CS711008Z Algorithm Design and Analysis

Lecture 5. Basic algorithm design technique: Divide-and-Conquer [1]

Dongbo Bu

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

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

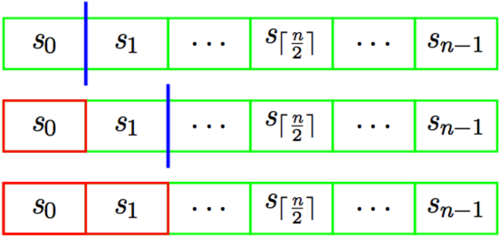
[1] The slides are prepared based on Chapter 2 and 28 of Introduction to algorithms, Chapter 5 of Algorithm design. Some slides are excerpted from the slides by Kevin Wayne with permission.

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 algorithm, SELECTION problem;
- Remarks:
  1. Divide-and-conquer technique is usually serving to reduce the running time though the brute-force algorithm is already polynomial-time. Say $O(n^2) \Rightarrow O(n\log(n))$ for CLOSESTPAIR problem.
  2. This technique is especially powerful when combined with randomization technique.

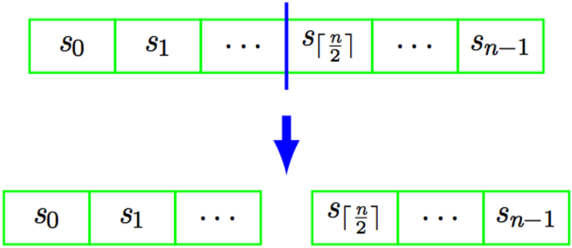
If a problem can be reduced into smaller sub-problems I

- We can consider two possible divide-and-conqueror strategies:
  1. **Incremental**: to solve the original problem, it suffices to solve a smaller sub-problem; thus the problem is shrunk step-by-step. In other words, a feasible solution can be constructed step-by-step. Say Gale-Shapley algorithm for Stable Matching problem, where a **stable, partial** matching is maintained at each step.



If a problem can be reduced into smaller sub-problems II

2. **Divide into two halves**: the original problem is decomposed into several independent sub-problems; thus, a feasible solution to the original problem can be constructed by assembling the solutions to independent sub-problems.



On what problems can we divide and conqueror?

- Suppose a problem is related to the following data structure, perhaps we can try to divide it into sub-problems.
  - An array with n elements;
  - A set of n elements;
  - A tree
  - A graph
  - ......

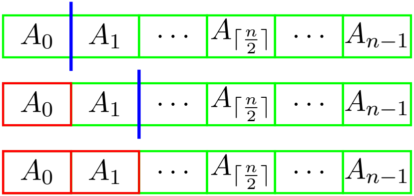
SORT problem: to sort an array of n integers

SORT problem

INPUT: An array of n integers, say $A[0..n-1]$;
OUTPUT: the items of A in increasing order;

Trial 1: Basic idea of INCREMENTAL strategy

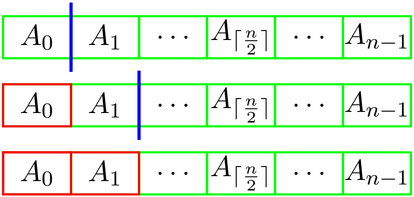
- Basic idea: At each step of the execution, we have **a partial solution** in its correct order, i.e., $A[0..j-1]$ has already been correctly sorted, and the objective is to put $A[j]$ in its correct position. This way, the final **complete solution** is constructed step-by-step.



Trial 1: InsertionSort algorithm

InsertionSort(A)

```
for j = 0 to n − 1 do
    key = A[j];
    i = j − 1;
    while  i ≥ 0 and A[i] > key do
        A[i + 1] = A[i];
        i − −;
    end while
    A[i + 1] = key;
end for
```



(See extra slides for a demo)

Trial 1: Analysis of InsertionSort algorithm

- Worst-case: if $A[0..n-1]$ has already been sorted.
- Time complexity: $O(n^2)$.
- In fact, the running time is $T(n) = T(n-1) + cn = O(n^2)$.

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

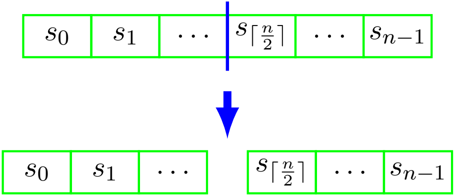
InsertSort: 28 ops

Trial 2: divide-and-conquer idea (MERGESORT algorithm [J. von Neumann, 1945, 1948])

Figure 1: von Neumann in 1940s

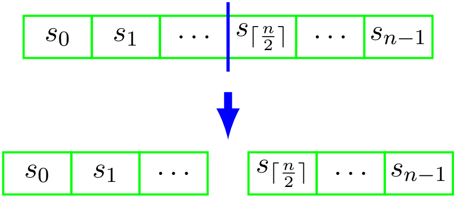
Trial 2: divide-and-conquer idea (MERGESORT algorithm)

- Key observation: the problem can be decomposed into two **independent sub-problems.**



1. **Divide** divide the n-element sequence into two subsequences; each has $n/2$ elements;
2. **Conquer** sort the subsequences recursively by calling MERGESORT itself;
3. **Combine** merge the two sorted subsequences to yield the answer to the original problem;

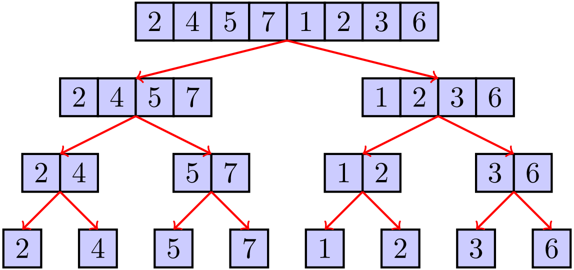
Trial 2: divide-and-conquer idea (MERGESORT algorithm)



MERGESORT(A, l, r)

```
/* To sort part of the array A[l..r]. */
if  l < r  then
    m = (l + r)/2; // m denotes the middle point;
    MERGESORT( A, l, m );
    MERGESORT( A,m, r);
    MERGE(A, l, m, r); // combining the sorted subsequences;
end if
```

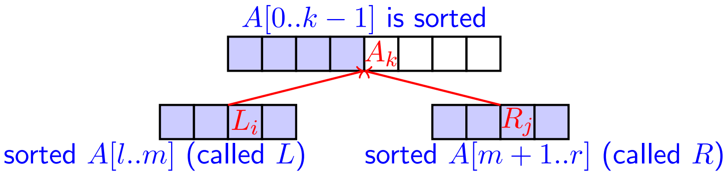
MergeSort algorithm: how to divide?



MergeSort algorithm: how to combine?

Merge (A, l, m, r)

```
/* to merge A[l..m](named as L) and A[m+1..r] (named as R). */
i = 0; j = 0;
for k = l to r do
    if L[i] < R[j] then
        A[k] = L[i];
        i++;
    else
        A[k] = R[j];
        j++;
    end if
end for
```



(See extra slides by K. Wayne.)

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

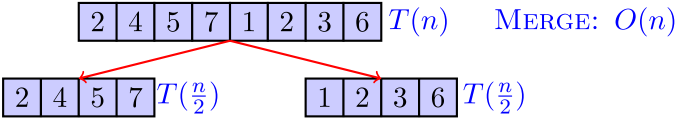
MERGE(A, l, m, r)

/* to 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**
        $A[k] = R[j]$;
        $j++$;
    else
        $A[k] = L[i]$;
        $i++$;
    end if
end for

Time complexity: $O(n)$. (See extra slides for a demo)

- Let $T(n)$ denote the running time on a problem of size n. We have the following recursion:

$$T(n) = \begin{cases} c & n = 2 \\ T(n/2) + T(n/2) + cn & otherwise \end{cases} \tag{1}$$



Time-complexity analysis technique for recursion tree

- Ways to analyse a recursion:
  1. **Unrolling the recurrence to find a pattern:** 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. **Generating function**

Analysis technique 1: Unrolling

- Unrolling the recurrence to find a pattern: unrolling a few levels to find a pattern, and then sum over all levels;

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

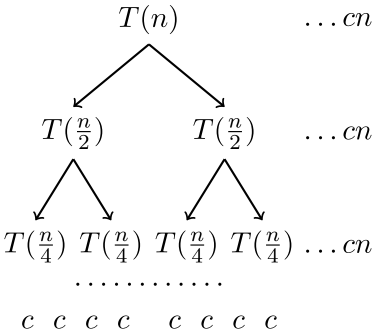
Analysis technique 1: Unrolling

- Unrolling the recurrence to find a pattern: unrolling a few levels to find a pattern, and then sum over all levels;

$$\begin{tikzcd}
 & & & T(n) \arrow[dll] \arrow[drr] & & & & & \ldots cn \\
 & T(\frac{n}{2}) \arrow[dl] \arrow[dr] & & & & T(\frac{n}{2}) \arrow[dl] \arrow[dr] & & & \ldots cn \\
T(\frac{n}{4}) & & T(\frac{n}{4}) & & T(\frac{n}{4}) & & T(\frac{n}{4}) & &
\end{tikzcd}$$

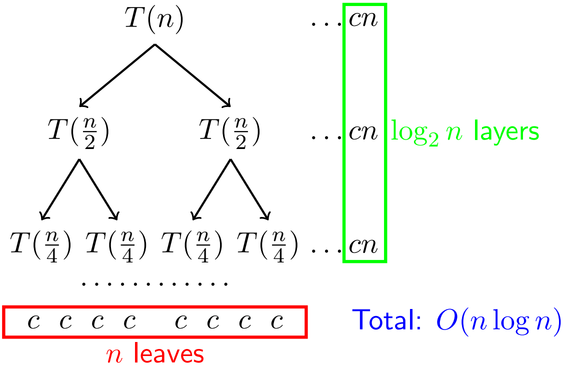
Analysis technique 1: Unrolling

- Unrolling the recurrence to find a pattern: unrolling a few levels to find a pattern, and then sum over all levels;



Analysis technique 1: Unrolling

- Unrolling the recurrence to find a pattern: 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: $T(n) \leq cn\log_2 n$ for all $n \geq 2$;
- Verification:
  - Case $n = 2$: $T(2) = c \leq cn\log_2 n$;
  - Case $n > 2$: Suppose $T(m) \leq cm\log_2 m$ holds for all $m \leq n$. We have

$$\begin{aligned} T(n) &= 2T(n/2) + cn && (2) \\ &\leq 2c(n/2)\log_2(n/2) + cn && (3) \\ &= 2c(n/2)\log_2 n - 2c(n/2) + cn && (4) \\ &= cn\log_2 n && (5) \end{aligned}$$

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 $T(n) = k \log_b n$, where k, b **will be determined later.**

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

Master theorem

Theorem

Let $T(n)$ be defined by $T(n) = aT(n/b) + f(n)$, then $T(n)$ can be bounded by:

1. *If $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$;*
2. *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a}) \log n$;*
3. *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$, then $T(n) = \Theta(f(n))$. Here, ϵ denotes a small, positive number.*

- Intuition: comparing the costs in **combining steps** (labelled at internal nodes) and the costs in **base cases** (labelled at leaves).

Master theorem: examples

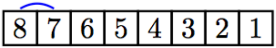
- Example 1: $T(n) \leq 3T(n/2) + cn$ (see a figure)
  $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

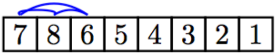
- Example 2: $T(n) \leq 2T(\frac{n}{2}) + cn^2$ (see a figure)

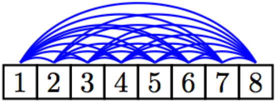
  $T(n) = \sum_{j=0}^{\log n} \frac{cn^2}{2^j} = cn^2 \sum_{j=0}^{\log n} \frac{1}{2^j} = 2cn^2$
  (Note: not $O(n^2 \log n)$)

- Example 3: $T(n) \leq T(n/3) + T(2n/3) + cn$ (see a figure)

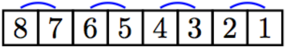
Question: from $O(n^2)$ to $O(n \log n)$, what did we save?



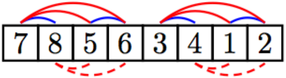




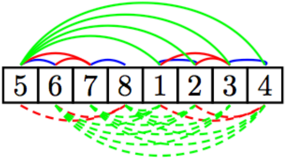
InsertSort: 28 ops



MergeSort step 1: 4 ops



MergeSort step 2: 4 ops, save: 4



MergeSort step 3: 4 ops, save: 12

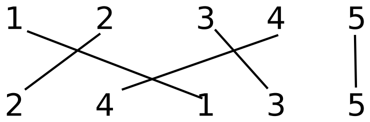
$\textsc{CountingInversion}$: to count inversions in an array of n integers

Practical problems:

1. to identify two persons with similar preference, i.e. ranking books, movies, etc.
2. In case of **meta search engine**, each engine produces a ranked pages for a given query. Comparison of the rankings help identify consensus or similar interests.

Formalized representation

INPUT: n (distinct) numbers $a_1, a_2, ..., a_n$;
OUTPUT: the number of **inversions**, i.e. a pair of indices such that $i < j$ but $a_i > a_j$;



Application 1: Genome comparison

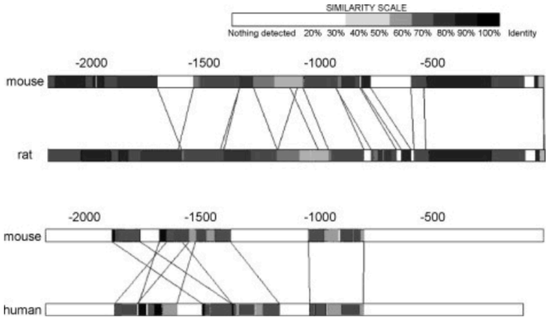


Figure 2: Sequence comparison of the 5' flanking regions of mouse, rat and human ERβ.

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.

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

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

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

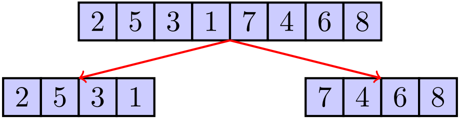
Ref: R. Wang, M. Waterman, H. Huang, PNAS, 2014

CountingInversion problem

- Solution: index pairs. The possible solution space has a size of $O(n^2)$.
- Brute-force: $O(n^2)$ (checking each pair (a_i, a_j)).
- Can we design a better algorithm?

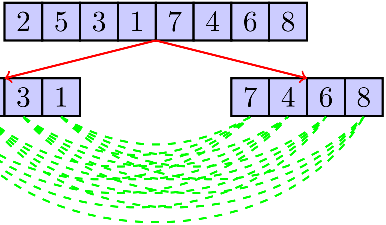
CountingInversion problem

- Key observation: the problem/solution can be divided into subproblems/solutions;
- Divide-and-conquer strategy:
  1. **Divide:** divide into two subproblems: $A[0..n/2]$ and $A[n/2+1...n-1]$;
  2. **Conquer:** counting inversion in each half by calling CountingInversion itself;

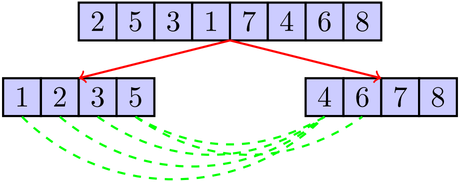


Combine strategy 1

- **Combine:** how to count inversion (a_i, a_j), when a_i and a_j are in different half?
- A simple enumeration will take $\frac{n^2}{4}$ steps. Thus, $T(n) = 2T(\frac{n}{2}) + \frac{n^2}{4} = O(n^2)$.



- **Combine:** how to count inversion (a_i, a_j), when a_i and a_j are in different half?
- A simple enumeration will take $\frac{n^2}{4}$ steps. Thus, $T(n) = 2T(\frac{n}{2}) + \frac{n^2}{4} = O(n^2)$.
- We will get a $O(n \log n)$ algorithm if we can perform "combine" step in $O(n)$ time.
- Thing will be easy provided each half has already been sorted!



(See extra slides for a demo)

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 $(RC = RC_L + RC_R + C, A)$;

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**
        $A[k] = R[j]$;
        $j++$;
        $RC += (\frac{n}{2} - i)$;
    else
        $A[k] = L[i]$;
        $i++$;
    end if
end for
return (RC, A);

Time complexity: $T(n) = O(n \log n)$.

Another view point

- A sorted array has an inversion number of 0.
- Thus, we can treat the sorting process as a process to decrease inversion number to 0.
- Suppose we can record the decrement of inversion number during the sorting process, the sum will be the inversion number.

The general DIVIDE-AND-CONQUER paradigm

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**.
- The divide-and-conquer paradigm contains three steps:
  1. **Divide** a problem into a number of **independent sub-problems**;
     How to divide? at middle-point; divide into two parts with odd- and even- indices; enumerate all cases of dividing point; randomly choose one, etc.
  2. **Conquer** the subproblems by solving them recursively;
  3. **Combine** the solutions to the subproblems into the solution to the original problem;
     Sometimes clever ideas are needed to combine.

QuickSort algorithm: an example of randomly-chosen splitter

QuickSort algorithm [C. A. R. Hoare, 1960]

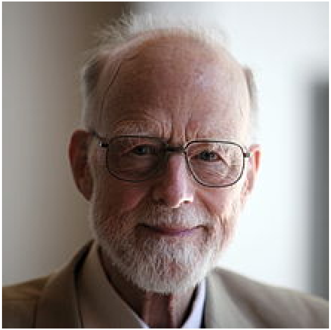


Figure 3: Sir Charles Antony Richard Hoare, 2011

QuickSort (A)

```
Choose a splitter A[j] randomly ;
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] ≥ A[j];
end for
QuickSort(S+);
QuickSort(S−);
Output S−, then A[j], then S+;
```

Note:

- The randomization operation makes this algorithm **simple** (relative to MergeSort algorithm) but **efficient**.
- However, the randomization also incurs a difficulty for analysis: Instead of selecting the median $A_{\lfloor \frac{n}{2} \rfloor}$, we use a randomly chosen A_j as splitter; 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/biggest element at each iteration;

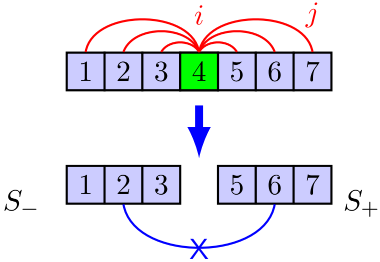
$$T(n) \leq T(n-1) + cn \Rightarrow T(n) = O(n^2)$$

- **Best-case:** if we select the median at each iteration;

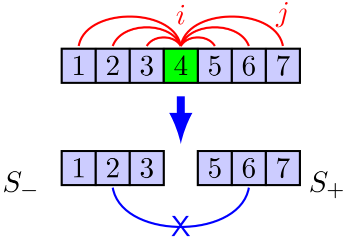
$$T(n) \leq 2T(n/2) + cn \Rightarrow T(n) = O(n \log n)$$

- **Most cases:** instead of selecting the median exactly, we can select a nearly central splitter with high probability. We can prove that the expected running time is still $T(n) = O(n \log n)$.

- Let X denote the number of comparison in Line 5 and 6;
- It is obvious that the running time of QuickSort is $O(n+X)$.
- We have the following two key observations:
- **Observation 1:** $A[i]$ and $A[j]$ are compared at most once for any i and j. (Why?)



Analysis cont'd



- Define index variable $X_{ij} = I\{A[i]$ is compared with $A[j]\}$.
- Thus we have $X = \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} X_{ij}$.

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

- **Observation 2:** $A[i]$ and $A[j]$ are compared iff either $A[i]$ or $A[j]$ is selected as pivot when processing numbers containing $A[i, i+1, ..., j]$. (Why?)
- We have $Pr\{A[i] \text{ is compared with } A[j]\} \leq \frac{2}{j-i+1}$.
- Thus we have:

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

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

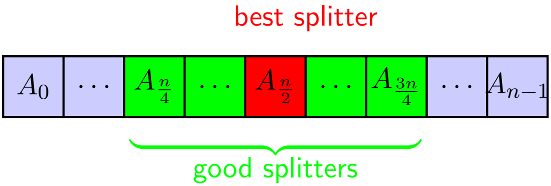
MODIFIED QUICKSORT: easier to analyze

MODIFIEDQUICKSORT(A)

while TRUE **do**
  randomly choose a splitter $A[j]$;
  for $i = 0$ to $n - 1$ **do**
    Put $A[i]$ in S_- if $A[i] < A[j]$;
    Put $A[i]$ in S_+ if $A[i] > A[j]$;
  end for
  if $\|S_+\| > \frac{n}{4}$ and $\|S_-\| > \frac{n}{4}$ **then**
    break;
  end if
end while
MODIFIEDQUICKSORT(S_+);
MODIFIEDQUICKSORT(S_-);
Output S_-, then $A[j]$, and finally S_+;

- Note:
  - This version is slower than the original version since it doesn't run when the splitter is "off-center".
  - MODIFIEDQUICKSORT works when all items are distinct.

MODIFIED QUICKSORT: analysis



1. $Pr\{$select the **centroid** splitter $\} = \frac{1}{n}$
2. $Pr\{$select a **nearly center** splitter$\} = \frac{1}{2}$
3. It is quick to get a nearly center splitter since $E(\#WHILE) = 2$; thus the expected time of this step is $2n$. (Note: $|S| = n$.)
4. The nearly center is good:
   - The recursion tree has a depth of $O(\log_{\frac{4}{3}} n)$.
   - And $O(n)$ work is needed for each level.
   - So $T(n) = O(n \log_{\frac{4}{3}} n)$.

(See extra slides.)

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.

MULTIPLICATION problem: to multiply two n-bits integers

MULTIPLICATION problem

- Problem: multiply two n-bits integer x and y;

```
  12
x 34
 ------
  68
 34
------
 408
```

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

- Conjecture: In 1952, 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:**

$$\begin{aligned} xy &= (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) && (6) \\ &= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l && (7) \end{aligned}$$

- 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: $T(n) = 4T(n/2) + cn \Rightarrow T(n) = O(n^2)$

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.

MULTIPLICATION problem: a clever **conquer** [Karatsuba-Ofman 1962]

Figure 4: 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:**

$$
\begin{aligned}
xy &= (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) && (8)\\
&= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l && (9)\\
&= x_h y_h 2^n + (P - x_h y_h - x_l y_l) 2^{\frac{n}{2}} + x_l y_l && (10)
\end{aligned}
$$

Karatsuba-Ofman algorithm

- Example:
  - Objective: to calculate 12×34
  - $x = 12 = 1 \times 10 + 2$, $y = 34 = 3 \times 10 + 4$
  - $P = (1 + 2) \times (3 + 4)$
  - $x \times y = (1 \times 3) \times 102 + (P - 1 \times 3 - 2 \times 4) \times 10 + 2 \times 4$
- Note: 3 sub-problems, 6 additions, and 2 shifts;
- Time-complexity:
  $T(n) = 3T(n/2) + cn \Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

(See an extra slide)

Theoretical analysis vs. empirical comparisons

- 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, the MULTIPLICATION can be finished in $O(n \log n)$ time.

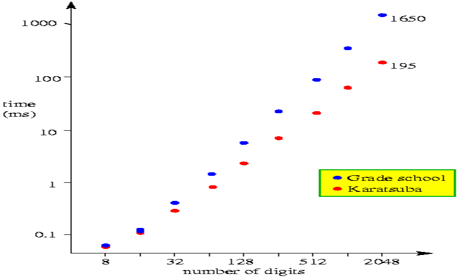


Figure 5: Sun SPARC4, g++ -O4, random input. See

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:

$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} && (11) \\ &= x_i - \frac{t - \frac{1}{x_i}}{\frac{1}{x_i^2}} && (12) \\ &= -t \times x_i^2 + 2x_i && (13) \end{aligned}$$

  - 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: 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:**

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

$$C_{11} = (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \tag{14}$$

$$C_{12} = (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \tag{15}$$

$$C_{21} = (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \tag{16}$$

$$C_{22} = (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \tag{17}$$

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

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

Figure 6: 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:**

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

$$P_1 = A_{11} \times (B_{12} - B_{22}) \tag{18}$$
$$P_2 = (A_{11} + A_{12}) \times B_{22} \tag{19}$$
$$P_3 = (A_{21} + A_{22}) \times B_{11} \tag{20}$$
$$P_4 = A_{22} \times (B_{21} - B_{11}) \tag{21}$$
$$P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) \tag{22}$$
$$P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \tag{23}$$
$$P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12}) \tag{24}$$

$$C_{11} = P_4 + P_5 + P_6 - P_2 \tag{25}$$
$$C_{12} = P_1 + P_2 \tag{26}$$
$$C_{21} = P_3 + P_4 \tag{27}$$
$$C_{22} = P_1 + P_5 - P_3 - P_7 \tag{28}$$

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

Advantages

- For large n, Strassen algorithm is faster than grade-school method. [2]
- Strassen algorithm can be used to solve other problems, say matrix inversion, determinant calculation, finding triangles in graphs, etc.
- Gaussian elimination is not optimal.

[2] 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.

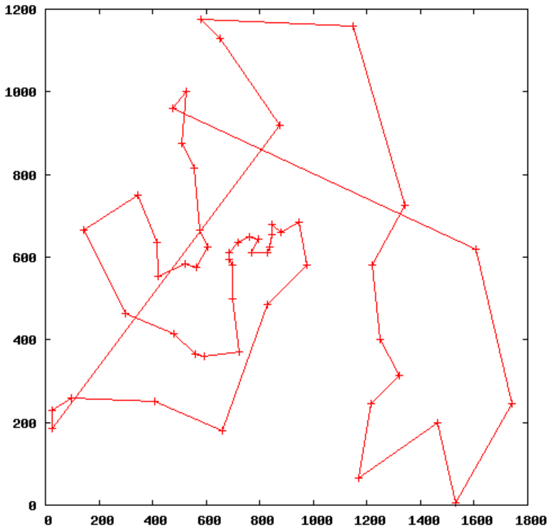
Fast matrix multiplication

- 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: $O(n^{2.376})$ [Coppersmit-Winograd, 1987]
- Conjecture: $O(n^{2+\epsilon})$ for any $\epsilon > 0$;

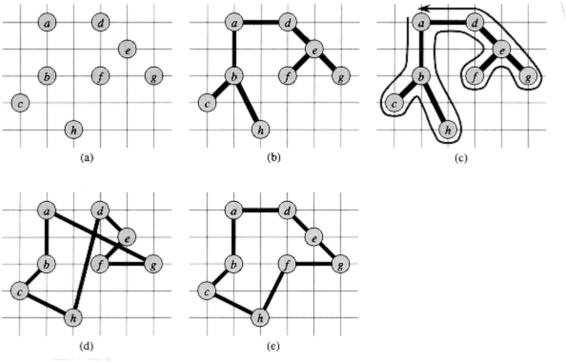
ClosestPair problem: to find the closest pair of points in a plane

Practical problem: Hierarchical clustering

Practical problem: Nearest neighbor heuristic for TSP



Practical problem: MST heuristic for TSP

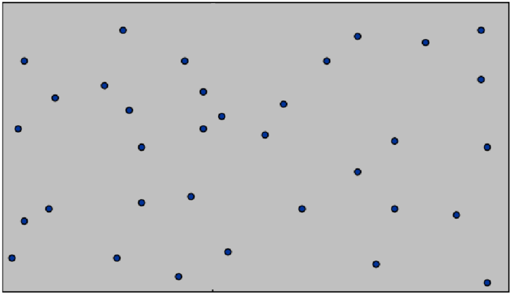


(a) (b) (c)

(d) (e)

Basic operation: CLOSESTPAIR problem

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



About ClosestPair problem

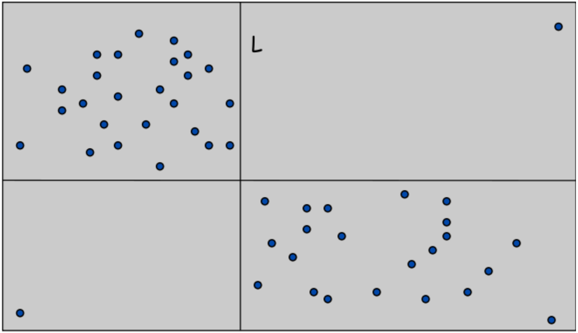
- Computational geometry: M. Shamos and D. Hoey were working out efficient algorithm for basic computational primitive in CG in 1970's. Does there exist an algorithm using less than $O(n^2)$ time?

- 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)

- Key observation: a point set can also be divided into subsets.
- Divide-and-conquer: divide into 4 subsets.

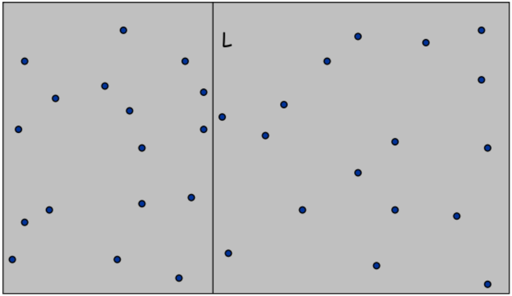


- Difficulty: The subsets might be unbalanced — we cannot guarantee that each subset has (roughly) $\frac{n}{4}$ points. Thus, it will take $O(n^2)$ time to combine. For example, we might have the following recursion $T(n) = 2T(\frac{n}{2}) + O(n^2)$.

Trial 2: Divide into 2 halves

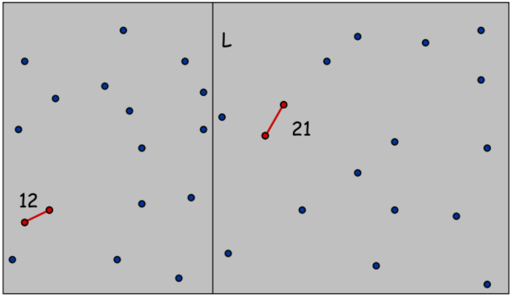
Trial 2: divide-and-conquer (2 subsets)

- **Divide:** divide into two halves;
  It is easy to achieve this through sorting by x coordinate first, and then select $x_{\lfloor \frac{n}{2} \rfloor}$ as splitter.



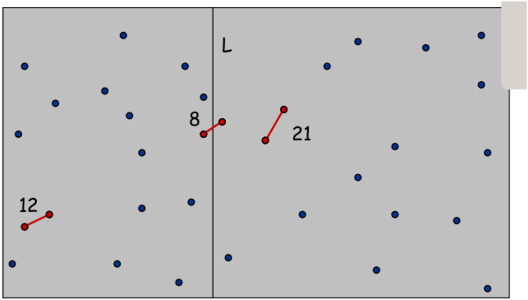
Trial 2: divide-and-conquer (2 subsets)

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

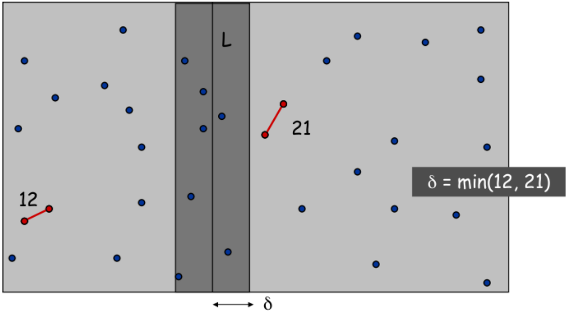


Trial 2: divide-and-conquer (2 subsets)

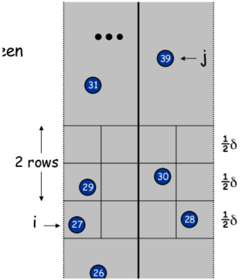
- **Divide:** dividing into two (roughly equal) subsets;
- **Conquer:** finding closest pairs in each half;
- **Combine:** It suffices to consider the pairs consisting of one point from left half and one point from right half.
  - There are $O(n^2)$ such pairs;
  - Can we find the closest pair in $O(n)$ time?



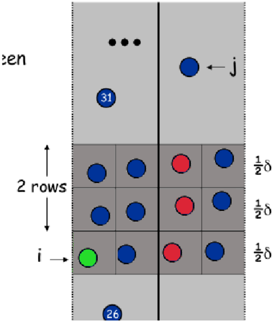
- **Observation 1:**
  - The closest pair is located in left part, or right part, or within δ of the middle line L.
  - The third type occurs in a narrow strip only!
  - Thus, it suffices to check point pairs in the 2δ-strip.
  - Here, δ is the minimum of $ClosestPair(LeftHalf)$ and $ClosestPair(RightHalf)$.



- **Observation 2:**
    - Moreover, it is unnecessary to explore **all** point pairs in the 2δ-strip.
    - 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 construct closest-pair.
    - Example: For point i, it suffices to search within 2 rows for possible closest partners ($< \delta$).

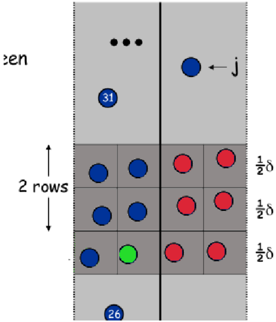


To detect potential closest pair: Case 1



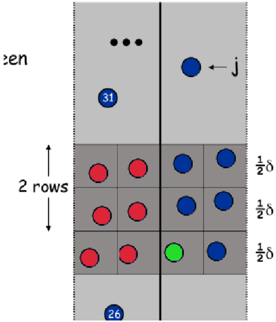
- Green: point i;
- Red: the possible closest partner (distance $< \delta$) of point i;

To detect potential closest pair: Case 2



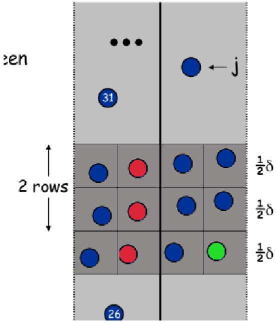
- Green: point i;
- Red: the possible closest partner (distance $< \delta$) of point i;

To detect potential closest pair: Case 3



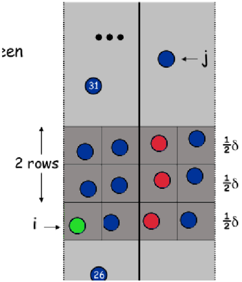
- Green: point i;
- Red: the possible closest partner (distance $< \delta$) of point i;

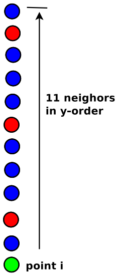
To detect potential closest pair: Case 4



- Green: point i;
- Red: the possible closest partner ($< \delta$) 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.
- Reason: All the points in red are within 3 rows, which have at most 12 points.

ClosestPair algorithm

ClosestPair($p_i, \ldots, p_j$) /* $p_i, \ldots, p_j$ have already been sorted according to x-coordinate; */

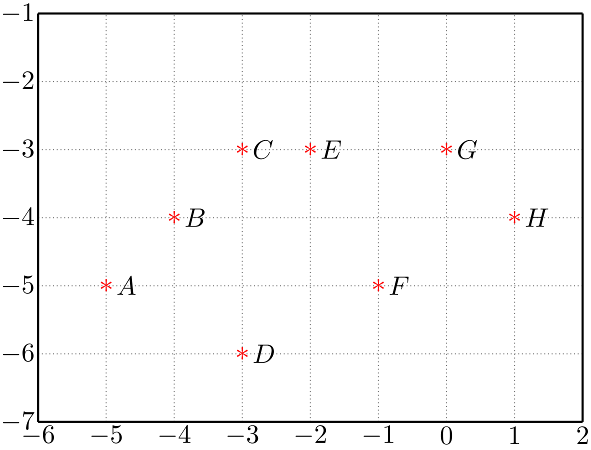
```
if  j − i == 1  then
    return d(p_i, p_j);
end if
Use the x-coordinate of p_⌊(i+j)/2⌋ to divide p_i, ..., p_j into two halves;
δ_1 = ClosestPair(left half); T(n/2)
δ_2 = ClosestPair(right half); T(n/2)
δ = min(δ_1, δ_2);
Sort points within the 2δ strip by y-coordinate; O(n log(n))
Scan points in y-order and calculate distance between each point with its next 11 neighbors. Update δ if finding a distance less than δ; O(n)
```

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

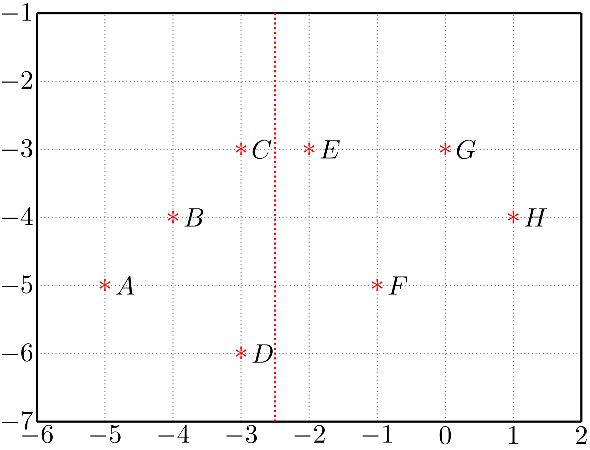
ClosestPair algorithm: improvement

- Note: can be improved to $O(n \log n)$ if we do not sort points within 2δ strip from the scratch every time.
    - Each recursion keeps two sorted list: one list by x, and the other list by y.
    - Merge pre-sorted lists into a list as MergeSort does. Thus it 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

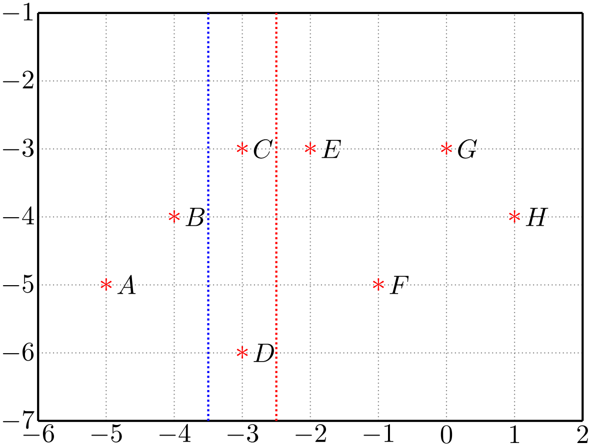


- Objective: to find the closest pair among these 8 points.

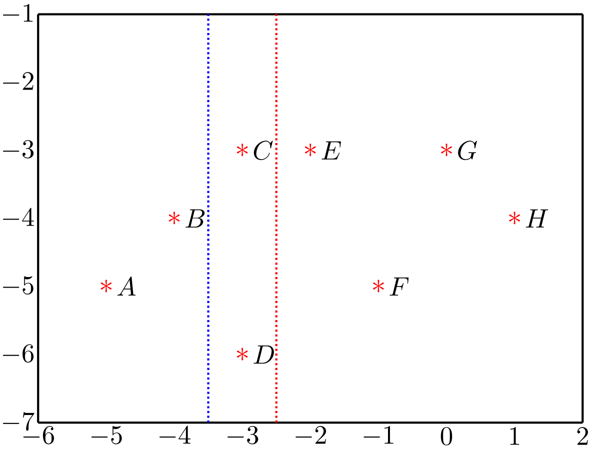


- Objective: to find the closest pair among these 8 points.

Left half: A, B, C, D

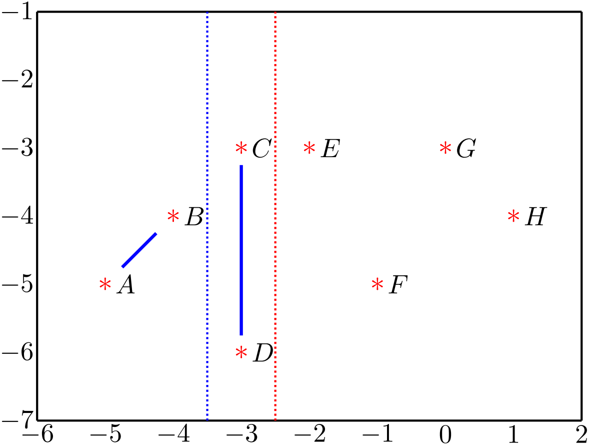


Left half: A, B, C, D



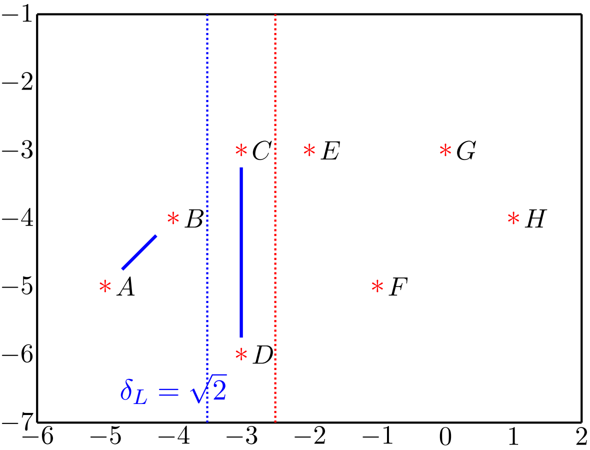
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Left half: A, B, C, D



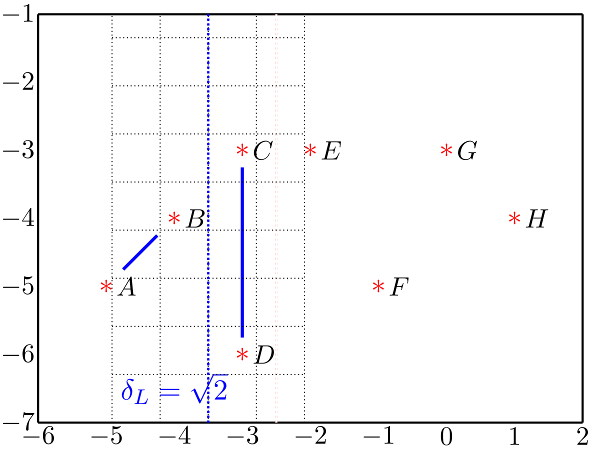
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Left half: A, B, C, D



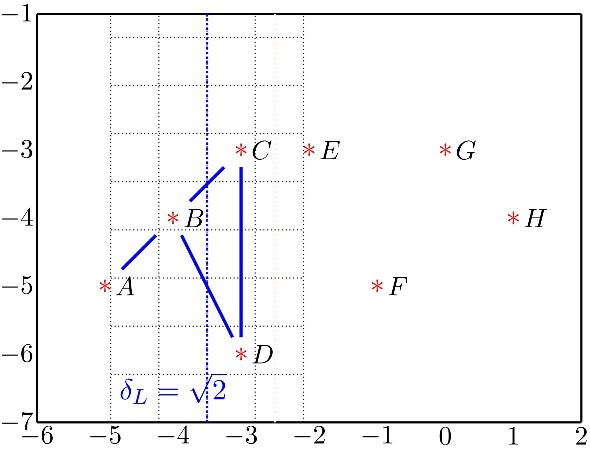
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Left half: A, B, C, D



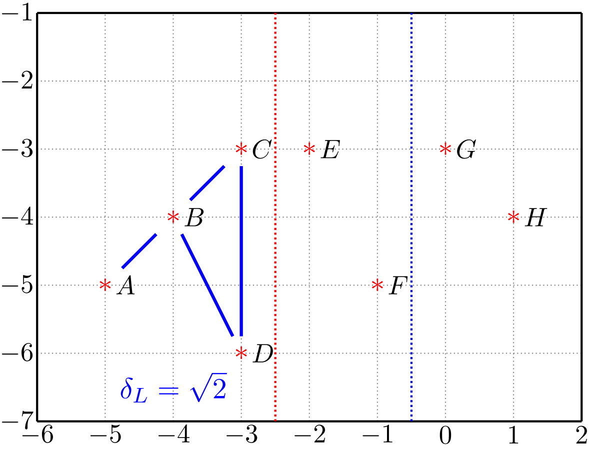
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Left half: A, B, C, D

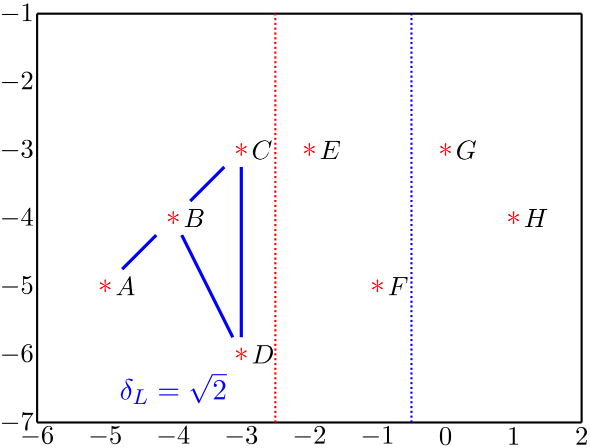


- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3; \Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}; \Rightarrow \delta_L = \sqrt{2}$.

Right half: E, F, G, H

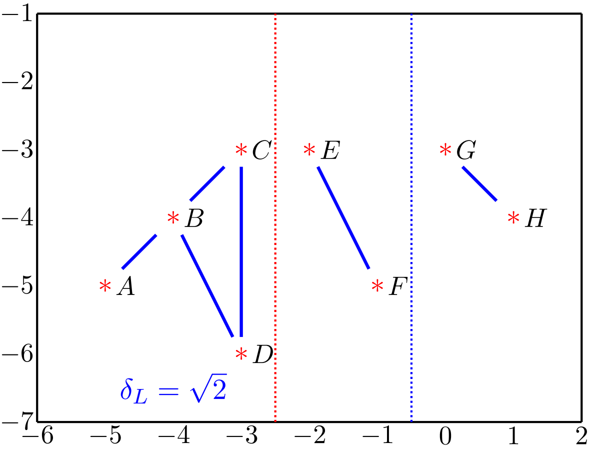


Right half: E, F, G, H



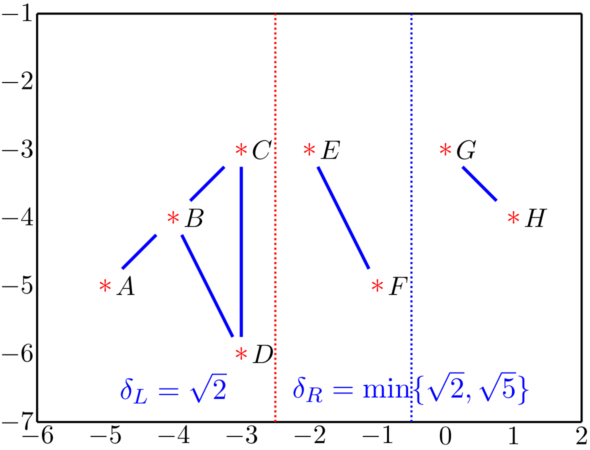
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



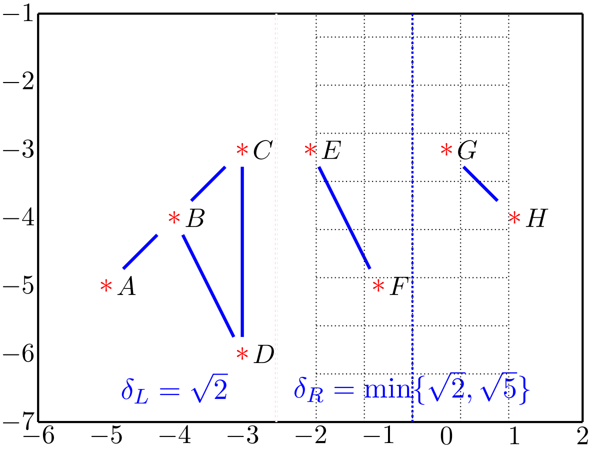
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



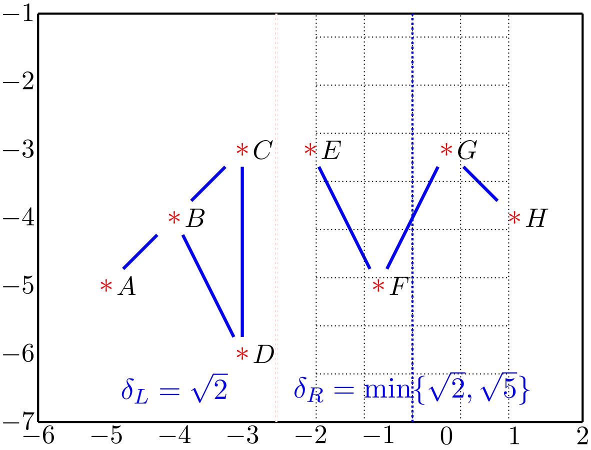
- Pair 5: $d(E,F) = \sqrt{5}$;
- Pair 6: $d(G,H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G,F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



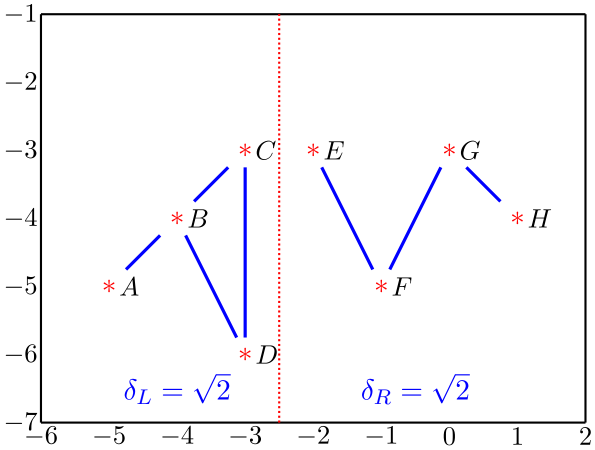
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



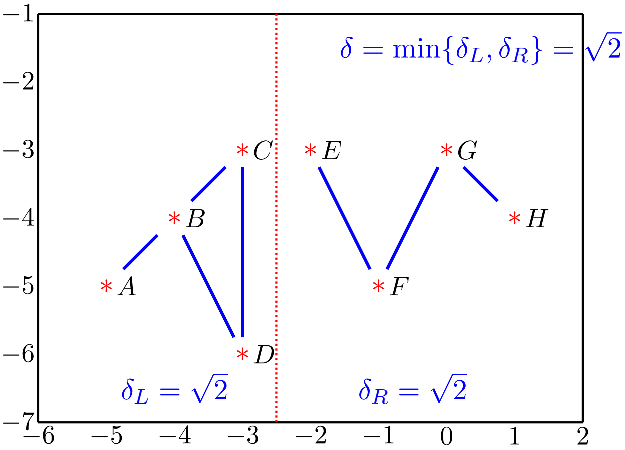
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Total: A, B, C, D, E, F, G, H



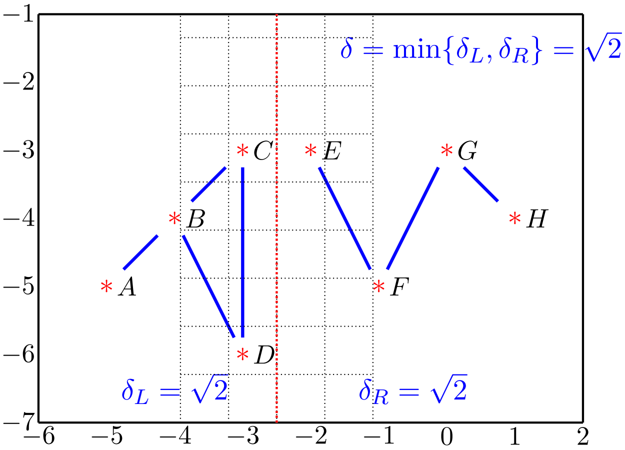
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

Total: A, B, C, D, E, F, G, H



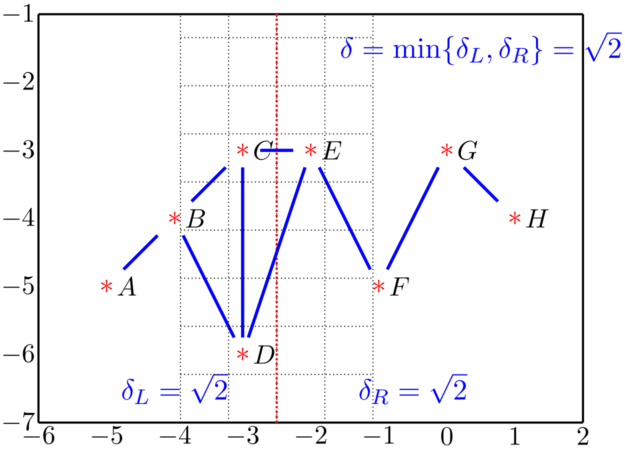
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

Total: A, B, C, D, E, F, G, H



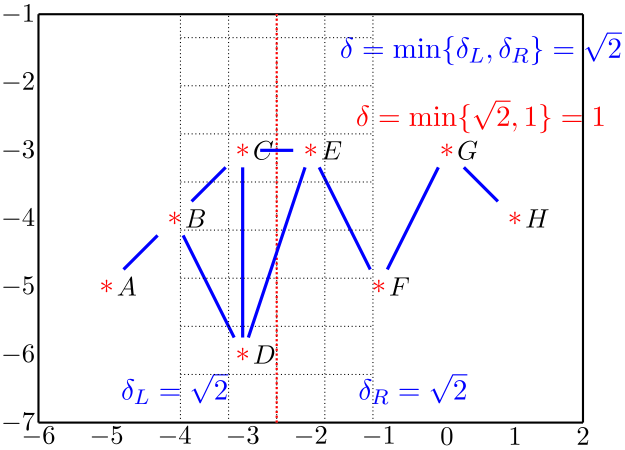
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

Total: A, B, C, D, E, F, G, H



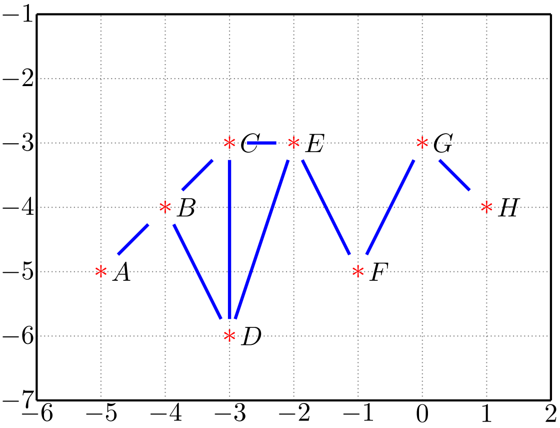
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

Total: 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 $O(n^2) \Rightarrow O(n\log(n))$, 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, Wards, 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.

SELECTION problem (to appear in Lec 14)