

# CS711008Z Algorithm Design and Analysis

## Lecture 5. Basic algorithm design technique: DIVIDE AND CONQUER

Dongbo Bu

Institute of Computing Technology  
Chinese Academy of Sciences, Beijing, China

- The basic idea of DIVIDE AND CONQUER technique;
- The first example: MERGESORT
  - Correctness proof by using **loop invariant** technique;
  - Time complexity analysis of recursive algorithm.
- Other examples: COUNTINGINVERSION, CLOSESTPAIR, MULTIPLICATION, FFT;
- Combining with randomization: QUICKSORT, QUICKSELECT, BFPRT and FLOYDRIVEST algorithm for SELECTION problem;
- Remarks:
  - 1 DIVIDE AND CONQUER could serve to reduce the running time though **the brute-force algorithm is already polynomial-time**, say the  $O(n^2)$  brute-force algorithm versus  $O(n \log n)$  divide and conquer algorithm for the CLOSESTPAIR problem.
  - 2 This technique is especially powerful when **combined with randomization technique**.

# The general DIVIDE AND CONQUER paradigm

- Basic idea: Many problems are recursive in structure, i.e., to solve a given problem, they call themselves several times to deal with closely related **sub-problems**. These sub-problems have the same form to the original problem but a smaller size.
- Three steps of the DIVIDE AND CONQUER paradigm:
  - ① **Divide** a problem into a number of **independent sub-problems**;
  - ② **Conquer** the subproblems by solving them recursively;
  - ③ **Combine** the solutions to the subproblems into the solution to the original problem.

4.将大问题分成小问题研究解决。我曾请教过薛定谔如何做研究，他回答，Divide and Command，即分而制之。这样难点就化开了。其实这句话是古罗马的凯撒大帝说的。实践证明这个方法相当有效，不过要做到这一点，要能将大问题分解开来也是需要相当深厚的功力的。

# DIVIDE AND CONQUER technique

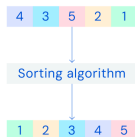
- To see whether the **DIVIDE AND CONQUER** technique applies on a given problem, we need to examine both **input** and **output** of the problem description.
  - Examine the **input** part to determine how to decompose the problem into subproblems of same structure but smaller size:  
It is relatively easy to decompose a problem into subproblems if the input part is related to the following data structures:
    - An **array** with  $n$  elements;
    - A **matrix**;
    - A **set** of  $n$  elements;
    - A **tree**;
    - A **directed acyclic graph**;
    - A **general graph**.
  - Examine the **output** part to determine how to construct the solution to the original problem using the solutions to its subproblems.

SORT problem: to sort an **array** of  $n$  integers

## SORT problem

**INPUT:** An array of  $n$  integers, denoted as  $A[0..n-1]$ ;

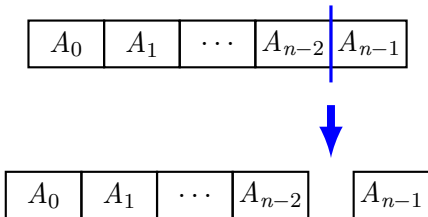
**OUTPUT:** The elements of  $A$  in increasing order.



- An array can be divided into smaller ones based on **indices** or **values** of elements.

# Divide strategy 1 based on indices of elements

- **Divide array  $A[0..n-1]$  into a  $n-1$ -length array  $A[0..n-2]$  and a single element:**  $A[n-1]$  has the same form to  $A[0..n-1]$  but smaller size; thus, sorting  $A[0..n-2]$  constructs a subproblem of the original problem. The **DIVIDE AND CONQUER** strategy might apply if we can sort  $A[0..n-1]$  using the sorted  $A[0..n-2]$ .





## Sort $A[0..n-1]$ using the sorted $A[0..n-2]$

- Basic idea: To sort  $A[0..n-1]$ , it suffices to put  $A[n-1]$  in its correct position among the sorted  $A[0..n-2]$ , which can be achieved through comparing  $A[n-1]$  with the elements in  $A[0..n-2]$ .

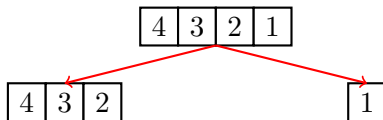
INSERTSORT(  $A, k$  )

```
1: if  $k \leq 1$  then  
2:   return ;  
3: end if  
4: INSERTSORT( $A, k-1$ );  
5:  $key = A[k]$ ;  
6:  $i = k-1$ ;  
7: while  $i \geq 0$  and  $A[i] > key$  do  
8:    $A[i+1] = A[i]$ ;  
9:    $i--$ ;  
10: end while  
11:  $A[i+1] = key$ ;
```

# An example

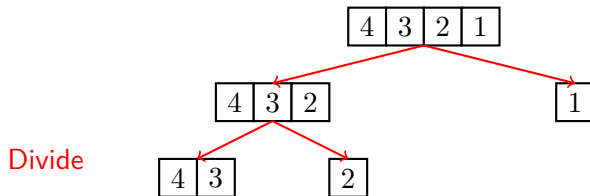
4	3	2	1
---	---	---	---

# An example

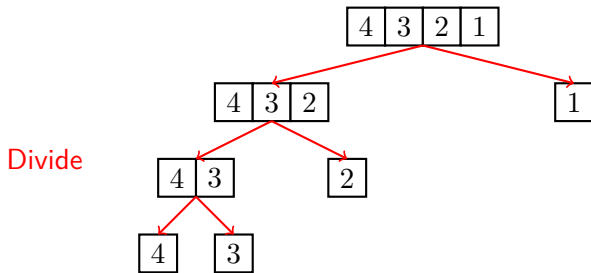


Divide

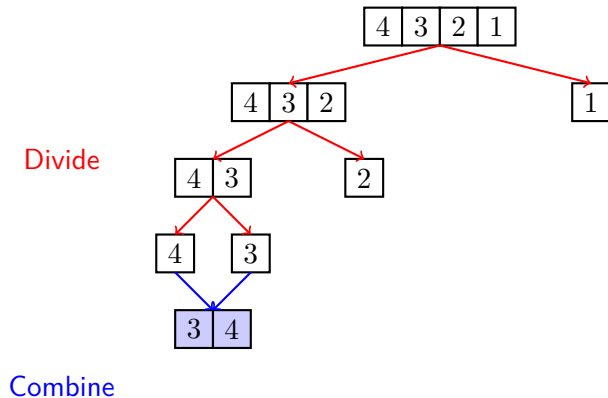
# An example



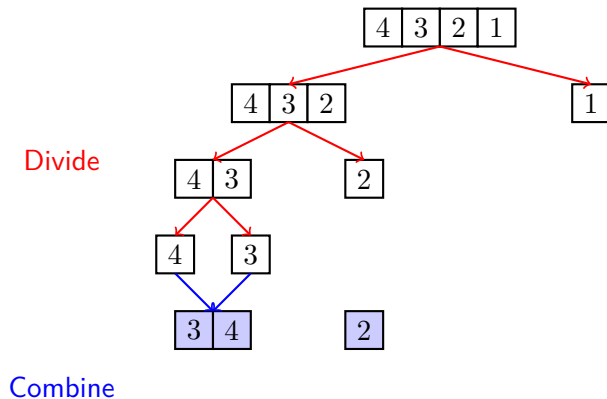
# An example



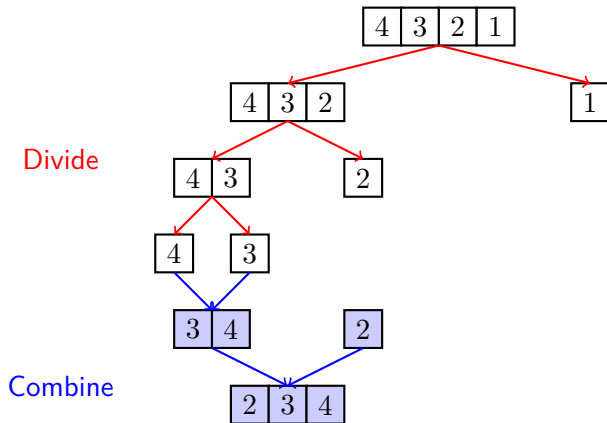
# An example



# An example

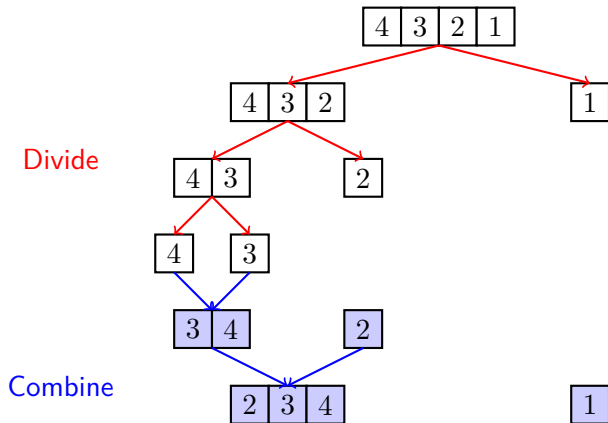


# An example

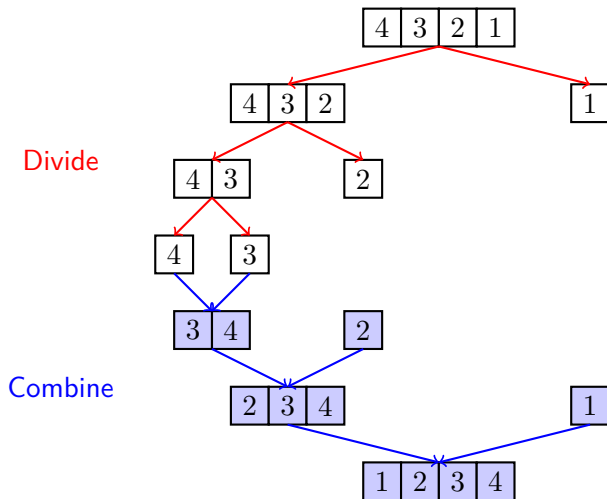




# An example

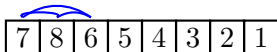
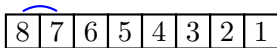


# An example

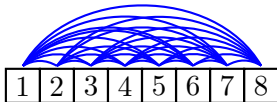


# Analysis of INSERTSORT algorithm

- Worst case: elements in  $A[0..n-1]$  are in decreasing order.
- Time complexity:  $T(n) = T(n-1) + O(n) = O(n^2)$ . The subproblems decrease **slowly in size** (linearly here, reducing by only one element each time); thus the sum of linear steps yields quadratic overall time.



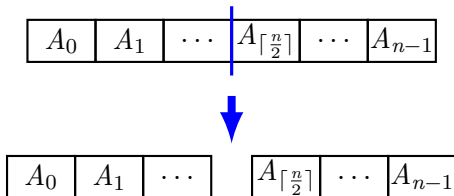
⋮



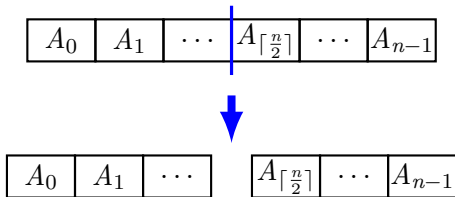
INSERTSORT: 28 ops

## Divide strategy 2 based on indices of elements

- **Divide the array  $A[0..n-1]$  into two arrays  $A[0..\lceil \frac{n}{2} \rceil - 1]$  and  $A[\lceil \frac{n}{2} \rceil..n-1]$ :** Both  $A[0..\lceil \frac{n}{2} \rceil - 1]$  and  $A[\lceil \frac{n}{2} \rceil..n-1]$  have same form to  $A[0..n-1]$  but smaller size; thus, sorting  $A[0..\lceil \frac{n}{2} \rceil - 1]$  and  $A[\lceil \frac{n}{2} \rceil..n-1]$  construct two subproblem of the original problem. The **DIVIDE AND CONQUER** technique might apply if we can sort  $A[0..n-1]$  using the sorted  $A[0..\lceil \frac{n}{2} \rceil - 1]$  and the sorted  $A[\lceil \frac{n}{2} \rceil..n-1]$ .



# MERGESORT algorithm [J. von Neumann, 1945, 1948]



MERGESORT( $A, l, r$ )

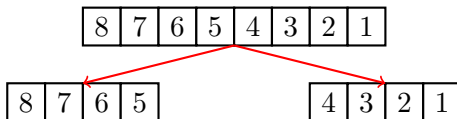
- 1: //Sort elements in  $A[l..r]$
- 2: **if**  $l < r$  **then**
- 3:      $m = (l + r)/2$ ; //  $m$  denotes the middle point
- 4:     MERGESORT( $A, l, m$ );
- 5:     MERGESORT( $A, m + 1, r$ );
- 6:     MERGE( $A, l, m, r$ ); //Combining the sorted arrays
- 7: **end if**

- Sort the entire array: MERGESORT( $A, 0, n - 1$ )

# An example

8	7	6	5	4	3	2	1
---	---	---	---	---	---	---	---

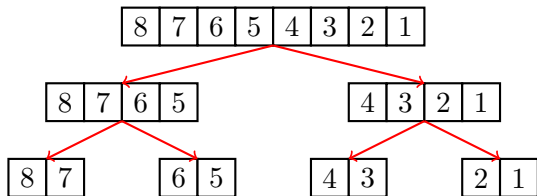
# An example



Divide

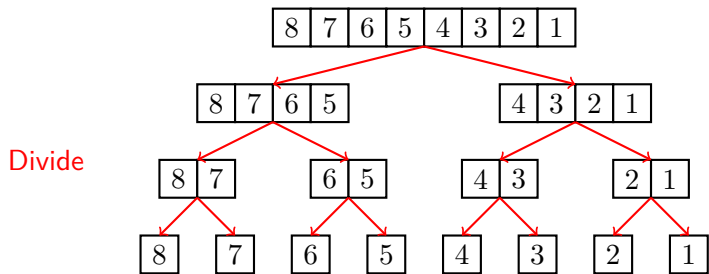
# An example

Divide

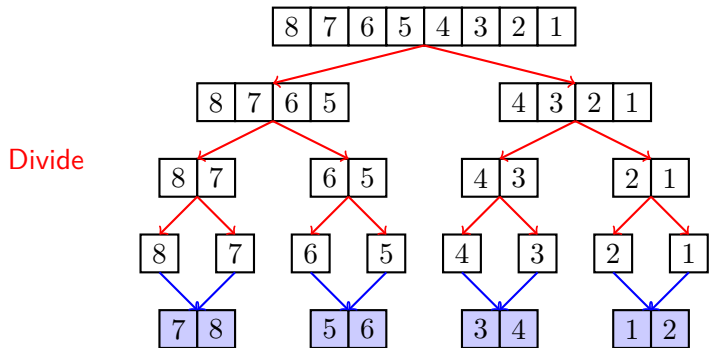




# An example

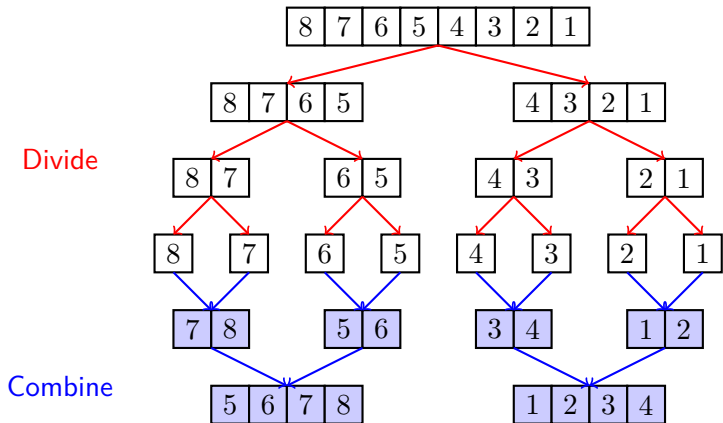


# An example

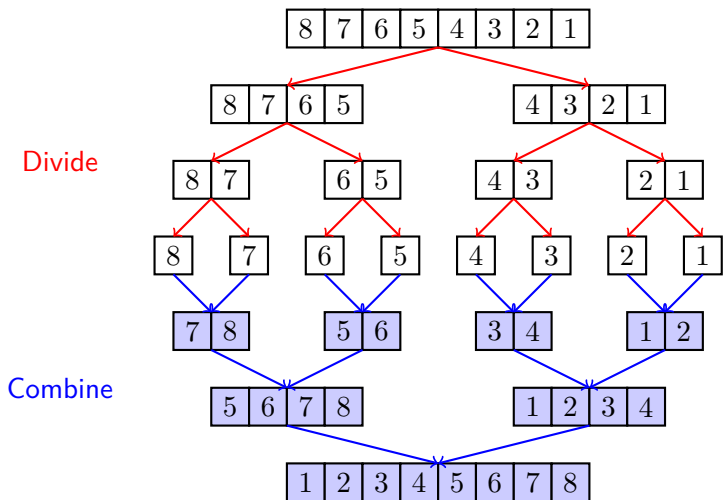


Combine

# An example



# An example

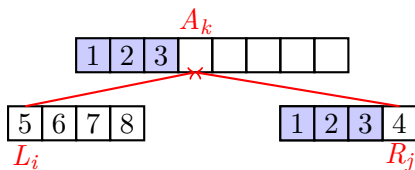


# MERGESORT algorithm: how to combine?

MERGE ( $A, l, m, r$ )

```
1: //Merge  $A[l..m]$  (denoted as  $L$ ) and  $A[m + 1..r]$  (denoted as  $R$ ).
2:  $i = 0; j = 0;$ 
3: for  $k = l$  to  $r$  do
4:   if  $L[i] < R[j]$  then
5:      $A[k] = L[i];$ 
6:      $i++;$ 
7:   if all elements in  $L$  have been copied then
8:     Copy the remainder elements from  $R$  into  $A$ ;
9:     break;
10:  end if
11: else
12:    $A[k] = R[j];$ 
13:    $j++;$ 
14:   if all elements in  $R$  have been copied then
15:     Copy the remainder elements from  $L$  into  $A$ ;
16:     break;
17:   end if
18: end if
19: end for
```

# MERGE algorithm



(see a demo)

## Correctness of MERGESORT algorithm

# Correctness of **Merge** procedure: **loop-invariant** technique [R. W. Floyd, 1967]

**Loop invariant:** (similar to **mathematical induction** proof technique)

- 1 At the start of each iteration of the **for** loop,  $A[l..k-1]$  contains the  $k-l$  smallest elements of  $L[1..n_1+1]$  and  $R[1..n_2+1]$ , in sorted order.
- 2  $L[i]$  and  $R[j]$  are the smallest elements of their array that have not been copied to  $A$ .

## Proof.

- Initialization:  $k = l$ . Loop invariant holds since  $A[l..k-1]$  is empty.
- Maintenance: Suppose  $L[i] < R[j]$ , and  $A[l..k-1]$  holds the  $k-l$  smallest elements. After copying  $L[i]$  into  $A[k]$ ,  $A[l..k]$  will hold the  $k-l+1$  smallest elements.





## Correctness of **Merge** procedure: **loop-invariant** technique [R. W. Floyd, 1967]

- Since the loop invariant holds initially, and is maintained during the **for** loop, thus it should hold when the algorithm terminates.
- Termination: At termination,  $k = r + 1$ . By loop invariant,  $A[l..k - 1]$ , i.e.  $A[l..r]$  must contain  $r - l + 1$  smallest elements, in sorted order.

## Time-complexity of MERGESORT algorithm

# Time-complexity of MERGE algorithm

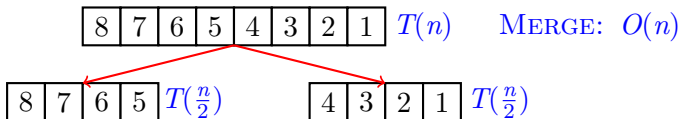
MERGE ( $A, l, m, r$ )

```
1: //Merge  $A[l..m]$  (denoted as  $L$ ) and  $A[m + 1..r]$  (denoted as  $R$ ).
2:  $i = 0; j = 0;$ 
3: for  $k = l$  to  $r$  do
4:   if  $L[i] < R[j]$  then
5:      $A[k] = L[i];$ 
6:      $i++;$ 
7:   if all elements in  $L$  have been copied then
8:     Copy the remainder elements from  $R$  into  $A$ ;
9:     break;
10:  end if
11: else
12:    $A[k] = R[j];$ 
13:    $j++;$ 
14:   if all elements in  $R$  have been copied then
15:     Copy the remainder elements from  $L$  into  $A$ ;
16:     break;
17:   end if
18: end if
19: end for
```

Time complexity:  $O(n)$ .

# Time-complexity of MERGESORT algorithm

- Let  $T(n)$  denote the running time of MERGESORT on an array of size  $n$ . As comparison of elements dominates the algorithm, we use the number of comparisons as  $T(n)$ .



- We have the following recursion:

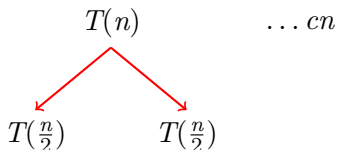
$$T(n) = \begin{cases} 1 & \text{if } n \leq 2 \\ T(\frac{n}{2}) + T(\frac{n}{2}) + O(n) & \text{otherwise} \end{cases} \quad (1)$$

- Note that the subproblems decrease **exponentially in size**, which is much faster than the linearly decrease in INSERTSORT.

- Ways to analyse a recursion:
  - 1 **Unrolling the recurrence:** unrolling a few levels to find a pattern, and then sum over all levels;
  - 2 **Guess and substitution:** guess the solution, substitute it into the recurrence relation, and check whether it works.
  - 3 **Master theorem**

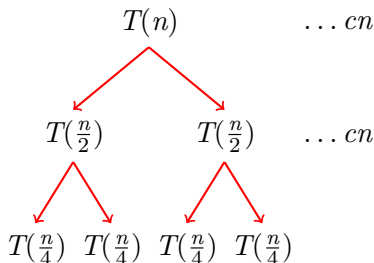
# Analysis technique 1: Unrolling the recurrence

- We have  $T(n) = 2T(\frac{n}{2}) + O(n) \leq 2T(\frac{n}{2}) + cn$  for a constant  $c$ . Let unrolling a few levels to find a pattern, and then sum over all levels.



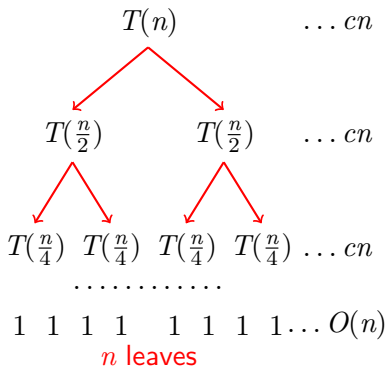
# Analysis technique 1: Unrolling the recurrence

- We have  $T(n) = 2T(\frac{n}{2}) + O(n) \leq 2T(\frac{n}{2}) + cn$  for a constant  $c$ . Let unrolling a few levels to find a pattern, and then sum over all levels.



# Analysis technique 1: Unrolling the recurrence

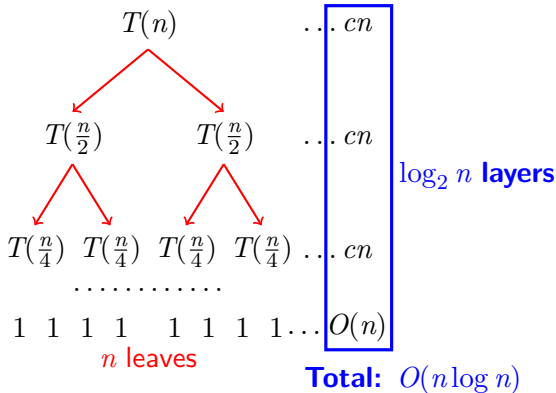
- We have  $T(n) = 2T(\frac{n}{2}) + O(n) \leq 2T(\frac{n}{2}) + cn$  for a constant  $c$ . Let unrolling a few levels to find a pattern, and then sum over all levels.





# Analysis technique 1: Unrolling the recurrence

- We have  $T(n) = 2T(\frac{n}{2}) + O(n) \leq 2T(\frac{n}{2}) + cn$  for a constant  $c$ . Let unrolling a few levels to find a pattern, and then sum over all levels.



## Analysis technique 2: Guess and substitution

- Guess and substitution: guess a solution, substitute it into the recurrence relation, and justify that it works.
- Guess:  $T(n) \leq cn \log_2 n$ .
- Verification:
  - Case  $n = 2$ :  $T(2) = 1 \leq cn \log_2 n$ ;
  - Case  $n > 2$ : Suppose  $T(m) \leq cm \log_2 m$  holds for all  $m \leq n$ .  
We have

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + cn \\&\leq 2c\frac{n}{2} \log_2\left(\frac{n}{2}\right) + cn \\&= 2c\frac{n}{2} \log_2 n - 2c\frac{n}{2} + cn \\&= cn \log_2 n\end{aligned}$$

## Analysis technique 2: a weaker version

- Guess and substitution: one guesses the overall form of the solution without pinning down the constants and parameters.
- A weaker guess:  $T(n) = O(n \log n)$ . Rewritten as  $T(n) \leq kn \log_b n$ , where  $k, b$  **will be determined later**.

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2k\frac{n}{2} \log_b\left(\frac{n}{2}\right) + cn \quad (\text{set } b = 2 \text{ for simplification}) \\ &= 2k\frac{n}{2} \log_2 n - 2k\frac{n}{2} + cn \\ &= kn \log_2 n - kn + cn \quad (\text{set } k = c \text{ for simplification}) \\ &= cn \log_2 n \end{aligned}$$

## Theorem

Let  $T(n)$  be defined by  $T(n) = aT(\frac{n}{b}) + O(n^d)$  for  $a > 1$ ,  $b > 1$  and  $d > 0$ , then  $T(n)$  can be bounded by:

- 1 If  $d < \log_b a$ , then  $T(n) = O(n^{\log_b a})$ ;
- 2 If  $d = \log_b a$ , then  $T(n) = O(n^{\log_b a} \log n)$ ;
- 3 If  $d > \log_b a$ , then  $T(n) = O(n^d)$ .

- Intuition: the ratio of cost between neighbouring layers is  $\frac{a}{b^d}$ .

Proof.

$$\begin{aligned}
 T(n) &= aT\left(\frac{n}{b}\right) + O(n^d) \\
 &\leq aT\left(\frac{n}{b}\right) + cn^d \\
 &\leq a\left(aT\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^d\right) + cn^d \\
 &\leq \dots\dots\dots \\
 &\leq cn^d\left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \dots + \left(\frac{a}{b^d}\right)^{\log_b n - 1}\right) + a^{\log_b n} \\
 &= \begin{cases} O(n^{\log_b a}) & \text{if } d < \log_b a \\ O(n^{\log_b a} \log n) & \text{if } d = \log_b a \\ O(n^d) & \text{if } d > \log_b a \end{cases}
 \end{aligned}$$

Here  $c > 0$  represents a constant. □

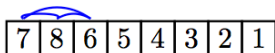
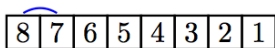
- Example 1:  $T(n) = 3T(\frac{n}{2}) + O(n)$

$$T(n) = O(n^{\log_2 3}) = O(n^{1.585})$$

- Example 2:  $T(n) = 2T(\frac{n}{2}) + O(n^2)$

$$T(n) = O(n^2)$$

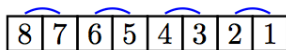
Question: from  $O(n^2)$  to  $O(n \log n)$ , what did we save?



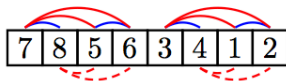
⋮



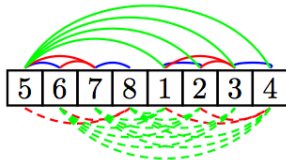
INSERTSORT: 28 ops



MERGESORT step 1: 4 ops



MERGESORT step 2: 4 ops, save: 4



MERGESORT step 3: 4 ops, save: 12

COUNTINGINVERSION: to count inversions in an **array** of  $n$  integers



# COUNTINGINVERSION problem

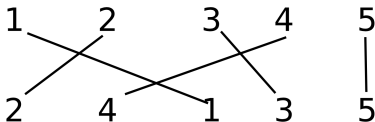
Practical problems:

- 1 To identify two users with similar preference, i.e. ranking books, movies, etc.

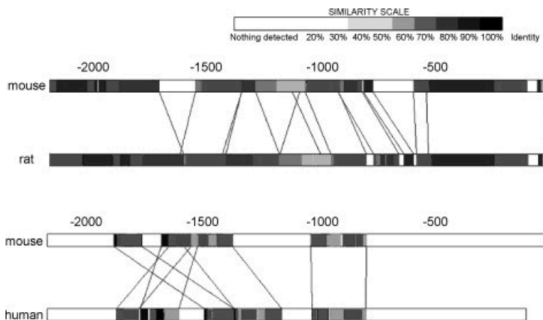
## COUNTINGINVERSION problem

**INPUT:** An array  $A[0..n-1]$  with  $n$  distinct numbers;

**OUTPUT:** the number of **inversions**. A pair of indices  $i$  and  $j$  constitutes an inversion if  $i < j$  but  $A[i] > A[j]$ .



# Application 1: Genome comparison



**Figure 1:** Sequence comparison of the 5' flanking regions of mouse, rat and human ER $\beta$ .

Reference: In vivo function of the 5' flanking region of mouse estrogen receptor  $\beta$  gene, The Journal of Steroid Biochemistry and Molecular Biology Volume 105, Issues 1-5, June-July 2007, pages 57-62.

## Application 2: A measure of bivariate association

- Motivation: how to measure the association between two genes when given expression levels across  $n$  time points?
- Existing measures:
  - Linear relationship: Pearson's CC (most widely used, but sensitive to outliers)
  - Monotonic relationship: Spearman, Kendall's correlation
  - General statistical dependence: Renyi correlation, mutual information, maximal information coefficient
- A novel measure:

$$W_1 = \sum_{i=1}^{n-k+1} (I_i^+ + I_i^-)$$

Here,  $I_i^+$  is 1 if  $X_{[i,..,i+k-1]}$  and  $Y_{[i,..,i+k-1]}$  has the same order and 0 otherwise, while  $I_i^-$  is 1 if  $X_{[i,..,i+k-1]}$  and  $-Y_{[i,..,i+k-1]}$  has the same order and 0 otherwise.

- Advantage: the association may exist across a subset of samples. For example,

$X$ : 1 3 4 2 5

$Y$ : 1 4 5 2 3

$W_1 = 2$  when  $k = 3$ . Much better than Pearson CC, et al.

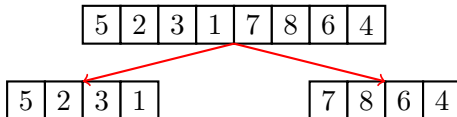
# COUNTINGINVERSION problem

- Solution: index pairs. The possible solution space has a size of  $O(n^2)$ .
- Brute-force:  $O(n^2)$  (Examining all index pairs  $(i, j)$ ).
- Can we design a better algorithm?

# COUNTINGINVERSION problem

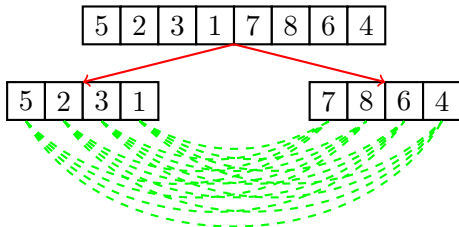
- DIVIDE AND CONQUER technique:

- 1 **Divide:** Divide  $A$  into two arrays  $A[0..\lceil \frac{n}{2} \rceil - 1]$  and  $A[\lceil \frac{n}{2} \rceil .. n - 1]$ ; thus counting inversions within  $A[0..\lceil \frac{n}{2} \rceil - 1]$  and  $A[\lceil \frac{n}{2} \rceil .. n - 1]$  constitutes two subproblems.
- 2 **Conquer:** Counting inversions within each half by calling COUNTINGINVERSION itself.



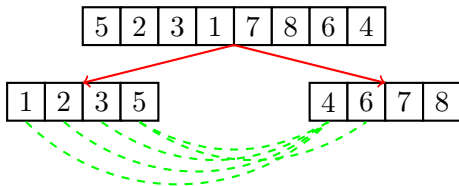
# Combine strategy 1

- **Combine:** How to count the inversions  $(i, j)$  with  $A[i]$  and  $A[j]$  from different halves?
- If the two halves  $A[0..\lceil \frac{n}{2} \rceil - 1]$  and  $A[\lceil \frac{n}{2} \rceil..n - 1]$  have no special structure, we have to examine all possible index pairs  $i \in [0, \lceil \frac{n}{2} \rceil - 1]$ ,  $j \in [\lceil \frac{n}{2} \rceil, n - 1]$  to count such inversions, which costs  $\frac{n^2}{4}$  time.
- Thus,  $T(n) = 2T(\frac{n}{2}) + \frac{n^2}{4} = O(n^2)$ .



## Combine strategy 2

- **Combine:** How to count the inversions  $(i, j)$  with  $A[i]$  and  $A[j]$  from different halves?
- If the two halves are unstructured, it would be inefficient to count inversions. Thus, we need to introduce some structures into  $A[0..\lceil \frac{n}{2} \rceil - 1]$  and  $A[\lceil \frac{n}{2} \rceil .. n - 1]$ .
- Note that it is relatively easy to count such inversions if **elements in both halves are in increasing order.**



(See a demo)

# SORT-AND-COUNT algorithm

$\text{SORT-AND-COUNT}(A)$

- 1: Divide  $A$  into two sub-sequences  $L$  and  $R$ ;
- 2:  $(RC_L, L) = \text{SORT-AND-COUNT}(L)$ ;
- 3:  $(RC_R, R) = \text{SORT-AND-COUNT}(R)$ ;
- 4:  $(C, A) = \text{MERGE-AND-COUNT}(L, R)$ ;
- 5: **return**  $(RC = RC_L + RC_R + C, A)$ ;

Time complexity:  $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$ .



# MERGE-AND-COUNT algorithm

MERGE-AND-COUNT ( $L, R$ )

```
1:  $RC = 0; i = 0; j = 0;$ 
2: for  $k = 0$  to  $\|L\| + \|R\| - 1$  do
3:   if  $L[i] > R[j]$  then
4:      $A[k] = R[j];$ 
5:      $j++;$ 
6:      $RC += (\|L\| - i);$ 
7:     if all elements in  $R$  have been copied then
8:       Copy the remainder elements from  $L$  into  $A$ ;
9:       break;
10:    end if
11:  else
12:     $A[k] = L[i];$ 
13:     $i++;$ 
14:    if all elements in  $L$  have been copied then
15:      Copy the remainder elements from  $R$  into  $A$ ;
16:      break;
17:    end if
18:  end if
19: end for
20: return  $(RC, A);$ 
```

QUICKSORT algorithm: divide based on **value of elements**



Figure 2: Sir Charles Antony Richard Hoare, 2011

# QUICKSORT: divide based on value of a randomly-selected element

QUICKSORT( $A$ )

```
1:  $S_- = \{\}; S_+ = \{\};$   
2: Choose a pivot  $A[j]$  uniformly at random;  
3: for  $i = 0$  to  $n - 1$  do  
4:   Put  $A[i]$  in  $S_-$  if  $A[i] < A[j]$ ;  
5:   Put  $A[i]$  in  $S_+$  if  $A[i] \geq A[j]$ ;  
6: end for  
7: QUICKSORT( $S_+$ );  
8: QUICKSORT( $S_-$ );  
9: Output  $S_-$ , then  $A[j]$ , then  $S_+$ ;
```

- The randomization operation makes this algorithm **simple** (relative to MERGESORT algorithm) but **efficient**.
- However, the randomization also makes it difficult to analyze time-complexity: When dividing based on indices, it is easy to divide into two halves with equal size; in contrast, we divide based on value of a randomly-selected pivot and thus we cannot guarantee that each sub-problem has exactly  $\frac{n}{2}$  elements.

# Various cases of the execution of QUICKSORT algorithm

- **Worst case:** selecting the smallest/largest element at each iteration. The subproblems decrease **linearly** in size.

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

- **Best case:** select the median exactly at each iteration. The subproblems decrease **exponentially** in size.

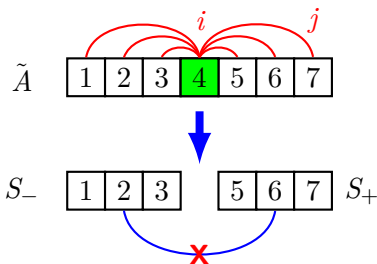
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$

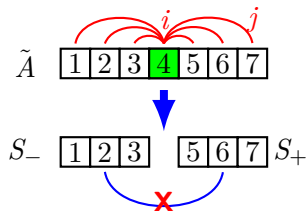
- **Most cases:** instead of selecting the median exactly, we can select a **nearly-central pivot** with high probability. We claim that the expected running time is still

$$T(n) = O(n \log n).$$

# Analysis

- Let  $X$  denote the number of comparisons performed in line 4 and 5. After expanding all recursive calls, it is obvious that the running time of QUICKSORT is  $O(n + X)$ . Our objective is to calculate  $E[X]$ .
- For simplicity, we represent each element using its index in the sorted array, denoted as  $\tilde{A}$ . We have two key observations:
- Observation 1:** Any two elements  $\tilde{A}[i]$  and  $\tilde{A}[j]$  are compared at most once.





- Define index variable

$$X_{ij} = \begin{cases} 1 & \text{if } \tilde{A}[i] \text{ is compared with } \tilde{A}[j] \\ 0 & \text{otherwise} \end{cases}$$

- Thus  $X = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}$ .

$$\begin{aligned} E[X] &= E\left[\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}\right] \\ &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} E[X_{ij}] \\ &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[j]) \end{aligned}$$

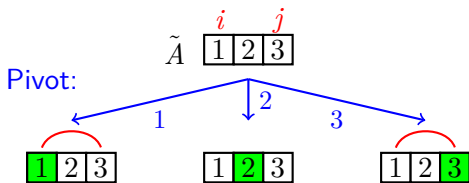
- **Observation 2:**  $\tilde{A}[i]$  and  $\tilde{A}[j]$  are compared iff either  $\tilde{A}[i]$  or  $\tilde{A}[j]$  is selected as pivot when processing elements containing  $\tilde{A}[i..j]$ .
- We claim  $\Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[j]) = \frac{2}{j-i+1}$ . (Why?)
- Then

$$\begin{aligned}
 E[X] &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[j]) \\
 &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1} \\
 &= \sum_{i=0}^{n-1} \sum_{k=1}^{n-i-1} \frac{2}{k+1} \\
 &\leq \sum_{i=0}^{n-1} \sum_{k=1}^{n-1} \frac{2}{k+1} \\
 &= O(n \log n)
 \end{aligned}$$

- Here  $k$  is defined as  $k = j - i$ .

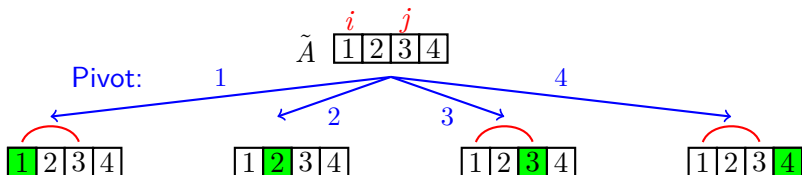


Why  $\Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[j]) = \frac{2}{j-i+1}$ ?



- Let's examine a simple example first: For an array with only 3 elements, each element will be selected as pivot with equal probability  $\frac{1}{3}$ .
- In two out of the three cases,  $\tilde{A}[i]$  is compared with  $\tilde{A}[j]$ . Hence,  $\Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[j]) = \frac{2}{3}$

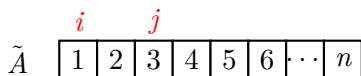
# Why $\Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[j]) = \frac{2}{j-i+1}$ ? cont'd



- Let's consider a larger array with 4 elements.
- Each element will be selected as pivot with equal probability  $\frac{1}{4}$ : the selection of  $\tilde{A}[i]$  or  $\tilde{A}[j]$  as pivot will lead to an immediate comparison of  $\tilde{A}[i]$  and  $\tilde{A}[j]$ . In contrast, the selection of  $\tilde{A}[3]$  as pivot produces a smaller problem, where  $\tilde{A}[i]$  will be compared with  $\tilde{A}[j]$  with probability  $\frac{2}{3}$  by induction. Hence,

$$\begin{aligned}\Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[j]) &= \frac{1}{4} + 0 + \frac{1}{4} + \frac{1}{4} \times \frac{2}{3} \\ &= \frac{3}{4} \times \frac{2}{3} + \frac{1}{4} \times \frac{2}{3} \\ &= \frac{2}{3}\end{aligned}$$

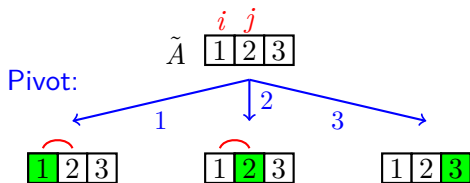
Why  $\Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[j]) = \frac{2}{j-i+1}$ ? cont'd



- Now let's extend these observations to general case that  $A$  has  $n$  elements. By induction over the size of  $A$ , we can calculate the probability as:

$$\begin{aligned}\Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[j]) &= \frac{1}{n} + \frac{1}{n} + \frac{n-(j-i+1)}{n} \times \frac{2}{j-i+1} \\ &= \left( \frac{j-i+1}{n} + \frac{n-(j-i+1)}{n} \right) \times \frac{2}{j-i+1} \\ &= \frac{2}{j-i+1}\end{aligned}$$

A special case: neighbors should definitely compare with each other



- Let's examine a simple example first: For an array with only 3 elements, each element will be selected as pivot with equal probability  $\frac{1}{3}$ .
- $\Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[i+1]) = \frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1$

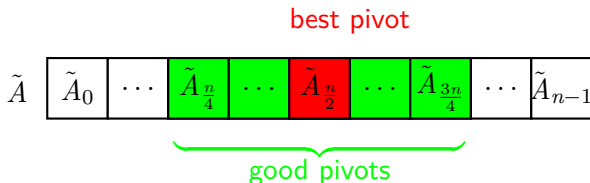
# MODIFIED QUICKSORT: easier to analyze

MODIFIEDQUICKSORT( $A$ )

```
1: while TRUE do
2:   Choose a pivot  $A[j]$  uniformly at random;
3:    $S_- = \{\}$ ;  $S_+ = \{\}$ ;
4:   for  $i = 0$  to  $n - 1$  do
5:     Put  $A[i]$  in  $S_-$  if  $A[i] < A[j]$ ;
6:     Put  $A[i]$  in  $S_+$  if  $A[i] \geq A[j]$ ;
7:   end for
8:   if  $\|S_+\| \geq \frac{n}{4}$  and  $\|S_-\| \geq \frac{n}{4}$  then
9:     break; // A fixed proportion of elements fall both below and
           above the pivot;
10:  end if
11: end while
12: MODIFIEDQUICKSORT( $S_+$ );
13: MODIFIEDQUICKSORT( $S_-$ );
14: Output  $S_-$ , then  $A[j]$ , and finally  $S_+$ ;
```

- MODIFIEDQUICKSORT works when all items are distinct.  
However, it is slower than the original version since it doesn't run when the pivot is "off-center".

# MODIFIED QUICKSORT: analysis



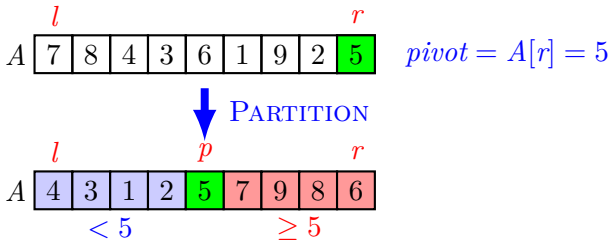
- It is easy to obtain a **nearly central pivot**:
  - $\Pr(\text{select the **centroid** as pivot}) = \frac{1}{n}$
  - $\Pr(\text{select a **nearly central element** as pivot}) = \frac{1}{2}$
  - Thus  $E(\#\text{WHILE}) = 2$ , i.e., the expected time of finding a nearly central pivot is  $2n$ .
- **Nearly central pivot** is good:
  - An element is a **good pivot** if a fixed proportion of elements fall both below and above it, thus making subproblems decrease **exponentially** in size.
  - Specifically, the recursion tree has a depth of  $O(\log_{\frac{4}{3}} n)$ , and  $O(n)$  work is needed at each level, hence  $T(n) = O(n \log_{\frac{4}{3}} n)$ .

# Lomuto's in-place algorithm

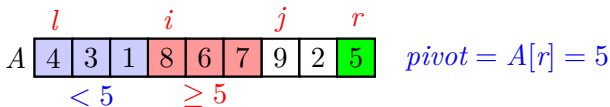
QUICKSORT( $A, l, r$ )

- 1: **if**  $l < r$  **then**
- 2:    $p = \text{PARTITION}(A, l, r)$  //Use  $A[r]$  as pivot;
- 3:   QUICKSORT( $A, l, p - 1$ );
- 4:   QUICKSORT( $A, p + 1, r$ );
- 5: **end if**

- Sort the entire array: QUICKSORT( $A, 0, n - 1$ ).



# Lomuto's PARTITION procedure



- Basic idea: Swap the elements (in  $A[l..j-1]$ ) to make elements in  $A[l..i-1] < pivot$  and elements in  $A[i..j-1] \geq pivot$ .

PARTITION( $A, l, r$ )

- 1:  $pivot = A[r]; i = l;$
- 2: **for**  $j = l$  to  $r - 1$  **do**
- 3:     **if**  $A[j] < pivot$  **then**
- 4:         Swap  $A[i]$  with  $A[j];$
- 5:          $i++;$
- 6:     **end if**
- 7: **end for**
- 8: Swap  $A[i]$  with  $A[r];$  //Put pivot in its correct position
- 9: **return**  $i;$

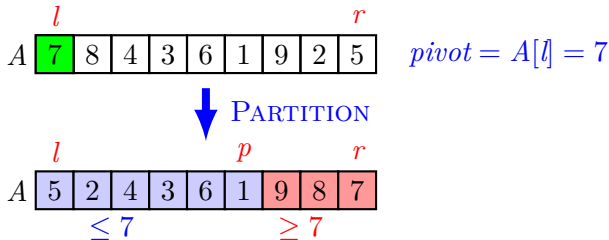


# Hoare's in-place algorithm [1961]

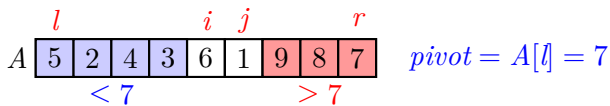
QUICKSORT( $A, l, r$ )

- 1: **if**  $l < r$  **then**
- 2:    $p = \text{PARTITION}(A, l, r)$  //Use  $A[l]$  as pivot;
- 3:   QUICKSORT( $A, l, p$ ); //Reason:  $A[p]$  might not be at its correct position
- 4:   QUICKSORT( $A, p + 1, r$ );
- 5: **end if**

- Sort the entire array: QUICKSORT( $A, 0, n - 1$ ).



# Hoares' PARTITION procedure



- Basic idea: Keep the elements in  $A[l..i-1] \leq pivot$  and the elements in  $A[j+1..r] \geq pivot$ .

PARTITION( $A, l, r$ )

```
1:  $i = l - 1; j = r + 1; pivot = A[l];$ 
2: while TRUE do
3:   repeat
4:      $j = j - 1$ ; //From right to left, find the first element  $\leq pivot$ 
5:   until  $A[j] \leq pivot$  or  $j == l$ ;
6:   repeat
7:      $i = i + 1$ ; //From left to right, find the first element  $\geq pivot$ 
8:   until  $A[i] \geq pivot$  or  $i == r$ ;
9:   if  $j \leq i$  then
10:    return  $j$ ;
11:  end if
12:  Swap  $A[i]$  with  $A[j]$ ;
13: end while
```

- Sort the entire array: QUICKSORT( $A, 0, n-1$ )

# Comparison with MERGESORT [Hoare, 1961]

NUMBER OF ITEMS	MERGE SORT	QUICKSORT
500	2 min 8 sec	1 min 21 sec
1,000	4 min 48 sec	3 min 8 sec
1,500	8 min 15 sec*	5 min 6 sec
2,000	11 min 0 sec*	6 min 47 sec

\* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

- Note: The preceding QUICKSORT algorithm works well for lists with **distinct elements** but exhibits poor performance when the input list contains many **repeated elements**. To solve this problem, an alternative PARTITION algorithm was proposed to divide the list into three parts: elements less than pivot, elements equal to pivot, and elements greater than pivot. Only the less-than and greater-than pivot partitions need to be recursively sorted.

## Extension: stability of sorting algorithm

- Stability: Stable sort algorithms sort equal elements in the same order that they appear in the input: if two items compare as equal (like the two 5 cards), then their relative order will be preserved, i.e. if one comes before the other in the input, it will come before the other in the output.
- Stability is important to preserve order over multiple sorts on the same data set.
- MERGESORT algorithm is stable while QUICKSORT and INTROSORT are unstable.

- Complexity attack: QUICKSORT has the expectation of running time of  $O(n \log n)$  but the worst-case time-complexity of  $O(n^2)$ . Thus, for elaborately-designed arrays, QUICKSORT runs very slowly.
- Improvement: D. R. Musser proposed INTROSORT: INTROSORT uses QUICKSORT when the iteration depth is less than  $O(n \log n)$  and uses HEAPSORT otherwise.

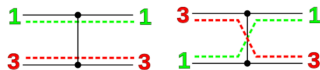
## Extension: sorting on dynamic data

- When the data changes gradually, the goal of a sorting algorithm is to sort the data at each time step, under the constraint that it only has limited access to the data each time.
- As the data is constantly changing and the algorithm might be unaware of these changes, it cannot be expected to always output the exact right solution; we are interested in algorithms that guarantee to output an approximate solution.
- In 2011, Eli Upfal et al. proposed an algorithm to sort dynamic data.
- In 2017, Liu and Huang proposed an efficient algorithm to determine top  $k$  elements of dynamic data.

AlphaDev: faster sorting algorithms discovered using deep reinforcement learning

# Sort an array with three to five numbers

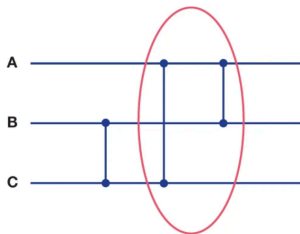
- Objective: improving sorting algorithms for shorter sequences of three to five elements. These algorithms are among the most widely used because they are often called many times as a part of larger sorting functions.
- Sorting networks:
  - Wire: each wire carries a value running from left to right
  - Comparator: each comparator connects two lines and swaps two values if and only if the top line's value is greater than or equal to the bottom line's value





# Sorting three numbers: an implementation designed by human

**a**



**b** Original

```
Memory[0] = A
Memory[1] = B
Memory[2] = C
```

```
mov Memory[0] P // P = A
mov Memory[1] Q // Q = B
mov Memory[2] R // R = C
```

```
mov R S
cmp P R
cmovg P R // R = max(A, C)
cmovl P S // S = min(A, C)
mov S P // P = min(A, C)
cmp S Q
cmovg Q P // P = min(A, B, C)
cmovg S Q // Q = max(min(A, C), B)
```

```
mov P Memory[0] // = min(A, B, C)
mov Q Memory[1] // = max(min(A, C), B)
mov R Memory[2] // = max(A, C)
```

- Question: can we find a more efficient implementation of the circled operations?

# The implementation discovered by AlphaDev

## Original

```
Memory[0] = A
Memory[1] = B
Memory[2] = C

mov Memory[0] P // P = A
mov Memory[1] Q // Q = B
mov Memory[2] R // R = C

mov R S
cmp P R
cmovg P R // R = max(A, C)
cmovl P S // S = min(A, C)
mov S P // P = min(A, C)
cmp S Q
cmovg Q P // P = min(A, B, C)
cmovg S Q // Q = max(min(A, C), B)

mov P Memory[0] // = min(A, B, C)
mov Q Memory[1] // = max(min(A, C), B)
mov R Memory[2] // = max(A, C)
```

## AlphaDev

```
Memory[0] = A
Memory[1] = B
Memory[2] = C

mov Memory[0] P // P = A
mov Memory[1] Q // Q = B
mov Memory[2] R // R = C

mov R S
cmp P R
cmovg P R // R = max(A, C)
cmovl P S // S = min(A, C)

cmp S Q
cmovg Q P // P = min(A, B)
cmovg S Q // Q = max(min(A, C), B)

mov P Memory[0] // = min(A, B)
mov Q Memory[1] // = max(min(A, C), B)
mov R Memory[2] // = max(A, C)
```

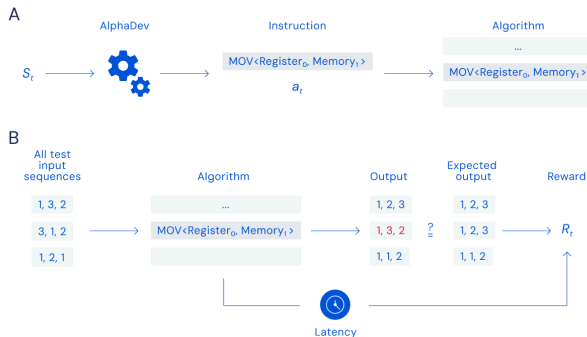
- AlphaDev: reduce 1 instruction

# Performance of the two implementations

<b>(a) Algorithm</b>	<b>AlphaDev</b>	<b>Human benchmarks</b>
	<b>Length</b>	<b>Length</b>
Sort 3	17	18
Sort 4	28	28
Sort 5	42	46
VarSort3	21	33
VarSort4	37	66
VarSort5	63	115
VarInt	27	31

<b>(b) Algorithm</b>	<b>AlphaDev</b>	<b>Human benchmarks</b>
	<b>Latency <math>\pm</math> (lower, upper)</b>	<b>Latency <math>\pm</math> (lower, upper)</b>
VarSort3	236,498 $\pm$ (235,898, 236,887)	246,040 $\pm$ (245,331, 246,470)
VarSort4	279,339 $\pm$ (278,791, 279,851)	294,963 $\pm$ (294,514, 295,618)
VarSort5	312,079 $\pm$ (311,515, 312,787)	331,198 $\pm$ (330,717, 331,850)
VarInt	97,184 $\pm$ (96,885, 97,847)	295,358 $\pm$ (293,923, 296,297)
Competitive	75,973 $\pm$ (75,420, 76,638)	86,056 $\pm$ (85,630, 86,913)

# How does AlphaDev discover new algorithms?



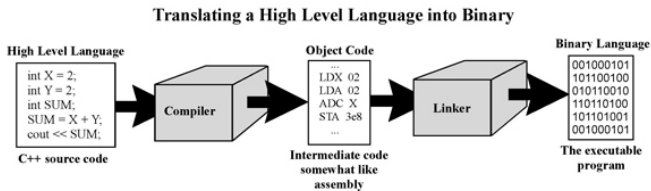
- Assembly game:

- State of the system:  $S_t = \langle P_t, Z_t \rangle$ , where  $P_t$  denotes algorithm and  $Z_t$  denotes CPU states after executing  $P_t$
- Action: selecting an instruction to add to the algorithm
- Reward: correctness and latency on test input sequences — for sort3, this corresponds to all combinations of sequences of three elements

# Why does AlphaDev use assembly program?

- Two advantages:
  - Easy to modify: just adding an instruction (removing an instruction can be accomplished by inserting a NOP instruction)
  - Easy to represent state: the state of system (CPU here) can be represented using the configuration of memory and registers
- Two extensions:
  - More algorithms: after discovering faster sorting algorithms, DeepMind tested whether AlphaDev could generalise and improve a different computer science algorithm: hashing. For 9-16 bytes range of the hashing function, the algorithm that AlphaDev discovered was 30% faster
  - High-level language: optimise algorithms directly in high-level languages such as C++ which would be more useful for developers

# What is an assembler?



- Assembly code is converted into executable machine code by a utility program referred to as an assembler
- The term “assembler” is generally attributed to Wilkes, Wheeler and Gill in their 1951 book *The Preparation of Programs for an Electronic Digital Computer*, who, however, used the term to mean “a program that assembles another program consisting of several sections into a single program”

Figure: excerpted from

<https://www.cs.uah.edu/~coleman/CS121/ClassTopics/ProgrammingLanguage>  
with courtesy

SELECTION problem: to select the  $k$ -th smallest items in **an array**

**INPUT:**

An array  $A = [A_0, A_1, \dots, A_{n-1}]$ , and a number  $k < n$ ;

**OUTPUT:**

The  $k$ -th smallest item in general case (or the median of  $A$  as a special case).

- Things will be easy when  $k$  is very small, say  $k = 1, 2$ .  
However, identification of the median is not that easy.
- The  $k$ -th smallest element could be readily determined after sorting  $A$ , which takes  $O(n \log n)$  time.
- In contrast, when using DIVIDE AND CONQUER technique, it is possible to develop a faster algorithm, say the deterministic linear algorithm ( $16n$  comparisons) by Blum et al.



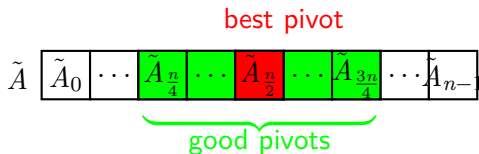
# Applying the general DIVIDE AND CONQUER paradigm

SELECT( $A, k$ )

```
1: Choose an element  $A_i$  from  $A$  as a pivot;  
2:  $S_+ = \{\}$ ;  $S_- = \{\}$ ;  
3: for all element  $A_j$  in  $A$  do  
4:   if  $A_j > A_i$  then  
5:      $S_+ = S_+ \cup \{A_j\}$ ;  
6:   else  
7:      $S_- = S_- \cup \{A_j\}$ ;  
8:   end if  
9: end for  
10: if  $|S_-| = k - 1$  then  
11:   return  $A_i$ ;  
12: else if  $|S_-| > k - 1$  then  
13:   return SELECT( $S_-, k$ );  
14: else  
15:   return SELECT( $S_+, k - |S_-| - 1$ );  
16: end if
```

Note: Unlike QUICKSORT, the SELECT algorithm needs to consider only one subproblem. The algorithm would be efficient if the subproblem size, i.e.,  $\|S_+\|$  or  $\|S_-\|$ , decreases exponentially as iteration proceeds.

# Question: How to choose a pivot?



- Worst choice: select the smallest/largest element as pivot at each iteration. The subproblems decrease **linearly** in size.

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

- Best choice: select the **exact median** at each iteration. The subproblems decrease **exponentially** in size.

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

- Good choice: select a **nearly-central element** such that a fixed proportion of elements fall both below and over it, i.e.,  $\|S_+\| \geq \epsilon n$ , and  $\|S_-\| \geq \epsilon n$  for a fixed  $\epsilon > 0$ , say  $\epsilon = \frac{1}{4}$ . In this case, the subproblems decrease **exponentially** in size, too.

$$\begin{aligned} T(n) &\leq T((1-\epsilon)n) + O(n) \\ &\leq cn + c(1-\epsilon)n + c(1-\epsilon)^2n + \dots \\ &= O(n) \end{aligned}$$

# How to efficiently get a **nearly-central** pivot?

- Selection of **nearly-central pivots** always leads to small subproblems, which will speed up the algorithm regardless of  $k$ . But how to obtain **nearly-central pivots**?
- We **estimate median of the whole set** through examining a **sample of the whole set**. The following samples have been tried:
  - ① Select a nearly-central pivot via **examining medians of groups**;
  - ② Select a nearly-central pivot via **randomly selecting an element**;
  - ③ Select a nearly-central pivot via **examining a random sample**.
- Note: In 1975, Sedgewick proposed a similar pivot-selecting strategy called **“median-of-three”** for QUICKSORT: selecting the median of the first, middle, and last elements as pivot. The “median-of-three” rule gives a good estimate of the best pivot.

Strategy 1: BFPRT algorithm uses median of medians as pivot

# Strategy 1: Median of medians [Blum et al, 1973]

	0	5	6	21	3	17	14	4	1	22	8
	2	9	11	25	16	19	31	20	36	29	18
Medians	7	10	13	26	27	32	34	35	38	42	44
	12	24	23	30	43	33	37	41	46	49	48
	15	51	28	40	45	53	39	47	50	54	52

SELECT( $A, k$ )

- 1: Line up elements in groups of 5 elements;
- 2: Find the median of each group; // Cost  $\frac{6}{5}n$  time
- 3: Find the median of medians (denoted as  $M$ ) through recursively running SELECT over the group medians; //  $T(\frac{n}{5})$  time
- 4: Use  $M$  as pivot to partition  $A$  into  $S_-$  and  $S_+$ ; //  $O(n)$  time
- 5: **if**  $|S_-| = k - 1$  **then**
- 6:     **return**  $M$ ;
- 7: **else if**  $|S_-| > k - 1$  **then**
- 8:     **return** SELECT( $S_-, k$ ); // at most  $T(\frac{7}{10}n)$  time
- 9: **else**
- 10:     **return** SELECT( $S_+, k - |S_-| - 1$ ); // at most  $T(\frac{7}{10}n)$  time
- 11: **end if**

$A = [51, 10, 24, 9, 5, 40, 30, 26, 25, 21, 15, 12, 7, 2, 0, 13, 11, 6, 28,$   
 $23, 43, 27, 45, 16, 3, 34, 37, 39, 31, 14, 32, 33, 53, 19, 17, 4, 35,$   
 $41, 47, 20, 8, 44, 18, 48, 52, 1, 36, 38, 50, 46, 22, 42, 54, 49, 29],$

G3	G1	G4	G2	G5	G7	G6	G8	G10	G11	G9
0	5	6	21	3	17	14	4	1	22	8
2	9	11	25	16	19	31	20	36	29	18
7	10	13	26	27	32	34	35	38	42	44
12	24	23	30	43	33	37	41	46	49	48
15	51	28	40	45	53	39	47	50	54	52

	0	5	6	21	3	17	14	4	1	22	8
	2	9	11	25	16	19	31	20	36	29	18
Medians	7	10	13	26	27	32	34	35	38	42	44
	12	24	23	30	43	33	37	41	46	49	48
	15	51	28	40	45	53	39	47	50	54	52

- Basic idea: Median of medians  $M = 32$  is a perfect approximate median as at least  $\frac{3n}{10}$  elements are larger (in red), and at least  $\frac{3n}{10}$  elements are smaller than  $M$  (in blue). Thus, at least  $\frac{3n}{10}$  elements will not appear in  $S_+$  and  $S_-$ .
- Running time:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n) = O(n).$$

Actually it takes at most  $24n$  comparisons.

# BFPRT algorithm: an in-place implementation

```
SELECT( $A, l, r, k$ )
1: while TRUE do
2:   if  $l == r$  then
3:     return  $l$ ;
4:   end if
5:    $p = \text{PIVOT}(A, l, r)$ ; //Use median of medians  $A[p]$  as pivot ;
6:    $pos = \text{PARTITION}(A, l, r, p)$ ; //  $pos$  represents the final
   position of the pivot,  $A[l..pos - 1]$  deposit  $S_-$  and
    $A[pos + 1..r]$  deposit  $S_+$ ;
7:   if  $(k - 1) == pos$  then
8:     return  $k - 1$ ;
9:   else if  $(k - 1) < pos$  then
10:     $r = pos - 1$ ;
11:   else
12:     $l = pos + 1$ ;
13:   end if
14: end while
```



## PIVOT( $A, l, r$ ): get median of medians

PIVOT( $A, l, r$ )

```
1: if  $(r - l) < 5$  then  
2:   return PARTITION5( $A, l, r$ ); //Get median for 5 or less  
   elements;  
3: end if  
4: for  $i = l$  to  $r$  by 5 do  
5:    $right = i + 4$ ;  
6:   if  $right > r$  then  
7:      $right = r$ ;  
8:   end if  
9:    $m = \text{PARTITION5}(A, i, right)$ ; //Get median of a group;  
10:  Swap  $A[m]$  and  $A[l + \lfloor \frac{i-l}{5} \rfloor]$ ;  
11: end for  
12: return SELECT( $A, l, l + \lfloor \frac{r-l}{5} \rfloor, l + \frac{r-l}{10}$ );
```

## PARTITION( $A, l, r, p$ ): Partition $A$ into $S_-$ and $S_+$

PARTITION( $A, l, r, p$ )

- 1:  $pivot = A[p]$ ;
  - 2: Swap  $A[p]$  and  $A[r]$ ; //Move pivot to the right end;
  - 3:  $i = l$ ;
  - 4: **for**  $j = l$  to  $r - 1$  **do**
  - 5:     **if**  $A[j] < pivot$  **then**
  - 6:         Swap  $A[i]$  and  $A[j]$ ;
  - 7:          $i++$ ;
  - 8:     **end if**
  - 9: **end for**
  - 10: Swap  $A[r]$  and  $A[i]$ ;
  - 11: **return**  $i$ ;
- Basic idea: Swap  $A[p]$  and  $A[r]$  to move pivot to the right end first, and then execute the PARTITION function used by Lomuto's QUICKSORT algorithm.

# An example: Iteration #1 of SELECT( $A, 0, 15, 7$ )

8	1	15	10	4	3	2	9	7	12	5	16	14	6	13	11
---	---	----	----	---	---	---	---	---	----	---	----	----	---	----	----

↓ Find group medians

8	1	15	10	4	3	2	9	7	12	5	16	14	6	13	11
---	---	----	----	---	---	---	---	---	----	---	----	----	---	----	----

↓ Swap medians to end

8	7	13	11	4	3	2	9	1	12	5	16	14	6	15	10
---	---	----	----	---	---	---	---	---	----	---	----	----	---	----	----

↓ Find **pivot** using SELECT( $A, 0, 3, 2$ )

8	7	13	11	4	3	2	9	1	12	5	16	14	6	15	10
---	---	----	----	---	---	---	---	---	----	---	----	----	---	----	----

↓ PARTITION( $A, 0, 15, 3$ )

8	7	10	4	3	2	9	1	5	6	11	16	14	12	15	13
---	---	----	---	---	---	---	---	---	---	----	----	----	----	----	----

## Iteration #2: SELECT( $A, 0, 9, 7$ )

8	7	10	4	3	2	9	1	5	6	11	16	14	12	15	13
---	---	----	---	---	---	---	---	---	---	----	----	----	----	----	----

↓ Find group medians

8	7	10	4	3	2	9	1	5	6	11	16	14	12	15	13
---	---	----	---	---	---	---	---	---	---	----	----	----	----	----	----

↓ Swap medians to end

7	5	10	4	3	2	9	1	8	6	11	16	14	12	15	13
---	---	----	---	---	---	---	---	---	---	----	----	----	----	----	----

↓ Find pivot using SELECT( $A, 0, 1, 1$ )

7	5	10	4	3	2	9	1	8	6	11	16	14	12	15	13
---	---	----	---	---	---	---	---	---	---	----	----	----	----	----	----

↓ PARTITION( $A, 0, 9, 1$ )

4	3	2	1	5	10	9	7	8	6	11	16	14	12	15	13
---	---	---	---	---	----	---	---	---	---	----	----	----	----	----	----

### Iteration #3: SELECT( $A, 5, 9, 7$ )

4	3	2	1	5	10	9	7	8	6	11	16	14	12	15	13
---	---	---	---	---	----	---	---	---	---	----	----	----	----	----	----

↓ Find group medians

4	3	2	1	5	10	9	7	8	6	11	16	14	12	15	13
---	---	---	---	---	----	---	---	---	---	----	----	----	----	----	----

↓ Move medians to end

4	3	2	1	5	8	9	7	10	6	11	16	14	12	15	13
---	---	---	---	---	---	---	---	----	---	----	----	----	----	----	----

↓ Find **pivot** using SELECT( $A, 5, 5, 1$ )

4	3	2	1	5	8	9	7	10	6	11	16	14	12	15	13
---	---	---	---	---	---	---	---	----	---	----	----	----	----	----	----

↓ PARTITION( $A, 5, 9, 5$ )

4	3	2	1	5	6	7	8	10	9	11	16	14	12	15	13
---	---	---	---	---	---	---	---	----	---	----	----	----	----	----	----

**Return**  $A[6] = 7$

## Question: How about setting other group size?

- It is easy to prove  $T(n) = O(n)$  when setting group size as 7 or larger.

- However, when we setting group size as 3, we have:

$$T(n) \leq T(\frac{n}{3}) + T(\frac{2n}{3}) + O(n) = O(n \log n)$$

- Note that BFPRT algorithm always selects the median of medians as pivot regardless of the value of  $k$ . In 2017, Zeng et al. proposed to use fractile of medians rather than median of medians as pivot and selected appropriate fractile of medians according to  $k$ .

Strategy 2: QUICKSELECT algorithm randomly select an element as pivot

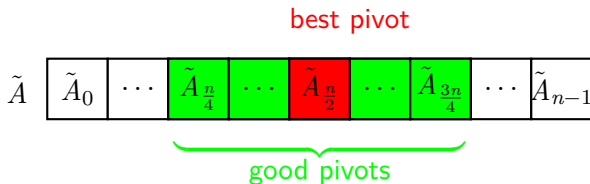
## Strategy 2: Selecting a pivot randomly [Hoare, 1961]

QUICKSELECT( $A, k$ )

```
1: Choose an element  $A_i$  from  $A$  uniformly at random;  
2:  $S_+ = \{\}$ ;  
3:  $S_- = \{\}$ ;  
4: for all element  $A_j$  in  $A$  do  
5:   if  $A_j > A_i$  then  
6:      $S_+ = S_+ \cup \{A_j\}$ ;  
7:   else  
8:      $S_- = S_- \cup \{A_j\}$ ;  
9:   end if  
10: end for  
11: if  $|S_-| = k - 1$  then  
12:   return  $A_i$ ;  
13: else if  $|S_-| > k - 1$  then  
14:   return QUICKSELECT( $S_-, k$ );  
15: else  
16:   return QUICKSELECT( $S_+, k - |S_-| - 1$ );  
17: end if
```



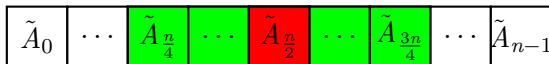
# Randomized DIVIDE AND CONQUER cont'd



- Basic idea: when selecting an element uniformly at random, it is highly likely to get a good pivot since a fairly large fraction of the elements are nearly-central.

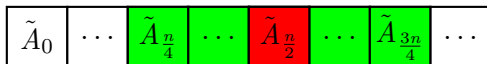
# An example

Iteration #1



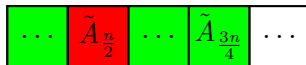
↓ Select  $\tilde{A}_{n-1}$  as pivot

Iteration #2



↓ Select  $\tilde{A}_{\frac{n}{4}}$  as pivot

Iteration #3



- Selecting a **nearly-central pivot** will lead to a  $\frac{3}{4}$  shrinkage of problem size.
- Two iterations are expected before selecting a **nearly-central pivot**.

## Theorem

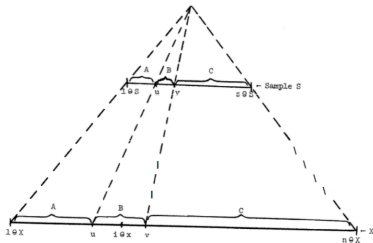
*The expected running time of QUICKSELECT is  $O(n)$ .*

## Proof.

- We divide the execution into a series of phases: phase  $j$  contains a collection of iterations when the size of set under consideration is in  $[n(\frac{3}{4})^{j+1} + 1, n(\frac{3}{4})^j]$ , say  $[\frac{3}{4}n + 1, n]$  for phase 0, and  $[\frac{9}{16}n + 1, \frac{3}{4}n]$  for phase 1.
- Let  $X$  be the number of comparison that QUICKSELECT uses, and  $X_j$  be the number of comparison in phase  $j$ . Thus,  
$$X = X_0 + X_1 + \dots$$
- Consider phase  $j$ . The probability to find a nearly-central pivot is  $\frac{1}{2}$  since half elements are nearly-central. Selecting a nearly-central pivot will lead to a  $\frac{3}{4}$  shrinkage of problem size and therefore make the execution enter phase  $(j+1)$ . Thus, the expected iteration number in phase  $j$  is 2.
- Each iteration in phase  $j$  performs at most  $cn(\frac{3}{4})^j$  comparison  $j$  since there are at most  $n(\frac{3}{4})^j$  elements. Thus,  $E[X_j] \leq 2cn(\frac{3}{4})^j$ .
- Hence  $E[X] = E[X_0 + X_1 + \dots] \leq \sum_j 2cn(\frac{3}{4})^j \leq 8cn$ .

Strategy 3: FLOYD-RIVEST algorithm selects a pivot based on random samples

## Strategy 3: Selecting pivots according to a random sample



- In 1973, Robert Floyd and Ronald Rivest proposed to select pivot using **random sampling** technique.
- Basic idea: A random sample, if sufficiently large, is a good representation of the whole set. Specifically, the median of a sample is an **unbiased point estimator** of the median of the whole set. We can also use **interval estimation**, i.e., a small interval that is expected to contain the median of the whole set with high probability.

# Floyd-Rivest algorithm for SELECTION [1973]

FLOYD-RIVEST-SELECT( $A, k$ )

- 1: Select a small random sample  $S$  (with replacement) from  $A$ .
  - 2: Select two pivots, denoted as  $u$  and  $v$ , from  $S$  through recursively calling FLOYD-RIVEST-SELECT. The interval  $[u, v]$ , although small, is expected to cover the  $k$ -th smallest element of  $A$ .
  - 3: Divide  $A$  into three dis-joint subsets:  $L$  contains the elements less than  $u$ ,  $M$  contains elements in  $[u, v]$ , and  $H$  contains the elements greater than  $v$ .
  - 4: Partition  $A$  into these three sets through comparing each element  $A_i$  with  $u$  and  $v$ : if  $k \leq \frac{n}{2}$ ,  $A_i$  is compared with  $v$  first and then to  $u$  only if  $A_i \leq v$ . The order is reversed if  $k > \frac{n}{2}$ .
  - 5: The  $k$ -th smallest element of  $A$  is selected through recursively running over an appropriate subset.
- Here we present a variant of Floyd-Rivest algorithm called LAZYSELECT, which is much easier to analyze.

# LAZYSELECTMEDIAN algorithm

LAZYSELECTMEDIAN( $A$ )

- 1: Randomly sample  $r$  elements (with replacement) from  $A = [A_0, A_1, A_2, \dots, A_{n-1}]$ . Denote the sample as  $S$ .
- 2: Sort  $S$ . Let  $u$  be the  $\frac{1-\delta}{2}r$ -th smallest element of  $S$  and  $v$  be the  $\frac{1+\delta}{2}r$ -th smallest element of  $S$ .
- 3: Divide  $A$  into three dis-joint subsets:

$$L = \{A_i : A_i < u\};$$

$$M = \{A_i : u \leq A_i \leq v\};$$

$$H = \{A_i : A_i > v\};$$

- 4: Check the following constraints of  $M$ :

- $M$  covers the median:  $|L| \leq \frac{n}{2}$  and  $|H| \leq \frac{n}{2}$
- $M$  should not be too large:  $|M| \leq c\delta n$

If one of the constraints was violated, got to STEP 1.

- 5: Sort  $M$  and return the  $(\frac{n}{2} - |L|)$ -th smallest of  $M$  as the median of  $A$ .

# An example

**Input:**  $A$ .  $n = |A| = 16$ . **Set**  $\delta = \frac{1}{2}$

8	1	15	10	4	3	2	9	7	12	5	16	14	6	13	11
---	---	----	----	---	---	---	---	---	----	---	----	----	---	----	----

↓ **Sample**  $r = 8$  **elements**

8	1	15	10	4	3	2	9	7	12	5	16	14	6	13	11
---	---	----	----	---	---	---	---	---	----	---	----	----	---	----	----

$$S = \{2, 4, 5, 8, 11, 13, 15, 16\}$$

↓ **Divide**  $A$  **into**  $L$ ,  $M$ , **and**  $H$

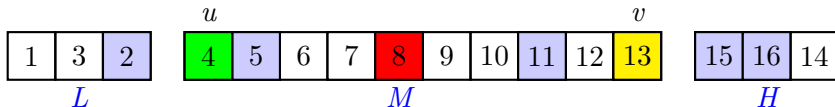
1	3	2	$u$										$v$				
1	3	2	4	5	6	7	8	9	10	11	12	13	15	16	14		
$L$			$M$												$H$		

**Return** 8 **as the median of**  $A$



# Elaborately-designed $\delta$ and $r$

$$S = \{2, 4, 5, 8, 11, 13, 15, 16\}$$



- We expect the following two properties of  $M$ :
  - On one side,  $|M|$  should be **sufficiently large** such that the median of  $A$  is covered by  $M$  with high probability.
  - On the other side,  $|M|$  should be **sufficiently small** such that the sorting operation in Step 5 will not take a long time.
- We claim that  $|M| = \Theta(n^{\frac{3}{4}})$  is an appropriate size that satisfies these two constraints simultaneously.
- To obtain such a  $M$ , we set  $r = n^{\frac{3}{4}}$ , and  $\delta = n^{-\frac{1}{4}}$  as  $M$  is expected to have a size of  $\delta n = n^{\frac{3}{4}}$ .

# Time-complexity analysis: linear time

LAZYSELECTMEDIAN( $A$ )

- 1: Randomly sample  $r$  elements (with replacement) from  $A = [A_0, A_1, A_2, \dots, A_{n-1}]$ . Denote the sample as  $S$ . **//Set  $r = n^{\frac{3}{4}}$**
- 2: Sort  $S$ . Let  $u$  be the  $\frac{1-\delta}{2}r$ -th smallest element of  $S$  and  $v$  be the  $\frac{1+\delta}{2}r$ -th smallest element of  $S$ . **//Take  $O(r \log r) = o(n)$  time**
- 3: Divide  $A$  into three disjoint subsets: **//Take  $2n$  steps**

$$L = \{A_i : A_i < u\};$$

$$M = \{A_i : u \leq A_i \leq v\};$$

$$H = \{A_i : A_i > v\};$$

- 4: Check the following constraints of  $M$ :

- $M$  covers the median:  $|L| \leq \frac{n}{2}$  and  $|H| \leq \frac{n}{2}$
- $M$  should not be too large:  $|M| \leq c\delta n$

If one of the constraints was violated, got to Step 1.

- 5: Sort  $M$  and return the  $(\frac{n}{2} - |L|)$ -th smallest of  $M$  as the median of  $A$ .

**//Take  $O(\delta n \log(\delta n)) = o(n)$  time when setting  $\delta = n^{-\frac{1}{4}}$**

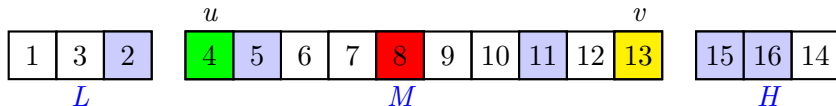
- Total running time (in one pass):  $2n + o(n)$ . The best known deterministic algorithm takes  $3n$  but it is too complicated. On the hand, it has been proved at least  $2n$  steps are needed.

# Analysis of the success probability in one pass

## Theorem

With probability  $1 - O(n^{-\frac{1}{4}})$ , LAZYSELECTMEDIAN reports the median in the first pass. Thus, the total running time is only  $2n + o(n)$ .

$$S = \{2, 4, 5, 8, 11, 13, 15, 16\}$$



- There are two types of failures in one pass, namely,  $M$  does not cover the median of the whole set  $A$ , and  $M$  is too large. We claim that the probability of both types of failures are as small as  $O(n^{-\frac{1}{4}})$ . Here we present proof for the first type only.

## $M$ covers the median of $A$ with high probability

- We argue that  $|L| > \frac{n}{2}$  occurs with probability  $O(n^{-\frac{1}{4}})$ . Note that  $|L| > \frac{n}{2}$  implies that  $u$  is greater than the median of  $A$ , and thus at least  $\frac{1+\delta}{2}r$  elements in  $S$  are greater than the median.
- Let  $X = x_1 + x_2 + \dots + x_r$  be the number of sampled elements greater than the median of  $A$ , where  $x_i$  is an index variable:  
$$x_i = \begin{cases} 1 & \text{if the } i\text{-th element in } S \text{ is greater than the median} \\ 0 & \text{otherwise} \end{cases}$$
- Then  $E(x_i) = \frac{1}{2}$ ,  $\sigma^2(x_i) = \frac{1}{4}$ ,  $E(X) = \frac{1}{2}r$ ,  $\sigma^2(X) = \frac{1}{4}r$ , and

$$\Pr(|L| > \frac{n}{2}) \leq \Pr(X \geq \frac{1+\delta}{2}r) \quad (2)$$

$$= \frac{1}{2} \Pr(|X - E(X)| \geq \frac{\delta}{2}r) \quad (3)$$

$$\leq \frac{\frac{1}{2} \sigma^2(X)}{(\frac{\delta}{2}r)^2} \quad (4)$$

$$= \frac{1}{2} \frac{1}{\delta^2 r} \quad (5)$$

$$= \frac{1}{2} n^{-\frac{1}{4}} \quad (6)$$

MULTIPLICATION problem: to multiply **two  $n$ -bits integers**

# MULTIPLICATION problem

**INPUT:** Two  $n$ -bits integers  $x$  and  $y$ . Here we represent  $x$  as an array  $x_0x_1\dots x_{n-1}$ , where  $x_i$  denotes the  $i$ -th bit of  $x$ . Similarly, we represent  $y$  as an array  $y_0y_1\dots y_{n-1}$ , where  $y_i$  denotes the  $i$ -th bit of  $y$ .

**OUTPUT:** The product  $x \times y$ .

- An example:

$$\begin{array}{r} 12 \\ \times 34 \\ \hline 48 \\ 36 \\ \hline 408 \end{array}$$

- Question: Is the grade-school  $O(n^2)$  algorithm optimal?



- Conjecture: In 1960, Andrey Kolmogorov conjectured that any algorithm for that task would require  $\Omega(n^2)$  elementary operations.



# MULTIPLICATION problem: Trial 1

- Key observation: both  $x$  and  $y$  can be decomposed into two parts;
- DIVIDE AND CONQUER:
  - 1 **Divide:**  $x = x_h \times 2^{\frac{n}{2}} + x_l$ ,  $y = y_h \times 2^{\frac{n}{2}} + y_l$ ,
  - 2 **Conquer:** calculate  $x_h y_h$ ,  $x_h y_l$ ,  $x_l y_h$ , and  $x_l y_l$ ;
  - 3 **Combine:**

$$xy = (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) \quad (7)$$

$$= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l \quad (8)$$

# MULTIPLICATION problem: Trial 1

- Example:
  - Objective: to calculate  $12 \times 34$
  - $x = 12 = 1 \times 10 + 2$ ,  $y = 34 = 3 \times 10 + 4$
  - $x \times y = (1 \times 3) \times 10^2 + ((1 \times 4) + (2 \times 3)) \times 10 + 2 \times 4$
- Note: 4 sub-problems, 3 additions, and 2 shifts;
- Time-complexity:  $T(n) = 4T(\frac{n}{2}) + O(n) = O(n^2)$

Question: can we reduce the number of sub-problems?

# Reduce the number of sub-problems

$\times$	$y_h$	$y_l$
$x_h$	$x_h y_h$	$x_h y_l$
$x_l$	$x_l y_h$	$x_l y_l$

- Our objective is to calculate  $x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l$ .
- Thus it is unnecessary to calculate  $x_h y_l$  and  $x_l y_h$  separately; we just need to calculate the sum  $(x_h y_l + x_l y_h)$ .
- It is obvious that
$$(x_h y_l + x_l y_h) + x_h y_h + x_l y_l = (x_h + x_l) \times (y_h + y_l).$$
- The sum  $(x_h y_l + x_l y_h)$  can be calculated using only **one** additional multiplication.
- This idea is dated back to Carl. F. Gauss: Calculation of the product of two complex numbers
$$(a + bi)(c + di) = (ac - bd) + (bc + ad)i$$
 seems to require four multiplications, three multiplications  $ac$ ,  $bd$ , and  $(a + b)(c + d)$  are sufficient because  $bc + ad = (a + b)(c + d) - ac - bd$ .

# MULTIPLICATION problem: a clever **conquer**

[Karatsuba-Ofman, 1962]



Figure 3: Anatolii Alexeevich Karatsuba

- Karatsuba algorithm was the first multiplication algorithm asymptotically faster than the quadratic "grade school" algorithm.

# MULTIPLICATION problem: a clever conquer

- DIVIDE AND CONQUER:

- ➊ **Divide:**  $x = x_h \times 2^{\frac{n}{2}} + x_l$ ,  $y = y_h \times 2^{\frac{n}{2}} + y_l$ ,
- ➋ **Conquer:** calculate  $x_h y_h$ ,  $x_l y_l$ , and  $P = (x_h + x_l)(y_h + y_l)$ ;
- ➌ **Combine:**

$$xy = (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) \quad (9)$$

$$= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l \quad (10)$$

$$= x_h y_h 2^n + (P - x_h y_h - x_l y_l) 2^{\frac{n}{2}} + x_l y_l \quad (11)$$

# Karatsuba-Ofman algorithm

- Example:
  - Objective: to calculate  $12 \times 34$
  - $x = 12 = 1 \times 10 + 2$ ,  $y = 34 = 3 \times 10 + 4$
  - $P = (1 + 2) \times (3 + 4)$
  - $x \times y = (1 \times 3) \times 10^2 + (P - 1 \times 3 - 2 \times 4) \times 10 + 2 \times 4$
- Note: 3 sub-problems, 6 additions, and 2 shifts;
- Time-complexity:
$$T(n) = 3T\left(\frac{n}{2}\right) + cn = O(n^{\log_2 3}) = O(n^{1.585})$$
- Karatsuba algorithm is a special case of Toom-Cook algorithm. Toom-3 algorithm decomposes both  $x$  and  $y$  into 3 parts, and calculates  $xy$  in  $O(n^{1.465})$  time.

# Theoretical analysis vs. empirical performance

- For large  $n$ , Karatsuba's algorithm will perform fewer shifts and single-digit additions.
- For small values of  $n$ , however, the extra shift and add operations may make it run slower.
- The crossover point depends on the computer platform and context.
- When applying FFT technique over ring, the MULTIPLICATION can be finished in  $O(n \log n \log \log n)$  time.

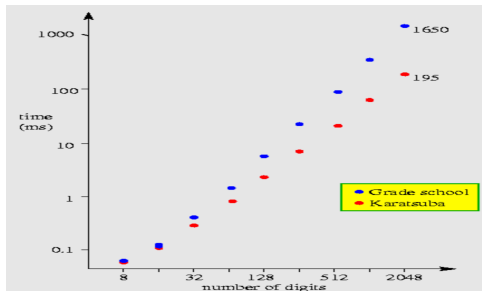


Figure 4: See <https://www.cs.cmu.edu/~cburch/251/karat/> for more details.



- Problem: Given two  $n$ -digit numbers  $s$  and  $t$ , to calculate  $q = s/t$  and  $r = s \bmod t$ .
- Method:

- 1 Calculate  $x = 1/t$  using Newton's method first:

$$x_{i+1} = 2x_i - t \times x_i^2$$

- 2 At most  $\log n$  iterations are needed.
- 3 Thus division is as fast as multiplication.

# Details of FAST DIVISION: Newton's method

- Objective: Calculate  $x = 1/t$ .
  - $x$  is the root of  $f(x) = 0$ , where  $f(x) = (t - \frac{1}{x})$ . (Why the form here?)
  - Newton's method:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (12)$$

$$= x_i - \frac{t - \frac{1}{x_i}}{\frac{1}{x_i^2}} \quad (13)$$

$$= -t \times x_i^2 + 2x_i \quad (14)$$

- Convergence speed: quadratic, i.e.  $\epsilon_{i+1} \leq M\epsilon_i^2$ , where  $M$  is a supremum of a ratio, and  $\epsilon_i$  denotes the distance between  $x_i$  and  $\frac{1}{t}$ . Thus the number of iterations is limited by  $\log \log t = O(\log n)$ .

# FAST DIVISION: an example

- Objective: to calculate  $\frac{1}{13}$ .

---

#Iteration	$x_i$	$\epsilon_i$
0	0.018700	-0.058223
1	0.032854	-0.044069
2	0.051676	-0.025247
3	0.068636	-0.008286
4	0.076030	-0.000892
5	0.076912	-1.03583e-05
6	0.076923	-1.39483e-09
7	0.076923	-2.77556e-17
8	...	...

---

- Note: the quadratic convergence implies that the error  $\epsilon_i$  has a form of  $O(e^{2^i})$ ; thus the iteration number is limited by  $\log \log(t)$ .

MATRIX MULTIPLICATION problem: to multiply two **matrices**

# MATRIX MULTIPLICATION problem

**INPUT:** Two  $n \times n$  matrices  $A$  and  $B$ ,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

**OUTPUT:** The product  $C = AB$ .

Grade-school algorithm:  $O(n^3)$ .

# MATRIX MULTIPLICATION problem: Trial 1 I

- Matrix multiplication: Given two  $n \times n$  matrices  $A$  and  $B$ , compute  $C = AB$ ;
  - Grade-school:  $O(n^3)$ .
- Key observation: matrix can be decomposed into four  $\frac{n}{2} \times \frac{n}{2}$  matrices;
- DIVIDE AND CONQUER:
  - 1 **Divide:** divide  $A$ ,  $B$ , and  $C$  into sub-matrices;
  - 2 **Conquer:** calculate products of sub-matrices;
  - 3 **Combine:**

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

$$C_{11} = (A_{11} \times B_{11}) + (A_{12} \times B_{21})$$

$$C_{12} = (A_{11} \times B_{12}) + (A_{12} \times B_{22})$$

$$C_{21} = (A_{21} \times B_{11}) + (A_{22} \times B_{21})$$

$$C_{22} = (A_{21} \times B_{12}) + (A_{22} \times B_{22})$$

- We need to solve 8 sub-problems, and 4 additions; each addition takes  $O(n^2)$  time.
- $T(n) = 8T(\frac{n}{2}) + cn^2 = O(n^3)$

Question: can we reduce the number of sub-problems?





Figure 5: Volker Strassen, 2009

- The first algorithm for performing matrix multiplication faster than the  $O(n^3)$  time bound.

# MATRIX MULTIPLICATION problem: a clever conquer I

- Matrix multiplication: Given two  $n \times n$  matrices  $A$  and  $B$ , compute  $C = AB$ ;
  - Grade-school:  $O(n^3)$ .
  - Key observation: matrix can be decomposed into four  $\frac{n}{2} \times \frac{n}{2}$  matrices;

DIVIDE AND CONQUER:

- 1 **Divide:** divide  $A$ ,  $B$ , and  $C$  into sub-matrices;
- 2 **Conquer:** calculate products of sub-matrices;
- 3 **Combine:**

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

# MATRIXMULTIPLICATION problem: a clever conquer II

$$P_1 = A_{11} \times (B_{12} - B_{22}) \quad (15)$$

$$P_2 = (A_{11} + A_{12}) \times B_{22} \quad (16)$$

$$P_3 = (A_{21} + A_{22}) \times B_{11} \quad (17)$$

$$P_4 = A_{22} \times (B_{21} - B_{11}) \quad (18)$$

$$P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) \quad (19)$$

$$P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \quad (20)$$

$$P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12}) \quad (21)$$

$$C_{11} = P_4 + P_5 + P_6 - P_2 \quad (22)$$

$$C_{12} = P_1 + P_2 \quad (23)$$

$$C_{21} = P_3 + P_4 \quad (24)$$

$$C_{22} = P_1 + P_5 - P_3 - P_7 \quad (25)$$

- We need to solve 7 sub-problems, and 18 additions/subtraction; each addition/subtraction takes  $O(n^2)$  time.
- $T(n) = 7T(\frac{n}{2}) + cn^2 = O(n^{\log_2 7}) = O(n^{2.807})$

- For large  $n$ , Strassen algorithm is faster than grade-school method.<sup>1</sup>
- Strassen algorithm can be used to solve other problems, say matrix inversion, determinant calculation, finding triangles in graphs, etc.
- Gaussian elimination is not optimal.

---

<sup>1</sup>This heavily depends on the system, including memory access property, hardware design, etc.

- Strassen algorithm performs better than grade-school method only for large  $n$ .
- The reduction in the number of arithmetic operations however comes at the price of a somewhat reduced numerical stability,
- The algorithm also requires significantly more memory compared to the naive algorithm.

# Fast matrix multiplication

- multiply two  $2 \times 2$  matrices: 7 scalar sub-problems:  
 $O(n^{\log_2 7}) = O(n^{2.807})$  [ Strassen 1969 ]
- multiply two  $2 \times 2$  matrices: 6 scalar sub-problems:  
 $O(n^{\log_2 6}) = O(n^{2.585})$  (impossible)[Hopcroft and Kerr 1971]
- multiply two  $3 \times 3$  matrices: 21 scalar sub-problems:  
 $O(n^{\log_3 21}) = O(n^{2.771})$  (impossible)
- multiply two  $20 \times 20$  matrices: 4460 scalar sub-problems:  
 $O(n^{\log_{20} 4460}) = O(n^{2.805})$
- multiply two  $48 \times 48$  matrices: 47217 scalar sub-problems:  
 $O(n^{\log_{48} 47217}) = O(n^{2.780})$
- Best known till 2010:  $O(n^{2.376})$  [Coppersmith-Winograd, 1987]
- Conjecture:  $O(n^{2+\epsilon})$  for any  $\epsilon > 0$

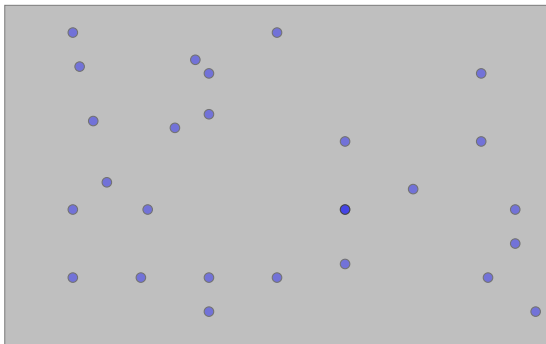
CLOSESTPAIR problem: given a **set** of points in a plane, to find the closest pair



# CLOSESTPAIR problem

**INPUT:**  $n$  points in a plane;

**OUTPUT:** The pair with the least Euclidean distance.



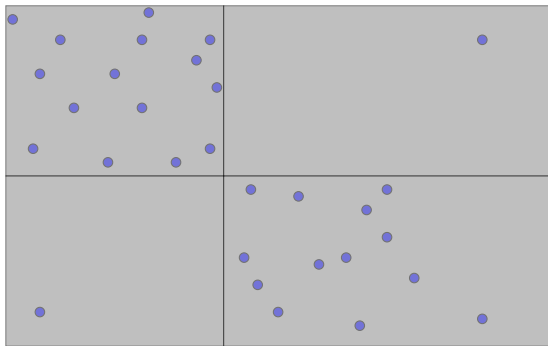
# About CLOSESTPAIR problem

- Computational geometry: M. Shamos and D. Hoey were working out efficient algorithm for basic computational primitive in CG in 1970's. They asked a question: does there exist an algorithm using less than  $O(n^2)$  time?
- 1D case: it is easy to solve the problem in  $O(n \log n)$  via sorting.
- 2D case: a brute-force algorithm works in  $O(n^2)$  time by checking all possible pairs.
- **Question:** can we find a faster method?

Trial 1: Divide into 4 subsets

# Trial 1: DIVIDE AND CONQUER (4 subsets)

- DIVIDE AND CONQUER: divide into 4 subsets.



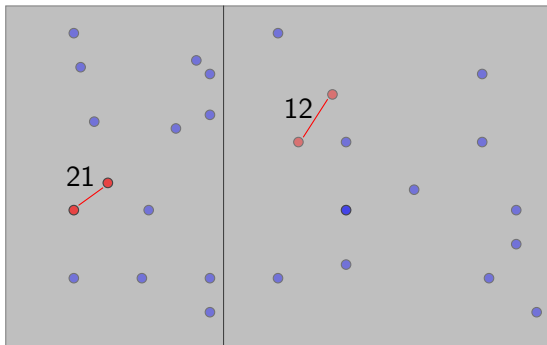
- Difficulties:
  - The subsets might be unbalanced — we cannot guarantee that each subset has approximately  $\frac{n}{4}$  points.
  - Since the closest pair might lie in different subsets, we need to consider all  $\binom{4}{2}$  pairs of subsets to avoid missing the closest pair, thus complicating the “combine” step.

Trial 2: Divide into 2 halves



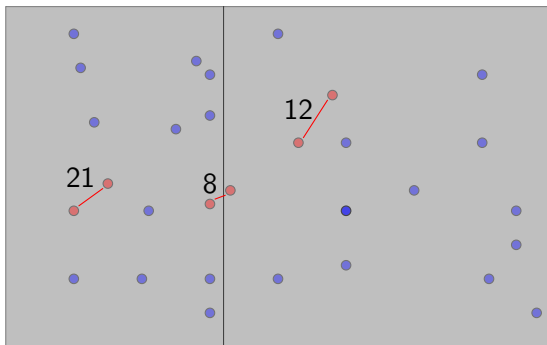
## Trial 2: DIVIDE AND CONQUER (2 subsets)

- **Divide:** dividing into two (roughly equal) subsets;
- **Conquer:** finding closest pairs in each half;



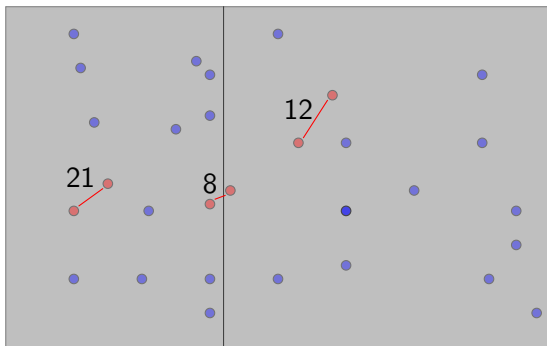
## Trial 2: DIVIDE AND CONQUER (2 subsets)

- **Combine:** It suffices to consider the pairs consisting of one point from left half and one point from right half. Simply examining all such pairs will take  $O(n^2)$  time.





# Two types of redundancy

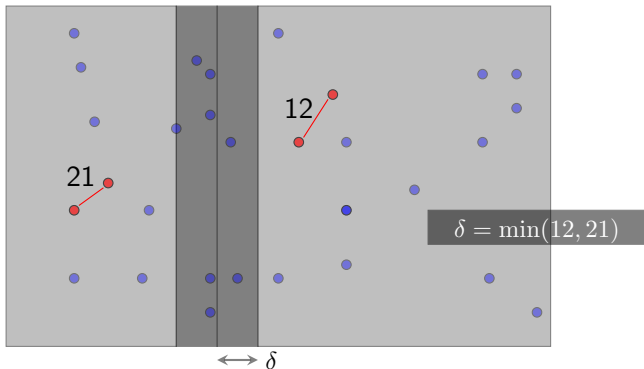


- It is redundant to calculate distance between  $p_i$  and  $p_j$  if
  - $|x_i - x_j| \geq 12$ , or
  - $|y_i - y_j| \geq 12$

# Remove redundancy of type 1

- **Observation 1:**

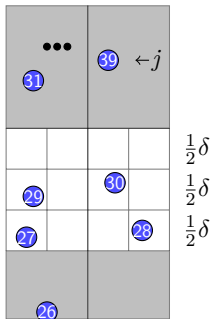
- The third type occurs in **a narrow strip** only; thus, it suffices to check point pairs within the  $2\delta$ -strip.
- Here,  $\delta$  is the minimum of  $\text{CLOSESTPAIR}(\text{LEFTHALF})$  and  $\text{CLOSESTPAIR}(\text{RIGHTHALF})$ .



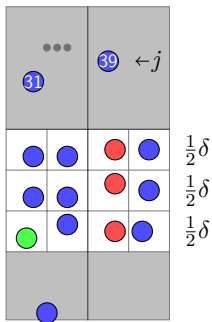
# Remove redundancy of type 2

- **Observation 2:**

- Moreover, it is unnecessary to explore **all** point pairs within the  $2\delta$ -strip. In fact, for each point  $p_i$ , it suffices to examine 11 points for possible closest partners.
- Let's divide the  $2\delta$ -strip into grids (size:  $\frac{\delta}{2} \times \frac{\delta}{2}$ ). A grid contains **at most one** point.
- If two points are 2 rows apart, the distance between them should be over  $\delta$  and thus cannot form closest pair.
- Example: For point 27, it suffices to search within 2 rows for possible closest partners ( $< \delta$ ).

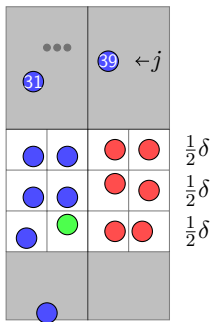


# To detect potential closest pair: Case 1



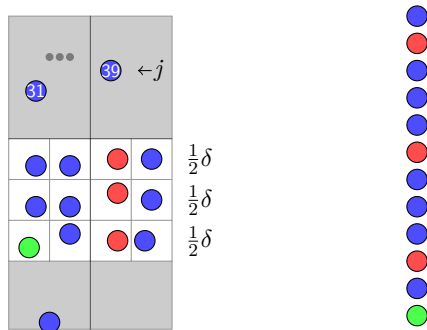
- Green: point  $i$ ;
- Red: the possible closest partner (distance  $< \delta$ ) of point  $i$ ;

## To detect potential closest pair: Case 2



- Green: point  $i$ ;
- Red: the possible closest partner (distance  $< \delta$ ) of point  $i$ ;

# To detect potential closest pair



- If all points within the strip were sorted by  $y$ -coordinates, it suffices to calculate distance between each point with its next 11 neighbors.
- Why 11 points here? All red points fall into the subsequent 11 points.

# CLOSESTPAIR algorithm

CLOSESTPAIR( $p_l, \dots, p_r$ )

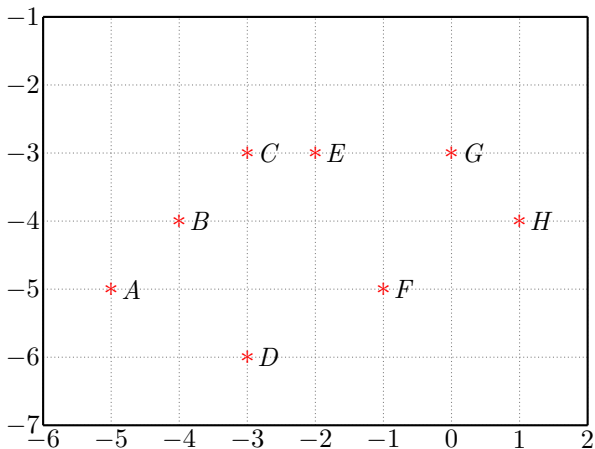
- 1: //To find the closest points within  $(p_l, \dots, p_r)$ . Here we assume that  $p_l, \dots, p_r$  have already been sorted according to  $x$ -coordinate;
  - 2: **if**  $r - l == 1$  **then**
  - 3:     **return**  $d(p_l, p_r)$ ;
  - 4: **end if**
  - 5: Use the  $x$ -coordinate of  $p_{\lfloor \frac{l+r}{2} \rfloor}$  to divide  $p_l, \dots, p_r$  into two halves;
  - 6:  $\delta_1 = \text{CLOSESTPAIR}(\text{LEFTHALF})$ ; //  $T(\frac{n}{2})$
  - 7:  $\delta_2 = \text{CLOSESTPAIR}(\text{RIGHTHALF})$ ; //  $T(\frac{n}{2})$
  - 8:  $\delta = \min(\delta_1, \delta_2)$ ;
  - 9: Sort points within the  $2\delta$  wide strip by  $y$ -coordinate; //  $O(n \log n)$
  - 10: Scan points in  $y$ -order and calculate distance between each point with its next 11 neighbors. Update  $\delta$  if finding a distance less than  $\delta$ ; //  $O(n)$
- Find closest pair within  $p_0, p_1, \dots, p_{n-1}$ :  
    CLOSESTPAIR( $p_0, \dots, p_{n-1}$ )
  - Time-complexity:  $T(n) = 2T(\frac{n}{2}) + O(n \log n) = O(n \log^2 n)$ .

# CLOSESTPAIR algorithm: improvement

- Note that if the points within the  $2\delta$ -wide strip have no structure, we have to sort them from the scratch, which will take  $O(n \log n)$  time.
- Let's try to introduce some structure into the points within the  $2\delta$ -wide: If the point within each  $\delta$ -wide strip were already sorted, it is relatively easy to sort the points within the  $2\delta$ -wide strip. Specifically,
  - Each recursion keeps two sorted list: one list by  $x$ , and the other list by  $y$ .
  - We merge two pre-sorted lists into a list as MERGESORT does, which costs only  $O(n)$  time.
- Time-complexity:  $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$ .

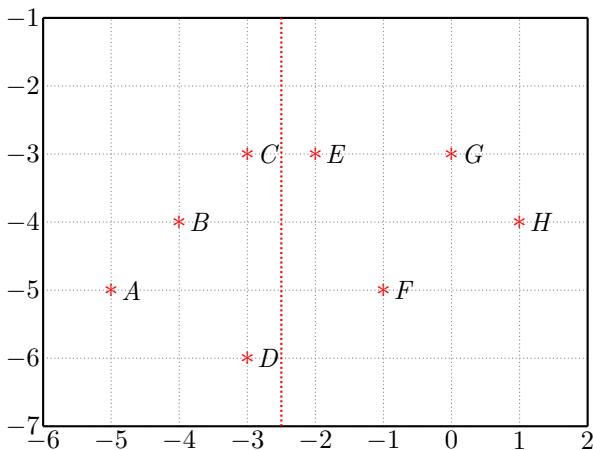


## CLOSESTPAIR: an example with 8 points



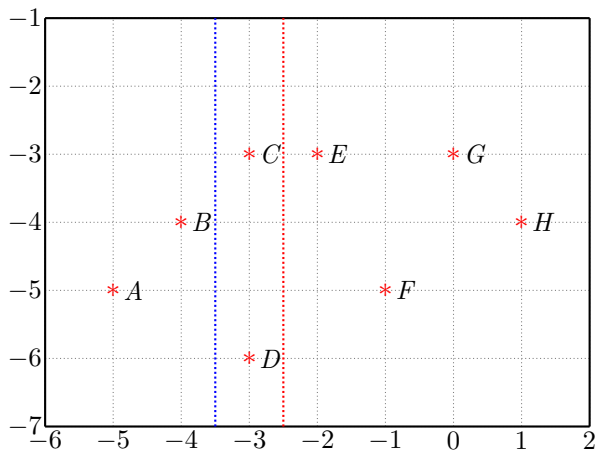
- Objective: to find the closest pair among these 8 points.

## CLOSESTPAIR: an example with 8 points

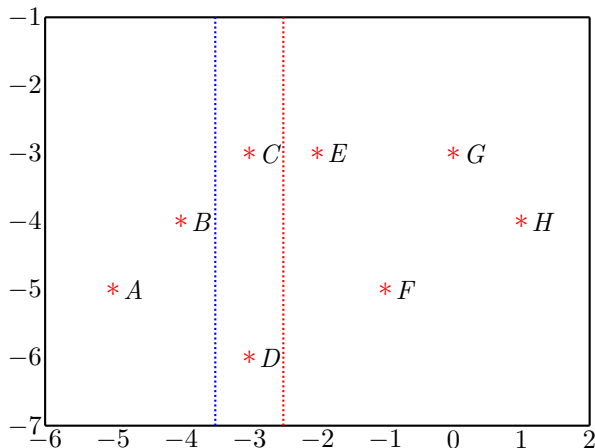


- Objective: to find the closest pair among these 8 points.

# Left half: A, B, C, D

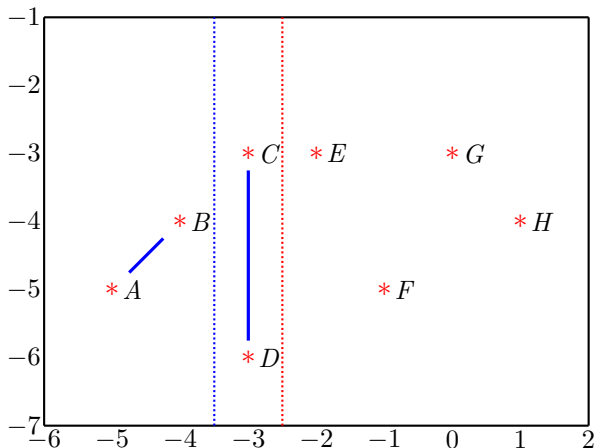


## Left half: A, B, C, D



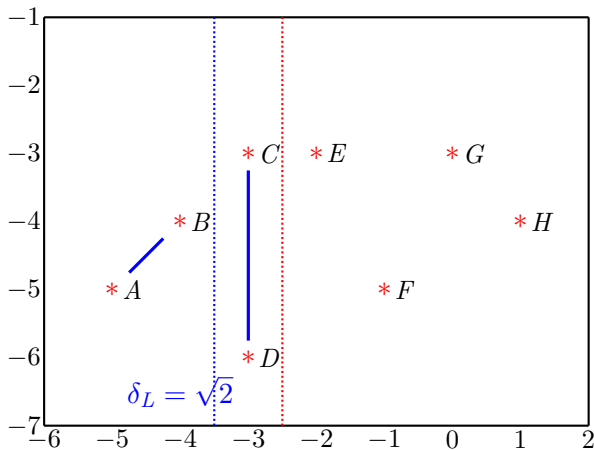
- Pair 1:  $d(A, B) = \sqrt{2}$ ;
- Pair 2:  $d(C, D) = 3$ ;  $\Rightarrow \min = \sqrt{2}$ ; Thus, it suffices to calculate:
- Pair 3:  $d(B, C) = \sqrt{2}$ ;
- Pair 4:  $d(B, D) = \sqrt{5}$ ;  $\Rightarrow \delta_L = \sqrt{2}$ .

## Left half: A, B, C, D



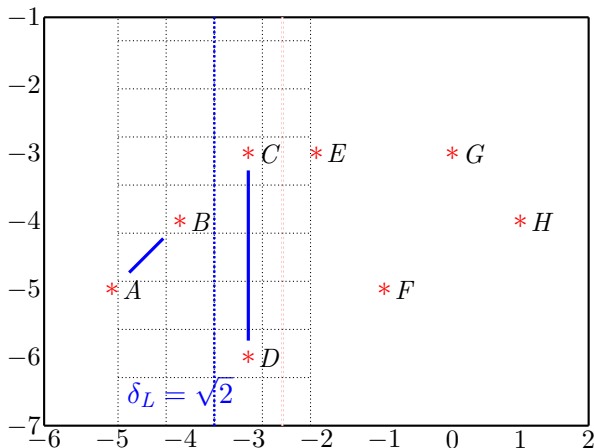
- Pair 1:  $d(A, B) = \sqrt{2}$ ;
- Pair 2:  $d(C, D) = 3$ ;  $\Rightarrow \min = \sqrt{2}$ ; Thus, it suffices to calculate:
- Pair 3:  $d(B, C) = \sqrt{2}$ ;
- Pair 4:  $d(B, D) = \sqrt{5}$ ;  $\Rightarrow \delta_L = \sqrt{2}$ .

## Left half: A, B, C, D



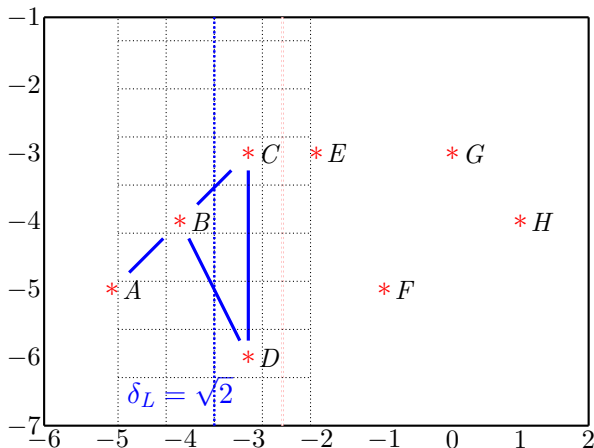
- Pair 1:  $d(A, B) = \sqrt{2}$ ;
- Pair 2:  $d(C, D) = 3$ ;  $\Rightarrow \min = \sqrt{2}$ ; Thus, it suffices to calculate:
- Pair 3:  $d(B, C) = \sqrt{2}$ ;
- Pair 4:  $d(B, D) = \sqrt{5}$ ;  $\Rightarrow \delta_L = \sqrt{2}$ .

## Left half: A, B, C, D



- Pair 1:  $d(A, B) = \sqrt{2}$ ;
- Pair 2:  $d(C, D) = 3$ ;  $\Rightarrow \min = \sqrt{2}$ ; Thus, it suffices to calculate:
- Pair 3:  $d(B, C) = \sqrt{2}$ ;
- Pair 4:  $d(B, D) = \sqrt{5}$ ;  $\Rightarrow \delta_L = \sqrt{2}$ .

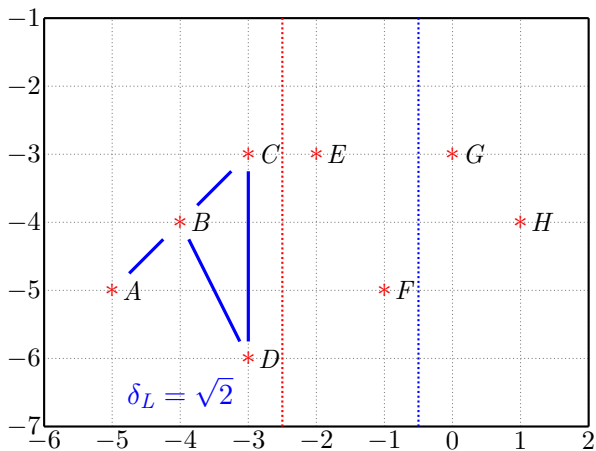
## Left half: A, B, C, D



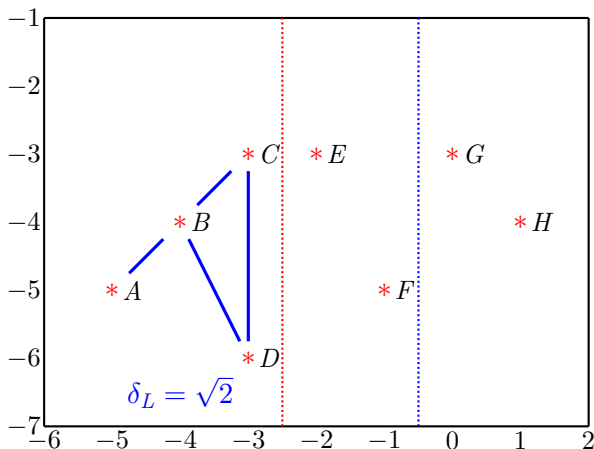
- Pair 1:  $d(A, B) = \sqrt{2}$ ;
- Pair 2:  $d(C, D) = 3$ ;  $\Rightarrow \min = \sqrt{2}$ ; Thus, it suffices to calculate:
- Pair 3:  $d(B, C) = \sqrt{2}$ ;
- Pair 4:  $d(B, D) = \sqrt{5}$ ;  $\Rightarrow \delta_L = \sqrt{2}$ .



## Right half: E, F, G, H

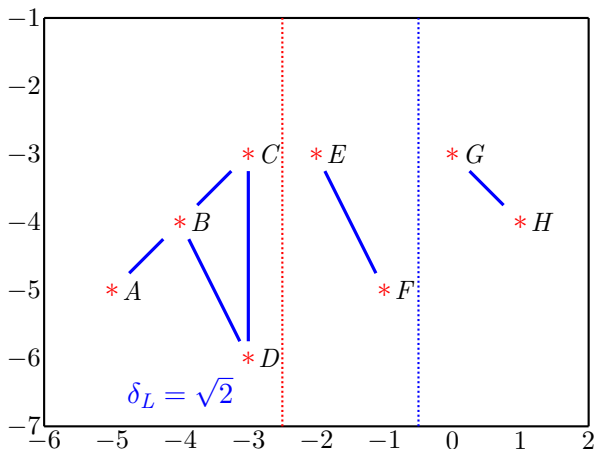


## Right half: E, F, G, H



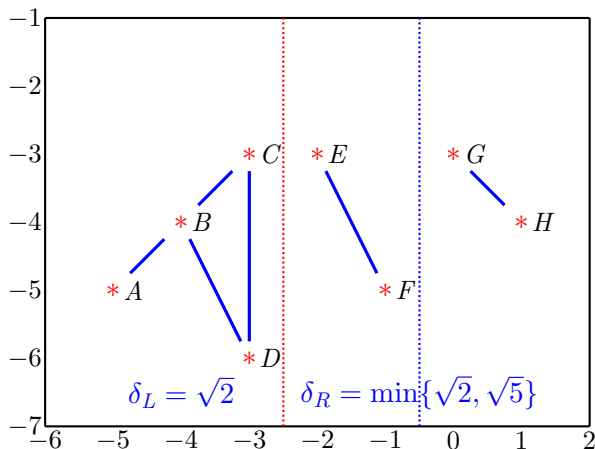
- Pair 5:  $d(E, F) = \sqrt{5}$ ;
- Pair 6:  $d(G, H) = \sqrt{2}$ ;  $\Rightarrow \min = \sqrt{2}$ ; Thus, it suffices to calculate:
- Pair 7:  $d(G, F) = \sqrt{5}$ ;  $\Rightarrow \delta_R = \sqrt{2}$ .

## Right half: E, F, G, H



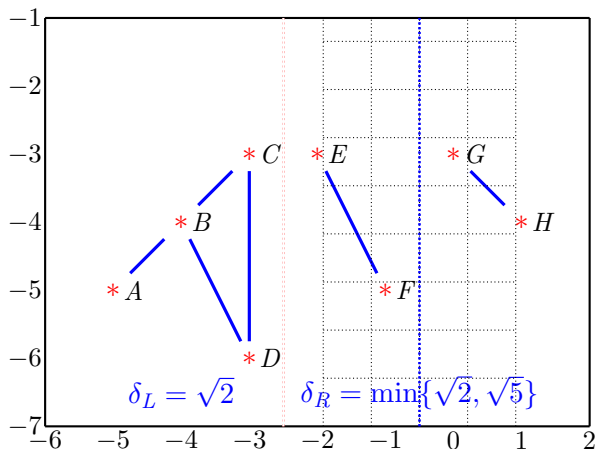
- Pair 5:  $d(E, F) = \sqrt{5}$ ;
- Pair 6:  $d(G, H) = \sqrt{2}$ ;  $\Rightarrow \min = \sqrt{2}$ ; Thus, it suffices to calculate:
- Pair 7:  $d(G, F) = \sqrt{5}$ ;  $\Rightarrow \delta_R = \sqrt{2}$ .

## Right half: E, F, G, H



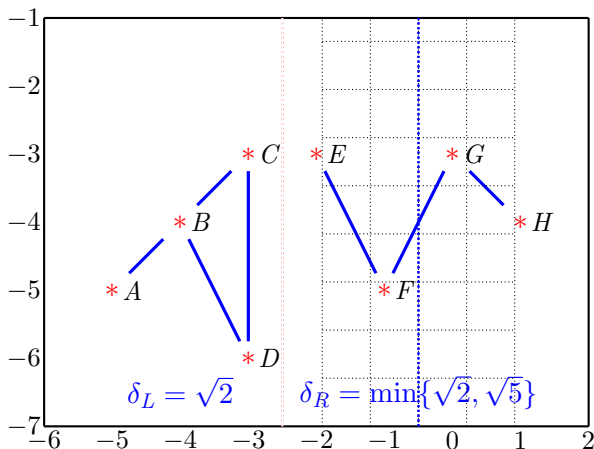
- Pair 5:  $d(E, F) = \sqrt{5}$ ;
- Pair 6:  $d(G, H) = \sqrt{2}$ ;  $\Rightarrow \min = \sqrt{2}$ ; Thus, it suffices to calculate:
- Pair 7:  $d(G, F) = \sqrt{5}$ ;  $\Rightarrow \delta_R = \sqrt{2}$ .

## Right half: E, F, G, H



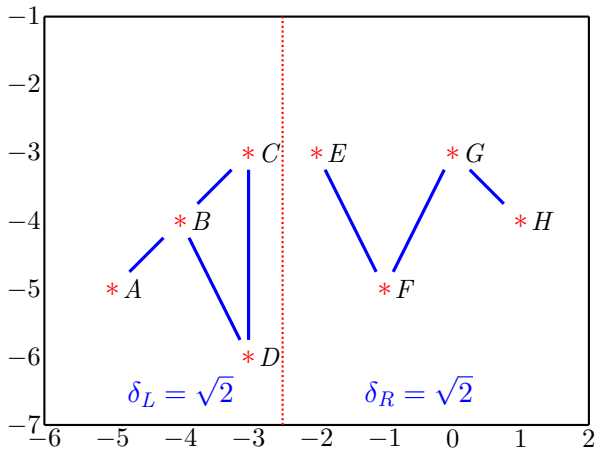
- Pair 5:  $d(E, F) = \sqrt{5}$ ;
- Pair 6:  $d(G, H) = \sqrt{2}$ ;  $\Rightarrow \min = \sqrt{2}$ ; Thus, it suffices to calculate:
- Pair 7:  $d(G, F) = \sqrt{5}$ ;  $\Rightarrow \delta_R = \sqrt{2}$ .

## Right half: E, F, G, H



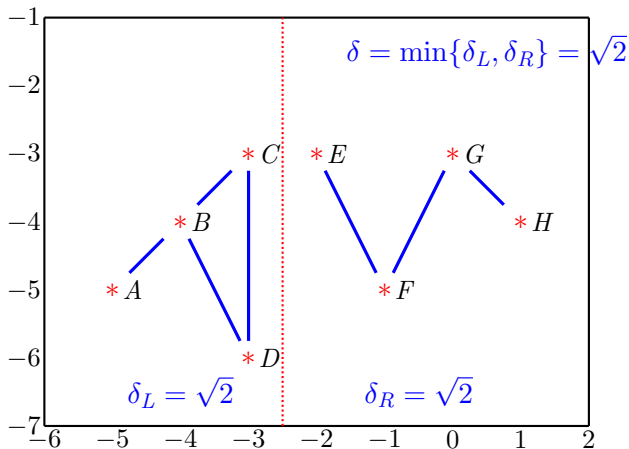
- Pair 5:  $d(E, F) = \sqrt{5}$ ;
- Pair 6:  $d(G, H) = \sqrt{2}$ ;  $\Rightarrow \min = \sqrt{2}$ ; Thus, it suffices to calculate:
- Pair 7:  $d(G, F) = \sqrt{5}$ ;  $\Rightarrow \delta_R = \sqrt{2}$ .

# The entire set: A, B, C, D, E, F, G, H



- Pair 8:  $d(C, E) = 1$ ;
- Pair 9:  $d(D, E) = \sqrt{10}$ ;  $\Rightarrow \delta = 1$ .

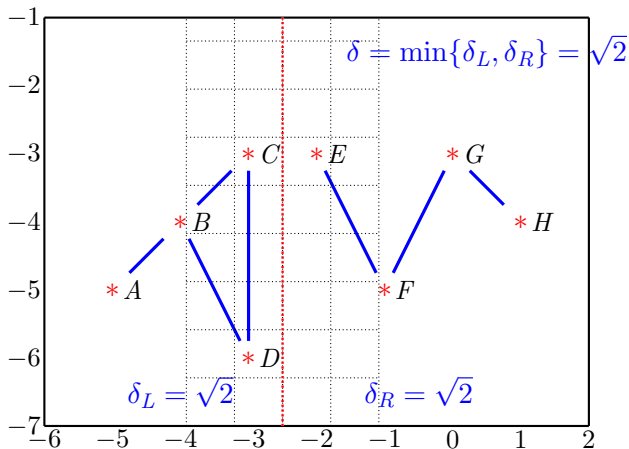
# The entire set: A, B, C, D, E, F, G, H



- Pair 8:  $d(C, E) = 1$ ;
- Pair 9:  $d(D, E) = \sqrt{10}$ ;  $\Rightarrow \delta = 1$ .

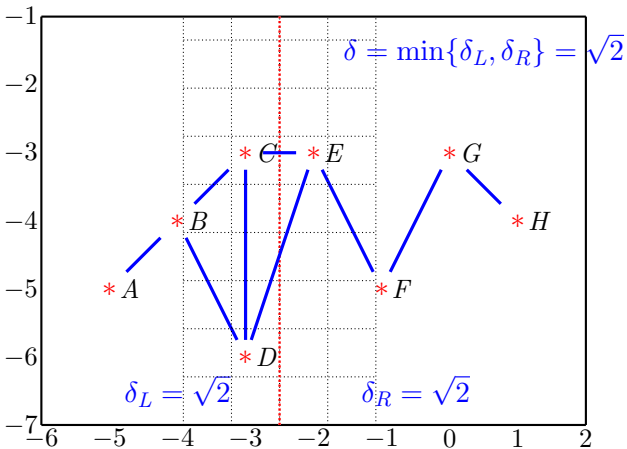


# The entire set: A, B, C, D, E, F, G, H



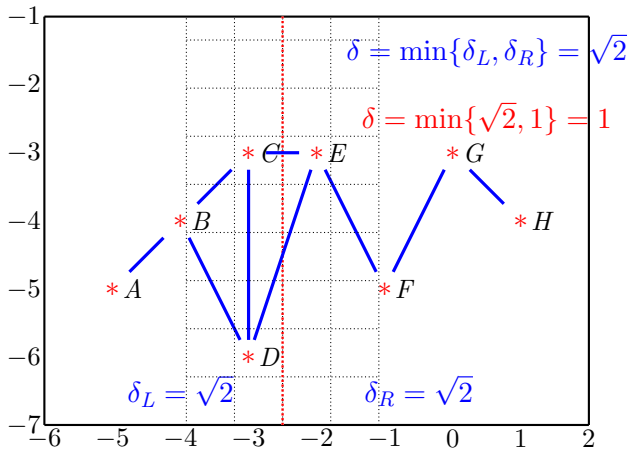
- Pair 8:  $d(C, E) = 1$ ;
- Pair 9:  $d(D, E) = \sqrt{10}$ ;  $\Rightarrow \delta = 1$ .

The entire set: A, B, C, D, E, F, G, H



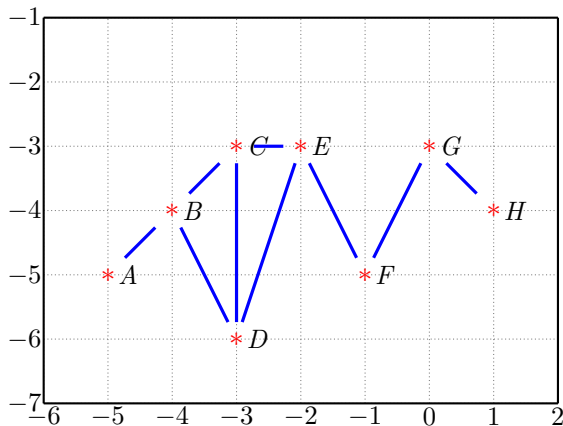
- Pair 8:  $d(C, E) = 1$ ;
- Pair 9:  $d(D, E) = \sqrt{10}$ ;  $\Rightarrow \delta = 1$ .

# The entire set: A, B, C, D, E, F, G, H



- Pair 8:  $d(C, E) = 1$ ;
- Pair 9:  $d(D, E) = \sqrt{10}$ ;  $\Rightarrow \delta = 1$ .

From  $O(n^2)$  to  $O(n \log n)$ , what did we save?

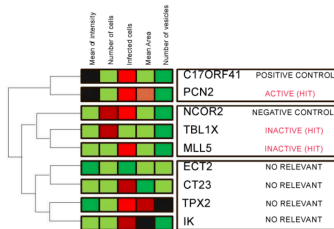


- We calculated distances for only 9 pairs of points (see 'blue' line). The other 19 pairs are redundant due to:
  - at least one of the two points lies out of  $2\delta$ -strip.
  - although two points appear in the same  $2\delta$ -strip, they are at least 2 rows of grids (size:  $\frac{\delta}{2} \times \frac{\delta}{2}$ ) apart.

# Extension: arbitrary (not necessarily geometric) distance functions

## Theorem

*We can perform bottom-up hierarchical clustering, for any cluster distance function computable in constant time from the distances between subclusters, in total time  $O(n^2)$ . We can perform median, centroid, Ward, or other bottom-up clustering methods in which clusters are represented by objects, in time  $O(n^2 \log^2 n)$  and space  $O(n)$ .*



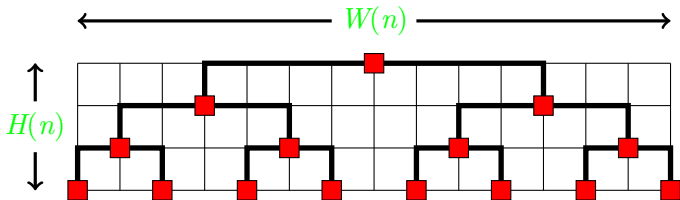
(See Eppstein 1998 for details.)

VLSI embedding: to embed a tree

# Embedding a tree

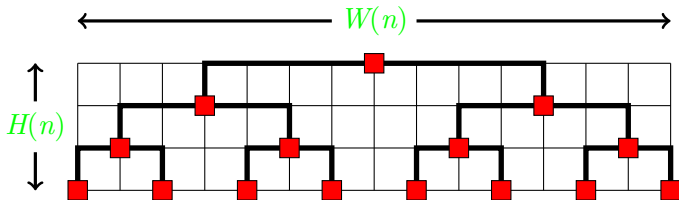
**INPUT:** Given a binary tree with  $n$  node;

**OUTPUT:** Embedding the tree into a VLSI with minimum area.



## Trial 1: divide into two sub-trees

- Let's divide into 2 sub-trees, each with a size of  $\frac{n}{2}$ .

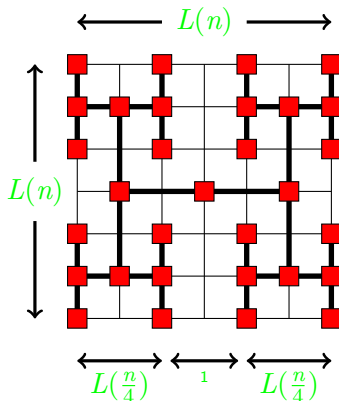


- We have:  
$$H(n) = H\left(\frac{n}{2}\right) + 1 = \Theta(\log n)$$
$$W(n) = 2W\left(\frac{n}{2}\right) + 1 = \Theta(n)$$
- The area is  $\Theta(n \log n)$ .



## Trial 2: divide into 4 sub-trees

- Let's divide into 4 sub-trees, each with a size of  $\frac{n}{4}$ .



- We have:
$$L(n) = 2L\left(\frac{n}{4}\right) + 1 = \Theta(\sqrt{n})$$
- Thus the area is  $\Theta(n)$ .