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

Outline

- 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.

- 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

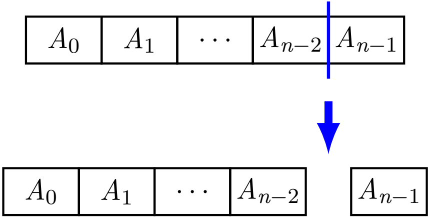
SORT problem

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**: $A[0..n-2]$ 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 $A[0..n-2]$.

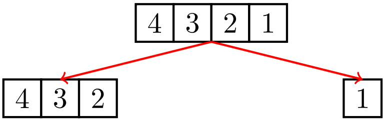
InsertionSort(A, k)

if $k \leq 1$ **then**
    return ;
end if
InsertionSort$(A, k-1)$;
$key = A[k]$;
$i = k - 1$;
while $i \geq 0$ and $A[i] > key$ **do**
    $A[i+1] = A[i]$;
    $i - -$;
end while
$A[i+1] = key$;

An example

4	3	2	1

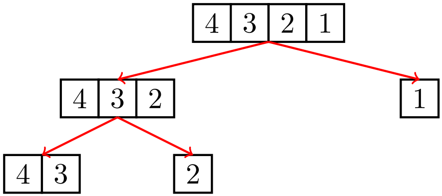
An example



Divide

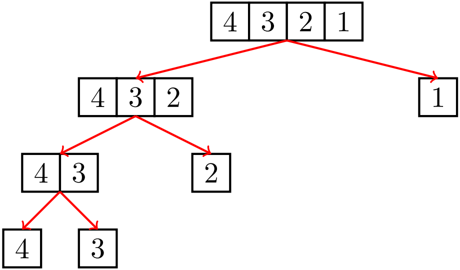
An example

Divide

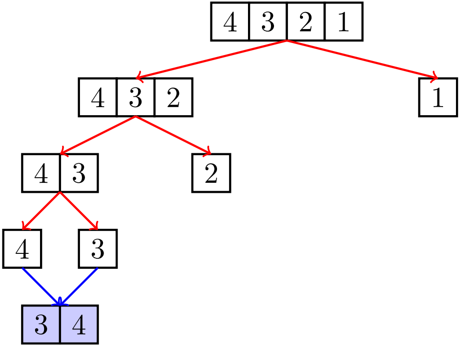


An example

Divide



An example

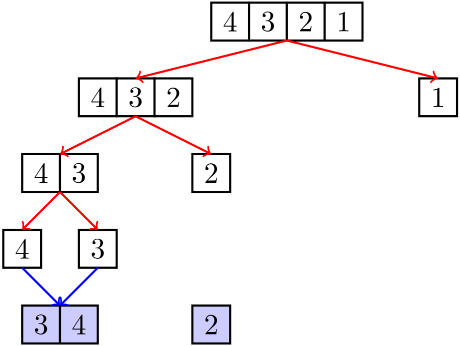


Divide

Combine

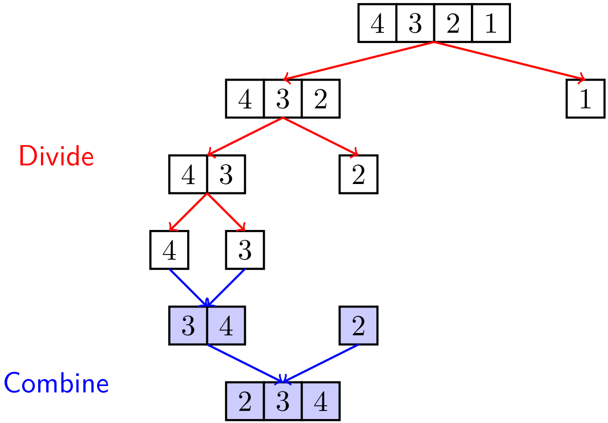
An example

Divide

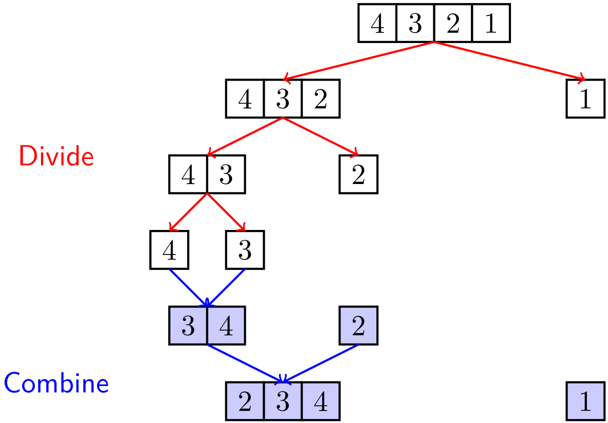


Combine

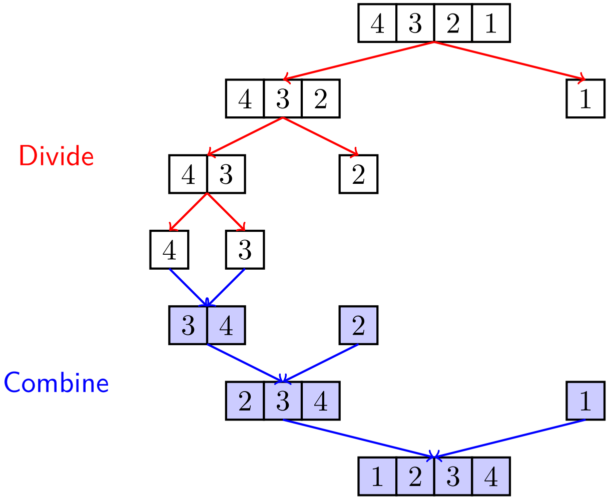
An example



An example



An example



- 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.

8	7	6	5	4	3	2	1

7	8	6	5	4	3	2	1

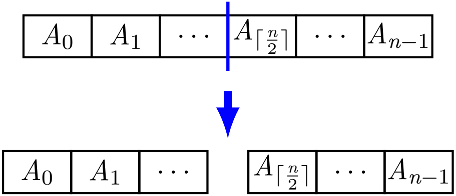
$\vdots$

1	2	3	4	5	6	7	8

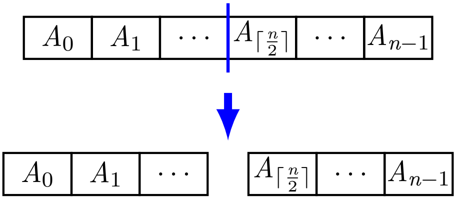
InsertSort: 28 ops

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)

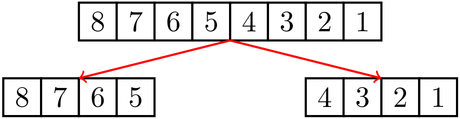
//Sort elements in $A[l..r]$
if $l < r$ **then**
  $m = (l + r)/2$; //m denotes the middle point
  MERGESORT(A, l, m);
  MERGESORT$(A, m + 1, r)$;
  MERGE(A, l, m, r); //Combining the sorted arrays
end if

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

An example

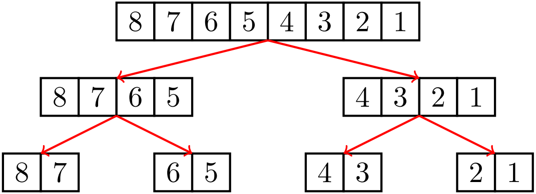
8	7	6	5	4	3	2	1

An example



Divide

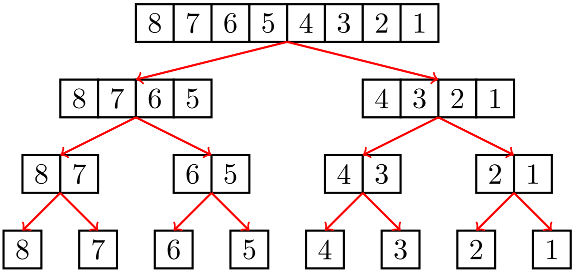
An example



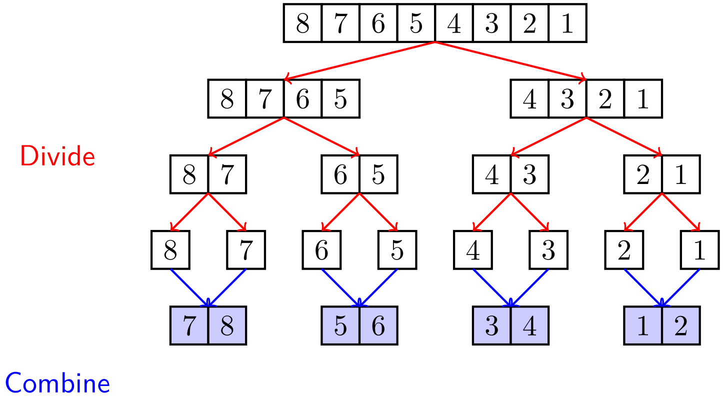
Divide

An example

Divide

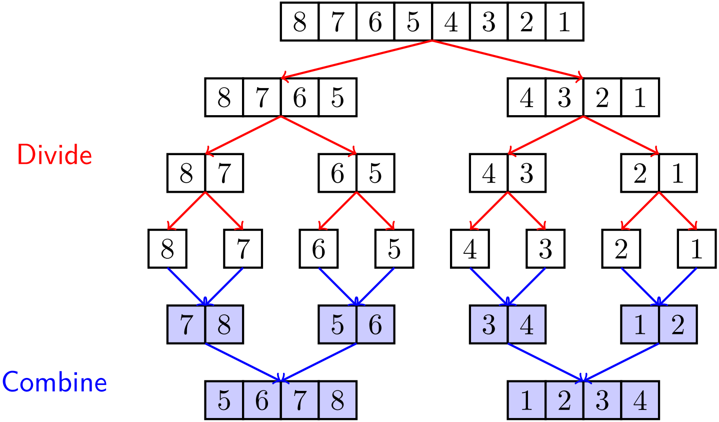


An example

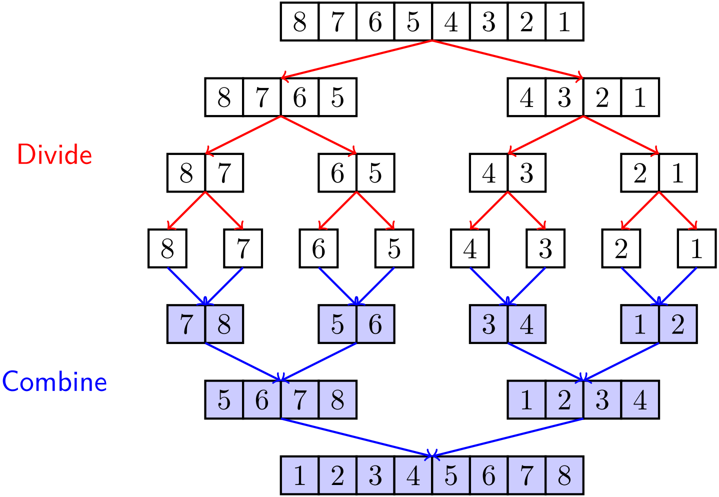


Divide
Combine
8 7 6 5 4 3 2 1
8 7 6 5
4 3 2 1
8 7
6 5
4 3
2 1
8
7
6
5
4
3
2
1
7 8
5 6
3 4
1 2

An example


Divide
Combine
8 7 6 5 4 3 2 1
8 7 6 5
4 3 2 1
8 7
6 5
4 3
2 1
8
7
6
5
4
3
2
1
7 8
5 6
3 4
1 2
5 6 7 8
1 2 3 4

An example

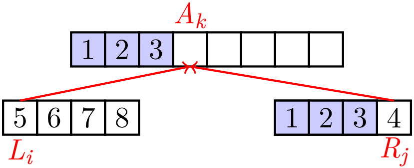

Divide
Combine
8 7 6 5 4 3 2 1
8 7 6 5
4 3 2 1
8 7
6 5
4 3
2 1
8
7
6
5
4
3
2
1
7 8
5 6
3 4
1 2
5 6 7 8
1 2 3 4
1 2 3 4 5 6 7 8

MergeSort algorithm: how to combine?

Merge (A, l, m, r)

```
//Merge A[l..m] (denoted as L) and A[m+1..r] (denoted as R).
i = 0; j = 0;
for k = l to r do
    if L[i] < R[j] then
        A[k] = L[i];
        i++;
        if all elements in L have been copied then
            Copy the remainder elements from R into A;
            break;
        end if
    else
        A[k] = R[j];
        j++;
        if all elements in R have been copied then
            Copy the remainder elements from L into A;
            break;
        end if
    end if
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

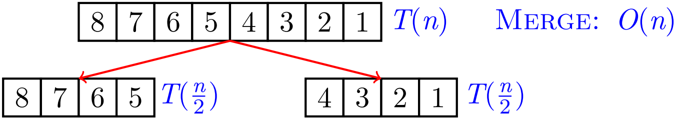
MERGE (A, l, m, r)

//Merge $A[l..m]$ (denoted as L) and $A[m+1..r]$ (denoted as R).
$i = 0$; $j = 0$;
for $k = l$ to r **do**
    if $L[i] < R[j]$ **then**
        $A[k] = L[i]$;
        $i++$;
        if all elements in L have been copied **then**
            Copy the remainder elements from R into A;
            break;
        end if
    else
        $A[k] = R[j]$;
        $j++$;
        if all elements in R have been copied **then**
            Copy the remainder elements from L into A;
            break;
        end if
    end if
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.

Analysis of recursion

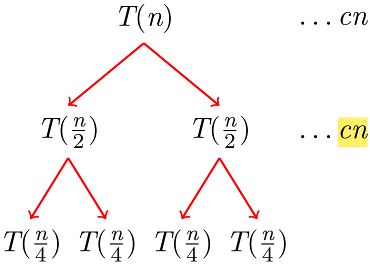
- Ways to analyse a recursion:
  1. **Unrolling the recurrence:** unrolling a few levels to find a pattern, and then sum over all levels;
  2. **Guess and substitution:** guess the solution, substitute it into the recurrence relation, and check whether it works.
  3. **Master theorem**

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

$$\begin{tikzcd}[column sep=small]
 & T(n) \arrow[dl] \arrow[dr] & & \ldots cn \\
T(\frac{n}{2}) & & T(\frac{n}{2}) &
\end{tikzcd}$$

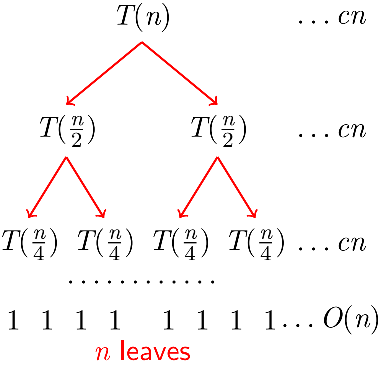
Analysis technique 1: Unrolling the recurrence

- 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.



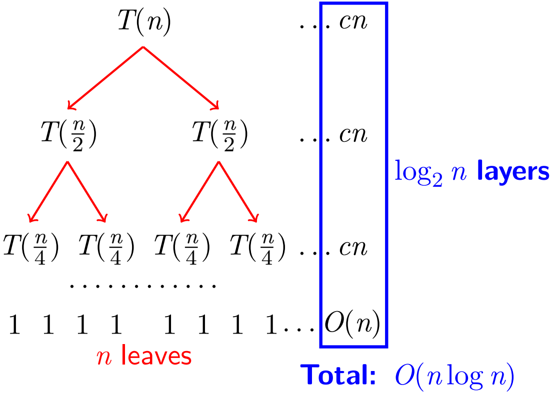
Analysis technique 1: Unrolling the recurrence

- 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

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

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.**

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

Master theorem

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.

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

Here $c > 0$ represents a constant. □

Master theorem: examples

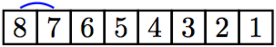
- Example 1: $T(n) = 3\,T(\frac{n}{2}) + O(n)$

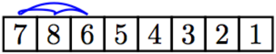
  $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

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

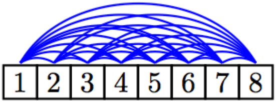
  $T(n) = O(n^2)$

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

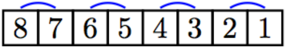




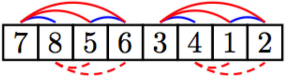
⋮



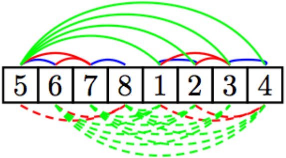
InsertSort: 28 ops



MergeSort step 1: 4 ops



MergeSort step 2: 4 ops, save: 4



MergeSort step 3: 4 ops, save: 12

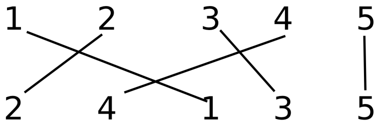
COUNTINGINVERSION: to count inversions in an array of n integers

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

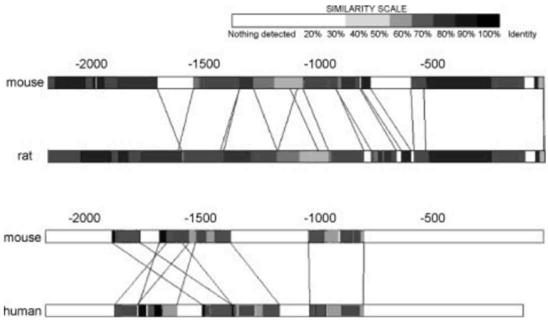


Figure 1: Sequence comparison of the 5' flanking regions of mouse, rat and human ERβ.

Reference: In vivo function of the 5' flanking region of mouse estrogen receptor β 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,

$$X: \ 1\ 3\ 4\ 2\ 5$$
$$Y: \ 1\ 4\ 5\ 2\ 3$$

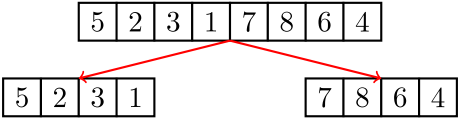
$W_1 = 2$ when $k = 3$. Much better than Pearson CC, et al.

Ref: R. Wang, M. Waterman, H. Huang, PNAS, 2014

COUNTINGINVERSION problem

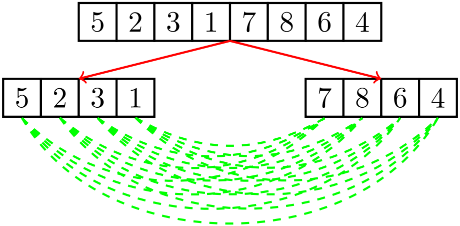
- Solution: index pairs. The possible solution space has a size of $O(n^2)$.
- Brute-force: $O(n^2)$ (Examining all index pairs (i, j)).
- Can we design a better algorithm?

- 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.



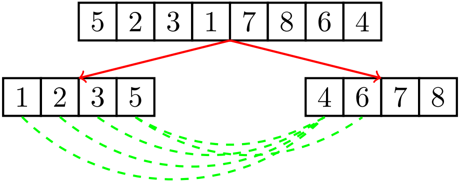
Combine strategy 1

- **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

Sort-and-Count(A)

Divide A into two sub-sequences L and R;
$(RC_L, L) = $ Sort-and-Count(L);
$(RC_R, R) = $ Sort-and-Count(R);
$(C, A) = $ Merge-and-Count(L, R);
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)

```
RC = 0; i = 0; j = 0;
for k = 0 to ||L|| + ||R|| - 1 do
    if L[i] > R[j] then
        A[k] = R[j];
        j++;
        RC += (||L|| - i);
        if all elements in R have been copied then
            Copy the remainder elements from L into A;
            break;
        end if
    else
        A[k] = L[i];
        i++;
        if all elements in L have been copied then
            Copy the remainder elements from R into A;
            break;
        end if
    end if
end for
return (RC, A);
```

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

QuickSort algorithm [C. A. R. Hoare, 1962]

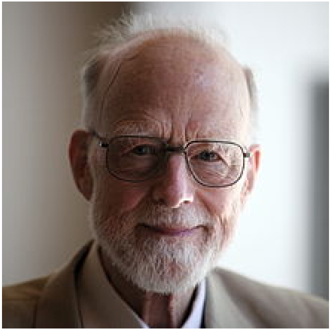


Figure 2: Sir Charles Antony Richard Hoare, 2011

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.

- **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.

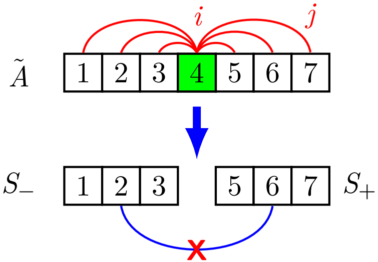
$$T(n) = 2T(\tfrac{n}{2}) + O(n) = O(n \log n)$$

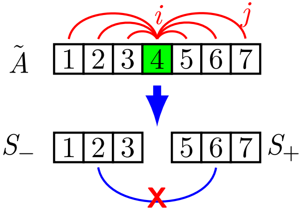
- **Most cases:** instead of selecting the median exactly, we can select a **nearly-central pivot** with high probability. We claim that the expected running time is still

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

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

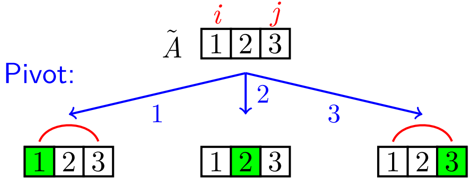
$$\begin{aligned} E[X] &= E[\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}] \\ &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} E[X_{ij}] \\ &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \Pr(\tilde{A}[i] \text{ is compared with } \tilde{A}[j]) \end{aligned}$$

- **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

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

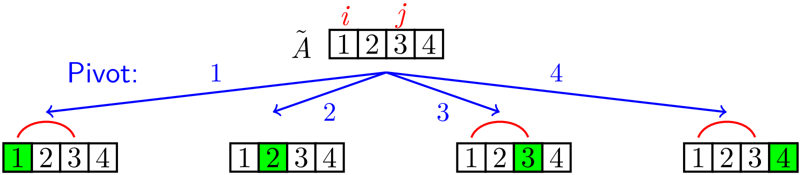
- Here k is defined as $k = j - i$.

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

	i		j					
$\tilde{A}$	1	2	3	4	5	6	$\cdots$	n

- 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:

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

ModifiedQuickSort: easier to analyze

ModifiedQuickSort(A)

while TRUE **do**
    Choose a pivot $A[j]$ **uniformly at random**;
    $S_- = \{\}; S_+ = \{\}$;
    for $i = 0$ to $n-1$ **do**
        Put $A[i]$ in S_- if $A[i] < A[j]$;
        Put $A[i]$ in S_+ if $A[i] \geq A[j]$;
    end for
    if $\|S_+\| \geq \frac{n}{4}$ and $\|S_-\| \geq \frac{n}{4}$ **then**
        break; //A fixed proportion of elements fall both below and above the pivot;
    end if
end while
ModifiedQuickSort(S_+);
ModifiedQuickSort(S_-);
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".

$$\tilde{A} \quad \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline \tilde{A}_0 & \cdots & \tilde{A}_{\frac{n}{4}} & \cdots & \overset{\text{best pivot}}{\tilde{A}_{\frac{n}{2}}} & \cdots & \tilde{A}_{\frac{3n}{4}} & \cdots & \tilde{A}_{n-1} \\ \hline \end{array}$$

$$\underbrace{\tilde{A}_{\frac{n}{4}} \ \cdots \ \tilde{A}_{\frac{n}{2}} \ \cdots \ \tilde{A}_{\frac{3n}{4}}}_{\text{good pivots}}$$

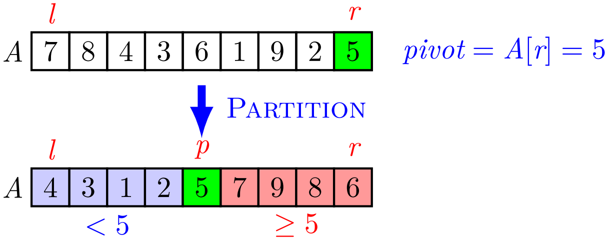
- It is easy to obtain a **nearly central pivot**:
  - $\Pr($select the **centroid** as pivot $) = \frac{1}{n}$
  - $\Pr($select a **nearly central element** as pivot$) = \frac{1}{2}$
  - Thus $E(\#\texttt{WHILE}) = 2$, i.e., the expected time of finding a nearly central pivot is $2n$.
- **Nearly central 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)

```
if l < r then
    p = Partition(A, l, r) //Use A[r] as pivot;
    QuickSort(A, l, p − 1);
    QuickSort(A, p + 1, r);
end if
```

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



	l			i			j		r
A	4	3	1	8	6	7	9	2	5
		< 5			≥ 5				

$pivot = A[r] = 5$

- 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)

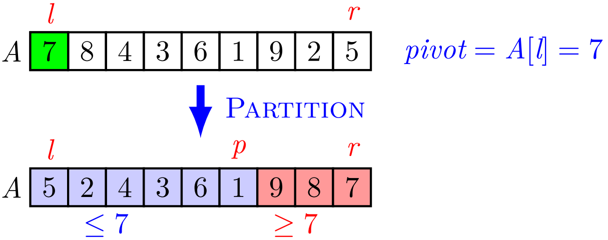
```
pivot = A[r]; i = l;
for j = l to r − 1 do
    if A[j] < pivot then
        Swap A[i] with A[j];
        i++;
    end if
end for
Swap A[i] with A[r]; //Put pivot in its correct position
return i;
```

Hoare's in-place algorithm [1961]

QuickSort(A, l, r)

if $l < r$ **then**
  $p =$ Partition(A, l, r) //Use $A[l]$ as pivot;
  QuickSort(A, l, p); //Reason: $A[p]$ might not be at its correct position
  QuickSort($A, p+1, r$);
end if

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



Hoares' PARTITION procedure

	l				i	j			r
A	5	2	4	3	6	1	9	8	7
		≤ 7						≥ 7	

$pivot = A[l] = 7$

- 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)

```
i = l − 1; j = r + 1; pivot = A[l];
while TRUE do
    repeat
        j = j − 1; //From right to left, find the first element ≤ pivot
    until A[j] ≤ pivot or j == l ;
    repeat
        i = i + 1; //From left to right, find the first element ≥ pivot
    until A[i] ≥ pivot or i == r;
    if j ≤ i then
        return j;
    end if
    Swap A[i] with A[j];
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.

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

INPUT:
An array $A = [A_0, A_1, ..., A_{n-1}]$, 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.

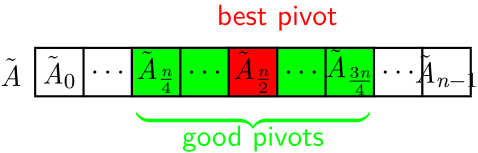
Applying the general DIVIDE AND CONQUER paradigm

SELECT(A, k)

Choose an element A_i from A as a pivot;
$S_+ = \{\}$; $S_- = \{\}$;
for all element A_j in A **do**
    if $A_j > A_i$ **then**
        $S_+ = S_+ \cup \{A_j\}$;
    else
        $S_- = S_- \cup \{A_j\}$;
    end if
end for
if $|S_-| = k - 1$ **then**
    return A_i;
else if $|S_-| > k - 1$ **then**
    return SELECT(S_-, k);
else
    return SELECT$(S_+, k - |S_-| - 1)$;
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.
$$T(n) = T(\tfrac{n}{2}) + O(n) = O(n)$$
- 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.

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

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

$\textsc{Select}(A, k)$

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

Analysis

	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:

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

Actually it takes at most $24n$ comparisons.

BFPRT algorithm: an in-place implementation

SELECT(A, l, r, k)

```
while TRUE do
    if l == r then
        return l;
    end if
    p = PIVOT(A, l, r); //Use median of medians A[p] as pivot ;
    pos = 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+;
    if (k − 1) == pos then
        return k − 1;
    else if (k − 1) < pos then
        r = pos − 1;
    else
        l = pos + 1;
    end if
end while
```

$\textsc{Pivot}(A, l, r)$: get median of medians

$\textsc{Pivot}(A, l, r)$

if $(r - l) < 5$ **then**
  return $\textsc{Partition5}(A, l, r)$; //Get median for 5 or less elements;
end if
for $i = l$ to r by 5 **do**
  $right = i + 4$;
  if $right > r$ **then**
    $right = r$;
  end if
  $m = \textsc{Partition5}(A, i, right)$; //Get median of a group;
  Swap $A[m]$ and $A[l + \lfloor \frac{i-l}{5} \rfloor]$;
end for
return $\textsc{Select}(A, l, l + \lfloor \frac{r-l}{5} \rfloor, l + \frac{r-l}{10})$;

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

Partition(A, l, r, p)

```
pivot = A[p];
Swap A[p] and A[r]; //Move pivot to the right end;
i = l;
for j = l to r - 1 do
    if A[j] < pivot then
        Swap A[i] and A[j];
        i++;
    end if
end for
Swap A[r] and A[i];
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 $\textsc{Select}(A, 0, 15, 7)$

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** $\textsc{Select}(A, 0, 3, 2)$

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

↓ $\textsc{Partition}(A, 0, 15, 3)$

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

Iteration #2: $\text{SELECT}(A, 0, 9, 7)$

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** $\text{SELECT}(A, 0, 1, 1)$

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

↓ $\text{PARTITION}(A, 0, 9, 1)$

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

Iteration #3: $\textsc{Select}(A, 5, 9, 7)$

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** $\textsc{Select}(A, 5, 5, 1)$

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

↓ $\textsc{Partition}(A, 5, 9, 5)$

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.

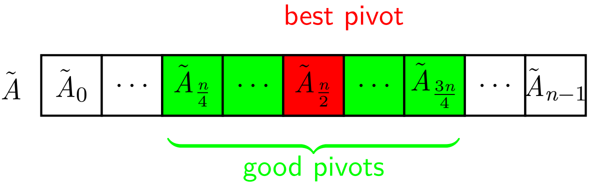
Strategy 2: QuickSelect algorithm randomly select an element as pivot

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

QuickSelect(A, k)

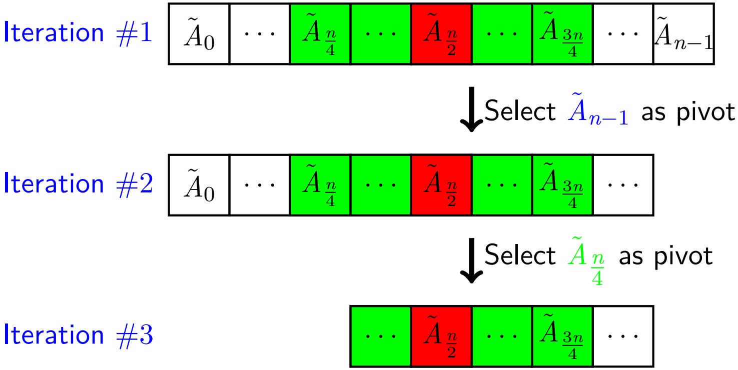
Choose an element A_i from A **uniformly at random**;
$S_+ = \{\}$;
$S_- = \{\}$;
for all element A_j in A **do**
    if $A_j > A_i$ **then**
        $S_+ = S_+ \cup \{A_j\}$;
    else
        $S_- = S_- \cup \{A_j\}$;
    end if
end for
if $|S_-| = k - 1$ **then**
    return A_i;
else if $|S_-| > k - 1$ **then**
    return QuickSelect(S_-, k);
else
    return QuickSelect$(S_+, k - |S_-| - 1)$;
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



- 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

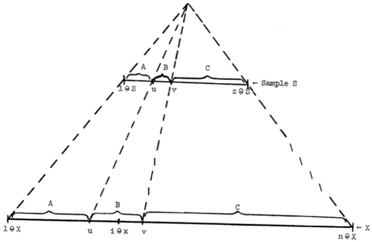
The expected running time of QUICKSELECT *is* $O(n)$.

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, $X = X_0 + X_1 + ...$
- 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 $E[X] = E[X_0 + X_1 + ...] \leq \sum_j 2cn(\frac{3}{4})^{j} \leq 8cn$.

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-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 Flyod-Rivest algorithm called LAZYSELECT, which is much easier to analyze.

LazySelectMedian(A)

1: Randomly sample r elements (with replacement) from $A = [A_0, A_1, A_2, ..., A_{n-1}]$. 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:

$$\begin{aligned} L &= \{A_i : A_i < u\}; \\ M &= \{A_i : u \leq A_i \leq v\}; \\ H &= \{A_i : A_i > v\}; \end{aligned}$$

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

$\downarrow$ **Sample $r = 8$ elements**

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

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

$\downarrow$ **Divide A into L, M, and H**

L:

1	3	2

M (u = 4, v = 13):

4	5	6	7	8	9	10	11	12	13

H:

15	16	14

Return 8 as the median of A

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

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

- 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-comlexity analysis: linear time

LazySelectMedian(A)

1: Randomly sample r elements (with replacement) from $A = [A_0, A_1, A_2, ..., A_{n-1}]$. 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** $O(rlogr) = o(n)$ **time**

3: Divide A into three dis-joint subsets: **//Take** $2n$ **steps**

$$
\begin{aligned}
L &= \{A_i : A_i < u\}; \\
M &= \{A_i : u \leq A_i \leq v\}; \\
H &= \{A_i : A_i > v\};
\end{aligned}
$$

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* $2n + o(n)$.

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

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

- 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 $X = x_1 + x_2 + ...x_r$ 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

$$
\begin{aligned}
\Pr(|L| > \tfrac{n}{2}) &\leq \Pr(X \geq \tfrac{1+\delta}{2}r) && (2)\\
&= \tfrac{1}{2}\Pr(|X - E(X)| \geq \tfrac{\delta}{2}r) && (3)\\
&\leq \frac{1}{2}\frac{\sigma^2(X)}{(\frac{\delta}{2}r)^2} && (4)\\
&= \frac{1}{2}\frac{1}{\delta^2 r} && (5)\\
&= \frac{1}{2}n^{-\frac{1}{4}} && (6)
\end{aligned}
$$

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

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

Grade-school algorithm

- 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?

Kolmogorov's conjecture

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

- 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:**

$$\begin{aligned} xy &= (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) && (7) \\ &= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l && (8) \end{aligned}$$

MULTIPLICATION problem: Trial 1

- Example:
  - Objective: to calculate 12×34
  - $x = 12 = 1 \times 10 + 2$, $y = 34 = 3 \times 10 + 4$
  - $x \times y = (1 \times 3) \times 10^2 + ((1 \times 4) + (2 \times 3)) \times 10 + 2 \times 4$
- 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.

- 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 $x_h y_h$, $x_l y_l$, and $P = (x_h + x_l)(y_h + y_l)$;
  3. **Combine:**

$$
\begin{aligned}
xy &= (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) && (9)\\
&= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l && (10)\\
&= x_h y_h 2^n + (P - x_h y_h - x_l y_l) 2^{\frac{n}{2}} + x_l y_l && (11)
\end{aligned}
$$

Karatsuba-Ofman algorithm

- Example:
  - Objective: to calculate 12×34
  - $x = 12 = 1 \times 10 + 2$, $y = 34 = 3 \times 10 + 4$
  - $P = (1 + 2) \times (3 + 4)$
  - $x \times y = (1 \times 3) \times 10^2 + (P - 1 \times 3 - 2 \times 4) \times 10 + 2 \times 4$
- Note: 3 sub-problems, 6 additions, and 2 shifts;
- Time-complexity:
  $T(n) = 3T(\frac{n}{2}) + cn = O(n^{\log_2 3}) = O(n^{1.585})$
- 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.

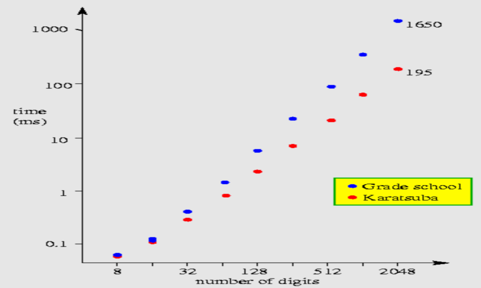


Figure 4: See https://www.cs.cmu.edu/ cburch/251/karat/ for more details.

Extension: Fast Division

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

$$
\begin{aligned}
x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} && (12)\\
&= x_i - \frac{t - \frac{1}{x_i}}{\frac{1}{x_i^2}} && (13)\\
&= -t \times x_i^2 + 2x_i && (14)
\end{aligned}
$$

  - Convergence speed: quadratic, i.e. $\epsilon_{i+1} \leq M\epsilon_i^2$, where M is a supremum of a ratio, and ϵ_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	ϵ_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 ϵ_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: 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:**

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

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

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

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

Strassen algorithm, 1969

Figure 5: Volker Strassen, 2009

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

- 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:**

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

$$
\begin{aligned}
P_1 &= A_{11} \times (B_{12} - B_{22}) && (15)\\
P_2 &= (A_{11} + A_{12}) \times B_{22} && (16)\\
P_3 &= (A_{21} + A_{22}) \times B_{11} && (17)\\
P_4 &= A_{22} \times (B_{21} - B_{11}) && (18)\\
P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) && (19)\\
P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) && (20)\\
P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) && (21)
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= P_4 + P_5 + P_6 - P_2 && (22)\\
C_{12} &= P_1 + P_2 && (23)\\
C_{21} &= P_3 + P_4 && (24)\\
C_{22} &= P_1 + P_5 - P_3 - P_7 && (25)
\end{aligned}
$$

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

Advantages

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

[1] This heavily depends on the system, including memory access property, hardware design, etc.

Shortcomings

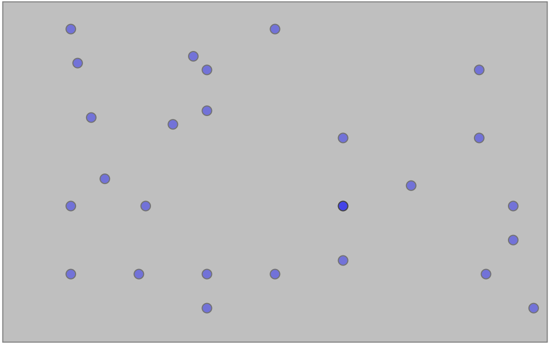
- Strassen algorithm performs better than grade-school method only for large n.
- The reduction in the number of arithmetic operations however comes at the price of a somewhat reduced numerical stability,
- The algorithm also requires significantly more memory compared to the naive algorithm.

Fast matrix multiplication

- multiply two 2×2 matrices: 7 scalar sub-problems: $O(n^{\log_2 7}) = O(n^{2.807})$ [Strassen 1969]
- multiply two 2×2 matrices: 6 scalar sub-problems: $O(n^{\log_2 6}) = O(n^{2.585})$ (impossible)[Hopcroft and Kerr 1971]
- multiply two 3×3 matrices: 21 scalar sub-problems: $O(n^{\log_3 21}) = O(n^{2.771})$ (impossible)
- multiply two 20×20 matrices: 4460 scalar sub-problems: $O(n^{\log_{20} 4460}) = O(n^{2.805})$
- multiply two 48×48 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

INPUT: n points in a plane;
OUTPUT: The pair with the least Euclidean distance.



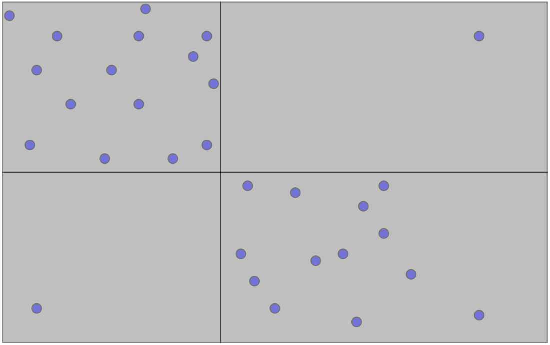
About ClosestPair problem

- 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)

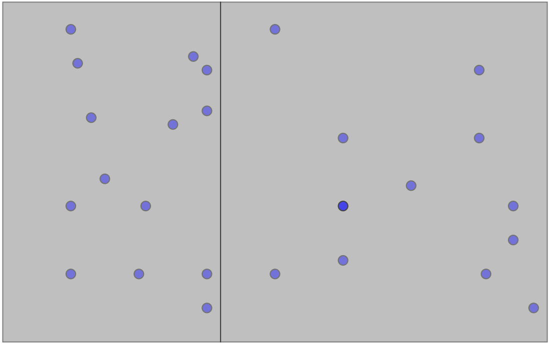
- Divide and Conquer: divide into 4 subsets.



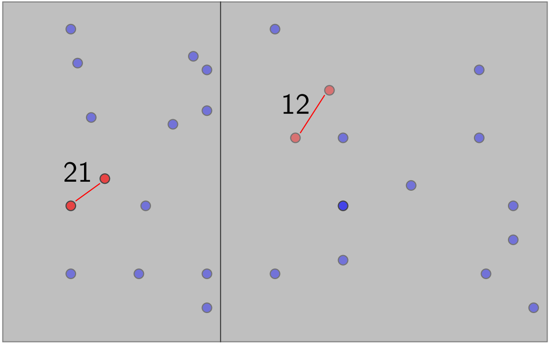
- Difficulties:
  - The subsets might be unbalanced — we cannot guarantee that each subset has approximately $\frac{n}{4}$ points.
  - Since the closest pair might lie in different subsets, we need to consider all $\binom{4}{2}$ pairs of subsets to avoid missing the closest pair, thus complicating the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

Trial 2: Divide into 2 halves

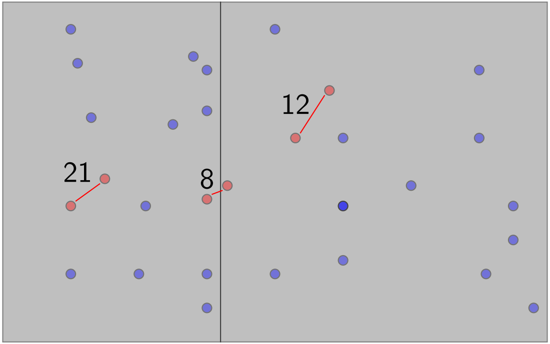
- **Divide:** divide into two halves with equal size.
  It is easy to achieve this through sorting by x coordinate first, and then select the median as pivot.



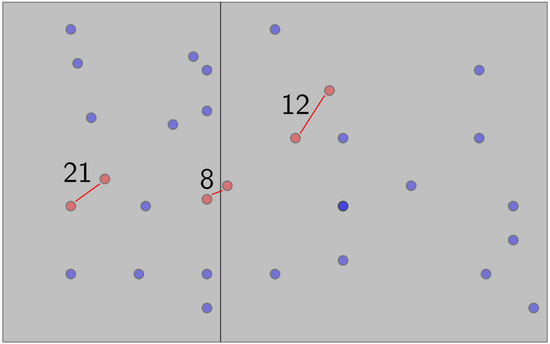
- **Divide:** dividing into two (roughly equal) subsets;
- **Conquer:** finding closest pairs in each half;



- **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.



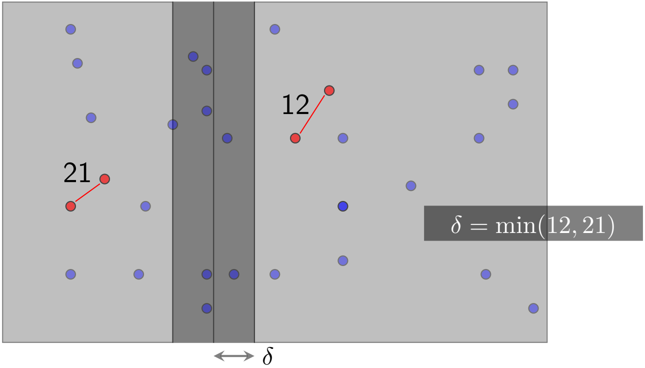
Two types of redundancy



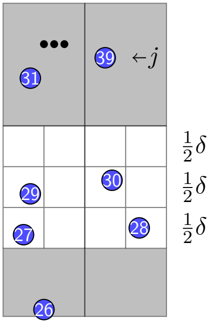
- It is redundant to calculate distance between p_i and p_j if
  - $|x_i - x_j| \geq 12$, or
  - $|y_j - y_j| \geq 12$

Remove redundancy of type 1

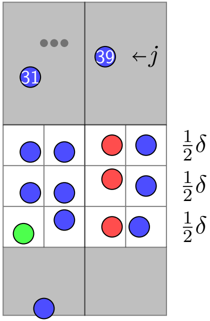
- **Observation 1:**
  - The third type occurs in **a narrow strip** only; thus, it suffices to check point pairs within the 2δ-strip.
  - Here, δ is the minimum of $\textsc{ClosestPair}(\textsc{LeftHalf})$ and $\textsc{ClosestPair}(\textsc{RightHalf})$.



- **Observation 2:**
  - Moreover, it is unnecessary to explore **all** point pairs within the 2δ-strip. In fact, for each point p_i, it suffices to examine 11 points for possible closest partners.
  - Let's divide the 2δ-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 δ and thus cannot form closest pair.
  - Example: For point 27, it suffices to search within 2 rows for possible closest partners ($< \delta$).

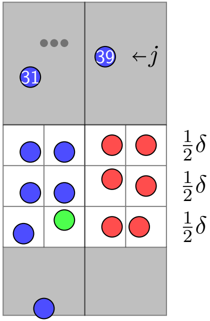


To detect potential closest pair: Case 1



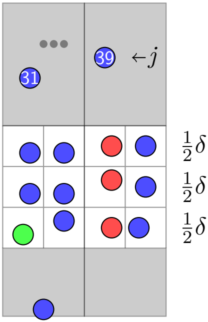
- Green: point i;
- Red: the possible closest partner (distance $< \delta$) of point i;

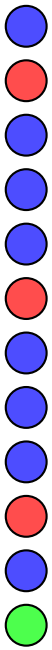
To detect potential closest pair: Case 2



- 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$(p_l, ..., p_r)$

//To find the closest points within $(p_l, ..., p_r)$. Here we assume that $p_l, ..., p_r$ have already been sorted according to x-coordinate;

if $r - l == 1$ **then**

    return $d(p_l, p_r)$;

end if

Use the x-coordinate of $p_{\lfloor \frac{l+r}{2} \rfloor}$ to divide $p_l, ..., p_r$ into two halves;

$\delta_1 =$ ClosestPair(LeftHalf); $//T(\frac{n}{2})$

$\delta_2 =$ ClosestPair(RightHalf); $//T(\frac{n}{2})$

$\delta = \min(\delta_1, \delta_2)$;

Sort points within the 2δ wide strip by y-coordinate; $//O(n \log n)$

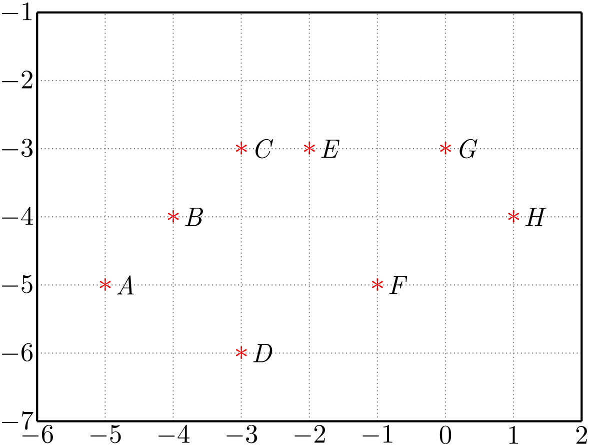
Scan points in y-order and calculate distance between each point with its next 11 neighbors. Update δ if finding a distance less than δ; $//O(n)$

- Find closest pair within $p_0, p_1, ..., p_{n-1}$: ClosestPair$(p_0, ..., p_{n-1})$
- Time-complexity: $T(n) = 2T(\frac{n}{2}) + O(n \log n) = O(n \log^2 n)$.

ClosestPair algorithm: improvement

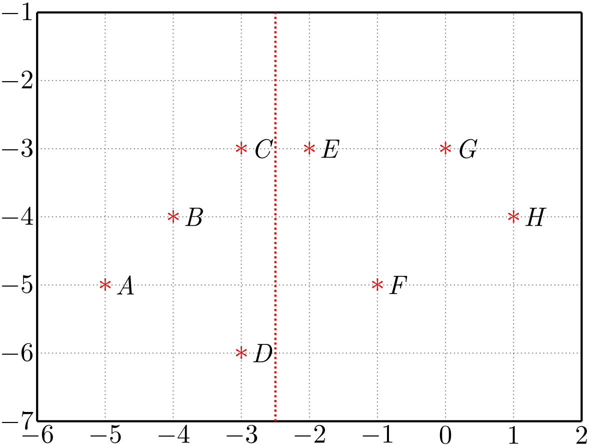
- Note that if the points within the 2δ-wide strip have no structure, we have to sort them from the scratch, which will take $O(n \log n)$ time.
- Let's try to introduce some structure into the points within the 2δ-wide: If the point within each δ-wide strip were already sorted, it is relatively easy to sort the points within the 2δ-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



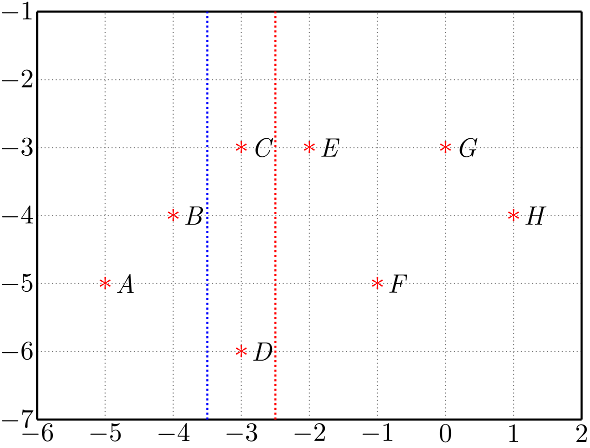
- Objective: to find the closest pair among these 8 points.

ClosestPair: an example with 8 points

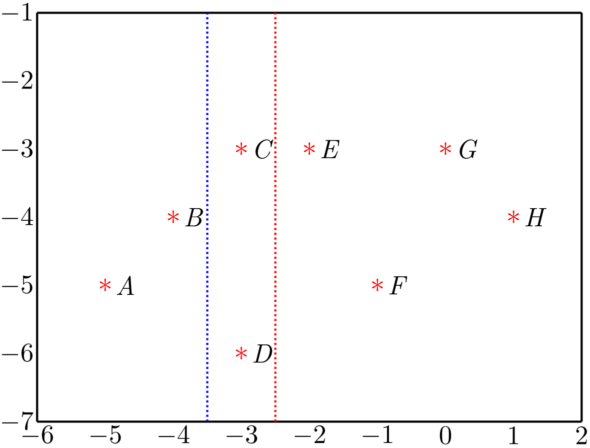


- Objective: to find the closest pair among these 8 points.

Left half: A, B, C, D

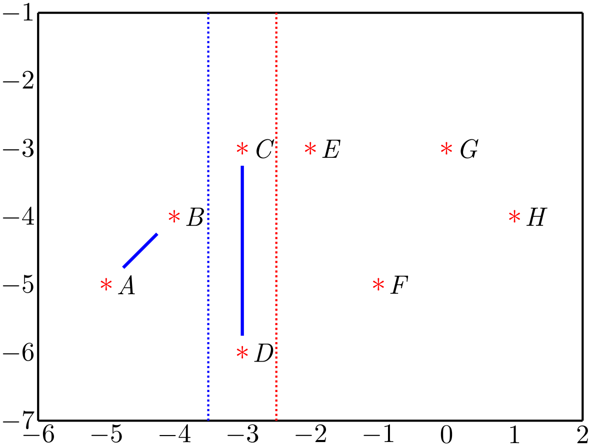


Left half: A, B, C, D



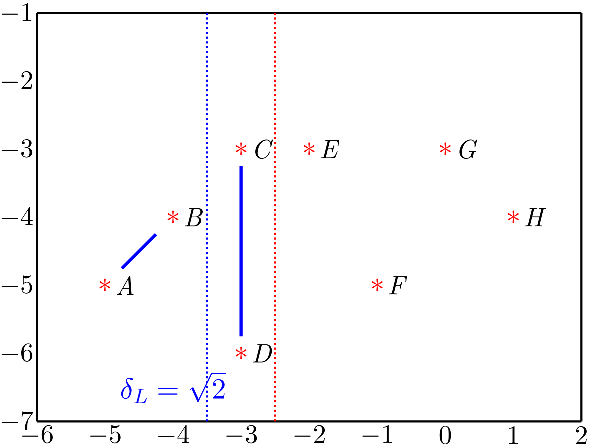
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Left half: A, B, C, D



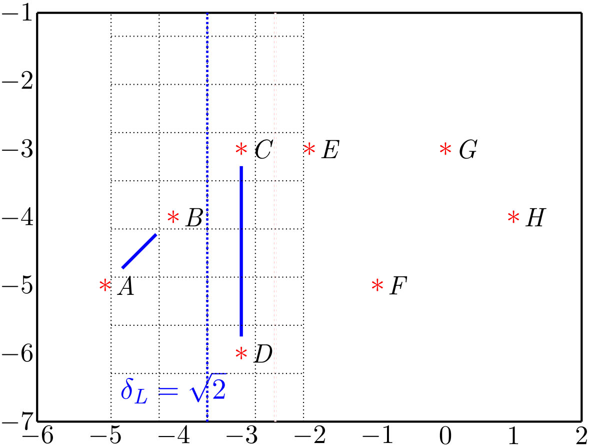
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Left half: A, B, C, D

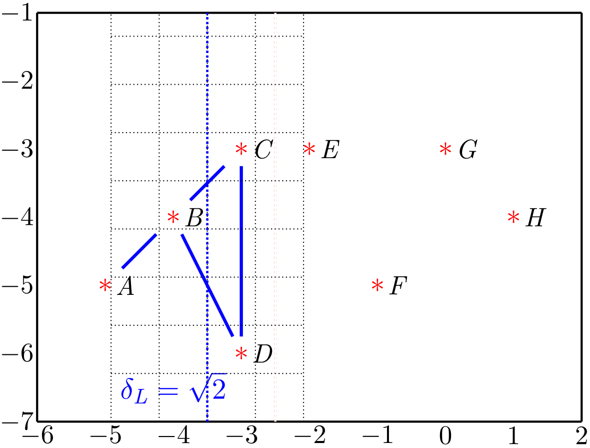


- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Left half: A, B, C, D

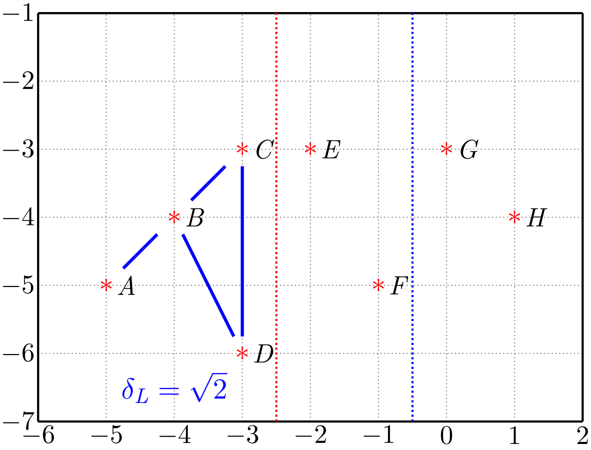


- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

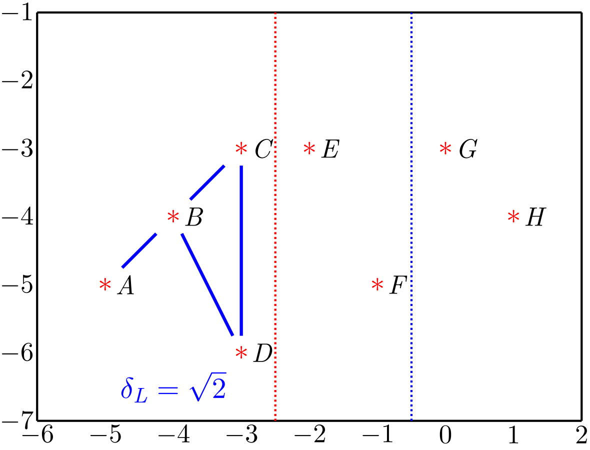


- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Right half: E, F, G, H

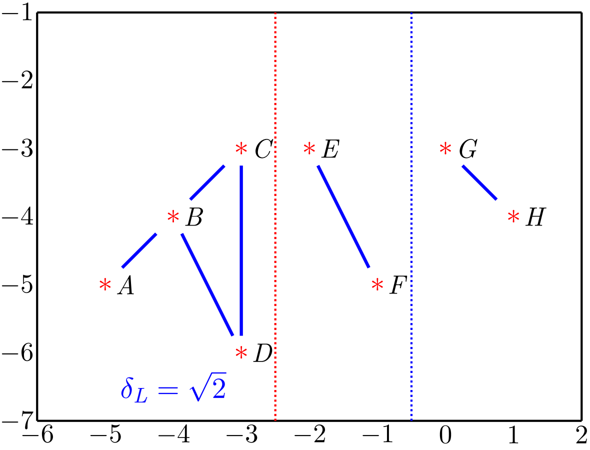


Right half: E, F, G, H



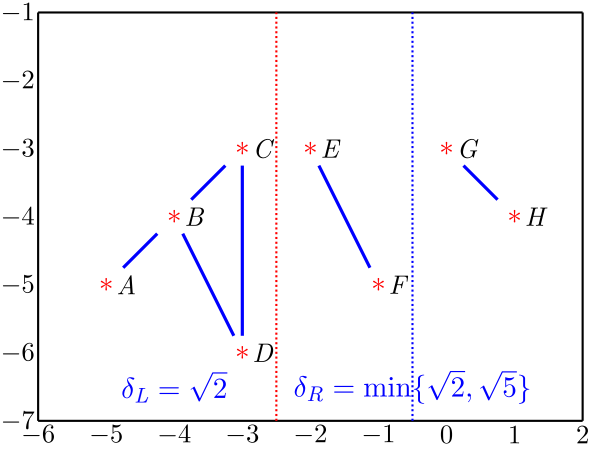
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



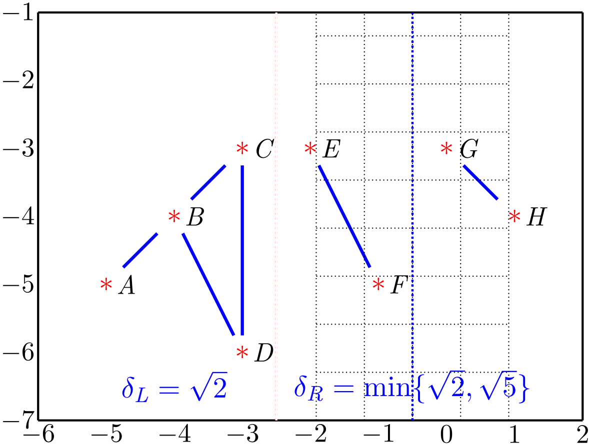
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



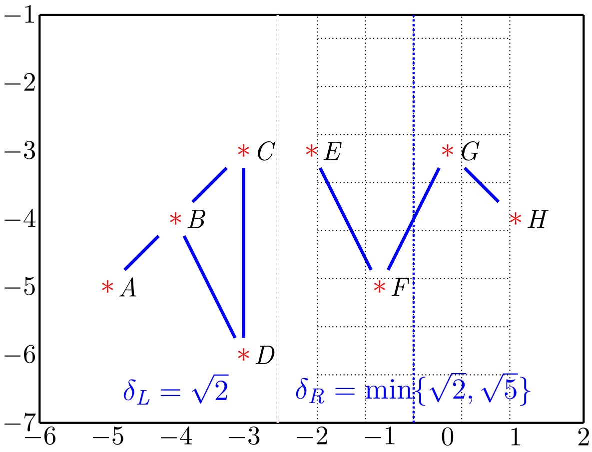
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



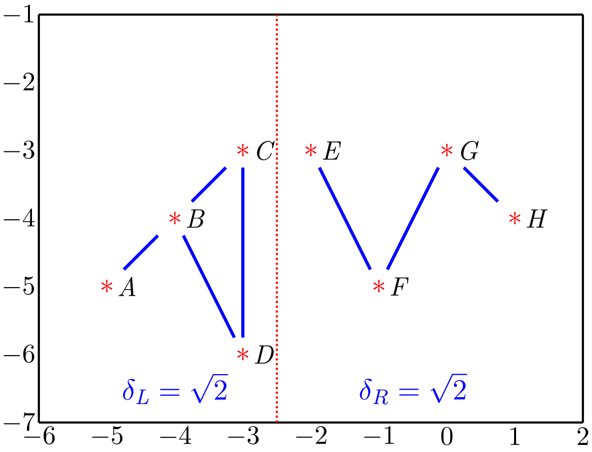
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



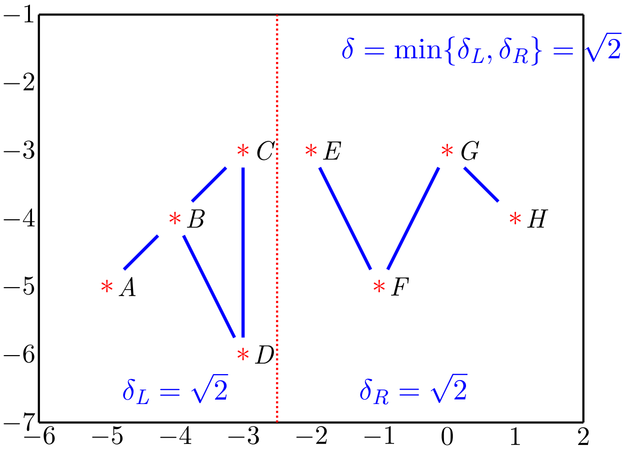
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

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



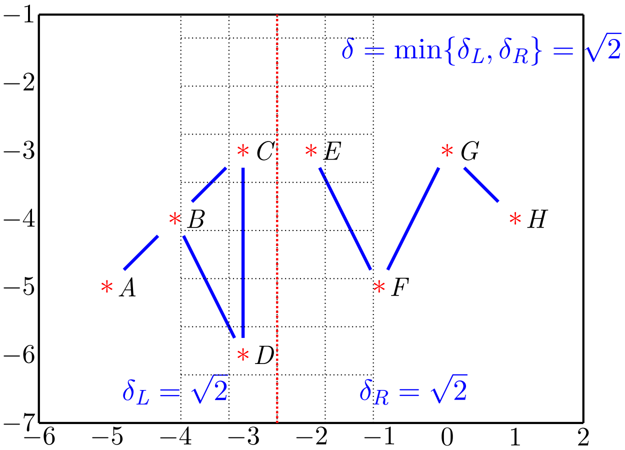
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

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



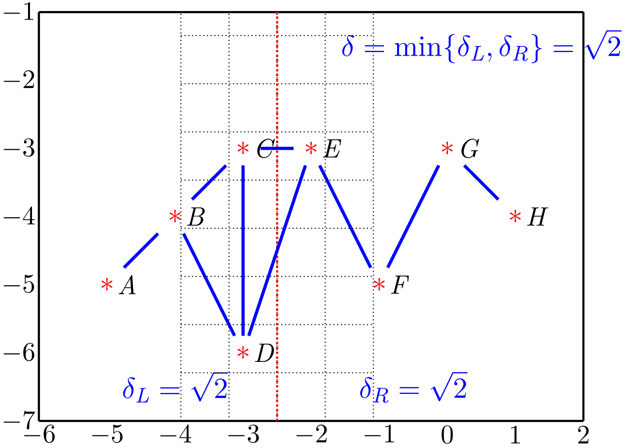
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

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



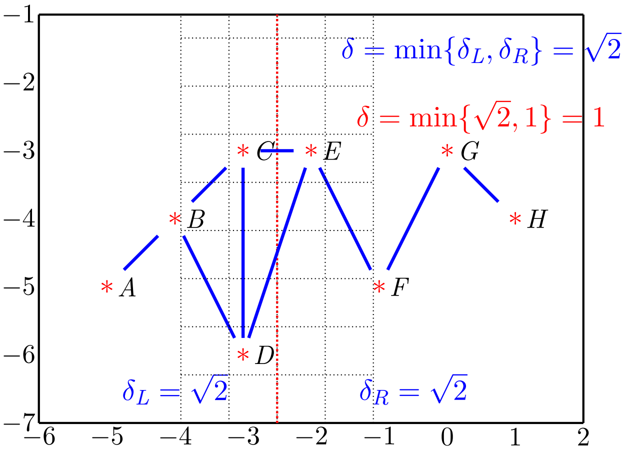
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

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



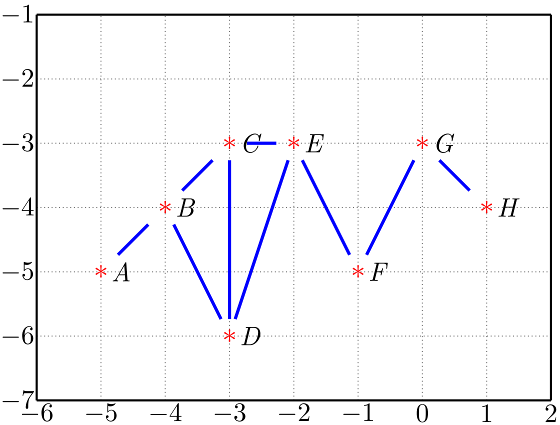
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

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 2δ-strip.
  - although two points appear in the same 2δ-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

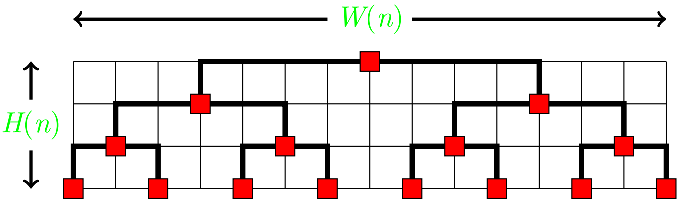
We can perform bottom-up hierarchical clustering, for any cluster distance function computable in constant time from the distances between subclusters, in total time $O(n^2)$. We can perform median, centroid, Ward, or other bottom-up clustering methods in which clusters are represented by objects, in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

(See Eppstein 1998 for details.)

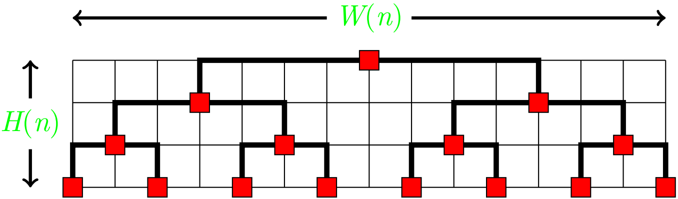
VLSI embedding: to embed a tree

Embedding a tree

INPUT: Given a binary tree with n node;
OUTOUT: Embedding the tree into a VLSI with minimum area.



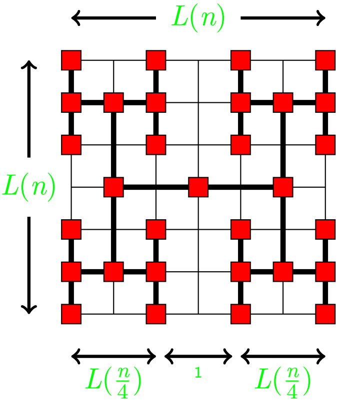
- Let's divide into 2 sub-trees, each with a size of $\frac{n}{2}$.



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