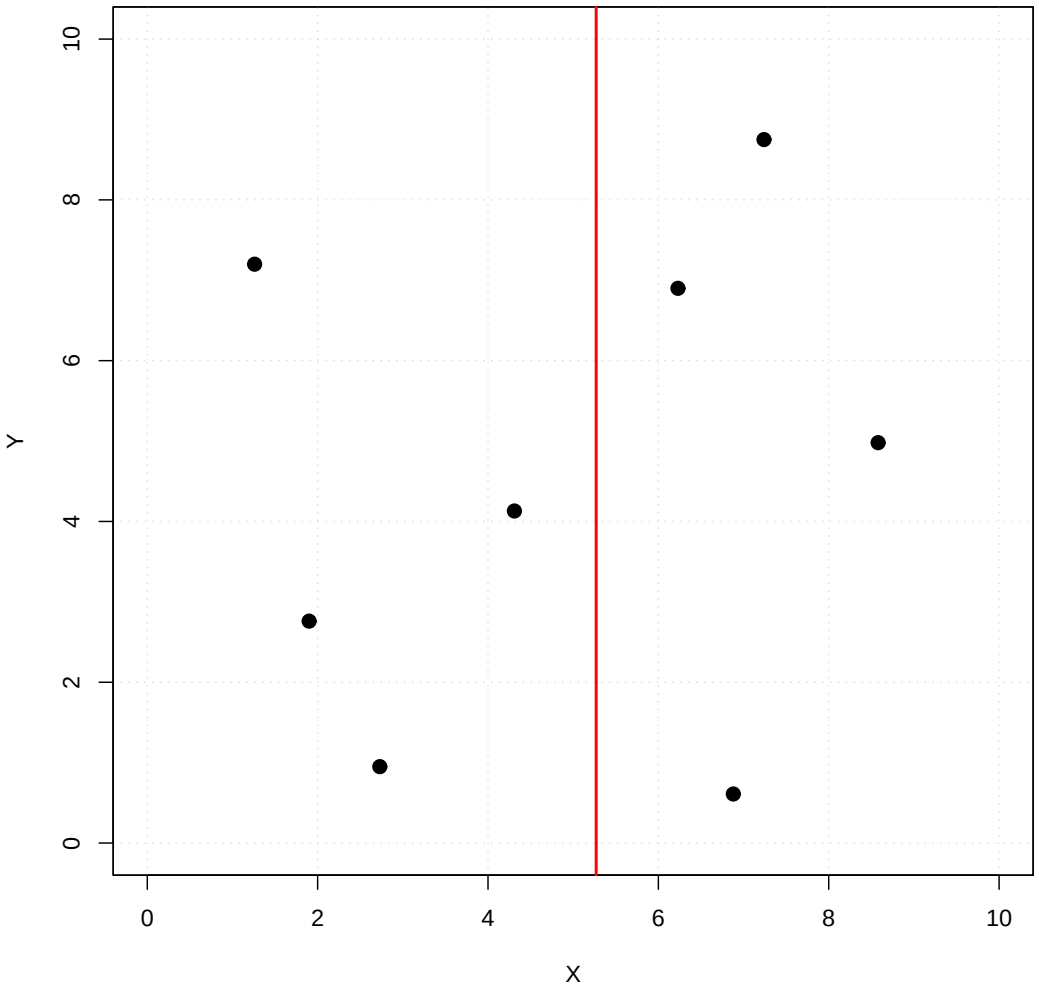
- Analysis of this is straightforward:
  - We count multiplications
  - We divide digits in half each time, apply multiplication three times when combining
  - Recurrence: $M(n) = 3M\left(\frac{n}{2}\right)$ for $n > 1,\ M(1) = 1$
  - $M(n) \in \Theta\left(n^{\lg 3} \approx n^{1.585}\right)$
- For small to moderate sized input, the standard multiplication method is faster
- Similar methods can be applied for matrix multiplication (Strassen's Method, in book)
  - Use algebra to reduce (by one) the eight multiplications performed when working with two 2×2 matrices
  - Accrue 18 addition operations, whereas traditional matrix multiplication requires only 4
  - Partition complete matrix into four submatrices, recursively apply method
  - Count additions
  - Recurrence: $A(n) = 7A\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$ for $n > 1,\ A(1) = 0$
  - $A(n) \in \Theta\left(n^{\lg 7} \approx n^{2.8}\right)$

D&C and the Closest Pair Problem



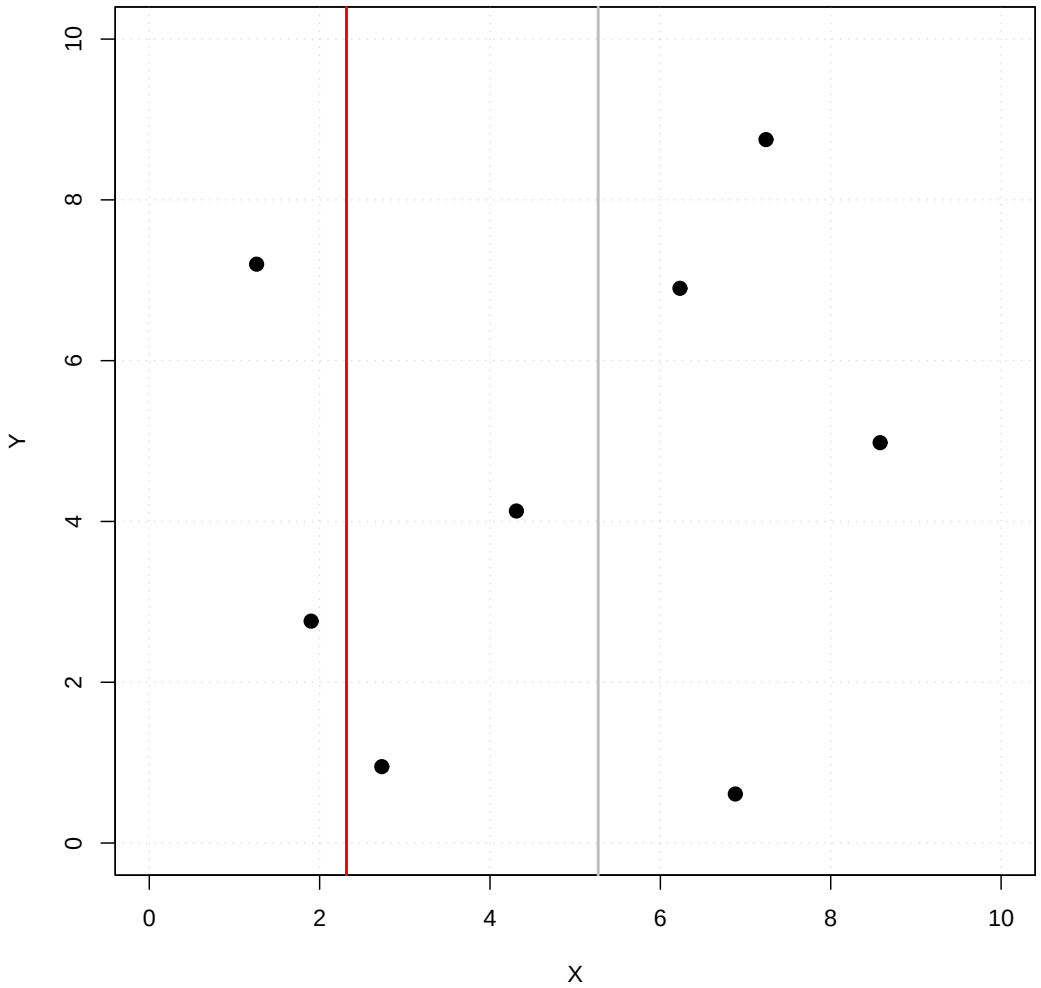
- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up

D&C and the Closest Pair Problem



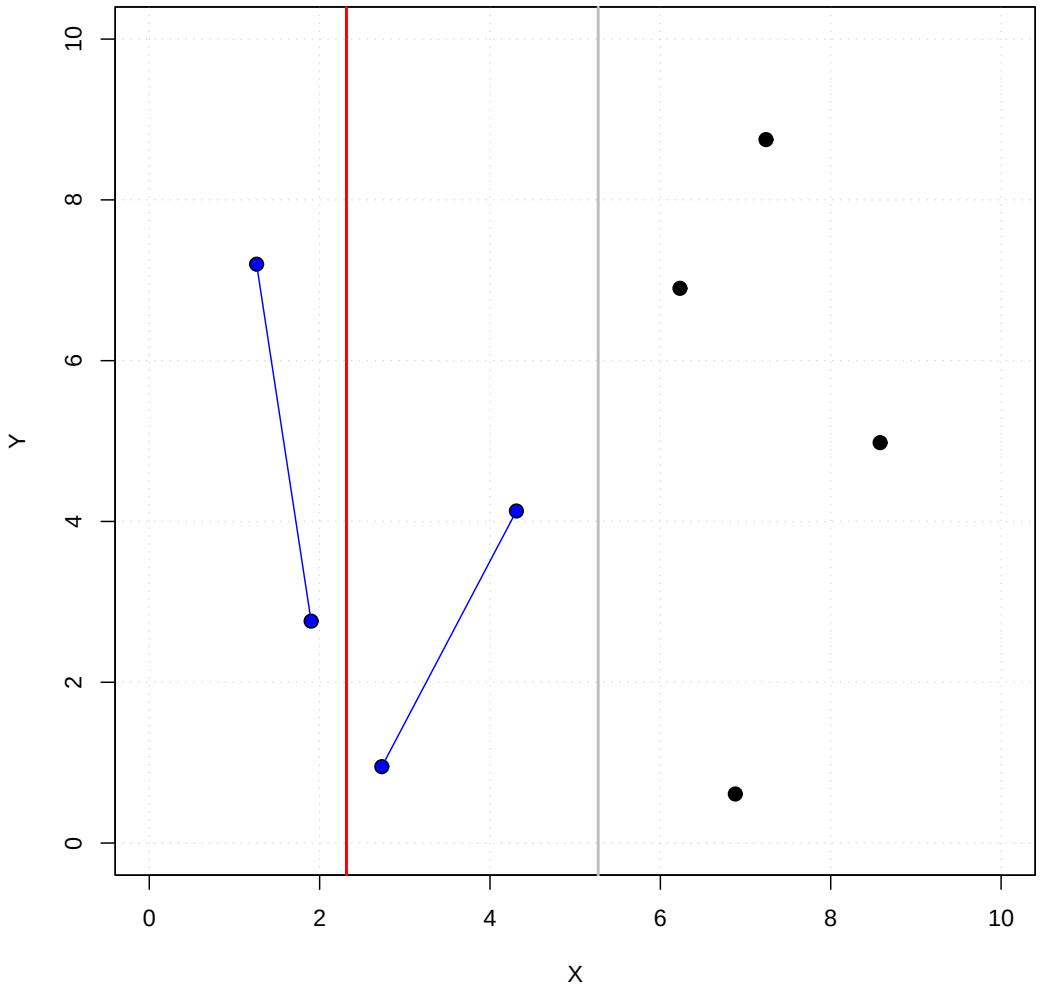
- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up

D&C and the Closest Pair Problem



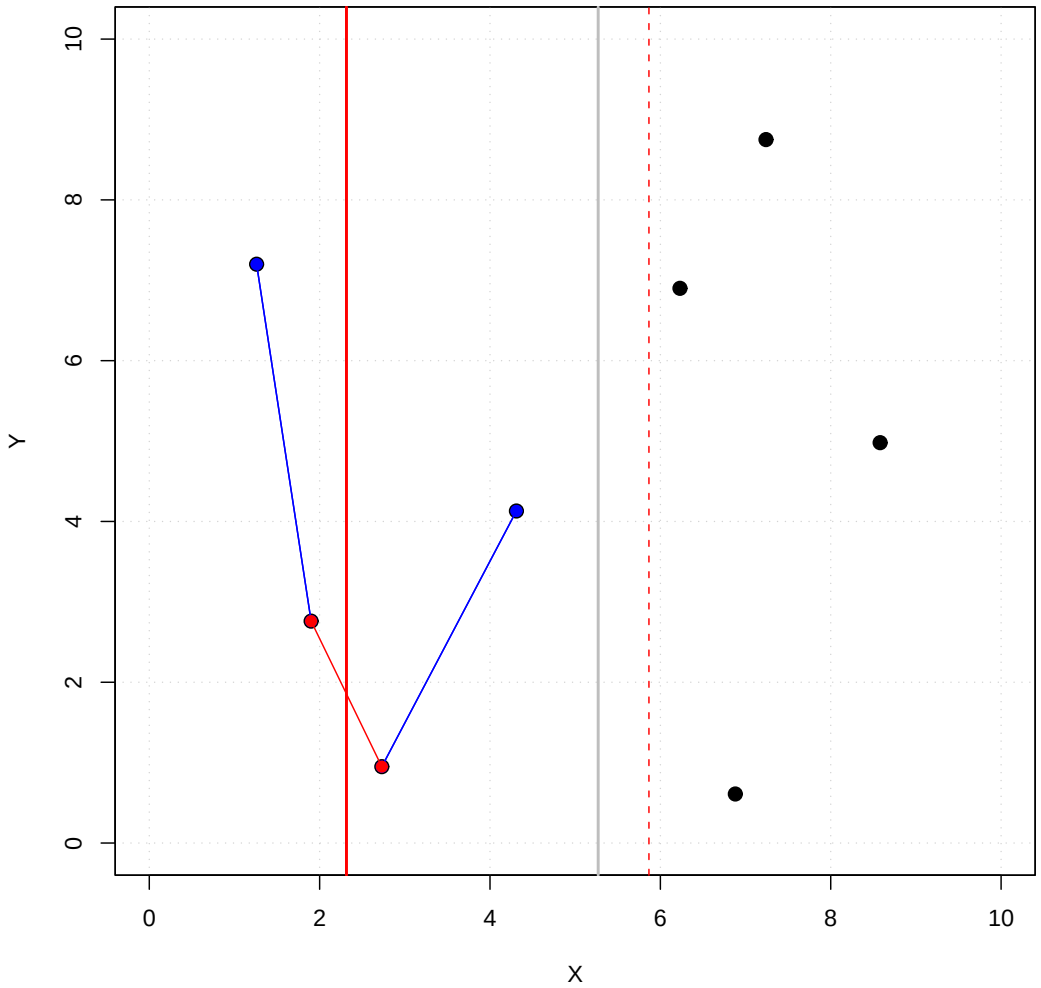
- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up

D&C and the Closest Pair Problem



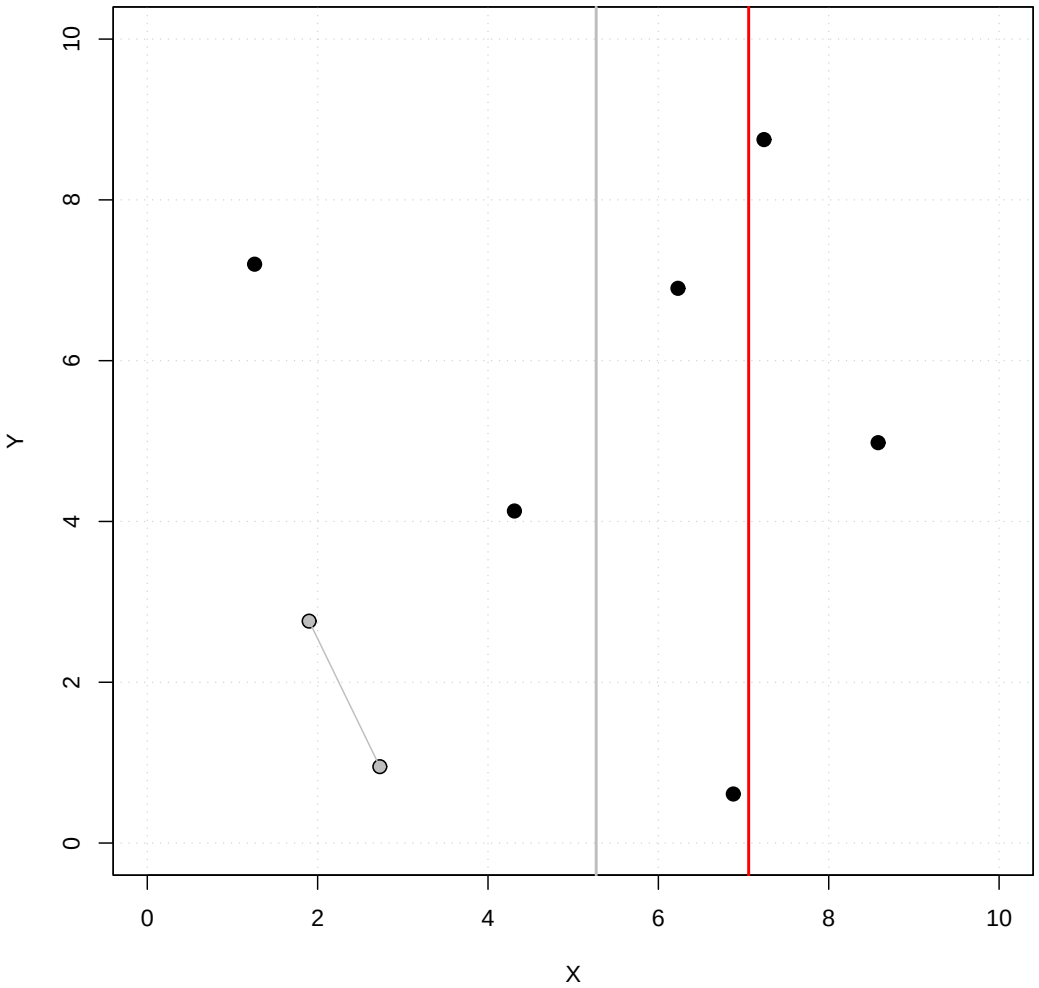
- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up

D&C and the Closest Pair Problem



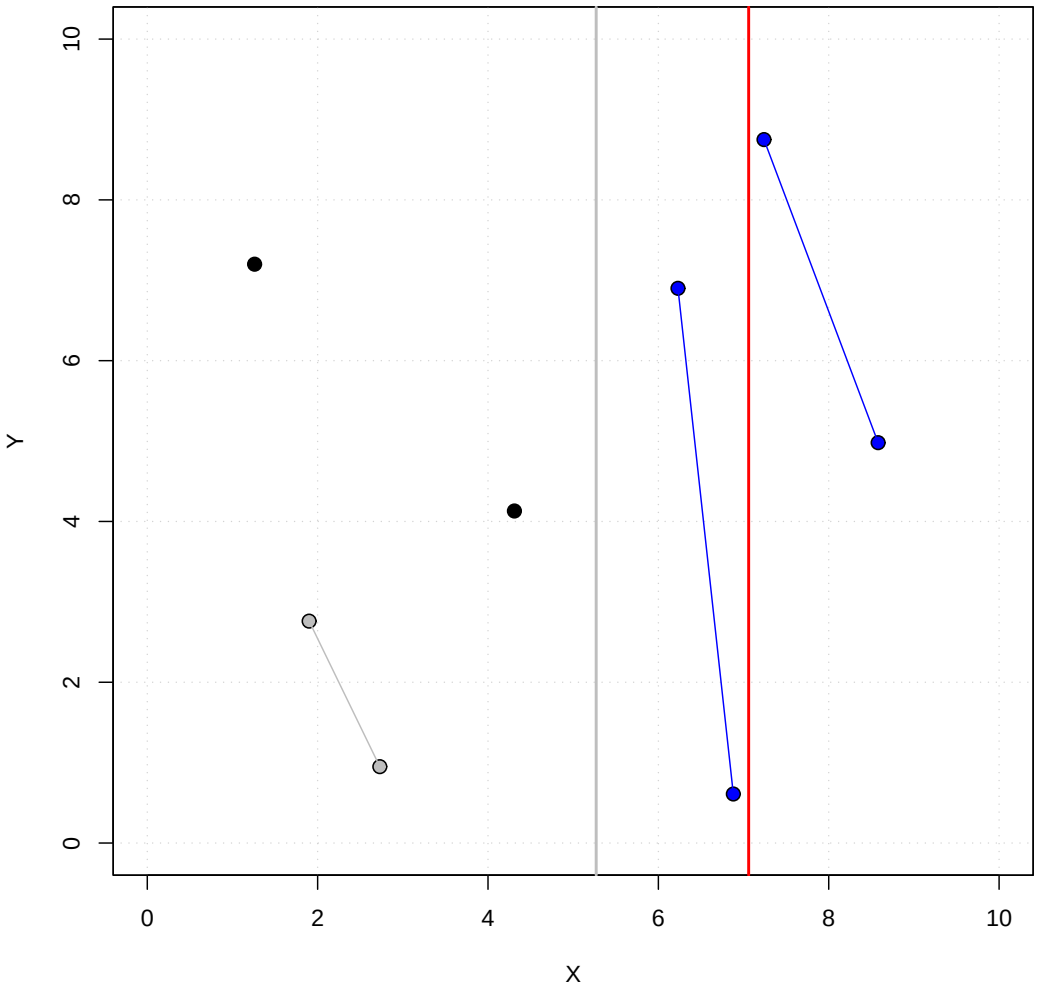
- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up

D&C and the Closest Pair Problem



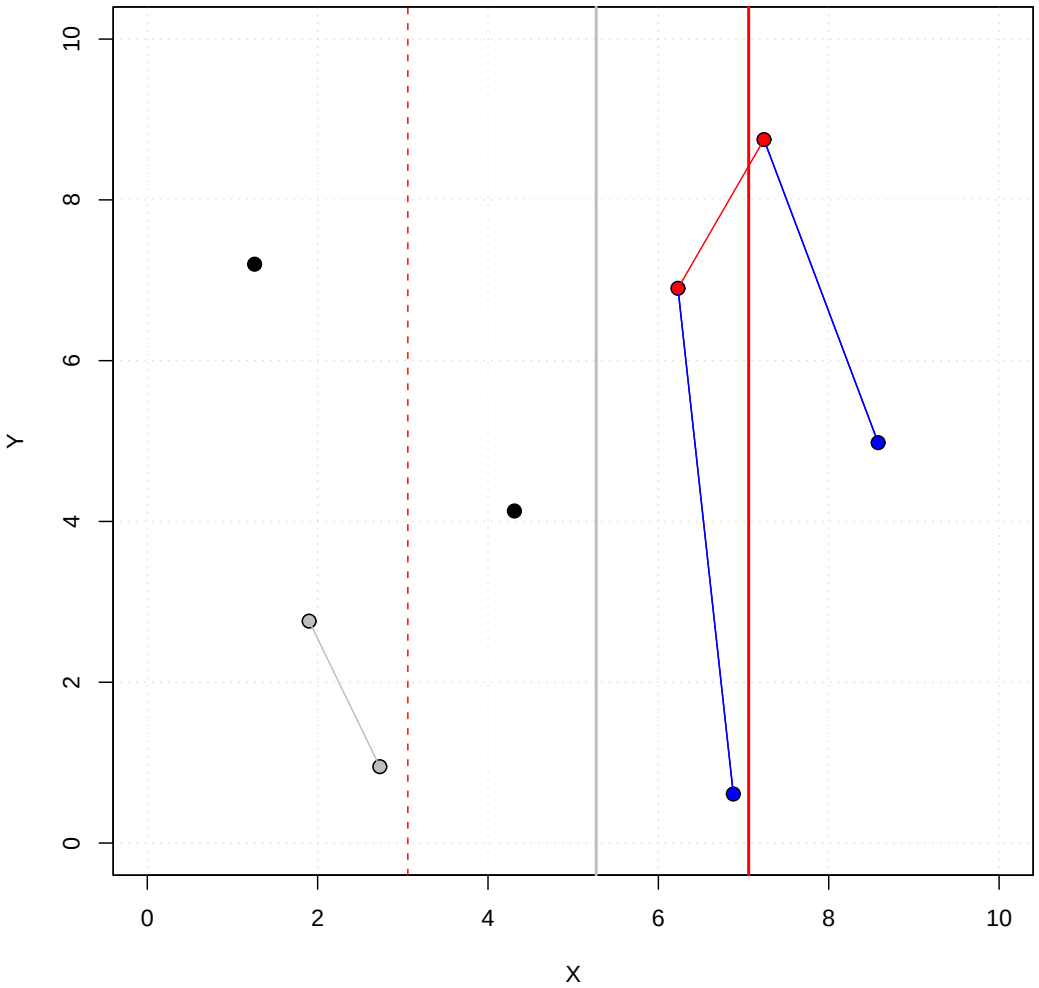
- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up

D&C and the Closest Pair Problem



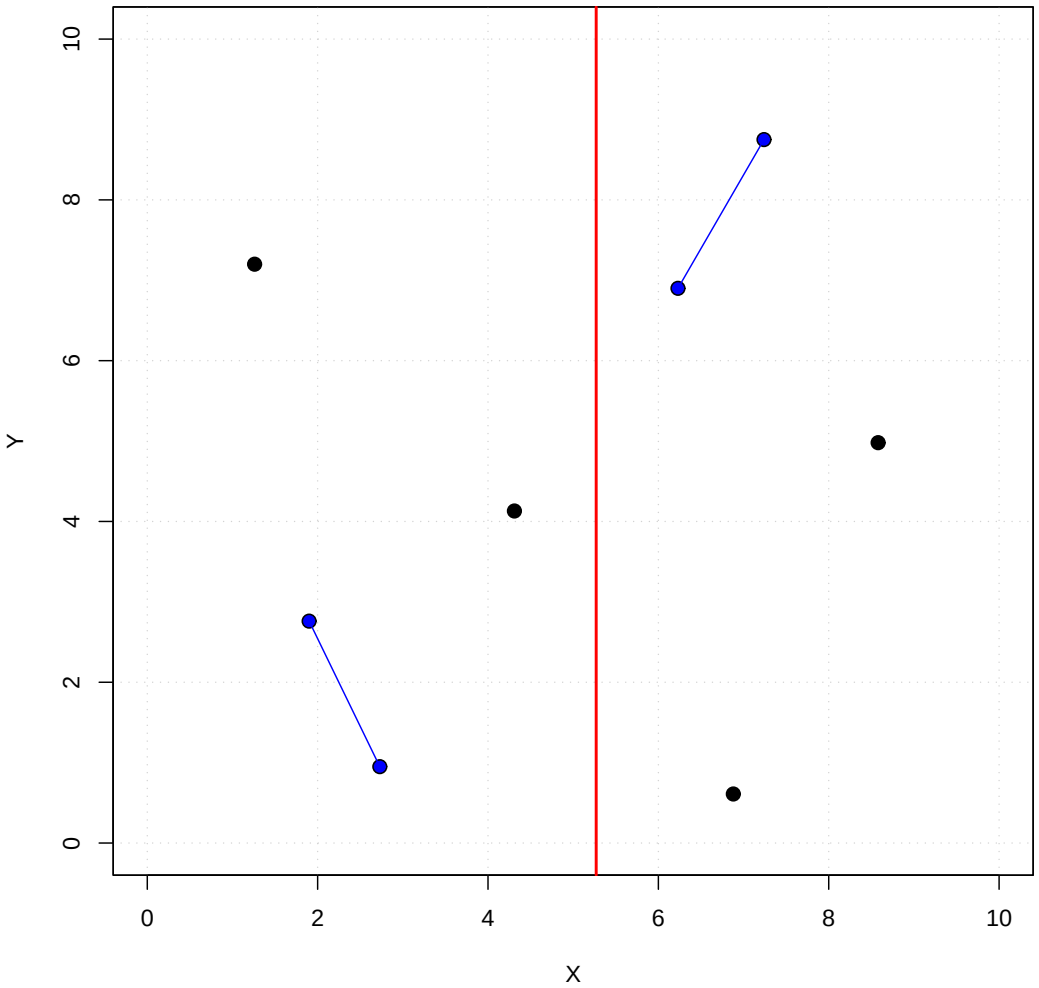
- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up

D&C and the Closest Pair Problem



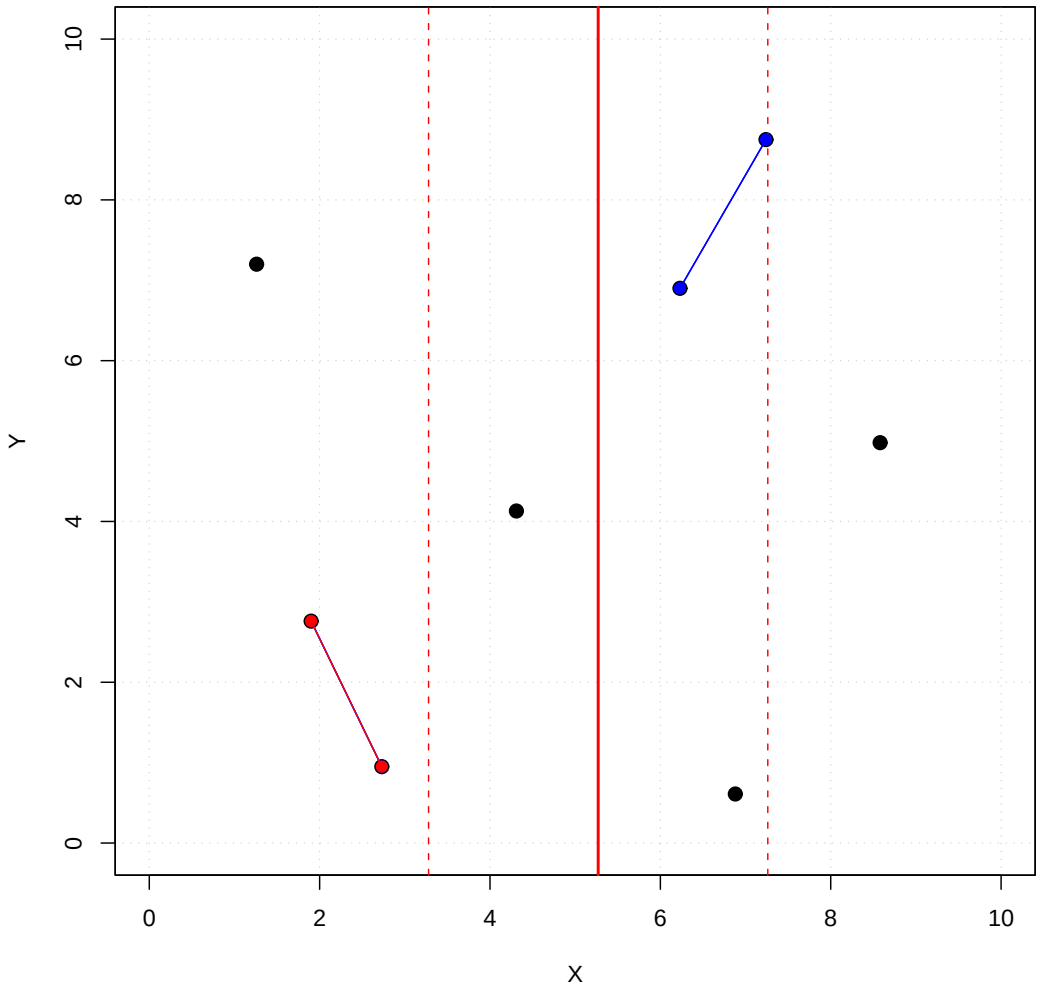
- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up

D&C and the Closest Pair Problem



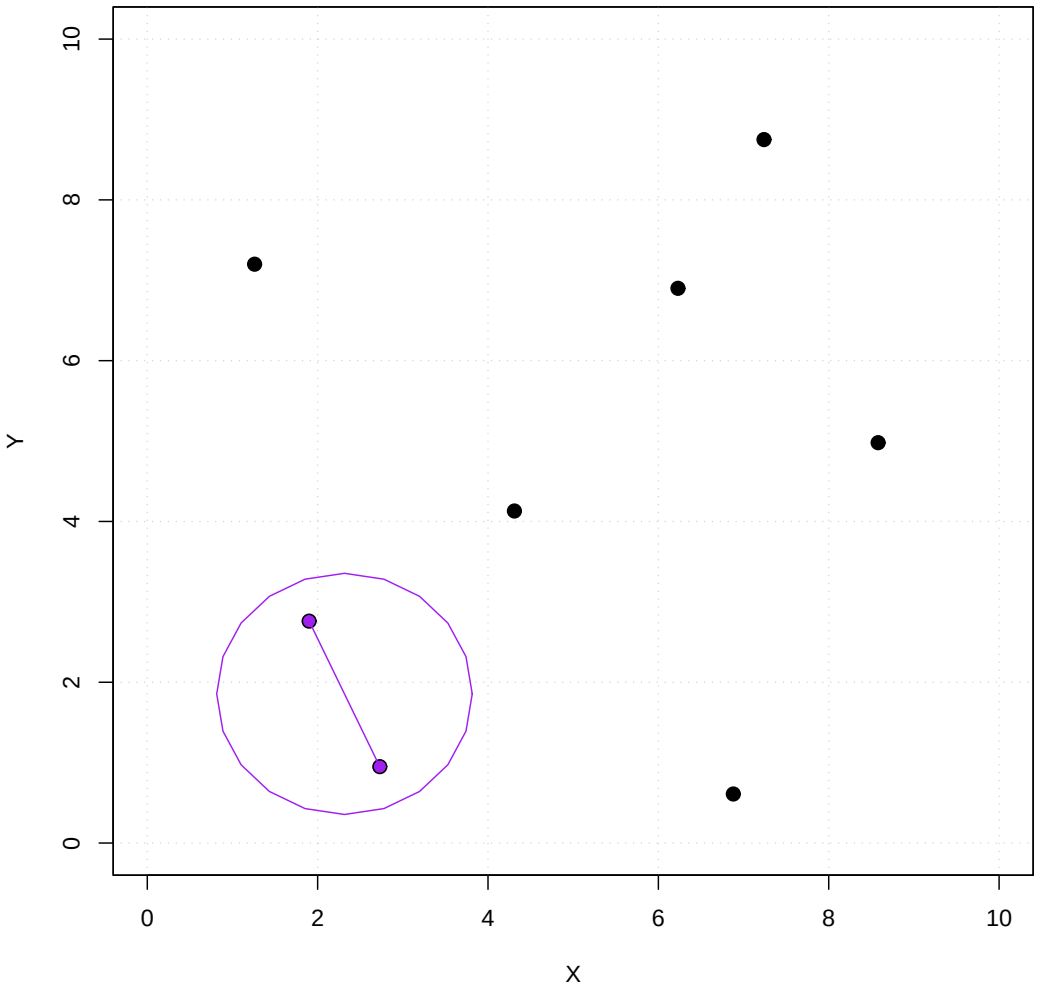
- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up

D&C and the Closest Pair Problem



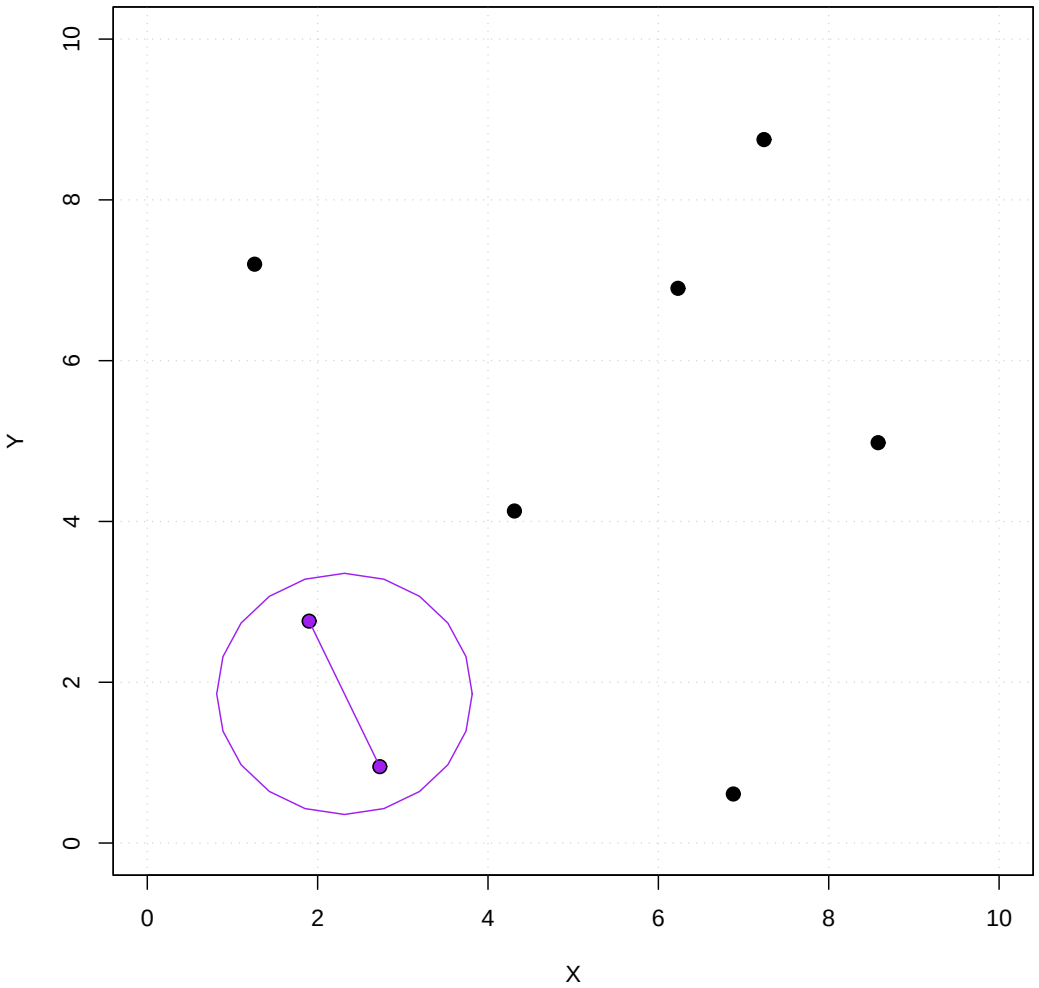
- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up

D&C and the Closest Pair Problem



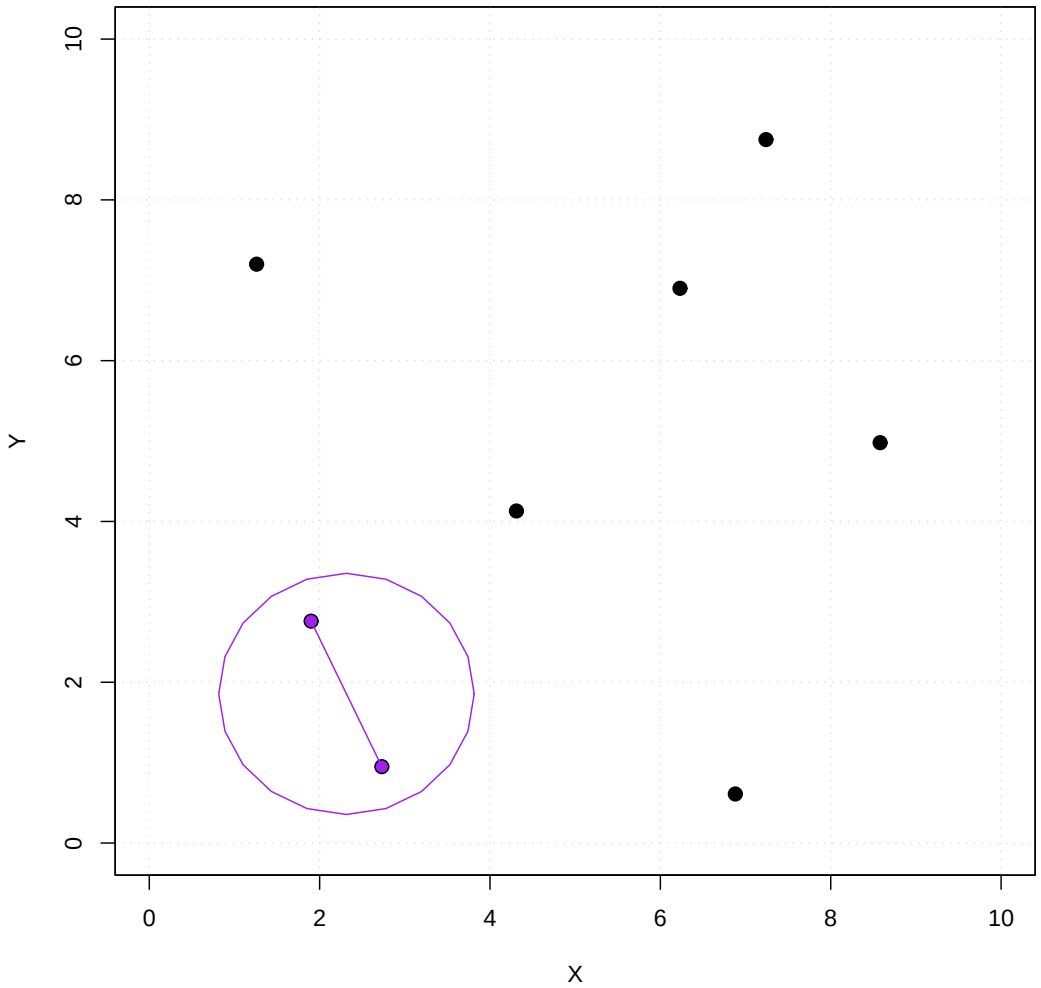
- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up

D&C and the Closest Pair Problem



- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up
- Recurrence: $T(n) = 2T\left(\frac{n}{2}\right) + M(n)$
- So $T(n) \in$ **?**

D&C and the Closest Pair Problem



- Order the points by x-axis
- Recursively partition points
- Find closest pair of 2 or 3 points
- Combine by checking within δ of split ($M(n) \in O(n)$)
- Pass the winning pair up
- Recurrence: $T(n) = 2T\left(\frac{n}{2}\right) + M(n)$
- So $T(n) \in O(n \lg n)$

QuickHull is similar, see book!

Assignments

- This week's assignments:
  - Section 4.1: Problems 5, 6, and 7
  - Section 4.2: Problems 1, 6, and 8
  - Section 4.3: Problems 1, 3, and 6
  - Section 4.4: Problems 2, 4, and 6
  - Section 4.5: Problems 2 and 6
  - Section 4.6: Problems 8 and 9

*Challenge problem