# Divide and Conquer*

Mohammad Javad Dousti

** Some slides are courtesy of Dr. Mahini.*

# Changelog

- Rev. 1
  - Updated the pseudocode to assume the array index is zero based (slide 16).
  - Clarified a note to define the root for a pair of nodes, where one of them is the parent or ancestor of the other (slide 60).
- Rev. 2
  - Updated the definition of array B to cover a corner case where the first element of array A is part of the solution (slide 33). Thanks to Mr. Sedghian.
- Rev. 3
  - Minor update in slide 33.

# Overview

- Introduction
  - Computing $a^n$
- Master theorem and its proof
- Review of sorting algorithm and their design principles
  - Insertion sort
  - Selection sort
  - Merge sort
  - Quick sort
  - Finding median
- Maximum subarray sum
- Polynomial multiplication
- Closest pair of points
- MapReduce: A Practical Example
- Sample Problems

# Computing $a^n$

# Computing $a^n$

- Problem statement: you are given a positive integer $a$ and a non-negative integer $n$. Design an algorithm to compute $a^n$.

```
power(a, n){
    result = 1
    for i = 1 to n
        result = result * a
    return result
}
```

Running time: $O(n)$

How to design a recursive algorithm?

# Computing $a^n$

- Problem statement: you are given a positive integer $a$ and a non-negative integer $n$. Design an algorithm to compute $a^n$.

```
power(a, n){
    result = 1
    for i = 1 to n
        result = result * a
    return result
}
```

Running time: $O(n)$

How to design a recursive algorithm?

Idea: $a^n = a^{n-1} \times a$

```
power(a, n) {
    if n == 0
        return 1
    return power(a, n-1) * a
}
```

Running time: $O(n)$

# Computing $a^n$

- Problem statement: you are given a positive integer $a$ and a non-negative integer $n$. Design an algorithm to compute $a^n$.

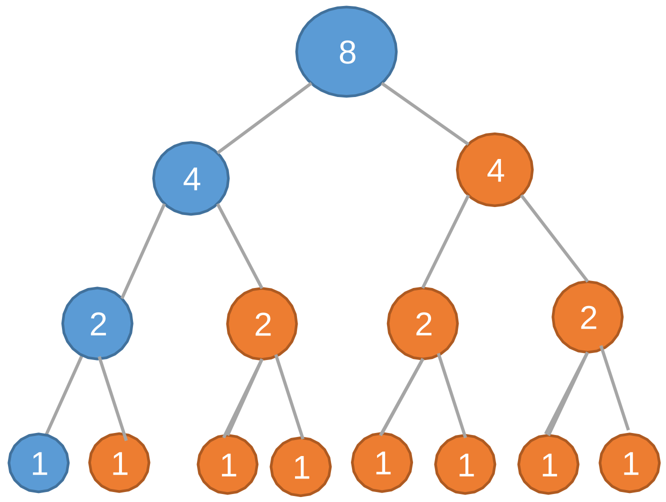
How to design a recursive algorithm?

Idea: $a^n = a^{n/2} \times a^{n/2}$

Running time: $T(n) = 2T\left(\frac{n}{2}\right) + O(1) \approx O(n)$

```
power(a, n){
    if n == 0
        return 1
    if n == 1
        return a

    return power(a, ⌊n/2⌋) * power(a, ⌈n/2⌉)
}
```

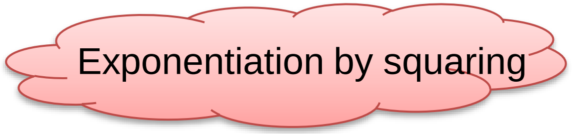



What is the problem of the above code?

# Computing $a^n$

- Problem statement: you are given a positive integer $a$ and a non-negative integer $n$. Design an algorithm to compute $a^n$.

How to design a recursive algorithm?

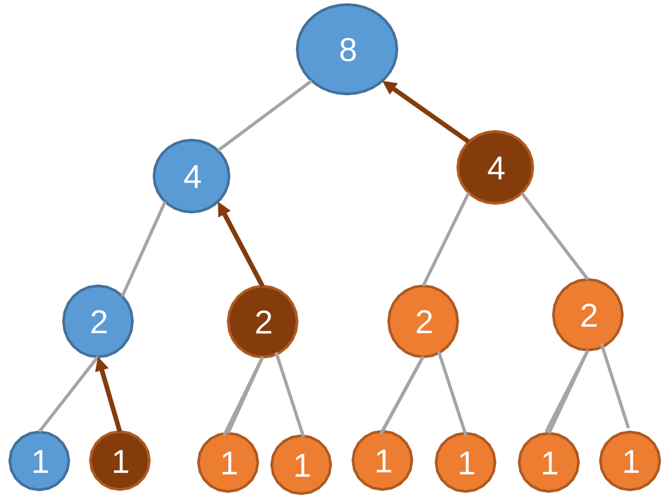



Idea: $a^n = a^{n/2} \times a^{n/2}$

Running time: $O(\log n)$

```
power(a, n) {
    if n == 0
        return 1
    if A[n] != null
        return A[n]

    A[n] = power(a, ⌊n/2⌋) * power(a, ⌊n/2⌋)
    if n is odd
        A[n] = A[n] * a
    return A[n]
}
```




Memoization: Caching results of expensive function calls

# Computing $a^n$

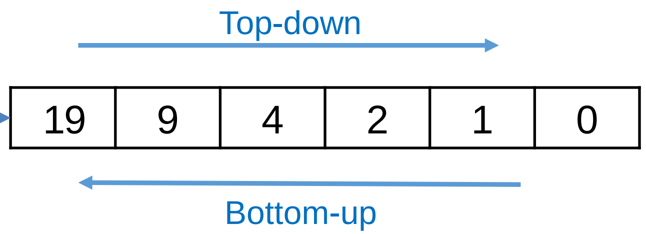
- Problem statement: you are given a positive integer $a$ and a non-negative integer $n$. Design an algorithm to compute $a^n$.

Top-down vs bottom-up approaches

Exponentiation by squaring

Idea: $a^n = a^{n/2} \times a^{n/2}$

Running time: $O(\log n)$




$19 = (1, 0, 0, 1, 1)$

```
power(a, n) {
    if n == 0
        return 1
    n = (n_k, n_k-1,..., n_0)_2  // binary representation of n
    result = a
    for i = k - 1 to 0
        if n_i = 0
            result = result * result
        else
            result = result * result * a

    return result
}
```

How can you implement the code with explicitly converting $n$ to binary?

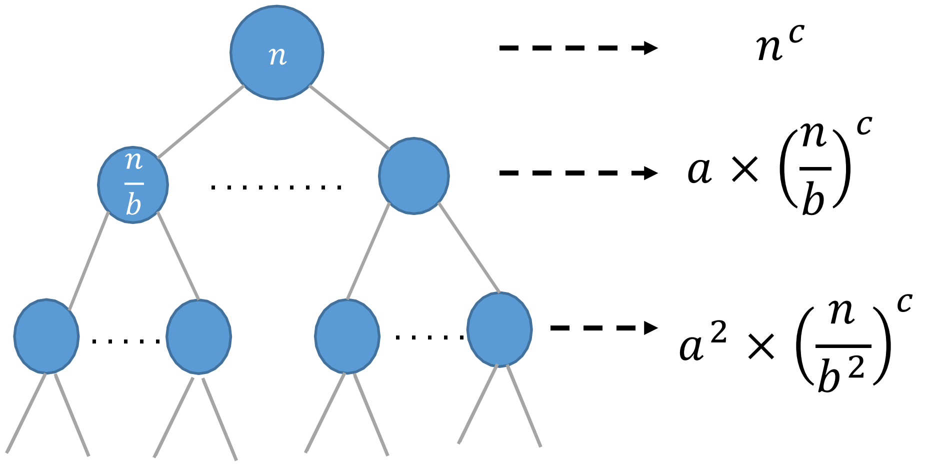
# Master Theorem

# Master Theorem

$$T(n) = a \times T\left(\frac{n}{b}\right) + O(n^c)$$

$d = \log_b a$

$$\begin{cases} d > c \rightarrow T(n) = O(n^d) \\ c > d \rightarrow T(n) = O(n^c) \\ d = c \rightarrow T(n) = O(n^c \log n) \end{cases}$$




Total: $n^c \times \left(1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^k\right)$ where $k = \log_b n$

# Master Theorem

$$T(n) = a \times T\left(\frac{n}{b}\right) + O(n^c)$$

$d = log_b a$

$$\begin{cases} d > c \rightarrow T(n) = O(n^d) \\ c > d \rightarrow T(n) = O(n^c) \\ d = c \rightarrow T(n) = O(n^c \log n) \end{cases}$$

Work is increasing as we go down the tree,
so this is the number of leaves in the recursion tree

Work is going down as we go down the tree, so dominated by the initial work at the root.

Work is the same at each level of the tree, so the work is the height, $\log_b n$, times work/level.

# Master Theorem

Total: $n^c \times \left(1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \cdots + \left(\frac{a}{b^c}\right)^k\right)$ where $k = \log_b n$

Assume $x = \frac{a}{b^c}$

$$T(n) = a \times T\left(\frac{n}{b}\right) + O(n^c)$$

$d = log_b a$

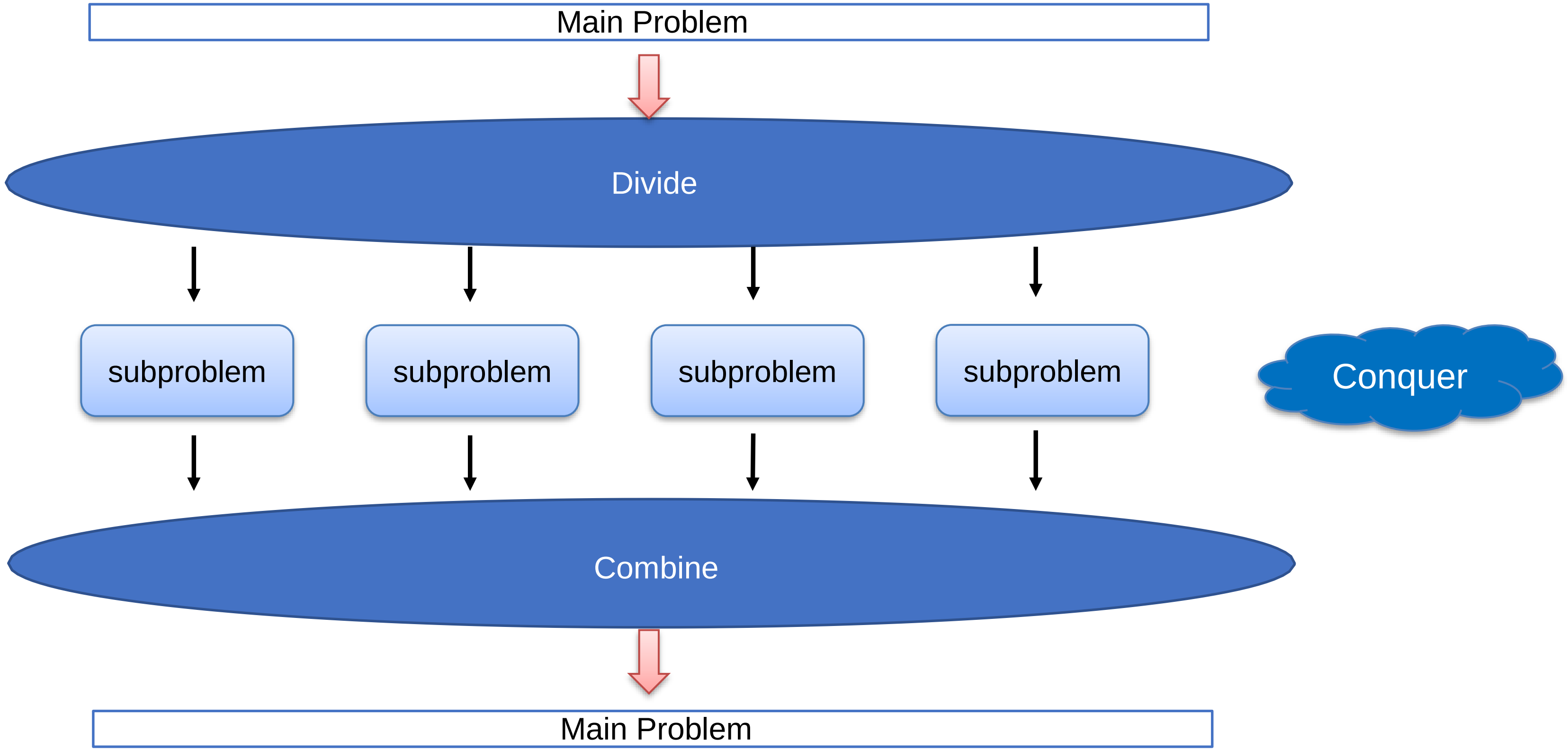
$$\begin{cases} d > c \rightarrow T(n) = O(n^d) \\ c > d \rightarrow T(n) = O(n^c) \\ d = c \rightarrow T(n) = O(n^c \log n) \end{cases}$$

$$T(n) = n^c \times \frac{(x^{k+1} - 1)}{x - 1} \approx O(n^c \times x^k)$$
$$= O\left(n^c \times \frac{a^{\log_b n}}{b^{c \times \log_b n}}\right) = O\left(n^c \times \frac{n^d}{n^c}\right) = O(n^d)$$

$$T(n) = n^c \times \frac{1 - x^{k+1}}{1 - x} \approx O(n^c \times \text{constant}) = O(n^c)$$

$$T(n) = n^c \times (1 + 1 + \cdots + 1) = n^c \times (k + 1) = O(nc \times \log_b n)$$

# Sorting Algorithms


Main Problem
Divide
subproblem
subproblem
subproblem
subproblem
Conquer
Combine
Main Problem

# Insertion Sort

- Divide: Sort the first $n-1$ elements
- Merge: Insert the last element into the right position
- **Runtime**: $T(n) = T(n-1) + O(n) = O(n^2)$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 4 | 7 | 10 | 15 | 16 | 18 | 12 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 4 | 7 | 10 | 12 | 15 | 16 | 18 |

Divide

Merge

```
insertion_sort(A, n) {
    if n == 1
        return

    insertion_sort(A, n-1)

    for i = n-2 to 0
        if A[i] <= A[i+1]
            break
        swap(A[i], A[i+1])
}
```

# Selection Sort

- ❑ Divide: Find the maximum element and put it at the end
- ❑ Merge: Nothing
- ❑ **Running time**: $T(n) = T(n-1) + O(n) = O(n^2)$

Divide

```
selection_sort(A, n) {
    if n == 1
        return

    max = n-1
    for i = 0 to n - 2
        if A[i] > A[max]
            max = i

    swap(A[max], A[n-1])
    selection_sort(A, n-1)
}
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 15 | 4 | 12 | 18 | 9 | 11 | 7 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 15 | 4 | 12 | 7 | 9 | 11 | 18 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 5 | 7 | 9 | 11 | 12 | 15 | 18 |

# Merge Sort

- Divide: Divide the array into two parts
- Merge: Merge two sorted arrays
- **Running time**: $T(n) = 2T\left(\frac{n}{2}\right) + O(?) = O(?)$

Divide

Merge

```
merge_sort(A, first, last) {
    if first == last
        return
    mid = (first + last) / 2
    merge_sort(first, mid)
    merge_sort(mid + 1, last)

    merge(A, first, mid, last)
}
```

```
merge(A, first, mid, last) {
   ????????
}
```

Main idea is to implement merge in $O(n)$

# Merge Sort

| | | | |
|---|---|---|---|
| 5 | 7 | 10 | 17 |
| 8 | 12 | 15 | 16 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | | | | | | | |

| | | | |
|---|---|---|---|
| 5 | 7 | 10 | 17 |
| 8 | 12 | 15 | 16 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 7 | | | | | | |

| | | | |
|---|---|---|---|
| 5 | 7 | 10 | 17 |
| 8 | 12 | 15 | 16 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 7 | 8 | | | | | |

| | | | |
|---|---|---|---|
| 5 | 7 | 10 | 17 |
| 8 | 12 | 15 | 16 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 7 | 8 | 10 | | | | |

# Merge Sort

- **Divide:** Divide the array into two parts
- **Merge:** Merge two sorted arrays.
- Running time: $T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$

```
merge_sort(A, first, last) {
    if first == last
        return
    mid = (first + last) / 2
    merge_sort(first, mid)
    merge_sort(mid + 1, last)

    merge(A, first, mid, last)
}
```

```
merge(A, first, mid, last){
    leftpos = first
    rightpos = mid
    for newpos = 0 to last - first {
        if leftpos < mid and (
          A[leftpos] <= A[rightpos] or rightpos > last) {
            newarray[newpos] = A[leftpos]
            leftpos++
        } else {
            newarray[newpos] = A[rightpos]
            rightpos++
        }
    }
    Copy newarray to A[first to last - 1]
}
```

# Quick Sort

- **Divide:** Divide the array into two parts such that all elements in the left subarray are less than or equal to all element in the right subarray
- **Merge:** Nothing

Divide

```
quick_sort(A, first, last){
    if first >= last
        return

    p = partition(A, first, last)
    quick_sort(A, first, p - 1)
    quick_sort(A, p + 1, last)
}
```

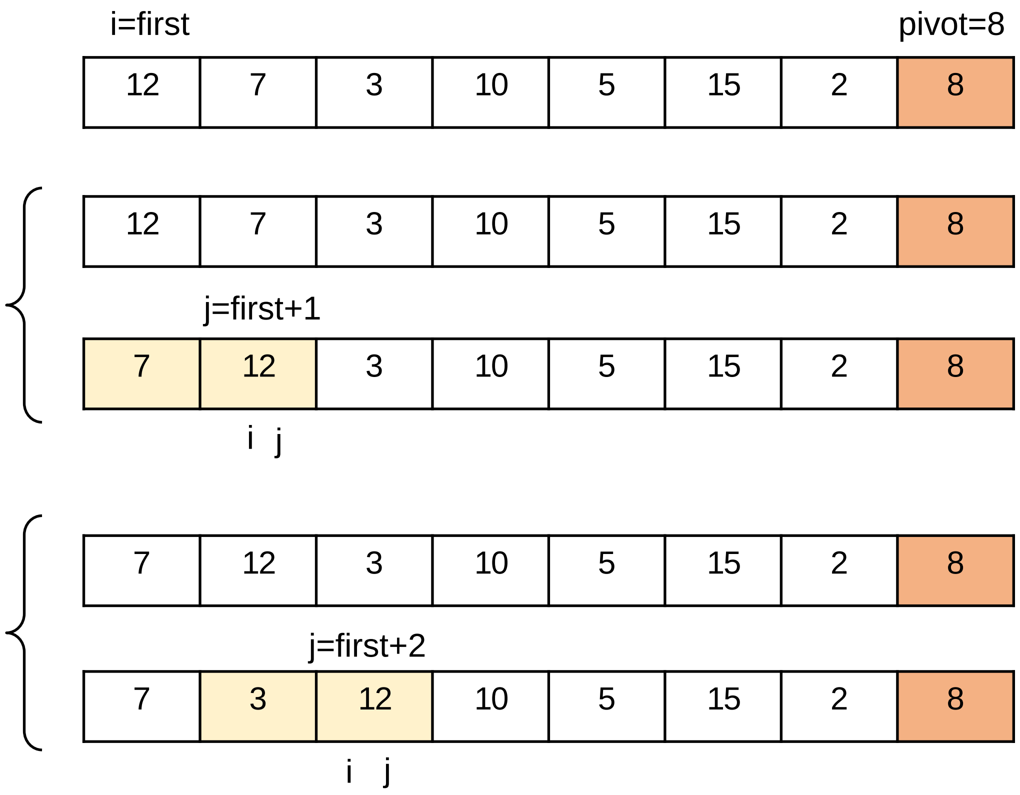
At each point if we look at all elements between A[first] and A[j], then

1) all elements in A[first] till A[i-1] are less than pivot
2) Others are greater than or equal to pivot

```
partition(A, first, last) {
    pivot = A[last]
    i = first

    for j = first to last-1 {
        if A[j] < pivot {
            swap(A[i], A[j])
            i = i+1
        }
    }
    swap(A[i], A[last])
    return i
}
```

# Quick Sort



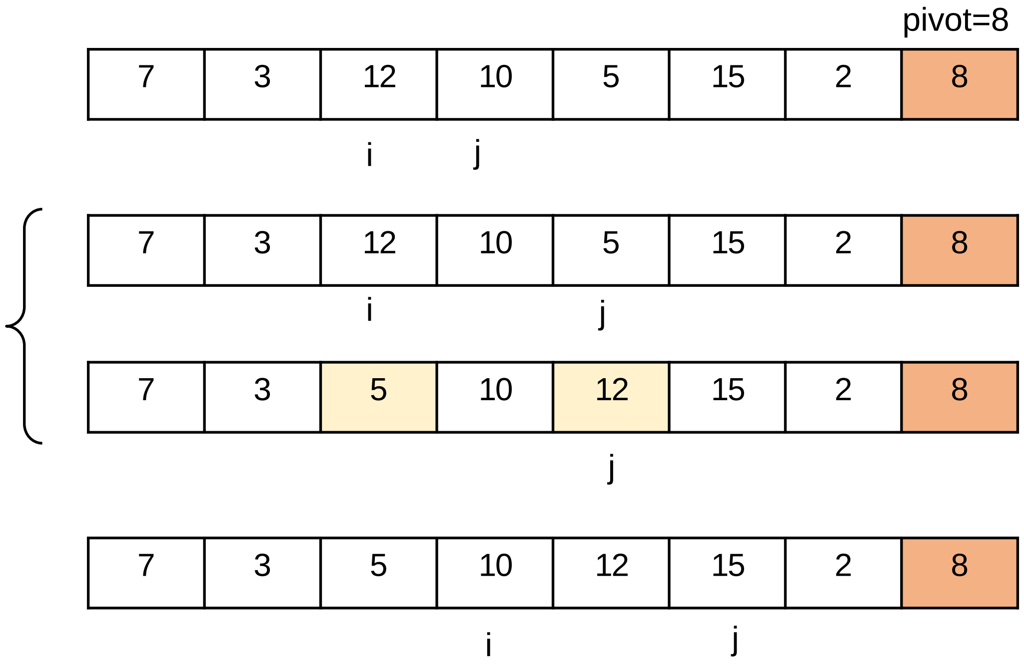

```
quick_sort(A, first, last){
    if first >= last
        return

    p = partition(A, first, last)
    quick_sort(A, first, p - 1)
    quick_sort(A, p + 1, last)
}
```

```
partition(A, first, last) {
    pivot = A[last]
    i = first

    for j = first to last-1 {
        if A[j] < pivot {
            swap(A[i], A[j])
            i = i+1
        }
    }
    swap(A[i], A[last])
    return i
}
```

# Quick Sort




```
quick_sort(A, first, last){
    if first >= last
        return

    p = partition(A, first, last)
    quick_sort(A, first, p - 1)
    quick_sort(A, p + 1, last)
}
```

```
partition(A, first, last) {
    pivot = A[last]
    i = first

    for j = first to last-1 {
        if A[j] < pivot {
            swap(A[i], A[j])
            i = i+1
        }
    }
    swap(A[i], A[last])
    return i
}
```

# Quick Sort

pivot=8

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 3 | 5 | 10 | 12 | 15 | 2 | 8 |
| | | | i | | | j | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 3 | 5 | 2 | 12 | 15 | 10 | 8 |
| | | | | i | | j | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 3 | 5 | 2 | 12 | 15 | 10 | 8 |
| | | | | i | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 3 | 5 | 2 | 8 | 15 | 10 | 12 |

```
quick_sort(A, first, last){
    if first >= last
        return

    p = partition(A, first, last)
    quick_sort(A, first, p - 1)
    quick_sort(A, p + 1, last)
}
```

```
partition(A, first, last) {
    pivot = A[last]
    i = first

    for j = first to last-1 {
        if A[j] < pivot {
            swap(A[i], A[j])
            i = i+1
        }
    }
    swap(A[i], A[last])
    return i
}
```

# Quick Sort – Running time

Running time: $T(n) = T(|L|) + T(|R|) + O(n)$

Best case: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Worst case: $T(n) = T(n-1) + O(n)$

Find an example for the worst case.

```
quick_sort(A, first, last){
    if first >= last
        return

    p = partition(A, first, last)
    quick_sort(A, first, p - 1)
    quick_sort(A, p + 1, last)
}
```

```
partition(A, first, last) {
    pivot = A[last]
    i = first

    for j = first to last-1 {
        if A[j] < pivot {
            swap(A[i], A[j])
            i = i+1
        }
    }
    swap(A[i], A[last])
    return i
}
```

# Randomized Quick Sort

Running time: $T(n) = T(|L|) + T(|R|) + O(n)$

Best case: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Worst case: $T(n) = T(n-1) + O(n)$

It can be proved that the average running time of randomized quick sort is $O(n \log n)$

```
quick_sort(A, first, last){
    if first >= last
        return

    p = partition(A, first, last)
    quick_sort(A, first, p - 1)
    quick_sort(A, p + 1, last)
}
```

```
random_partition(A, first, last) {
    k = a random number between first and last
    swap(A[k], A[last])

    pivot = A[last]
    i = first
    for j = first to last-1 {
        if A[j] < pivot {
            swap(A[i], A[j])
            i = i+1
        }
    }
    swap(A[i], A[last])
    return i
}
```

# A Conservative Implementation of Randomized Quick Sort

Running time: $T(n) = T(|L|) + T(|R|) + O(n)$

Probability of having a good pivot is 1/2


Probability of not having good pivot after 1000 iteration is $\left(\frac{1}{2}\right)^{1000} < \left(\frac{1}{1000}\right)^{100} \approx 0$

Worst case: $T(n) = T\left(\frac{3n}{4}\right) + T\left(\frac{n}{4}\right) + O(n) \approx O(n \log n)$

```
quick_sort(A, first, last){
    if first >= last
        return
    len = last - first + 1
    p = first - 1
    while p < first + len / 4 or p > last - len / 4
        p = random_partition(A, first, last)

    quick_sort(A, first, p - 1)
    quick_sort(A, p + 1, last)
}
```

```
random_partition(A, first, last) {
    k = a random number between first and last
    swap(A[k], A[last])

    pivot = A[last]
    i = first
    for j = first to last-1 {
        if A[j] < pivot {
            swap(A[i], A[j])
            i = i+1
        }
    }
    swap(A[i], A[last])
    return i
}
```

# Finding Median

- Problem: You are given an array of numbers. Find the median of this array.

I can solve it in $O(n \log n)$

Can you design a faster one?

# Finding Median

- **Problem:** You are given an array of numbers. Find the median of this array.

Idea: Let's solve a more general problem.
Rank Problem: You are given an array of numbers and an integer $k$. Find the $k^{\text{th}}$ smallest elements.

case 1: $k^{\text{th}}$ element is exactly pivot


case 2: $k^{\text{th}}$ element is before pivot

case 3: $k^{\text{th}}$ element is after pivot

```
rank(A, first, last, k) {
    if first >= last
        return A[first]

    p = partition(A, first, last)
    if k = p - first + 1
        return A[p] // case 1
    else if k < p - first + 1
        return rank(A, first, p-1, k) // case 2
    else
        return rank (A, p+1, last, k-p+first-1) // case 3
}
```

```
partition(A, first, last) {
    pivot = A[last]
    i = first
    for j = first to last-1 {
        if A[j] < pivot {
            swap(A[i], A[j])
            i = i+1
        }
    }
    swap(A[i], A[last])
    return i
}
```

# Finding Median

- **Problem:** You are given an array of numbers. Find the median of this array.

What is the running time of the proposed algorithm?

With an idea like the implementation of randomized quicksort, the worst case is:

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

```
rank(A, first, last, k) {
    if first >= last
        return A[first]

    p = partition(A, first, last)
    if k = p - first + 1
        return A[p] // case 1
    else if k < p - first + 1
        return rank(A, first, p-1, k) // case 2
    else
        return rank (A, p+1, last, k-p+first-1) // case 3
}
```

```
partition(A, first, last) {
    k = a random number between first and last
    swap(A[k], A[last])

    pivot = A[last]
    i = first
    for j = first to last-1 {
        if A[j] < pivot {
            swap(A[i], A[j])
            i = i+1
        }
    }
    swap(A[i], A[last])
    return i
}
```

# Maximum Subarray Sum

# Maximum Subarray Sum

- **Input:** You are given an array of numbers (positive and negative).
- **Goal:** Find the sum of contiguous subarray of numbers which has the largest sum.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -2 | -5 | 6 | -2 | -3 | 1 | 5 | -6 |

Maximum sum is 7

I can solve it in $O(n^2)$

Can you design a faster one?

# Maximum Subarray Sum

- **Input:** You are given an array of numbers (positive and negative).
- **Goal:** Find the sum of contiguous subarray of numbers which has the largest sum.

| A | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | -2 | -5 | 6 | -2 | -3 | 1 | 5 | -6 |

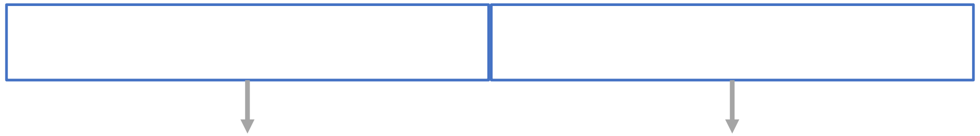
Maximum sum is 7

| B | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | -2 | -7 | -1 | -3 | -6 | -5 | 0 | -6 |
| | | | i | | | | | j | |

Idea:
- Let $B[0] = 0$ and $B[i] = \Sigma_0^{i-1} A[i]$ for $i > 0$
- Find $i < j$ to maximize $B[j] - B[i]$
- $(i, j]$ is the answer to the original problem

# Maximum Subarray Sum

- **Input:** You are given array $B$ of numbers (positive and negative).
- **Goal:** Find $i < j$ to maximize $B[j] - B[i]$



$i_L, j_L$ $\qquad$ $i_R, j_R$

Divide

Options:

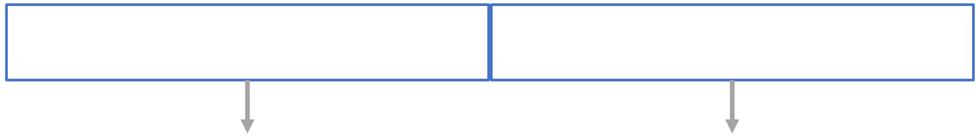
- Both $i$ and $j$ are in the left subarray — $B[j_L] - B[i_L]$ — Solve for the left subarray ($T\left(\frac{n}{2}\right)$)
- Both $i$ and $j$ are in the right subarray — $B[j_R] - B[i_R]$ — Solve for the right subarray ($T\left(\frac{n}{2}\right)$)
- $i$ is in the left and $j$ is the right subarray — $B[max._R] - B[min._L]$ — Find the minimum of the left and the maximum of the right subarray (Loop of $O(n)$)

Merge

**Running time**: $\mathrm{T}(n) = 2T\left(\frac{n}{2}\right) + O(n) \approx O(n \log n)$

# Maximum Subarray Sum

- Input: You are given array $B$ of numbers (positive and negative).
- Goal: Find $i < j$ to maximize $B[j] - B[i]$



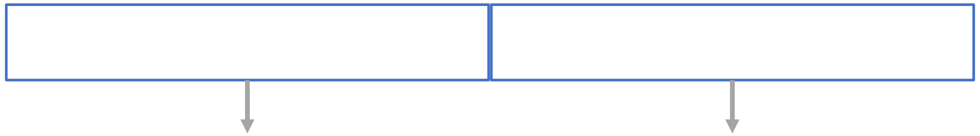
$i_L, j_L, \min._L, \max._L$ $\quad$ $i_R, j_R, \min._R, \max._R$

How to compute $i, j, \min, \max$ in $O(1)$?

**Running time**: $T(n) = 2T\left(\frac{n}{2}\right) + O(1) \approx O(n)$

# Maximum Subarray Sum

- Input: You are given array B of numbers (positive and negative).
- Goal: Find $i < j$ to maximize $B[j] - B[i]$



$i_L, j_L, \min._L, \max._L$ $\qquad$ $i_R, j_R, \min._R, \max._R$

$$\begin{cases} min = \underset{m \in \{min._L, min._R\}}{\operatorname{argmin}} B[m] \\ \max = \underset{m \in \{max._L, max._R\}}{argmax} B[m] \\ i \text{ and } j \text{ are selected based on maximum of} \\ B[j_L] - B[i_L], B[j_R] - B[i_R] \text{ and } B[max._R] - B[min._L] \end{cases}$$

**Running time**: $T(n) = 2T\left(\frac{n}{2}\right) + O(1) \approx O(n)$

Divide

Merge

```
best_index(B, first, last){
    if first >= last
        return first, first, first, first

    mid = (first + last)/2
    iL, jL, minL, maxL = best_index(B, first, mid)
    iR, jR, minR, maxR = best_index (B, mid+1, last)

    if B[minL] < B[minR]    // Finding minimum
        min = minL
    else
        min = minR

    if B[maxL] > B[maxR]    // Finding maximum
        max = maxL
    else
        max= maxR

    if B[jL]-B[iL] > B[jR]-B[iR] and B[jL]-B[iL] > B[maxR]-B[minL]
        i = iL, j = jL
    else if B[jR]-B[iR] > B[maxR] - B[minL]    // Finding i and j
        i = iR, j = jR
    else
        i = minL, j = maxR

    return i, j, min, max
}
```

# Polynomial Multiplication

# Polynomial Multiplication

- **Input:** You are given two polynomial A(x) and $B(x)$ of order $n$
- **Goal:** Find $C(x) = A(x) \times B(x)$

Fact: Polynomial $A(x) = \Sigma_{i=0}^{n} a_i x^i$ of order $\boldsymbol{n}$ can be represented by an array of size $\boldsymbol{n+1}$

$A(x) = 1 + 2x^2 - 4x^3$ can be represented by array (1, 0, 2, -4)

Example:

- $A(x) = 1 + 2x^2 - 4x^3$
- $B(x) = -1 + x - x^3$
- $C(x) = A(x) \times B(x) = -1 + x - 2x^2 + 5x^3 - 4x^4 - 2x^5 + 4x^6$

# Polynomial Multiplication

- **Input:** You are given two polynomial A(x) and $B(x)$ of order $n$
- **Goal:** Find $C(x) = A(x) \times B(x)$

I can solve it in $O(n^2)$

Since $c_k = \Sigma_{i=0}^{k} a_i b_{k-i}$ each $c_k$ can be computed in $O(n)$

Can you design a faster one?

# Polynomial Multiplication

- **Input:** You are given two polynomial $A(x)$ and $B(x)$ of order $n$
- **Goal:** Find $C(x) = A(x) \times B(x)$

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{2}}$$

| $A_0(x)$ | $A_1(x)$ |
|---|---|

| $B_0(x)$ | $B_1(x)$ |
|---|---|

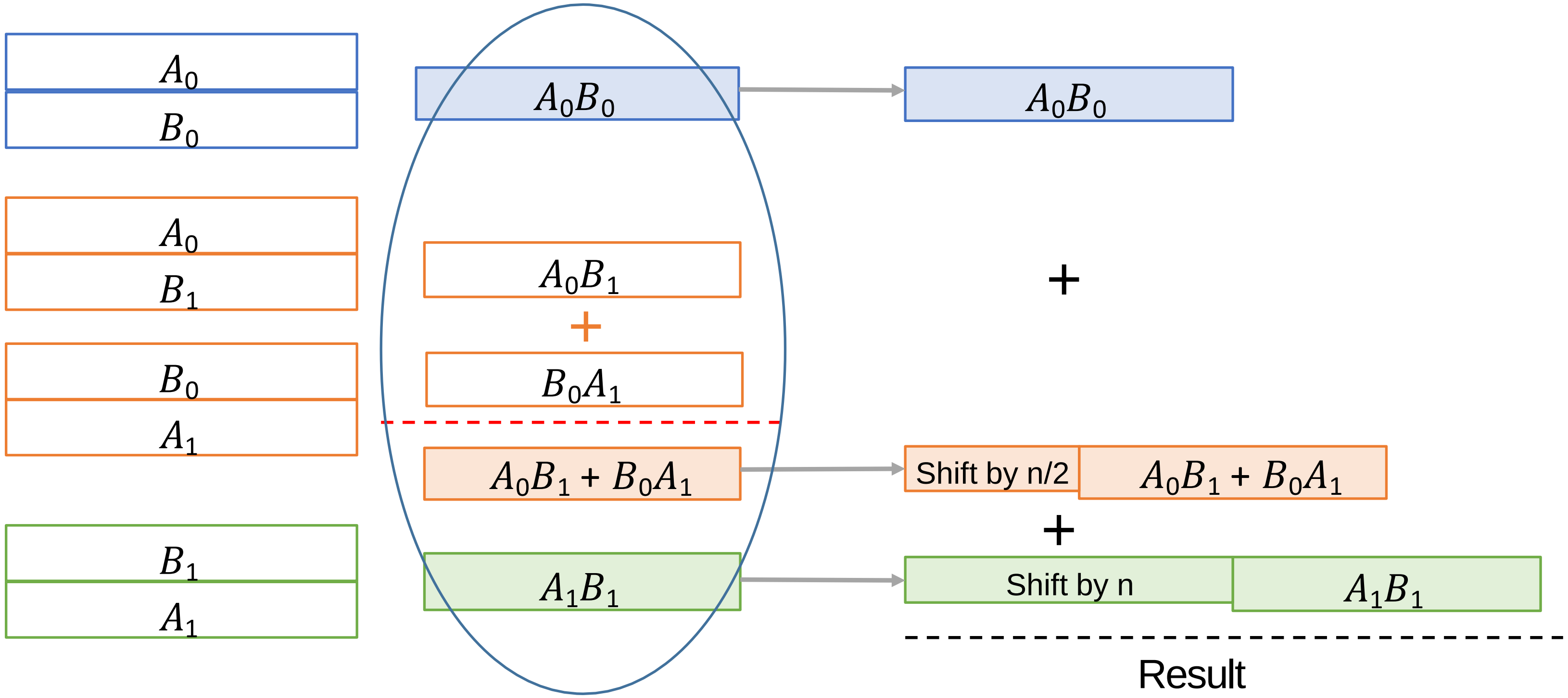
$$B(x) = B_0(x) + B_1(x)x^{\frac{n}{2}}$$

$$A(x)B(x) = \underbrace{\boxed{A_0(x)B_0(x)} + x^{\frac{n}{2}}\left[\boxed{A_1(x)B_0(x)} + \boxed{A_0(x)B_1(x)}\right] + x^{n}\boxed{A_1(x)B_1(x)}}_{\text{4 sub-problem}}$$

# Polynomial Multiplication

- **Input:** You are given two polynomial $A(x)$ and $B(x)$ of order $n$
- **Goal:** Find $C(x) = A(x) \times B(x)$

$$A(x)B(x) = \boxed{A_0(x)B_0(x)} + x^{\frac{n}{2}}[\boxed{A_1(x)B_0(x)} + \boxed{A_0(x)B_1(x)}] + x^n \boxed{A_1(x)B_1(x)}$$



# Polynomial Multiplication

- **Input:** You are given two polynomial $A(x)$ and $B(x)$ of order $n$
- **Goal:** Find $C(x) = A(x) \times B(x)$

$$A(x)B(x) = \underbrace{\boxed{A_0(x)B_0(x)} + x^{\frac{n}{2}}[\boxed{A_1(x)B_0(x)} + \boxed{A_0(x)B_1(x)}] + x^n \boxed{A_1(x)B_1(x)}}_{\text{4 sub-problem}}$$

Running time: $T(n) = 4T\left(\frac{n}{2}\right) + O(n) \approx O(n^2)$

Bottleneck is the number of sub-problems. How to reduce them?

# Polynomial Multiplication

- **Input:** You are given two polynomial $A(x)$ and $B(x)$ of order $n$
- **Goal:** Find $C(x) = A(x) \times B(x)$

$$A(x)B(x) = A_0(x)B_0(x) + x^{\frac{n}{2}}[A_1(x)B_0(x) + A_0(x)B_1(x)] + x^n A_1(x)B_1(x)$$

Idea: One can write $A_1B_0 + A_0B_1$ as $(A_1+A_0)(B_0 + B_1) - A_0B_0 - A_1B_1$ .
This reduces the number of sub-problems to three:

- $A_0B_0$
- $A_1B_1$
- $(A_0 + A_1)(B_0 + B_1)$

Running time: $T(n) = 3T\left(\frac{n}{2}\right) + O(n) \approx O(n^{\log_2 3}) \approx O(n^{1.58})$

# Polynomial Multiplication - Example

- **Input:** You are given two polynomial $A(x)$ and $B(x)$ of order $n$
- **Goal:** Find $C(x) = A(x) \times B(x)$

> - $\mathrm{A}(\mathrm{x}) = 2 + 5x + 2x^3$
> - $\mathrm{B}(\mathrm{x}) = 1 + 2x^2 + 3x^3$

$$\mathrm{A}: \overbrace{\boxed{2}\,\boxed{5}}^{A_0}\,\overbrace{\boxed{0}\,\boxed{2}}^{A_1}$$

$$\mathrm{B}: \underbrace{\boxed{1}\,\boxed{0}}_{B_0}\,\underbrace{\boxed{2}\,\boxed{3}}_{B_1}$$

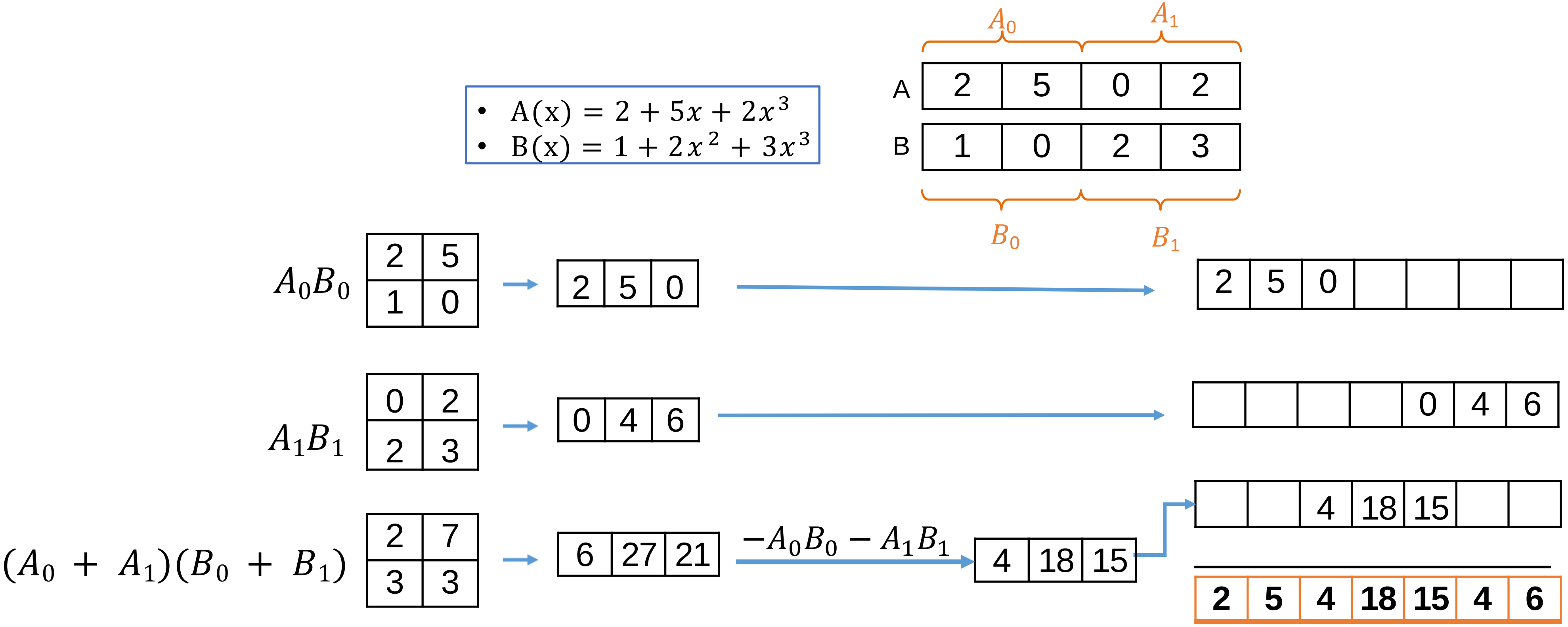
$$A_0B_0 \quad \begin{bmatrix} 2 & 5 \\ 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 5 & 0 \end{bmatrix}$$

$$A_1B_1 \quad \begin{bmatrix} 0 & 2 \\ 2 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 4 & 6 \end{bmatrix}$$

$$(A_0 + A_1)(B_0 + B_1) \quad \begin{bmatrix} 2 & 7 \\ 3 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 27 & 21 \end{bmatrix} \xrightarrow{-A_0B_0 - A_1B_1} \begin{bmatrix} 4 & 18 & 15 \end{bmatrix}$$

# Polynomial Multiplication - Example

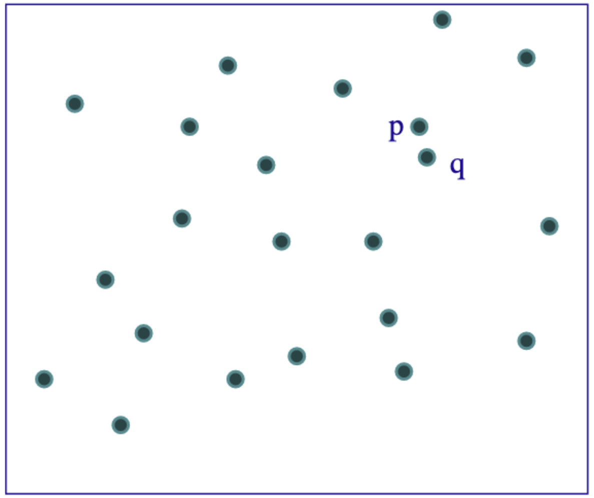
- **Input:** You are given two polynomial $A(x)$ and $B(x)$ of order $n$
- **Goal:** Find $C(x) = A(x) \times B(x)$



# Closest Pair of Points

# Closest Pair of Points

- **Input:** You are given an array of $n$ points.
- **Goal:** Find the closest pair of points




I can solve it in $O(n^2)$

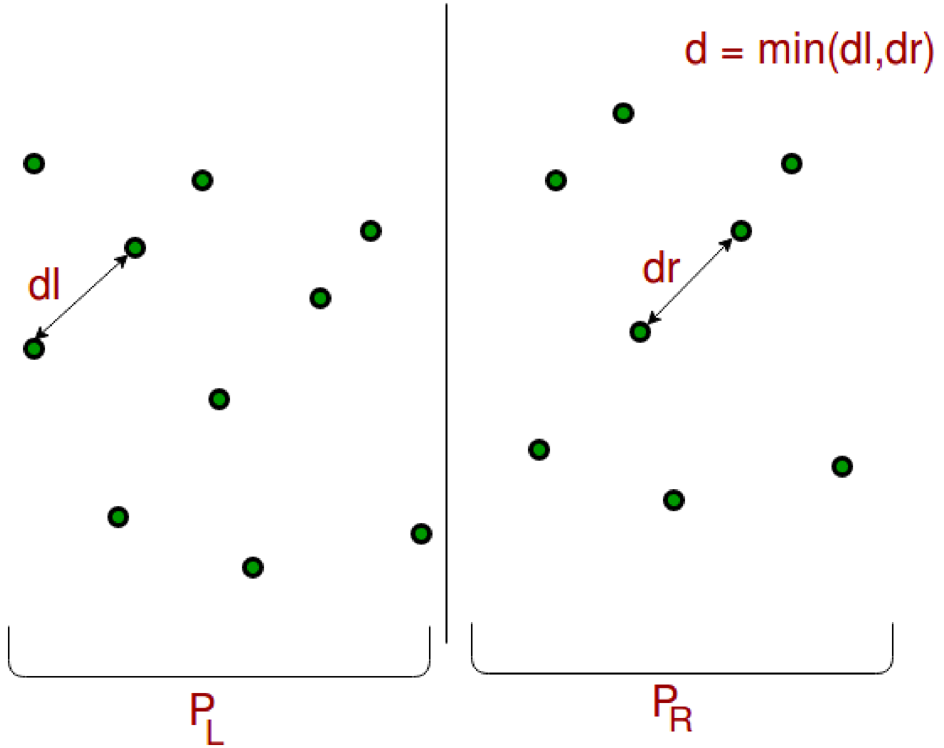
Check all pairs and choose the best one!

Can you design a faster one?

# Closest Pair of Points

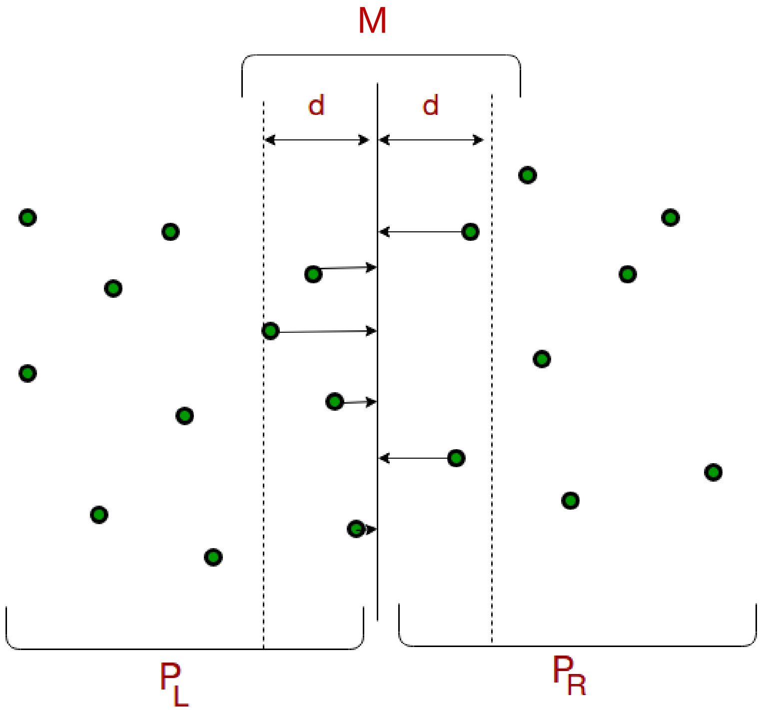
- Let's try the divide and conquer idea!
  1. Assume points are sorted based on their $x$ value
  2. Divide points into half (boundary is $x = M$)
  3. $dL$ = best solution for the left
  4. $dR$ = best solution for the right
  5. $d = \min(dL, dR)$
     - Is $d$ the right answer?

Options $\begin{cases} \text{Both points are in the left (recursive)} \\ \text{Both points are in the right (recursive)} \\ \text{One point is in the left and the other one in the right (How to find in } O(n)\text{?)} \end{cases}$



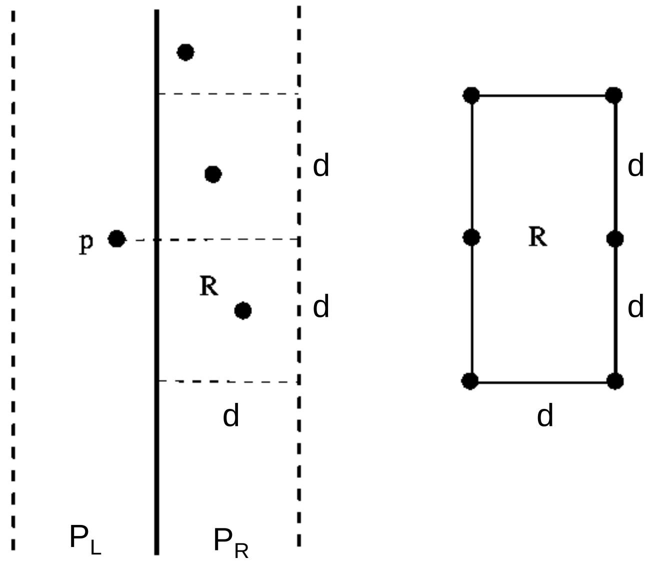
# Closest Pair of Points

1. Assume points are sorted based on their $x$ value
2. Divide points into half (boundary is $x=M$)
3. $dL$ = best solution for the left
4. $dR$ = best solution for the right
5. $d = \min(dL, dR)$
6. Discard any point with $x<M-d$ and $x>M+d$



# Closest Pair of Points

1. Assume points are sorted based on their $x$ value
2. Divide points into half (boundary is $x=M$)
3. $dL$ = best solution for the left
4. $dR$ = best solution for the right
5. $d = \min(dL, dR)$
6. Discard any point with $x<M-d$ and $x>M+d$
7. For the remaining points, sort them based on $y$
8. We just need to check each point with the next 11 points and update $d$ if we find a better pair



# Closest Pair of Points - Implementation

1. Assume points are sorted based on their $x$ value
2. Divide points into half (boundary is $x=M$)
3. $dL$ = best solution for the left
4. $dR$ = best solution for the right
5. $d$ = min($dL$, $dR$)
6. Discard any point with $x<M-d$ and $x>M+d$ // how to implement in $O(n)$ ?
7. For the remaining points, sort them based on $y$ // how to implement in $O(n)$ ?
8. We just need to check each point with the next 11 points and update $d$ if we find a better pair // how to implement in $O(n)$ ?

Divide

Merge

# Closest Pair of Points - Implementation

- Main idea: Preprocess
  - Maintain three arrays:
    - P: original points
    - X: index of points assuming they're sorted based on their x value
    - Y: index of points assuming they're sorted based on their y value
- We can build X and Y in $O(n \log n)$ in the preprocess phase.

How to implement the algorithm based on this above idea?

# Closest Pair of Points - Implementation

1. Assume points are sorted based on their $x$ value
2. Divide points into half (boundary is $x=M$)
3. $dL$ = best solution for the left
4. $dR$ = best solution for the right
5. $d = \min(dL, dR)$
6. Discard any point with $x<M-d$ and $x>M+d$
7. For the remaining points, sort them based on $y$
8. We just need to check each point with the next 11 points and update $d$ if we find a better pair

- M = P[X[n/2]].x
- We can easily divide array P, X, and Y with a simple loop. For example for array Y, here is the solution:

```
for i = 1 to n
    if P[Y[i]].x < M
        Add Y[i] to Y_l
    else
        Add Y[i] to Y_r
```

- Note that we prepare P_l, X_l, and Y_l for the left sub-problem, and P_r, X_r, Y_r for the right sub-problem.

Handling corner cases properly

```
for i = 1 to n
    if P[Y[i]].x < M
        Add Y[i] to Y_l
    else if P[Y[i]].x > M
        Add Y[i] to Y_r
    else if P[Y[i]].x == M and Y_l.length < n/2
        Add Y[i] to Y_l
    else
        Add Y[i] to Y_r
```

# Closest Pair of Points - Implementation

1. Assume points are sorted based on their $x$ value
2. Divide points into half (boundary is $x=M$)
3. $dL$ = best solution for the left
4. $dR$ = best solution for the right
5. $d = \min(dL, dR)$
6. Discard any point with $x<M-d$ and $x>M+d$
7. For the remaining points, sort them based on $y$
8. We just need to check each point with the next 11 points and update $d$ if we find a better pair

We can discard points in a loop. For example for array Y:

```
for i = 1 to n
    if P[Y[i]].x > M – d and P[Y[i]].x < M + d
        Add Y[i] to Y_filtered
```

Already sorted in array Y

# Closest Pair of Points - Implementation

1. Assume points are sorted based on their $x$ value
2. Divide points into half (boundary is $x = M$)
3. $dL$ = best solution for the left
4. $dR$ = best solution for the right
5. $d = \min(dL, dR)$
6. Discard any point with $x < M - d$ and $x > M + d$
7. For the remaining points, sort them based on $y$
8. We just need to check each point with the next 11 points and update $d$ if we find a better pair

We can find the smallest distance in the middle region:

```
k = size of Y_filtered array
for i = 1 to k
    p1 = P[Y_filtered[i]]
    for j = i + 1 to i + 11
        if j > k
            break
        p2 = P[Y_filtered[j]]
        if distance(p1, p2) < d
            d = distance(p1, p2)
```

Running time:

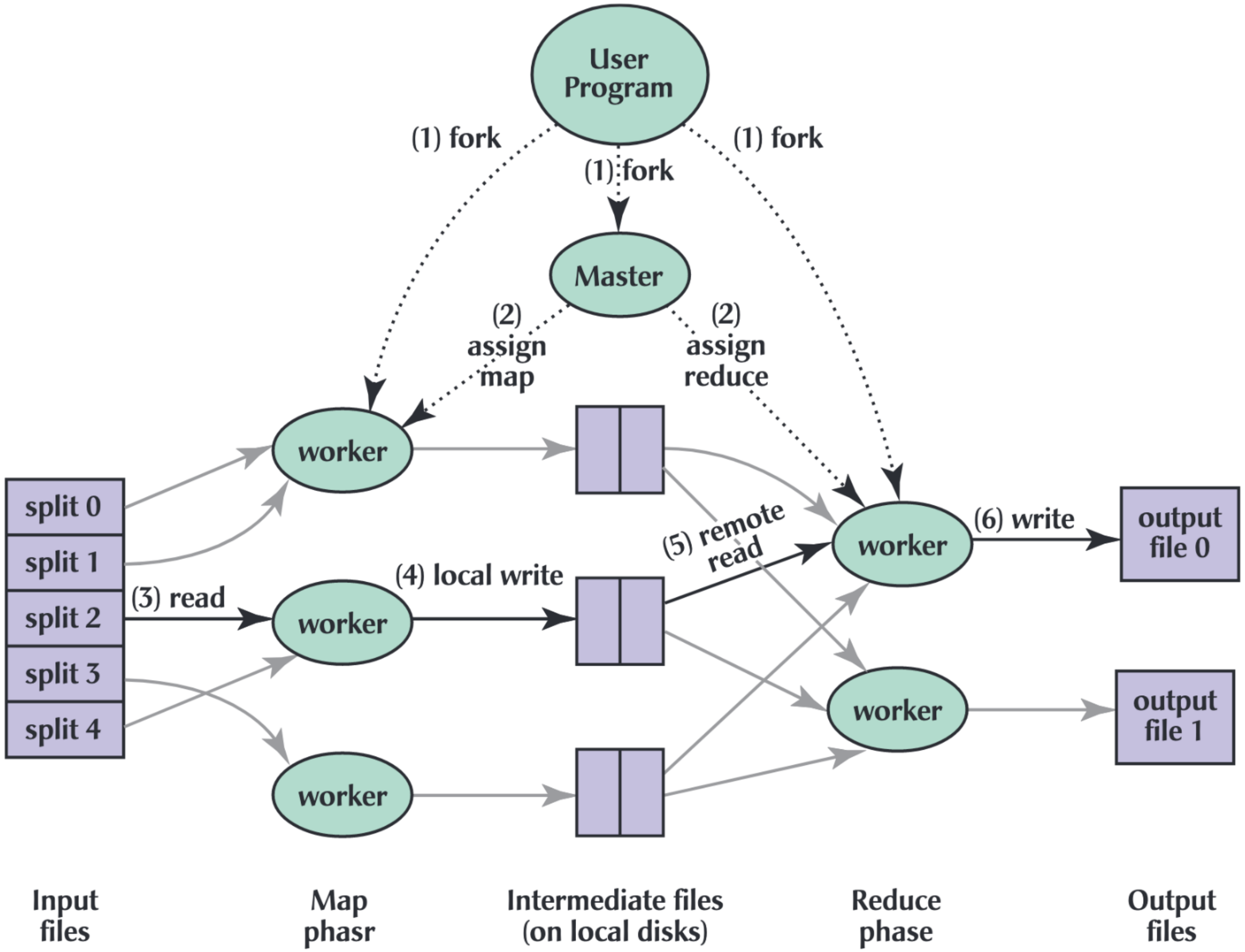
- *Preprocess:* $O(n \log n)$
- $T(n) = 2T\left(\frac{n}{2}\right) + O(n) \approx O(n \log n)$

# MapReduce: A Practical Example

# What is big data?

- How much data is actually considered *big*?
  - 1 GB, 10GB, 100GB, 1TB, ...?
- Big data means your memory is small!
- How to handle big data?
  - Sampling
  - Streaming
  - Distributing

# MapReduce Architecture

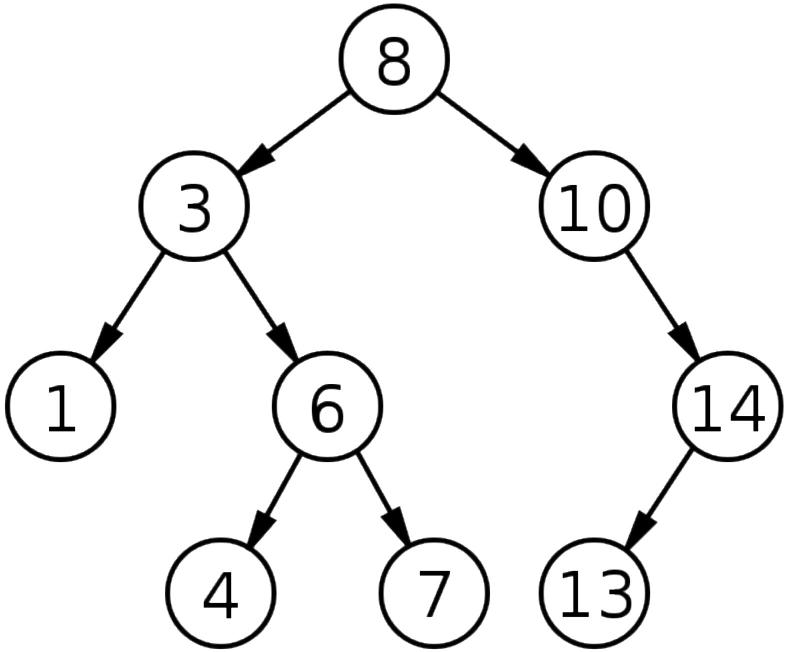



J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

# Sample Problems

# Lowest Common Ancestry in BST

- **Input:** A binary search tree (BST) $T$, its root node $r$, and two random nodes $x$ and $y$ in the tree.
- **Goal:** Find the lowest common ancestry of two nodes $x$ and $y$ in $O(\log n)$, where $n$ is the number of nodes in the tree.
  - *Each node is a descendant of itself, so if v has a direct connection from w, w is the lowest common ancestor of v and w.*
  - Note that a parent node $p$ has pointers to its children `p.leftChild` and `p.rightChild`, but a child node **does not** have a pointer to its parent node.
- Recall that in a BST, the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.



# Largest $m$ Integers

- Assume that $A$ is a very large unsorted array of integers with length $n$.
- Find $m$ largest integers in $A$, where $m \ll n$ in less than $O(n \log n)$.
- Also, the amount of additional memory that you are given is $O(1)$.

# Find The Largest Ratio

- You are given an array of $n$ positive numbers $A[1], A[2], \ldots, A[n]$.
- Give a divide and conquer algorithm to find indices $i < j$ such that $\frac{A[j]}{A[i]}$ is maximized.
- Your algorithm should run in $O(n)$ time.