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, QuickSort, and FloydRivest algorithm for 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 the ClosestPair problem.
  2. This technique is especially powerful when combined with randomization technique.

On what problems can we divide and conqueror?

- Suppose the input of a problem is related to the following data structures, perhaps we can try to divide it into sub-problems, i.e., problems with the same structure but smaller size.
    - An **array** with n elements;
    - A **matrix**;
    - A **set** of n elements;
    - A **tree**;
    - A **directed acyclic graph**;
    - A **general graph**;
    - ......

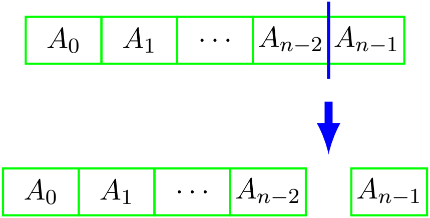
SORT problem: to sort an array of n integers

SORT problem

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

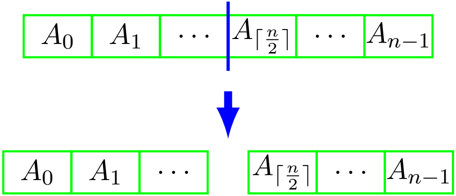
Two possible divide-and-conqueror strategies I

1. **Divide into a $n-1$-length array and an element**: 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.



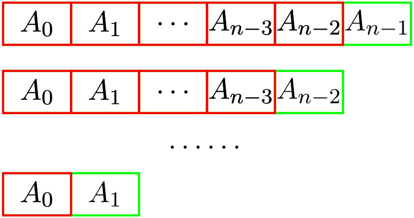
Two possible divide-and-conqueror strategies II

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.



Trial 1: The first divide strategy

- Basic idea: At each step of the execution, we have several elements 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 solution is constructed step-by-step.



InsertionSort(A, n)

```
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
```

$$\boxed{A_0}\boxed{A_1}$$

$$\boxed{A_0}\boxed{A_1}\boxed{A_2}$$

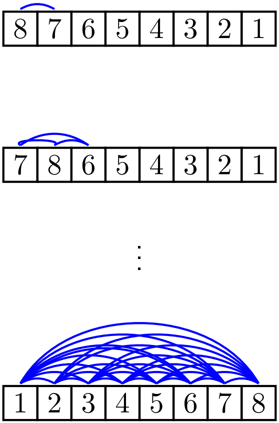
$$\cdots\cdots$$

$$\boxed{A_0}\boxed{A_1}\boxed{\cdots}\boxed{A_{\lceil \frac{n}{2} \rceil}}\boxed{\cdots}\boxed{A_{n-1}}$$

(see a demo here)

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



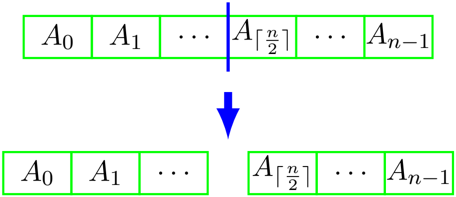
InsertSort: 28 ops

Trial 2: the second divide strategy (MergeSort algorithm [J. von Neumann, 1945, 1948])

Figure 1: von Neumann in 1940s

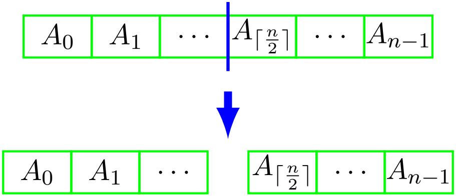
Trial 2: MERGESORT algorithm

- 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;

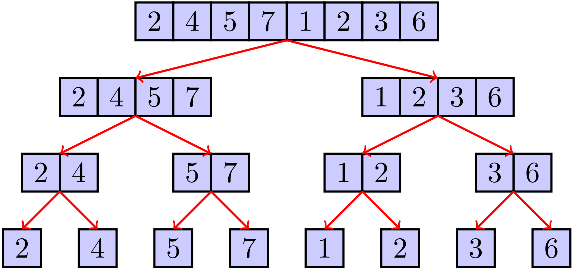
MergeSort algorithm



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
```

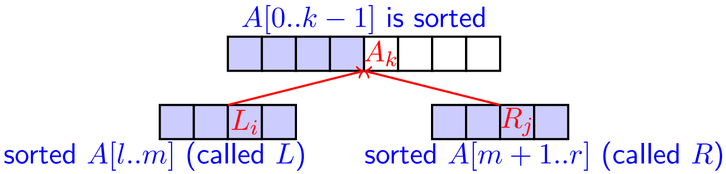
An example



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 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)

/* 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

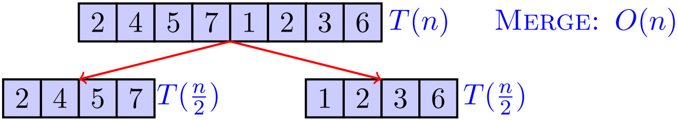
end for

Time complexity: $O(n)$. (see a demo)

Time-complexity of MergeSort algorithm

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

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



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;

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

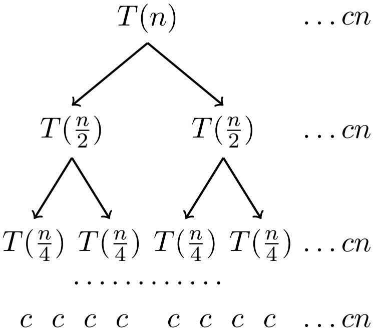
Analysis technique 1: Unrolling

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

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

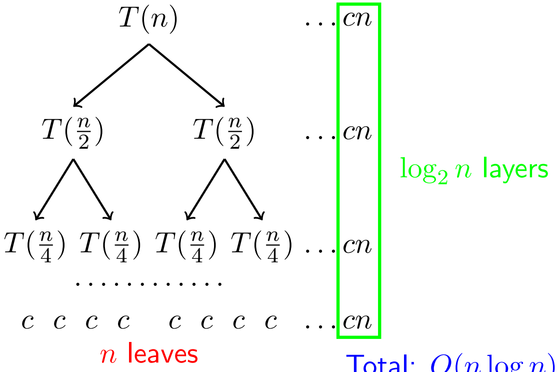
Analysis technique 1: Unrolling

- 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;



Total: $O(n \log n)$

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* $T(n) = aT(n/b) + n^d$ *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.

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

□

Master theorem: examples

- Example 1: $T(n) \leq 3T(n/2) + cn$

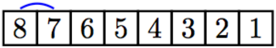
  $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

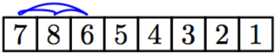
- Example 2: $T(n) \leq 2T(\frac{n}{2}) + cn^2$

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

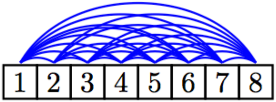
- Example 3: $T(n) \leq T(n/3) + T(2n/3) + cn$

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

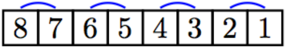




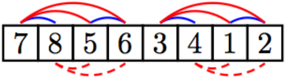
$\vdots$



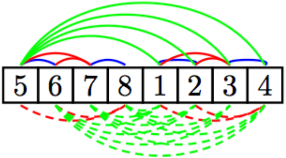
InsertSort: 28 ops



MergeSort step 1: 4 ops



MergeSort step 2: 4 ops, save: 4



MergeSort step 3: 4 ops, save: 12

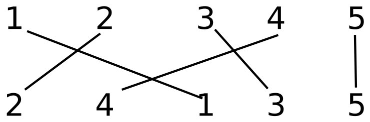
COUNTINGINVERSION: to count inversions in an array of n integers

Practical problems:

1. to identify two 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

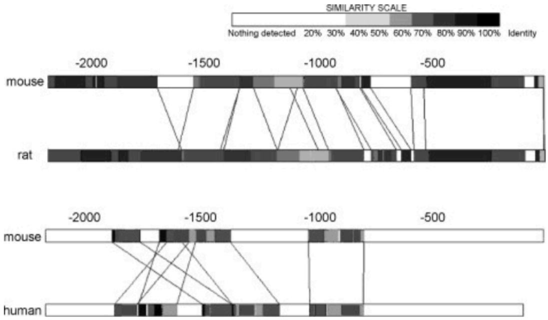


Figure 2: Sequence comparison of the 5' flanking regions of mouse, rat and human ERβ.

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, while I_i^- is 1 if $X_{[i,..,i+k-1]}$ and $-Y_{[i,..,i+k-1]}$ has the same order and 0 otherwise.
- Advantage: the association may exist across a subset of samples. For example,

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

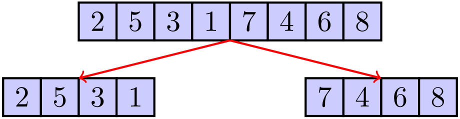
$W_1 = 2$ when $k = 3$. Much better than Pearson CC, et al.

CountingInversion problem

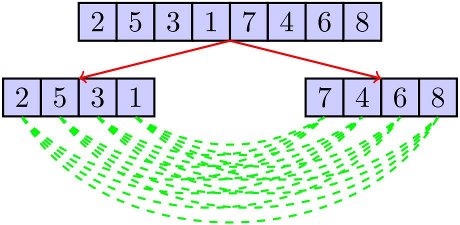
- Solution: index pairs. The possible solution space has a size of $O(n^2)$.
- Brute-force: $O(n^2)$ (checking each pair (a_i, a_j)).
- Can we design a better algorithm?

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;

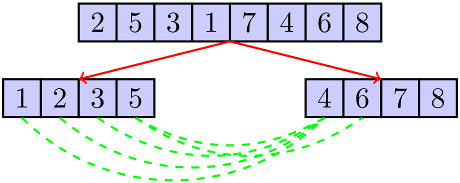


- **Combine:** how to count inversion (a_i, a_j), when a_i and a_j are in different half?
- A simple enumeration will take $\frac{n^2}{4}$ steps. Thus, $T(n) = 2T(\frac{n}{2}) + \frac{n^2}{4} = O(n^2)$.



Combine strategy 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 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**. These sub-problems have the same form to the original problem but a smaller size.
- 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: divide according to **a randomly-selected pivot**

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

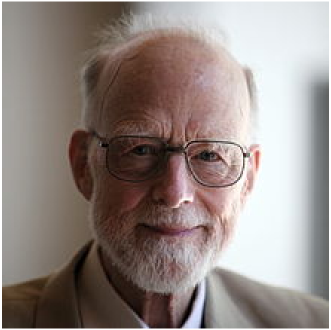


Figure 3: Sir Charles Antony Richard Hoare, 2011

QUICKSORT: divide randomly

```
QuickSort(A)
S_- = {}; S_+ = {};
Choose a pivot A[j] uniformly at random;
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_+;
```

- 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 pivot and divide based on its value; thus, we cannot guarantee that each sub-problem has exactly $\frac{n}{2}$ elements.

- **Worst case:** selecting the smallest/biggest element at each iteration;

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

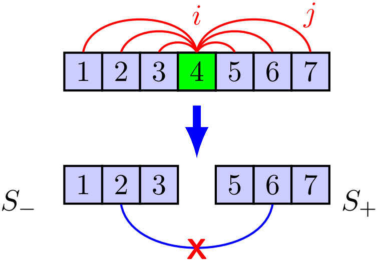
- **Best case:** select the median exactly at each iteration;

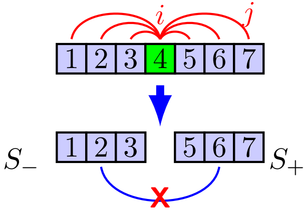
$$T(n) \leq 2T(n/2) + cn \Rightarrow T(n) = O(n \log n)$$

- **Most cases:** instead of selecting the median exactly, we can select a **nearly-central 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 comparison in line 3 and 4;
- 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.





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

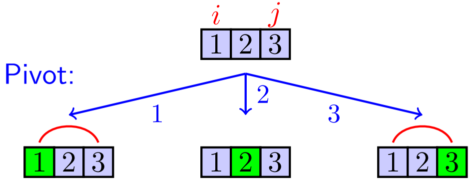
$$
\begin{aligned}
E[X] &= E[\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}] \\
&= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} E[X_{ij}] \\
&= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \Pr(A[i] \text{ is compared with } A[j])
\end{aligned}
$$

Analysis cont'd

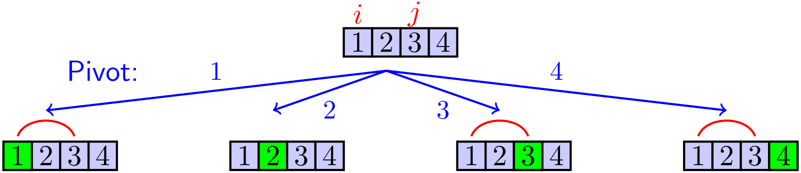
- **Observation 2:** $A[i]$ and $A[j]$ are compared iff either $A[i]$ or $A[j]$ is selected as pivot when processing elements containing $A[i, i+1, ..., j]$.
- We claim $\Pr(A[i]$ is compared with $A[j]) = \frac{2}{j-i+1}$. (Why?)
- Then:

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

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



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



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

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

1	2	$\cdots$	i	$\cdots$	j	$\cdots$	n

- Now let's extend these observations to general cases. By induction over the size of A, we can calculate the probability as:

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

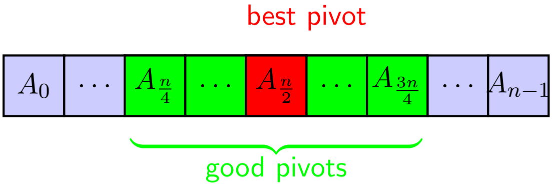
ModifiedQuickSort: easier to analyze

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 $A[i] > A[j]$;
    end for
    if $\|S_{+}\| \geq \frac{n}{4}$ and $\|S_{-}\| \geq \frac{n}{4}$ **then**
        break;
    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".

ModifiedQuickSort: analysis



- It is easy to obtain a **nearly central pivot**:
  - $\Pr(\text{select the } \textbf{centroid} \text{ pivot }) = \frac{1}{n}$
  - $\Pr(\text{select a } \textbf{nearly central} \text{ pivot}) = \frac{1}{2}$
  - Thus $E(\#\texttt{WHILE}) = 2$, i.e., the expected time of this step is $2n$.
- **Nearly central pivot** is good:
  - The recursion tree has a depth of $O(\log_{\frac{4}{3}} n)$, and $O(n)$ work is needed at each level.
  - So $T(n) = O(n \log_{\frac{4}{3}} n)$.

Lomuto's implementation

$\textsc{QuickSort}(A, l, h)$

if $l < h$ **then**
  $p = \textsc{Partition}(A, l, h)$;
  $\textsc{QuickSort}(A, l, \mathbf{p-1})$;
  $\textsc{QuickSort}(A, p+1, h)$;
end if

$\textsc{Partition}(A, l, h)$

$pivot = A[h]$; $i = l - 1$;
for $j = l$ to $h - 1$ **do**
  if $A[j] < pivot$ **then**
    $i++$;
    Swap $A[i]$ with $A[j]$;
  end if
end for
if $A[h] < A[i+1]$ **then**
  Swap $A[i+1]$ with $A[j]$;
end if
return $i + 1$;

- Basic idea: elements in $A[l..i] \leq pivot$; elements in $A[i+1..j-1] > pivot$.
- Sorting the entire array: $\textsc{QuickSort}(A, 0, n-1)$.

Hoare's implementation [1961]

```
QuickSort(A, l, h)
if l < h then
    p = Partition(A, l, h);
    QuickSort(A, l, p);
    QuickSort(A, p + 1, h);
end if
```

```
Partition(A, l, h)
i = l - 1; j = h + 1; pivot = A[l];
while TRUE do
    repeat
        j = j - 1;
    until A[j] ≤ pivot;
    repeat
        i = i + 1;
    until A[i] ≥ pivot;
    if i ≥ j then
        return j;
    end if
    Swap A[i] with A[j];
end while
```

- Sorting the entire array: $\textsc{QuickSort}(A, 0, n-1)$.

Comparison of MergeSort and QuickSort [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: 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.

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

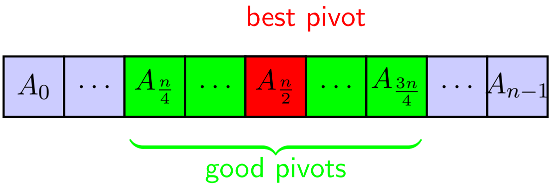
INPUT:
An array $A = [A_0, A_1, ..., A_{n-1}]$, and a number $k < n$;
OUTPUT:
The k-th smallest item in general case (or the median of A as a specical case).

- For example, given a set $A = [18, 15, 27, 13, 1, 7, 25]$, the objective is to find the median of A.
- A feasible strategy is to sort A first, and then report the k-th one, 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 j = 1 to n do
    if A_j > A_i then
        S_+ = S_+ ∪ {A_j};
    else
        S_- = S_- ∪ {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
```

Question: How to choose a pivot?



- We have the following three options:
  - Worst choice: select the smallest element at each iteration.
    $T(n) = T(n-1) + O(n) = O(n^2)$
  - Best choice: select the median at each iteration.
    $T(n) = T(\frac{n}{2}) + O(n) = O(n)$
  - Good choice: select **a nearly-central element** A_i, i.e., $|S_+| \geq \epsilon n$, and $|S_-| \geq \epsilon n$ for a fixed $\epsilon > 0$.

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

- The problem of finding the median turns into finding **an element close to the median**, say within $\frac{n}{4}$ from the median.
- How can we efficiently get **a nearly-central pivot**?
- We estimate median of **the whole set** through examining a **sample of the whole set**. The following samples have been tried:
  1. Selecting a central pivot via **examining medians of groups**;
  2. Selecting a central pivot via **randomly selecting an element**;
  3. Selecting a 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.

Median of group medians algorithm [Blum et al, 1973]

SELECTMEDIAN(A)

1: Line up elements in groups of 5 elements;
2: Find the median of each group; $O(\frac{6n}{5})$ time
3: Find the median of medians (denoted as M); $T(\frac{n}{5})$ time
4: Use M as pivot to partition the input and call the algorithm recursively on one of the partitions. at most $O(\frac{7n}{10})$ time

- Basic idea: "median of group medians" is nearly central.

	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

Analysis

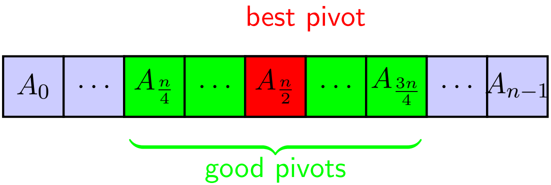
- Advantages:
  1. Median of medians M is nearly-central as at least $\frac{3n}{10}$ elements are larger, and at least $\frac{3n}{10}$ elements are smaller than M. Thus, at least $\frac{3n}{10}$ elements can be deleted at each iteration.
  2. It takes only $T(\frac{n}{5})$ time to find the median of medians.
- Running time:
  $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + \frac{6n}{5} = O(n)$. Actually it takes at most $24n$ comparisons.
- Question: what happens if we divide the set into groups of 3 elements?

QuickSelect: Selecting a pivot randomly [Hoare, 1961]

QuickSelect(A, k)

Choose an element A_i from A **uniformly at random**;
$S_+ = \{\}$;
$S_- = \{\}$;
for $j = 1$ to n **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 a pivot A_i uniformly at random, it is highly likely to get a good pivot since a fairly large fraction of the elements are nearly-central.

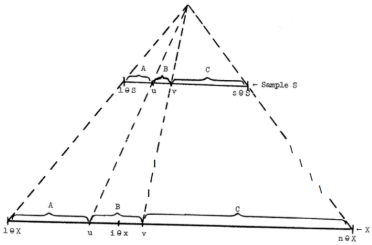
Theorem

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

Proof.

- We divide the execution into a series of phases: We say that the execution is in phase j 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 steps that QuickSelect uses, and X_j be the number of steps 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.
- There are at most $cn(\frac{3}{4})^j$ steps in phase j since there are at most $n(\frac{3}{4})^j$ elements. Thus, $E(X_j) \leq 2cn(\frac{3}{4})^j$.
- Hence $E(X) = E(X_0 + X_1 + ...) \leq \sum_j 2cn(\frac{3}{4})^j \leq 8cn$.

Folyd-Rivest algorithm: Selecting pivots according to a random sample



- In 1973, Floyd and 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 estimator of the median of the whole set, and we can find a small interval that is expected to contain the median of the whole set with high probability.

FLOYD-RIVEST-SELECT(A)

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 with values less than u, M contains elements with values in $[u, v]$, and H contains the elements with values greater than v.
4: The partition of A into these three sets is completed through comparing each element e in $A - S$ with u and v: if $k \leq \frac{n}{2}$, e is compared with v first and then to u only if $e \leq v$. 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 algorithm

LazySelectMedian(A)

1: Randomly sample r elements (with replacement) from $A = \{a_1, a_2, ..., a_n\}$. Denote the sample as S.
2: Sort S. Let u be the $(1-\delta)\frac{r}{2}$-th smallest element of S and v be the $(1+\delta)\frac{r}{2}$-th smallest element of S. **//The median is expected to be in the interval $[u, v]$ with high probability.**
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 constraint 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. **Set** $n = |A| = 16$ **and** $\delta = \frac{1}{2}$

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									v
4	5	6	7	8	9	10	11	12	13

H:

15	16	14

Return 7 **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 a 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 has a size of $\delta n = n^{\frac{3}{4}}$.

Time-comlexity analysis: linear time

LAZYSELECTMEDIAN(A)

1: Randomly sample r elements (with replacement) from $A = \{a_1, a_2, ..., a_n\}$. Denote the sample as S. **//Set $r = n^{\frac{3}{4}}$**
2: Sort S. Let u be the $(1-\delta)\frac{r}{2}$-th smallest element of S and v be the $(1+\delta)\frac{r}{2}$-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* $2n + o(n)$.

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

					u									v				
1	3	2			4	5	6	7	8	9	10	11	12	13		15	16	14
	L								M								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:
$$x_i = \begin{cases} 1 & x_i \text{ is greater than the median of } A \\ 0 & otherwise \end{cases}$$
- 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

$$\begin{aligned}
\Pr(|L| > \tfrac{n}{2}) &\leq \Pr(X \geq \tfrac{1+\delta}{2}r) && (6)\\
&= \tfrac{1}{2}\Pr(|X - E(X)| \geq \tfrac{\delta}{2}r) && (7)\\
&\leq \frac{1}{2}\frac{\sigma^2(X)}{(\frac{\delta}{2}r)^2} && (8)\\
&= \frac{1}{2}\frac{1}{\delta^2 r} && (9)\\
&= \frac{1}{2}n^{-\frac{1}{4}} && (10)
\end{aligned}$$

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

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

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

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

Kolmogorov's conjecture

- Conjecture: In 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:**

$$
\begin{aligned}
xy &= (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) && (11)\\
&= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l && (12)
\end{aligned}
$$

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

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.
- 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 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:**

$$
\begin{aligned}
xy &= (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) && (13)\\
&= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l && (14)\\
&= x_h y_h 2^n + (P - x_h y_h - x_l y_l) 2^{\frac{n}{2}} + x_l y_l && (15)
\end{aligned}
$$

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:
  $T(n) = 3T(n/2) + cn \Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

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

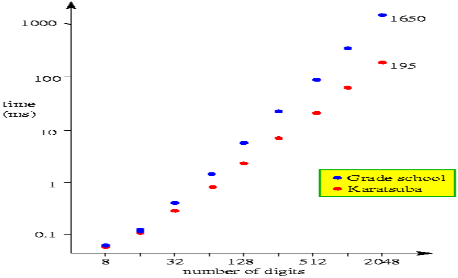


Figure 5: Sun SPARC4, g++ -O4, random input. See

- 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.

- 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:

$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} && (16) \\ &= x_i - \frac{t - \frac{1}{x_i}}{\frac{1}{x_i^2}} && (17) \\ &= -t \times x_i^2 + 2x_i && (18) \end{aligned}$$

  - 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:**

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

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) && (19) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) && (20) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) && (21) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) && (22) \end{aligned}$$

- 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?

Strassen algorithm, 1969

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:**

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

MatrixMultiplication problem: a clever conquer II

$$
\begin{aligned}
P_1 &= A_{11} \times (B_{12} - B_{22}) && (23)\\
P_2 &= (A_{11} + A_{12}) \times B_{22} && (24)\\
P_3 &= (A_{21} + A_{22}) \times B_{11} && (25)\\
P_4 &= A_{22} \times (B_{21} - B_{11}) && (26)\\
P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) && (27)\\
P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) && (28)\\
P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) && (29)
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= P_4 + P_5 + P_6 - P_2 && (30)\\
C_{12} &= P_1 + P_2 && (31)\\
C_{21} &= P_3 + P_4 && (32)\\
C_{22} &= P_1 + P_5 - P_3 - P_7 && (33)
\end{aligned}
$$

- 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. [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.

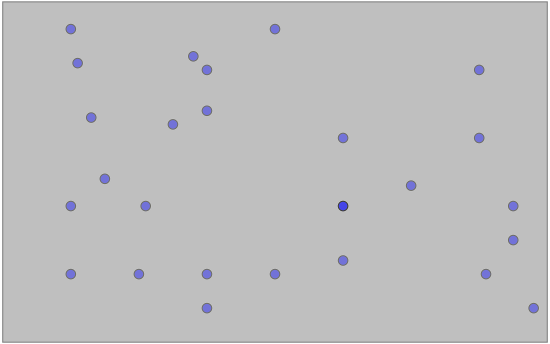
Fast matrix multiplication

- 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

Basic operation: ClosestPair problem

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



About ClosestPair problem

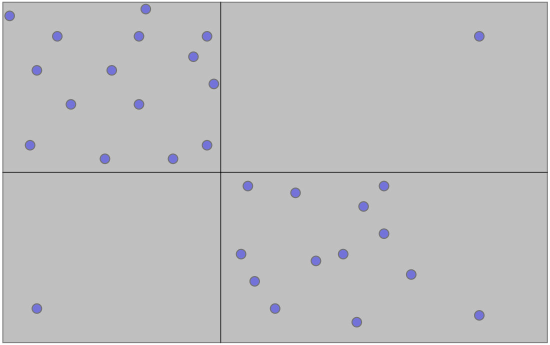
- Computational geometry: M. Shamos and D. Hoey were working out efficient algorithm for basic computational primitive in CG in 1970's. Does there exist an algorithm using less than $O(n^2)$ time?

- 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)

- Divide-and-conquer: divide into 4 subsets.

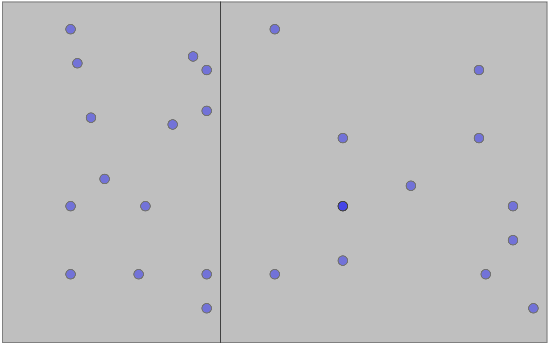


- Difficulties:
  - The subsets might be unbalanced — we cannot guarantee that each subset has approximately $\frac{n}{4}$ points.
  - Since the closest-pair might lie in different subsets, we need to consider all $\binom{4}{2}$ pairs of subsets to avoid missing, thus complicating the "combine" step.

Trial 2: Divide into 2 halves

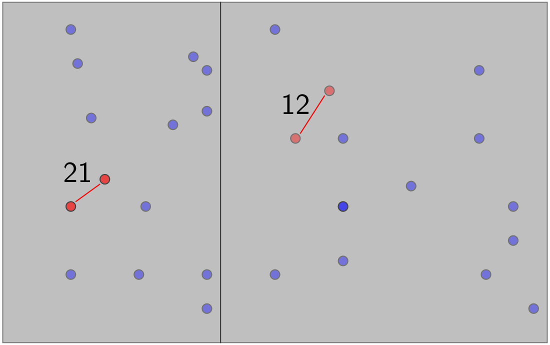
Trial 2: divide-and-conquer (2 subsets)

- **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.



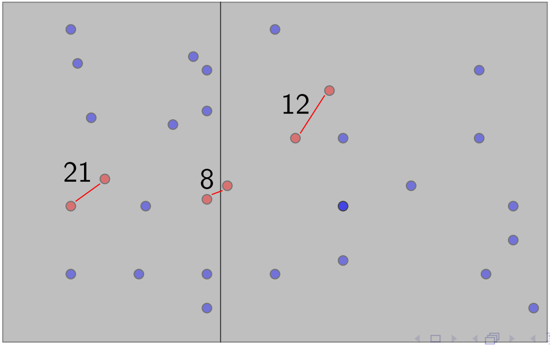
Trial 2: divide-and-conquer (2 subsets)

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



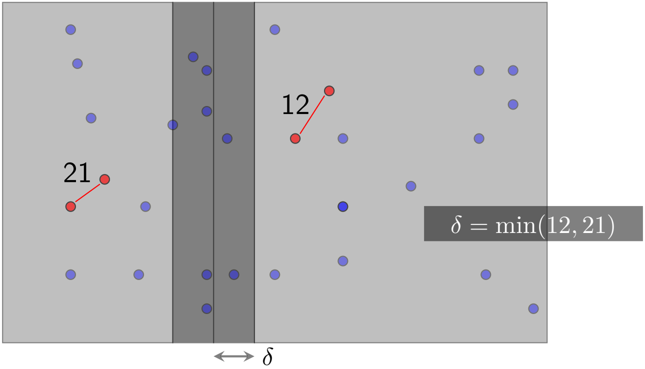
Trial 2: divide-and-conquer (2 subsets)

- **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?

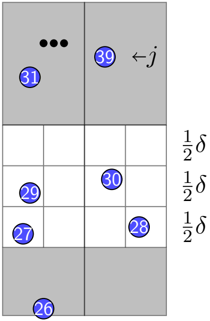


- **Observation 1:**
  - The closest pair is located in left part, or right part, or within δ of the middle line L.
  - The third type occurs in a narrow strip only!
  - Thus, it suffices to check point pairs in the 2δ-strip.
  - Here, δ is the minimum of $ClosestPair(LeftHalf)$ and $ClosestPair(RightHalf)$.

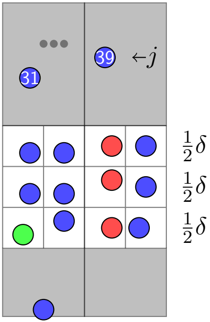
It is unnecessary to check all pairs (I) II



- **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 ($< \delta$).

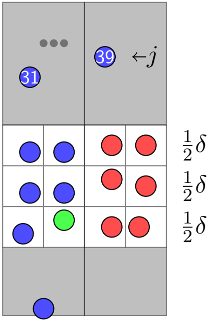


To detect potential closest pair: Case 1



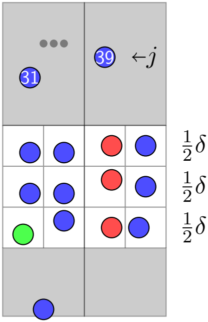
- Green: point i;
- Red: the possible closest partner (distance $< \delta$) of point i;

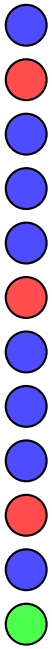
To detect potential closest pair: Case 2



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

To detect potential closest pair





- 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($p_i, ..., p_j$) /* $p_i, ..., p_j$ 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 $p_{\lfloor \frac{i+j}{2} \rfloor}$ to divide $p_i, ..., p_j$ into two halves;

δ_1 = ClosestPair(left half); $T(\frac{n}{2})$

δ_2 = ClosestPair(right half); $T(\frac{n}{2})$

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

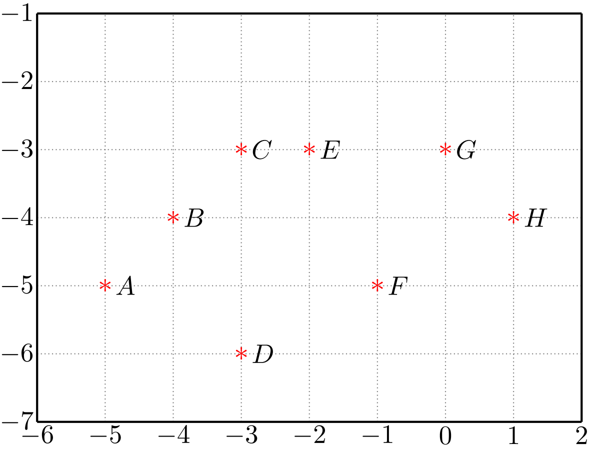
Sort points within the 2δ wide strip by y-coordinate; $O(n \log n)$

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; $O(n)$

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

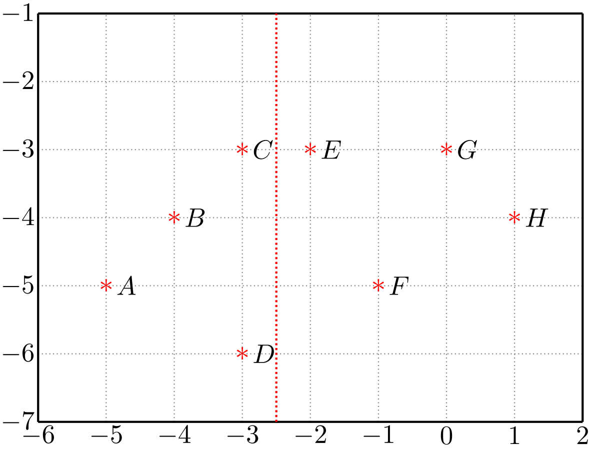
- Note: The algorithm can be improved to $O(n \log n)$ if we do not sort points within 2δ strip from the scratch every time.
  - Each recursion keeps two sorted list: one list by x, and the other list by y.
  - We merge two pre-sorted lists into a list as MergeSort does, which costs only $O(n)$ time.
- Time-complexity: $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

ClosestPair: an example with 8 points



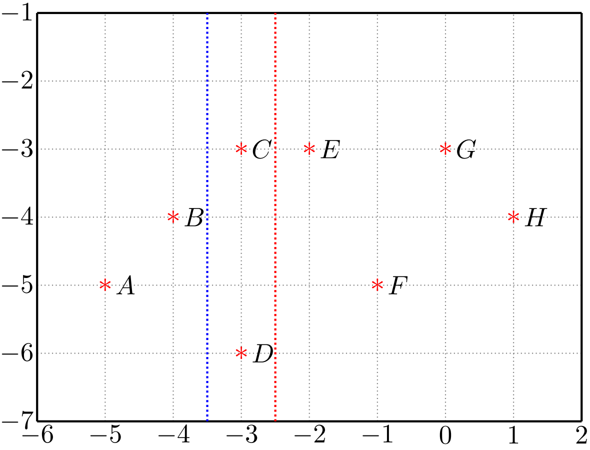
- Objective: to find the closest pair among these 8 points.

ClosestPair: an example with 8 points

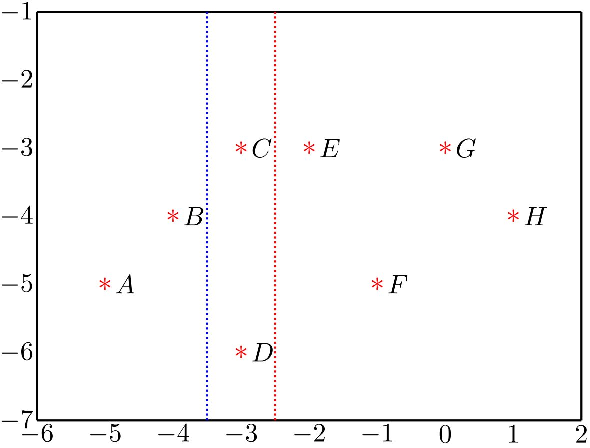


- Objective: to find the closest pair among these 8 points.

Left half: A, B, C, D

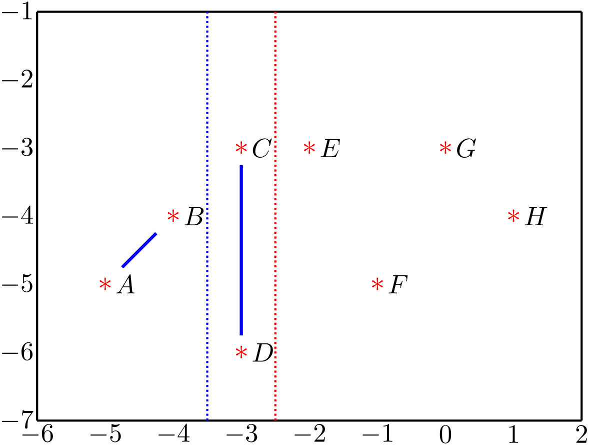


Left half: A, B, C, D



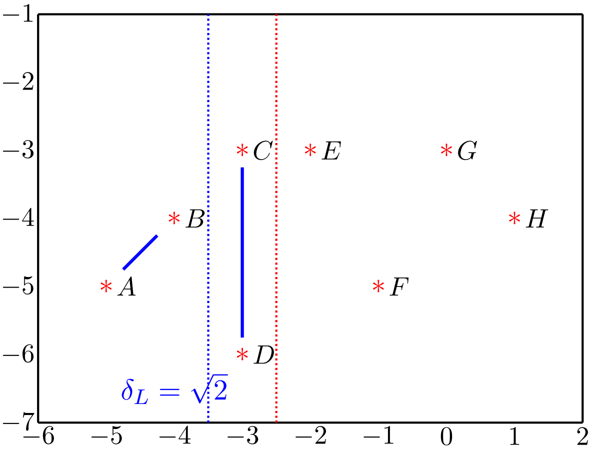
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Left half: A, B, C, D



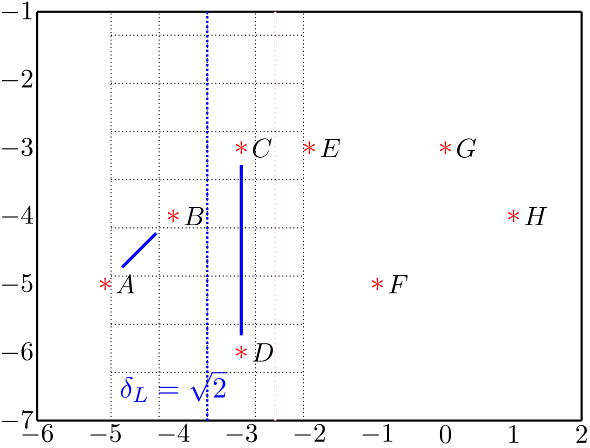
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Left half: A, B, C, D



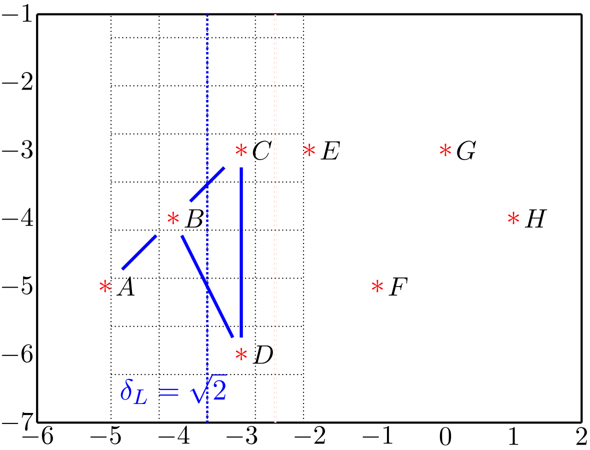
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Left half: A, B, C, D



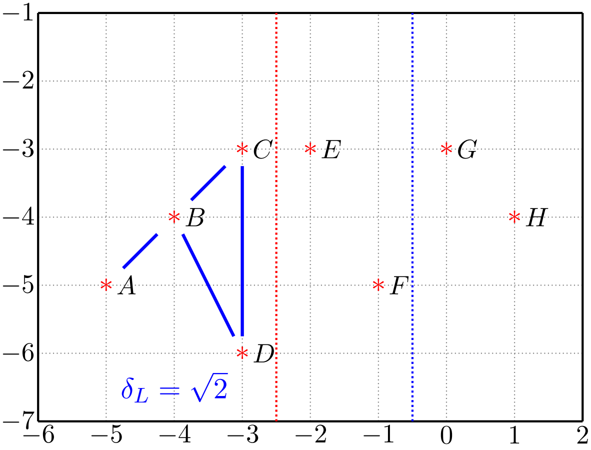
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Left half: A, B, C, D

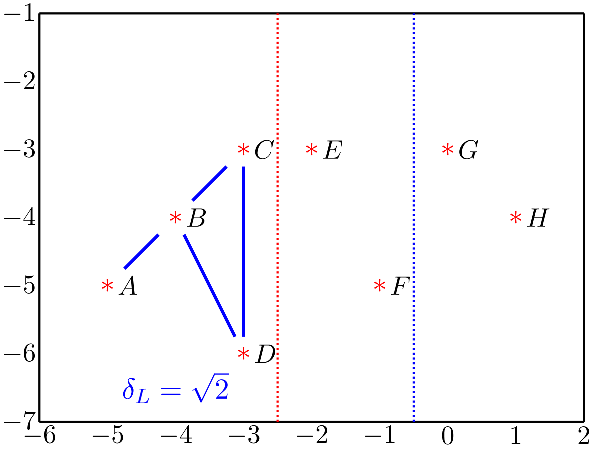


- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.

Right half: E, F, G, H

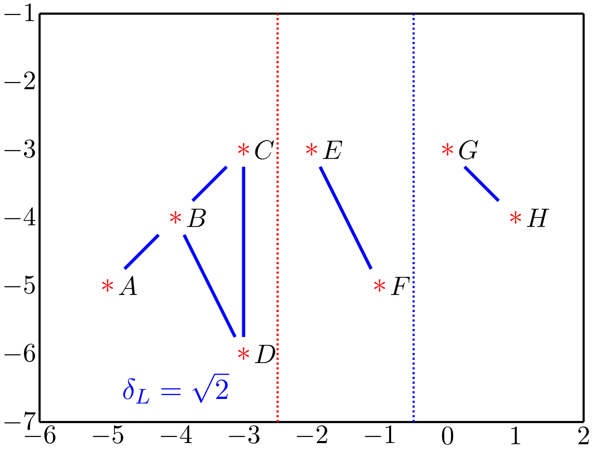


Right half: E, F, G, H



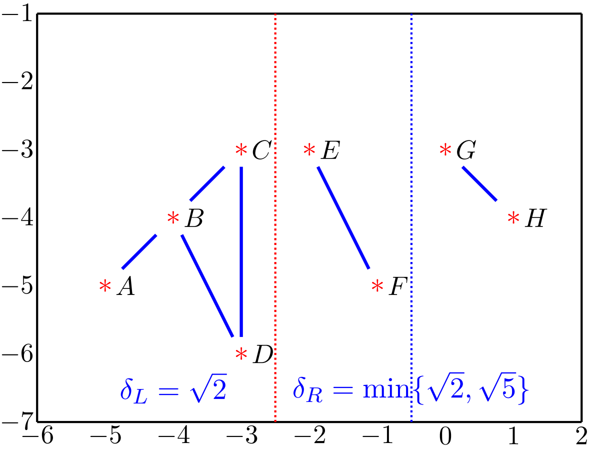
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



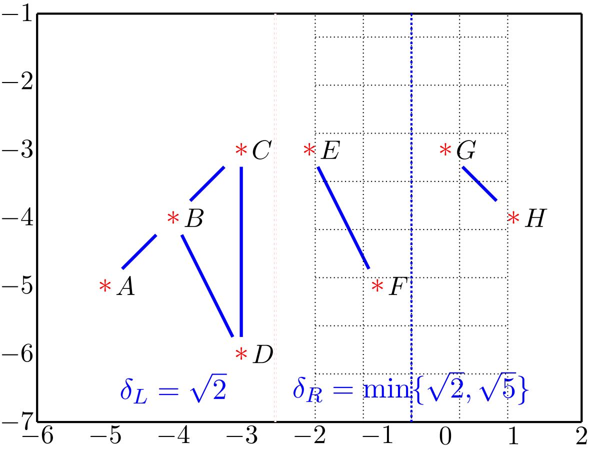
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



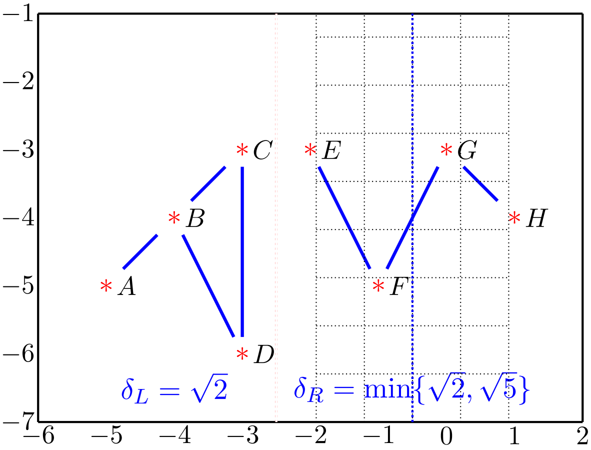
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



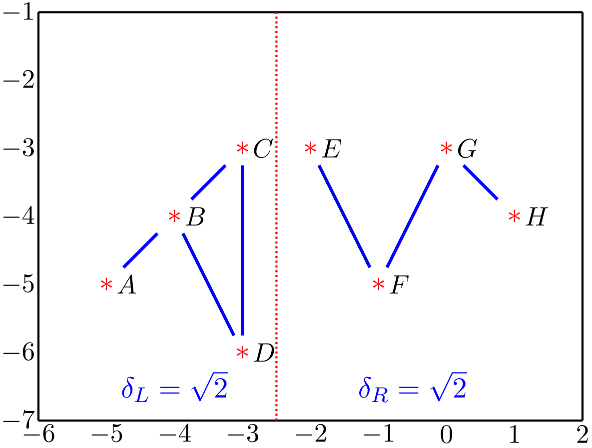
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

Right half: E, F, G, H



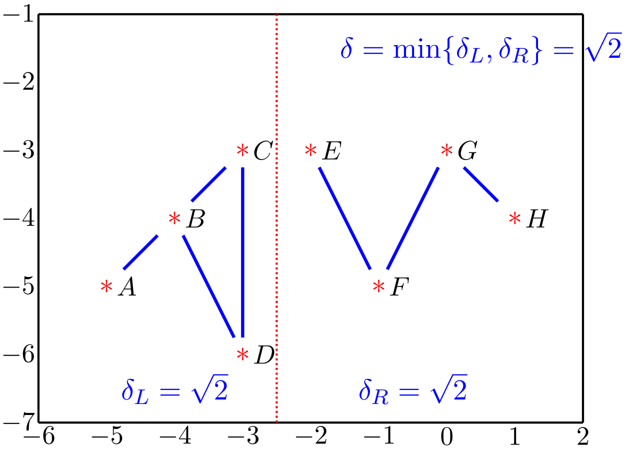
- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.

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



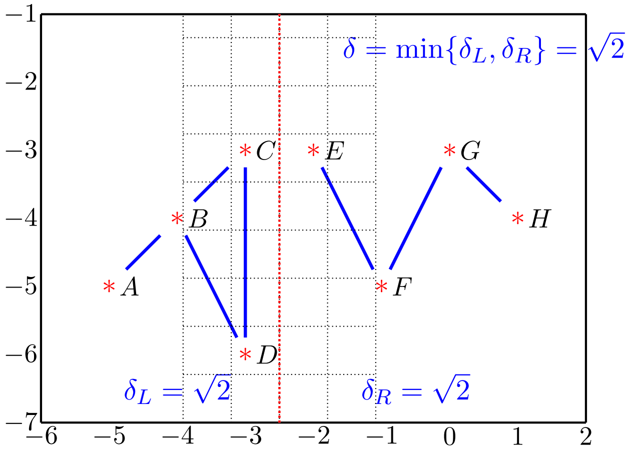
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

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



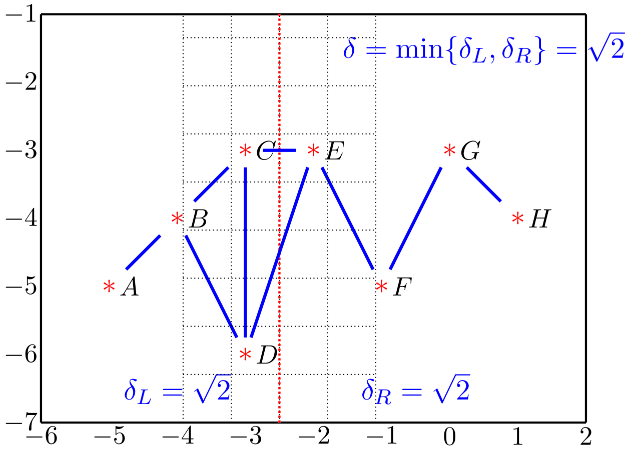
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

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



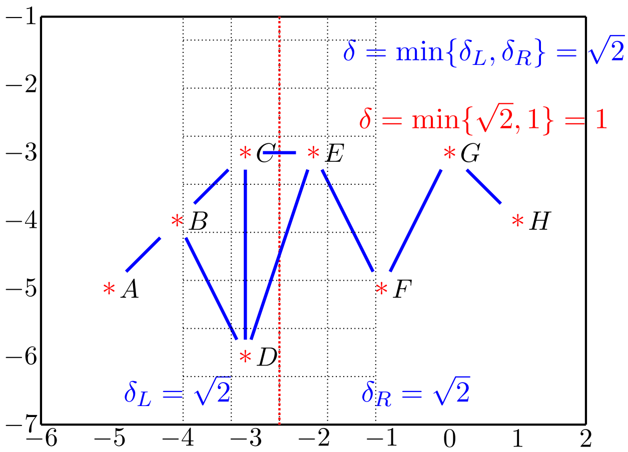
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

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



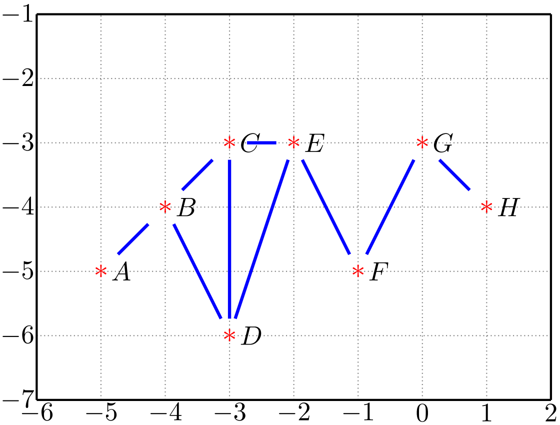
- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.

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



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

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

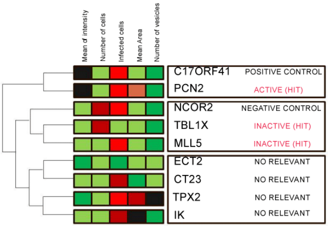


- We calculated distances for only 9 pairs of points (see 'blue' line). The other 19 pairs are redundant due to:
  - at least one of the two points lies out of 2δ-strip.
  - although two points appear in the same 2δ-strip, they are at least 2 rows of grids (size: $\frac{\delta}{2} \times \frac{\delta}{2}$) 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, Ward, or other bottom-up clustering methods in which clusters are represented by objects, in time* $O(n^2 \log^2 n)$ *and space* $O(n)$.

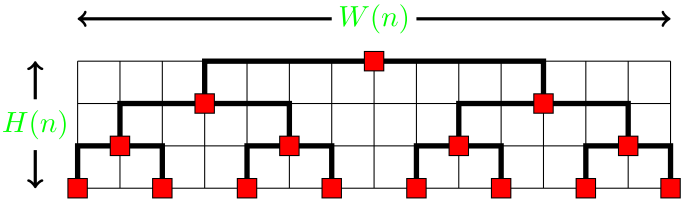


(See Eppstein 1998 for details.)

VLSI embedding: to embed a tree

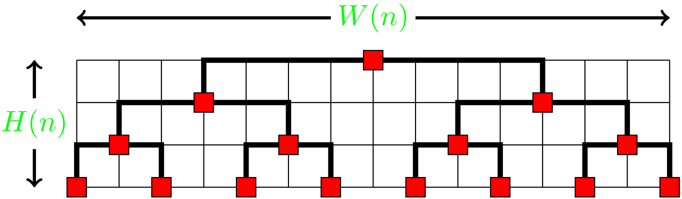
Embedding a tree

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



Trial 1: divide into two sub-trees

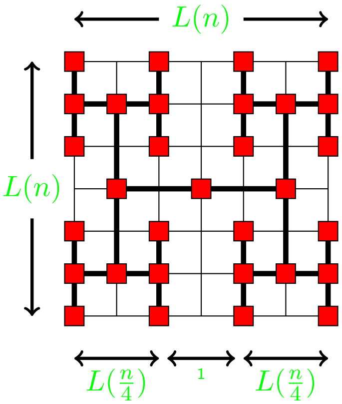
- Let's divide into 2 sub-trees, each with a size of $\frac{n}{2}$.



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