

# Algorithms Week 2

Ljubomir Perković, DePaul University

# Course overview: core techniques

Algorithm design techniques that we will use in this class include:

- 1 divide-and-conquer
- 2 backtracking
- 3 dynamic programming
- 4 greedy

All of the above techniques rely on **reductions**.

As your textbook says:

*“Reductions are the single most common technique used in designing algorithms.*

*Reducing one problem  $X$  to another problem  $Y$  means to write an algorithm for  $X$  that uses an algorithm for  $Y$  as a black box or subroutine.*

*Crucially, the correctness of the resulting algorithm for  $X$  cannot depend in any way on how the algorithm for  $Y$  works. The only thing we can assume is that the black box solves  $Y$  correctly. The inner workings of the black box are simply none of our business; they're somebody else's problem. It's often best to literally think of the black box as functioning purely by magic.”*

Recursion is a special type of reduction that reduces a problem instance to one or more simpler instances of the same problem.

All four algorithm design techniques

- 1 divide-and-conquer
- 2 backtracking
- 3 dynamic programming
- 4 greedy

that we will use in this class can be described using recursion.

# Divide-and-conquer approach

A divide-and-conquer algorithm works as follows:

- 1 If the problem is small enough, solve it directly (and quickly).
- 2 Otherwise, divide it into subproblems (**Divide**) that you solve recursively (“as a black box functioning purely by magic”!)
- 3 Combine the solutions to the subproblems into a solution of the original problem (**Conquer**)

A fundamental problem is sorting an array of numbers.

**Input:** An array  $A[1..n]$  of  $n$  numbers

**Output:** A re-ordering of the numbers in the array such that  
 $A[1] \leq A[2] \leq \dots \leq A[n]$

There are many different ways of approaching the problem

InsertionSort works by repeatedly moving elements into their proper relative positions

A fundamental problem is sorting an array of numbers.

**Input:** An array  $A[1..n]$  of  $n$  numbers

**Output:** A re-ordering of the numbers in the array such that  
 $A[1] \leq A[2] \leq \dots \leq A[n]$

There are many different ways of approaching the problem

MergeSort recursively sorts each half of the array recursively and then merges the two sorted halves

```
MergeSort(A[1..n])  
  if n > 1  
    m ← ⌊n/2⌋  
    MergeSort(A[1..m])  
    MergeSort(A[m+1..n])  
    Merge(A[1..n], m)
```

We describe Merge next

# Merging the sorted halves

The specification for the Merge procedure is:

**Input:** An array  $A[1..N]$  and index  $m$  such that  $A[1..m]$  and  $A[m + 1..n]$  are sorted in non-decreasing order.

**Output:** A re-ordering of the numbers in the array such that  $A[1] \leq A[2] \leq \dots \leq A[n]$

```
Merge(A[1..n], m)
  i ← 1; j ← m+1
  for k ← 1 to n
    if j > n
      B[k] ← A[i]; i ← i+1
    else if i > m
      B[k] ← A[j]; j ← j+1
    else if A[i] < A[j]
      B[k] ← A[i]; i ← i+1
    else
      B[k] ← A[j]; j ← j+1
  for k ← 1 to n
    A[k] ← B[k]
```

```
Merge(A[1..n], m)
  i ← 1; j ← m+1
  for k ← 1 to n
    if j > n
      B[k] ← A[i]; i ← i+1
    else if i > m
      B[k] ← A[j]; j ← j+1
    else if A[i] < A[j]
      B[k] ← A[i]; i ← i+1
    else
      B[k] ← A[j]; j ← j+1
  for k ← 1 to n
    A[k] ← B[k]
```

Correctness: After iteration  $k$ ,  $B[1..k]$  holds the smallest  $k$  elements of  $A$  in sorted order.

```
Merge(A[1..n], m)
  i ← 1; j ← m+1
  for k ← 1 to n
    if j > n
      B[k] ← A[i]; i ← i+1
    else if i > m
      B[k] ← A[j]; j ← j+1
    else if A[i] < A[j]
      B[k] ← A[i]; i ← i+1
    else
      B[k] ← A[j]; j ← j+1
  for k ← 1 to n
    A[k] ← B[k]
```

Running time:  $O(n)$

# Running time analysis of MergeSort

Let  $T(n)$  be the time needed to sort an array of  $n$  elements.

- Lines 1 and 2 take constant time each.
- The first recursive call to MergeSort takes  $T(\lfloor \frac{n}{2} \rfloor)$  time.
- The second recursive call to MergeSort takes  $T(\lceil \frac{n}{2} \rceil)$  time.
- The call to Merge takes  $O(n)$  time.

and we thus we get the **recurrence relation**:

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(n)$$

# Running time analysis of MergeSort

Let  $T(n)$  be the time needed to sort an array of  $n$  elements.

- Lines 1 and 2 take constant time each.
- The first recursive call to MergeSort takes  $T(\lfloor \frac{n}{2} \rfloor)$  time.
- The second recursive call to MergeSort takes  $T(\lceil \frac{n}{2} \rceil)$  time.
- The call to Merge takes  $O(n)$  time.

and we thus we get the **recurrence relation**:

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

We can safely ignore the floors and ceilings (see textbook).

# Running time analysis of MergeSort

Let  $T(n)$  be the time needed to sort an array of  $n$  elements.

- Lines 1 and 2 take constant time each.
- The first recursive call to MergeSort takes  $T(\lfloor \frac{n}{2} \rfloor)$  time.
- The second recursive call to MergeSort takes  $T(\lceil \frac{n}{2} \rceil)$  time.
- The call to Merge takes  $O(n)$  time.

and we thus we get the **recurrence relation**:

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

We can safely ignore the floors and ceilings (see textbook).

A recurrence relation is a recursive way to define a function. It has two parts:

- ① The **recursive definition**, that is, the part that describes  $T(n)$  in terms of itself on smaller sized inputs.
  - For example,  $T(n) = 2T(n/2) + f(n)$
  - Or, a bit more loosely,  $T(n) = 2T(n/2) + O(n)$ .
- ② The **initial conditions**, that is, the part that determines when the recurrence stops.
  - For example,  $T(1) = 1$
  - Or,  $T(1) = O(1)$ , which we will in general assume and not write down.

## Solving recurrence relations

We can solve recurrence relations using recursion trees, which is a rooted tree with one node for each recursive subproblem.

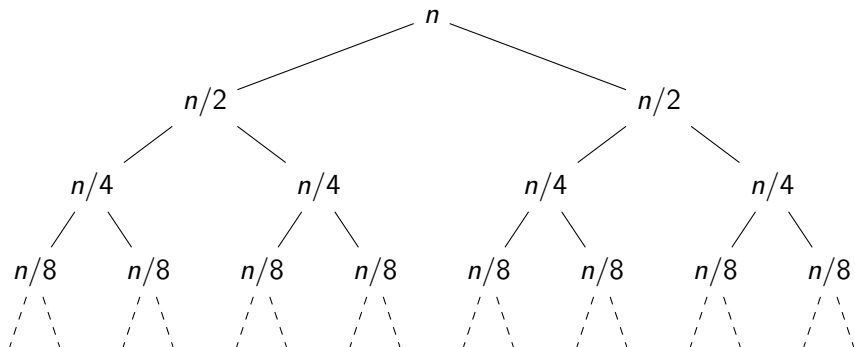
The value of each node is the amount of time spent on the corresponding subproblem excluding recursive calls.

## Solving recurrence relations

We can solve recurrence relations using recursion trees, which is a rooted tree with one node for each recursive subproblem.

The value of each node is the amount of time spent on the corresponding subproblem excluding recursive calls.

Example:  $T(n) = 2T(n/2) + O(n)$

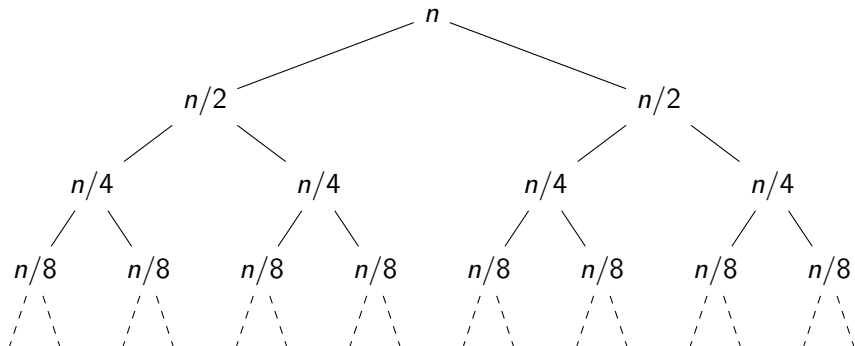


## Solving recurrence relations

Thus, the overall running time of the algorithm is the sum of the values of all nodes in the tree.

Since each row adds up to  $n$  and since there are  $\log_2 n$  rows (why?) the total running time  $T(n)$  is  $O(n \log n)$ .

Example:  $T(n) = 2T(n/2) + O(n) = O(n \log n)$



A fundamental problem is sorting an array of numbers.

**Input:** An array  $A[1..n]$  of  $n$  numbers

**Output:** A re-ordering of the numbers in the array such that  
 $A[1] \leq A[2] \leq \dots \leq A[n]$

There are many different ways of approaching the problem

InsertionSort works by repeatedly moving elements into their proper relative positions

A fundamental problem is sorting an array of numbers.

**Input:** An array  $A[1..n]$  of  $n$  numbers

**Output:** A re-ordering of the numbers in the array such that  
 $A[1] \leq A[2] \leq \dots \leq A[n]$

There are many different ways of approaching the problem

MergeSort recursively sorts each half of the array recursively and then merges the two sorted halves

A fundamental problem is sorting an array of numbers.

**Input:** An array  $A[1..n]$  of  $n$  numbers

**Output:** A re-ordering of the numbers in the array such that  
 $A[1] \leq A[2] \leq \dots \leq A[n]$

There are many different ways of approaching the problem

QuickSort partitions the array into three parts: an element chosen to be the pivot, a subarray containing elements smaller than the pivot, and a subarray containing elements larger than the pivot. QuickSort then recursively sorts the two subarrays.

```
QuickSort(A[1..n])
  if (n > 1)
    Choose a pivot element A[p]
    r ← Partition(A[1..n], p)
    QuickSort(A[1..r-1])
    QuickSort(A[r+1..n])
```

We describe Partition next

The Partition procedure has the following specification:

**Input:** An array  $A[1..N]$  and a pivot element  $P = A[p]$  where  $1 \leq p \leq N$ .

**Output:** Modified array  $A$  with an index  $r$  such that entries in  $A[1..r - 1]$  are less than  $P$ , entries in  $A[r + 1..n]$  are greater than or equal to  $P$ , and  $A[r] = P$ .

```
Partition(A[1..n], p)
  swap A[p] ↔ A[n]
  l ← 0
  for i ← 1 to n - 1
    if A[i] < A[n]
      l ← l+1
      swap A[l] A[i]
  swap A[n] ↔ A[l + 1]
  return l + 1
```

```
Partition(A[1..n], p)
  swap A[p] ↔ A[n]
  l ← 0
  for i ← 1 to n - 1
    if A[i] < A[n]
      l ← l+1
      swap A[l] A[i]
  swap A[n] ↔ A[l + 1]
  return l + 1
```

Correctness: At the end of each iteration of the main loop, everything in the subarray  $A[1..l]$  is  $< A[n]$  and everything in the subarray  $A[l + 1..i]$  is  $\geq A[n]$ .

```
Partition(A[1..n], p)
  swap A[p] ↔ A[n]
  l ← 0
  for i ← 1 to n - 1
    if A[i] < A[n]
      l ← l+1
      swap A[l] A[i]
  swap A[n] ↔ A[l + 1]
  return l + 1
```

Correctness: At the end of each iteration of the main loop, everything in the subarray  $A[1..l]$  is  $< A[n]$  and everything in the subarray  $A[l + 1..i]$  is  $\geq A[n]$ .

Running time:  $O(n)$ .

# Running time analysis of QuickSort

Let  $T(n)$  be the time needed to sort an array of  $n$  elements.

- Lines 1 and 2 take constant time each.
- The first call to Quicksort takes  $T(r - 1)$  time.
- The second call to QuickSort takes  $T(n - r)$  time.
- The call to Partition takes  $O(n)$  time.

and we thus we get the **recurrence relation**:

$$T(n) = T(r - 1) + T(n - r) + O(n)$$

# Running time analysis of QuickSort

Let  $T(n)$  be the time needed to sort an array of  $n$  elements.

- Lines 1 and 2 take constant time each.
- The first call to Quicksort takes  $T(r - 1)$  time.
- The second call to QuickSort takes  $T(n - r)$  time.
- The call to Partition takes  $O(n)$  time.

and we thus we get the **recurrence relation**:

$$T(n) = T(r - 1) + T(n - r) + O(n)$$

If we could magically choose the pivot to be the median then

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

# Running time analysis of QuickSort

Let  $T(n)$  be the time needed to sort an array of  $n$  elements.

- Lines 1 and 2 take constant time each.
- The first call to Quicksort takes  $T(r - 1)$  time.
- The second call to QuickSort takes  $T(n - r)$  time.
- The call to Partition takes  $O(n)$  time.

and we thus we get the **recurrence relation**:

$$T(n) = T(r - 1) + T(n - r) + O(n)$$

If we could magically choose the pivot to be the median then

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

# Running time analysis of QuickSort

Let  $T(n)$  be the time needed to sort an array of  $n$  elements.

- Lines 1 and 2 take constant time each.
- The first call to Quicksort takes  $T(r - 1)$  time.
- The second call to QuickSort takes  $T(n - r)$  time.
- The call to Partition takes  $O(n)$  time.

and we thus we get the **recurrence relation**:

$$T(n) = T(r - 1) + T(n - r) + O(n)$$

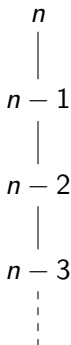
If the pivot happens to be the smallest or largest entry then

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

# Running time analysis of QuickSort

If the pivot happens to be the smallest or largest entry then

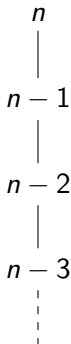
$$T(n) = T(n - 1) + O(n)$$



# Running time analysis of QuickSort

If the pivot happens to be the smallest or largest entry then

$$T(n) = T(n - 1) + O(n) = O(n^2)$$



# Running time analysis of QuickSort

What if we can guarantee that the pivot is in the middle half of the array? Then

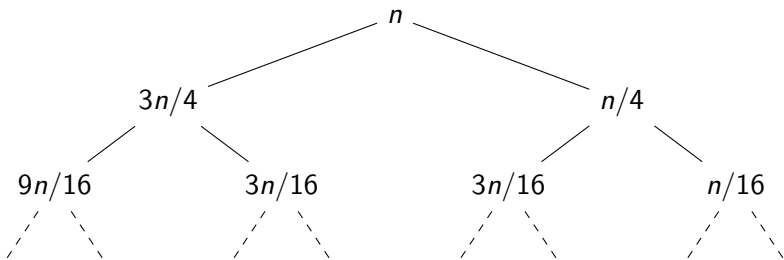
$$T(n) \leq T(3n/4) + T(n/4) + O(n)$$

# Running time analysis of QuickSort

What if we can guarantee that the pivot is in the middle half of the array? Then

$$T(n) = T(3n/4) + T(n/4) + O(n) = O(n \log n)$$

since all rows sum to  $n$  and the depth of the tree is  $\log_{4/3} n$ .



# Running time analysis of QuickSort

In Section 1.8, your textbook describes an algorithm that can be used to find the pivot in  $O(n)$  time.

Therefore the worst-case running time of QuickSort can be made  $O(n \log n)$ .

## (Last week) Exponentiation

Consider the problem of computing  $a^n$ :

**Input:** number  $a$  and positive integer  $n$

**Output:**  $a^n$

**Example:** given  $a = 1.5$  and  $n = 2$ ,  $a^n = 1.5^2 = 2.25$

The obvious algorithm:

```
SlowPower(a, n)
  res ← a
  for i ← 2 to n
    res ← res * a
  return res
```

## (Last week) Exponentiation

Consider the problem of computing  $a^n$ :

**Input:** number  $a$  and positive integer  $n$

**Output:**  $a^n$

**Example:** given  $a = 1.5$  and  $n = 2$ ,  $a^n = 1.5^2 = 2.25$

The obvious algorithm:

```
SlowPower(a, n)
  res ← a
  for i ← 2 to n
    res ← res * a
  return res
```

Running time:  $T(n) = O(n)$

## (Last week) Exponentiation

Consider the problem of computing  $a^n$ :

**Input:** number  $a$  and positive integer  $n$

**Output:**  $a^n$

**Example:** given  $a = 1.5$  and  $n = 2$ ,  $a^n = 1.5^2 = 2.25$

The obvious algorithm:

```
SlowPower(a, n)
  res ← a
  for i ← 2 to n
    res ← res * a
  return res
```

Running time:  $T(n) = O(n)$

How could divide-and-conquer possibly help?

## (Last week) Fast exponentiation

Indian scholar Pingala proposed the following recursive formula in 2nd century BCE!

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^{\lfloor n/2 \rfloor})^2 a & \text{otherwise} \end{cases}$$

which leads to the following exponentiation algorithm:

```
PingalaPower(a, n)
  if n ← 1
    return a
  tmp ← PingalaPower(a, ⌊n/2⌋)
  if n is even
    return tmp*tmp
  else
    return tmp*tmp*a
```

# Running time analysis of PingalaPower

The running time  $T(n)$  for input  $n$  satisfies

$$T(n) = T(n/2) + O(1)$$

1  
|  
1  
|  
1  
|  
1  
|  
⋮

# Running time analysis of PingalaPower

The running time  $T(n)$  for input  $n$  satisfies

$$T(n) = T(n/2) + O(1) = O(\log n)$$

since the depth of the tree is  $\log_2 n$ .

1  
|  
1  
|  
1  
|  
1  
|  
⋮

## Exercise: What is the running time of BinarySearch?

The recurrence relation for BinarySearch is

## Exercise: What is the running time of BinarySearch?

The recurrence relation for BinarySearch is

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

## Exercise: What is the running time of BinarySearch?

The recurrence relation for BinarySearch is

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

Multiplying two  $n$ -digit numbers  $x$  and  $y$  using a traditional multiplication algorithm requires  $O(n^2)$  single-digit multiplications.

$$\begin{array}{r} x_{n-1} \quad x_{n-2} \quad x_{n-3} \quad \dots \quad x_3 \quad x_2 \quad x_1 \quad x_0 \\ \times y_{n-1} \quad y_{n-2} \quad y_{n-3} \quad \dots \quad y_3 \quad y_2 \quad y_1 \quad y_0 \end{array}$$

Multiplying two  $n$ -digit numbers  $x$  and  $y$  using a traditional multiplication algorithm requires  $O(n^2)$  time.

$$\begin{array}{r} x_{n-1} \quad x_{n-2} \quad x_{n-3} \quad \dots \quad x_3 \quad x_2 \quad x_1 \quad x_0 \\ \times y_{n-1} \quad y_{n-2} \quad y_{n-3} \quad \dots \quad y_3 \quad y_2 \quad y_1 \quad y_0 \end{array}$$

Multiplying two  $n$ -digit numbers  $x$  and  $y$  using a traditional multiplication algorithm requires  $O(n^2)$  time.

$$\begin{array}{rcccccccc} & x_{n-1} & x_{n-2} & x_{n-3} & \dots & x_3 & x_2 & x_1 & x_0 \\ \times & y_{n-1} & y_{n-2} & y_{n-3} & \dots & y_3 & y_2 & y_1 & y_0 \end{array}$$

How could divide-and-conquer possibly help?

# Fast multiplication

$$\begin{array}{r} x_{n-1} \quad x_{n-2} \quad x_{n-3} \quad \dots \quad x_3 \quad x_2 \quad x_1 \quad x_0 \\ \times y_{n-1} \quad y_{n-2} \quad y_{n-3} \quad \dots \quad y_3 \quad y_2 \quad y_1 \quad y_0 \end{array}$$

# Fast multiplication

$$\begin{array}{cccccccc} & x_{n-1} & x_{n-2} & x_{n-3} & \dots & x_3 & x_2 & x_1 & x_0 \\ \times & y_{n-1} & y_{n-2} & y_{n-3} & \dots & y_3 & y_2 & y_1 & y_0 \end{array}$$

Let  $a$ ,  $b$ ,  $c$ , and  $d$  be numbers defined as follows:

$$\begin{aligned} a &= x_{n-1} & x_{n-2} & \dots & x_{m+1} & x_m \\ b &= x_{m-1} & x_{m-2} & \dots & x_1 & x_0 \\ c &= y_{n-1} & y_{n-2} & \dots & y_1 & y_0 \\ d &= y_{m-1} & y_{m-2} & \dots & y_1 & y_0 \end{aligned}$$

# Fast multiplication

$$\begin{array}{cccccccc} & x_{n-1} & x_{n-2} & x_{n-3} & \dots & x_3 & x_2 & x_1 & x_0 \\ \times & y_{n-1} & y_{n-2} & y_{n-3} & \dots & y_3 & y_2 & y_1 & y_0 \end{array}$$

Let  $a$ ,  $b$ ,  $c$ , and  $d$  be numbers defined as follows:

$$\begin{aligned} a &= x_{n-1} x_{n-2} \dots x_{m+1} x_m \\ b &= x_{m-1} x_{m-2} \dots x_1 x_0 \\ c &= y_{n-1} y_{n-2} \dots y_1 y_0 \\ d &= y_{m-1} y_{m-2} \dots y_1 y_0 \end{aligned}$$

Then  $xy = (10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$ .

## Fast multiplication

$$\begin{array}{cccccccc} & x_{n-1} & x_{n-2} & x_{n-3} & \dots & x_3 & x_2 & x_1 & x_0 \\ \times & y_{n-1} & y_{n-2} & y_{n-3} & \dots & y_3 & y_2 & y_1 & y_0 \end{array}$$

Let  $a$ ,  $b$ ,  $c$ , and  $d$  be numbers defined as follows:

$$\begin{aligned} a &= x_{n-1} x_{n-2} \dots x_{m+1} x_m \\ b &= x_{m-1} x_{m-2} \dots x_1 x_0 \\ c &= y_{n-1} y_{n-2} \dots y_1 y_0 \\ d &= y_{m-1} y_{m-2} \dots y_1 y_0 \end{aligned}$$

Then  $xy = (10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$ .

If  $m \approx n/2$ , multiplying two  $n$ -digit numbers is reduced to multiplying 4  $n/2$ -digit numbers:  $ac$ ,  $bc$ ,  $ad$ , and  $bd$ .

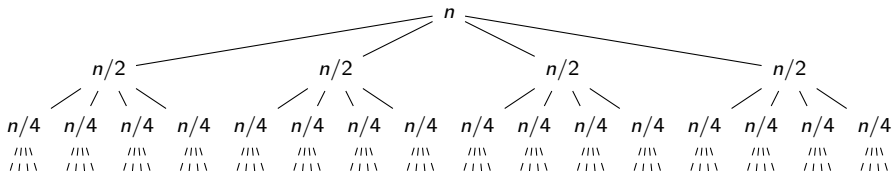
```
SplitMultiply(x, y, n):  
  if n = 1  
    return xy  
  else  
    m ← ⌈n/2⌉  
    a ← ⌊x/10m⌋; b ← x mod 10m  
    c ← ⌊y/10m⌋; d ← y mod 10m  
    e ← SplitMultiply(a, c, m)  
    f ← SplitMultiply(b, d, m)  
    g ← SplitMultiply(b, c, m)  
    h ← SplitMultiply(a, d, m)  
    return 102me + 10m(g + h) + f
```

Running time:  $T(n) = 4T(n/2) + O(n)$

# Running time analysis

The running time of this divide-and-conquer multiplication algorithm is

$$T(n) = 4T(n/2) + O(n)$$

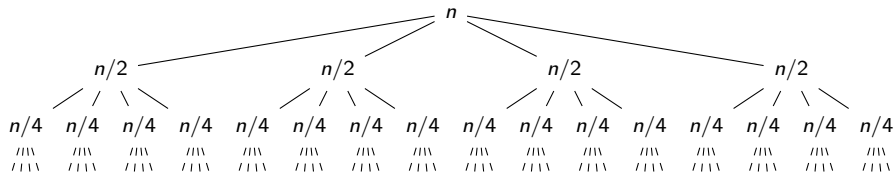


# Running time analysis

The running time of this divide-and-conquer multiplication algorithm is

$$T(n) = 4T(n/2) + O(n) = O(n^2)$$

since the rows add up to the geometric progression  $n, 2n, 2^2n, 2^3n, \dots, 2^{\log n}n$  which adds up to  $O(n^2)$ .

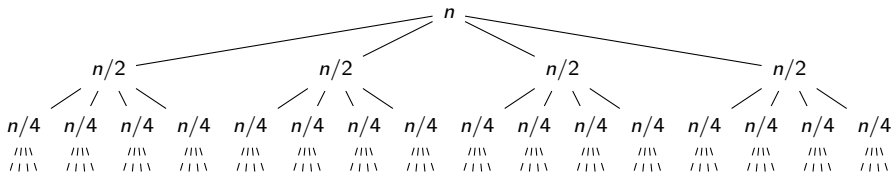


# Running time analysis

The running time of this divide-and-conquer multiplication algorithm is

$$T(n) = 4T(n/2) + O(n) = O(n^2)$$

since the rows add up to the geometric progression  $n, 2n, 2^2n, 2^3n, \dots, 2^{\log n}n$  which adds up to  $O(n^2)$ .



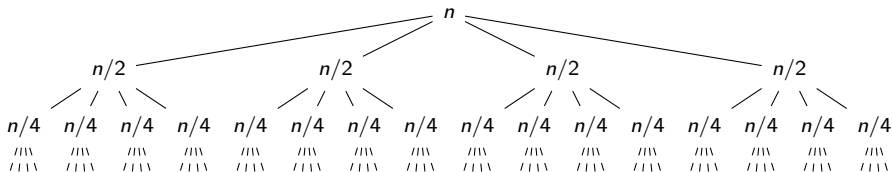
No gain!

# Running time analysis

The running time of this divide-and-conquer multiplication algorithm is

$$T(n) = 4T(n/2) + O(n) = O(n^2)$$

since the rows add up to the geometric progression  $n, 2n, 2^2n, 2^3n, \dots, 2^{\log n}n$  which adds up to  $O(n^2)$ .



No gain! So why did we bother ..?

## Fast multiplication

$$\begin{array}{r} x_{n-1} \quad x_{n-2} \quad x_{n-3} \quad \dots \quad x_3 \quad x_2 \quad x_1 \quad x_0 \\ \times y_{n-1} \quad y_{n-2} \quad y_{n-3} \quad \dots \quad y_3 \quad y_2 \quad y_1 \quad y_0 \end{array}$$

Let  $a$ ,  $b$ ,  $c$ , and  $d$  be numbers defined as follows:

$$\begin{aligned} a &= x_{n-1} \quad x_{n-2} \quad \dots \quad x_{m+1} \quad x_m \\ b &= x_{m-1} \quad x_{m-2} \quad \dots \quad x_1 \quad x_0 \\ c &= y_{n-1} \quad y_{n-2} \quad \dots \quad y_1 \quad y_0 \\ d &= y_{m-1} \quad y_{m-2} \quad \dots \quad y_1 \quad y_0 \end{aligned}$$

Then  $xy = (10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$ .

## Fast multiplication

$$\begin{array}{cccccccc} & x_{n-1} & x_{n-2} & x_{n-3} & \dots & x_3 & x_2 & x_1 & x_0 \\ \times & y_{n-1} & y_{n-2} & y_{n-3} & \dots & y_3 & y_2 & y_1 & y_0 \end{array}$$

Let  $a$ ,  $b$ ,  $c$ , and  $d$  be numbers defined as follows:

$$\begin{aligned} a &= x_{n-1} x_{n-2} \dots x_{m+1} x_m \\ b &= x_{m-1} x_{m-2} \dots x_1 x_0 \\ c &= y_{n-1} y_{n-2} \dots y_1 y_0 \\ d &= y_{m-1} y_{m-2} \dots y_1 y_0 \end{aligned}$$

Then  $xy = (10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m(bc + ad) + bd$ .

In the mid-1950s, Karatsuba, a 23 year old student, realized that we do not necessary need to compute  $bc$  and  $ad$  to get  $(bc + ad)$ . One can compute  $(bc + ad)$  as follows:

$$ac + bd - (a - b)(c - d) = bc + ad$$

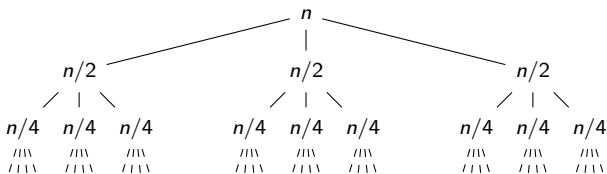
```
FastMultiply(x, y, n):  
  if n  $\leftarrow$  1  
    return xy  
  else  
    m  $\leftarrow$   $\lceil n/2 \rceil$   
    a  $\leftarrow$   $\lfloor x/10^m \rfloor$ ; b  $\leftarrow$   $x \bmod 10^m$   
    c  $\leftarrow$   $\lfloor y/10^m \rfloor$ ; d  $\leftarrow$   $y \bmod 10^m$   
    e  $\leftarrow$  FastMultiply(a, c, m)  
    f  $\leftarrow$  FastMultiply(b, d, m)  
    g  $\leftarrow$  FastMultiply(a - b, c - d, m)  
    return  $10^{2m} e + 10^m (e + f - g) + f$ 
```

Running time:  $T(n) = 3T(n/2) + O(n)$ .

## Solving recurrence relations

The running time of the 2nd divide-and-conquer multiplication algorithm is

$$T(n) = 3T(n/2) + O(n)$$

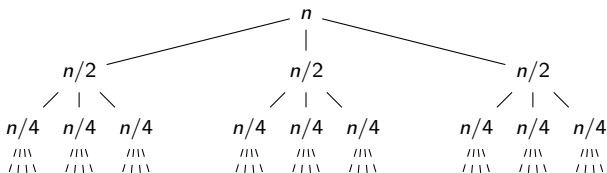


# Solving recurrence relations

The running time of the 2nd divide-and-conquer multiplication algorithm is

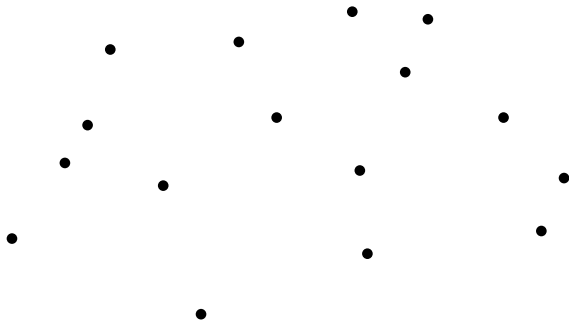
$$T(n) = 3T(n/2) + O(n) = O(n^{1.58496})$$

since the rows add up to the geometric progression  $n, 3n/2, (3/2)^2n, (3/2)^3n, \dots, (3/2)^{\log_2 n}n$  which adds up to  $O(n^{\log_2 3}) = O(n^{1.58496})$ .



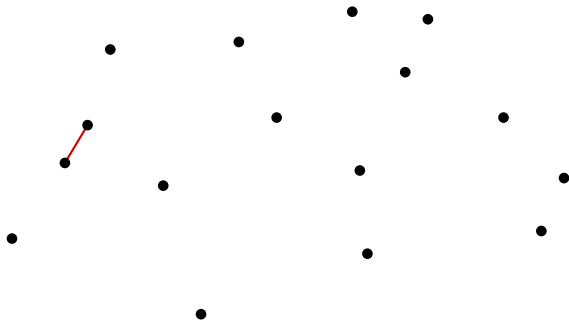
# Finding the closest pair of points

Given  $n$  points on the plane,



# Finding the closest pair of points

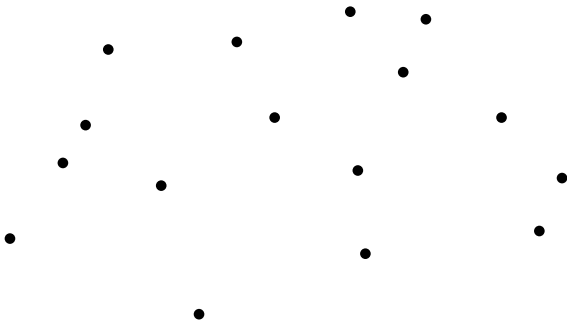
Given  $n$  points on the plane, find the pair that is closest together.



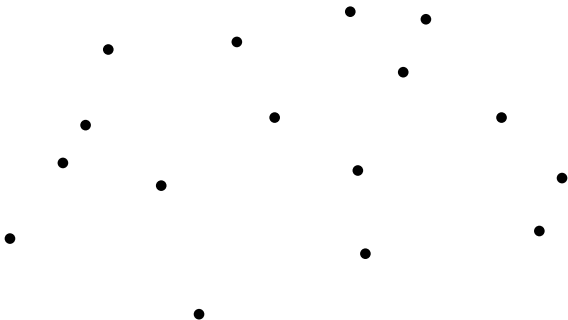
# Finding the closest pair of points

**Input:** Points  $p_1, \dots, p_n$  where  $p_i$  has coordinates  $(x_i, y_i)$ .

**Output:** Pair  $p_i, p_j$  whose distance  $d(p_i, p_j)$  is smallest.

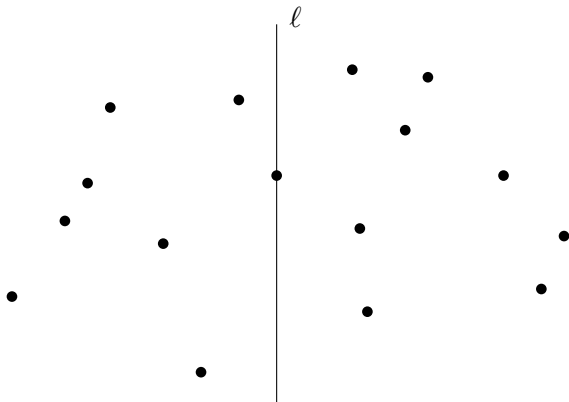


# Divide-and-conquer approach



# Divide-and-conquer approach

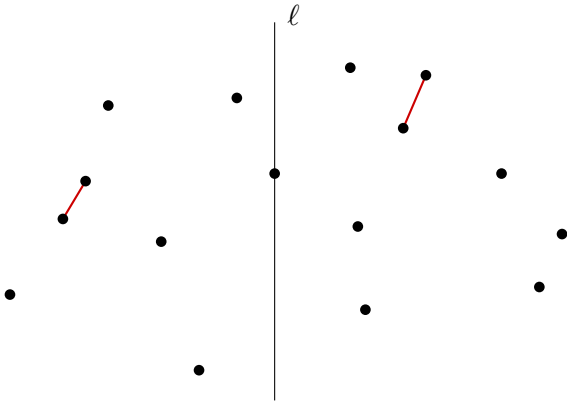
Step 1: Sort the points by  $x$ -coordinate



# Divide-and-conquer approach

Step 1: Sort the points by  $x$ -coordinate

Step 2: Split the plane at the median point and recursively find the closest pair of points in each half

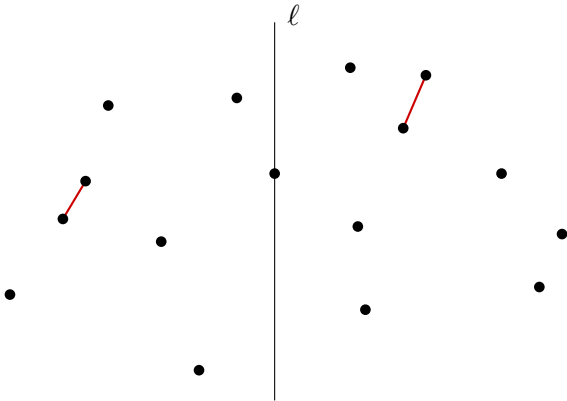


# Divide-and-conquer approach

Step 1: Sort the points by  $x$ -coordinate

Step 2: Split the plane at the median point and recursively find the closest pair of points in each half

Step 3: Return the closer of the two pairs!

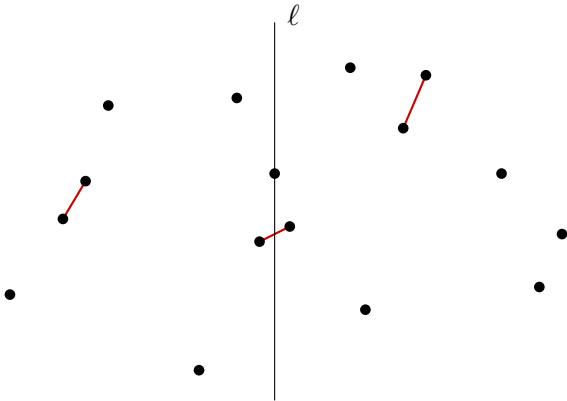


# Divide-and-conquer approach

Step 1: Sort the points by  $x$ -coordinate

Step 2: Split the plane at the median point and recursively find the closest pair of points in each half

Step 3: ~~Return the closer of the two pairs!~~

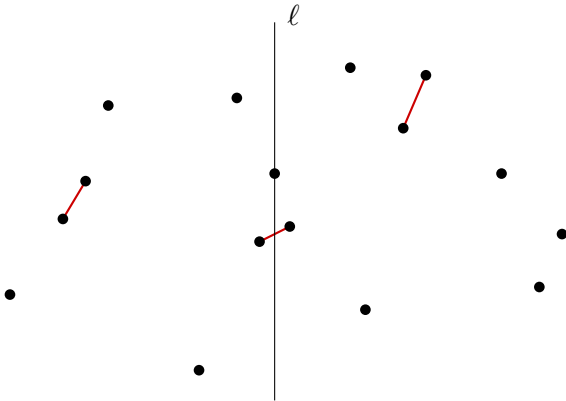


# Divide-and-conquer approach

Step 1: Sort the points by  $x$ -coordinate

Step 2: Split the plane at the median point and recursively find the closest pair of points in each half

Step 3: Need to also find the closest pair of points that lie on opposite sides of  $\ell$

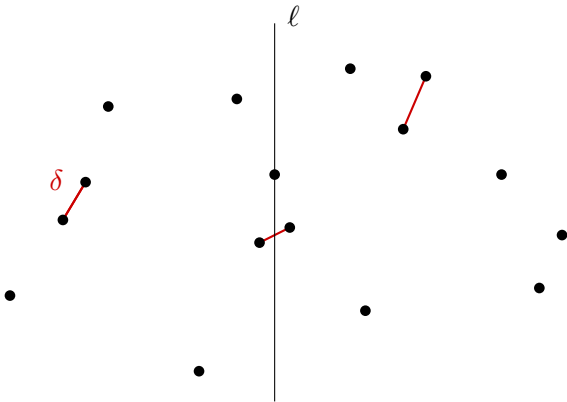


# Divide-and-conquer approach

Step 1: Sort the points by  $x$ -coordinate

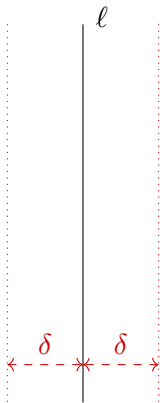
Step 2: Split the plane at the median point and recursively find the closest pair of points in each half

Step 3: Need to also find the closest pair of points that lie on opposite sides of  $\ell$  but only if their distance is less than  $\delta$



## Finding the closest *opposite* pair of points

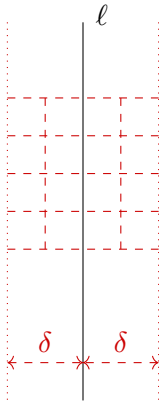
Insight 1: If there is a pair of opposite points whose distance is less than  $\delta$  their distance from line  $\ell$  must be less than  $\delta$



## Finding the closest *opposite* pair of points

Step 1.1: Sort the points by  $y$ -coordinate to get ordered list  $S_y$ .  
Let  $S'_y$  be the sublist of  $S_y$  consisting of points in the narrow  $2\delta$  band.

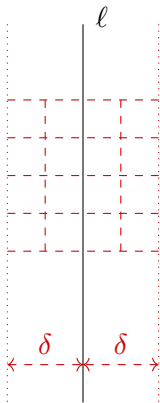
Insight 2: If two points in  $S'_y$  are closer than  $\delta$  then the points must be within 15 positions of each other in the list.



## Finding the closest *opposite* pair of points

Step 1.1: Sort the points by  $y$ -coordinate to get ordered list  $S_y$ .

Step 3: Construct sublist  $S'_y$  from  $S_y$  and then go through list  $S'_y$  and for each point compute the distance to the next 15 points in  $S'_y$  to find closest pair of opposite points whose distance is  $< \delta$ , if any.



# Running Time Analysis

- Steps 1 and 1.1 take  $O(n \log(n))$  time each and are done only once.
- Step 2 consists of two recursive calls, each taking  $T(n/2)$  time.
- Step 3 consists of a sequential search through  $S_y$  followed by a sequential search through  $S'_y$  which takes a total time of  $O(n)$ .

If  $T(n)$  is the running time of steps 2 and 3 then

$$T(n) = 2T(n/2) + O(n) = O(n \log n).$$

By homework problem 1(c), the running time of the whole algorithm is  $O(n \log n)$ .