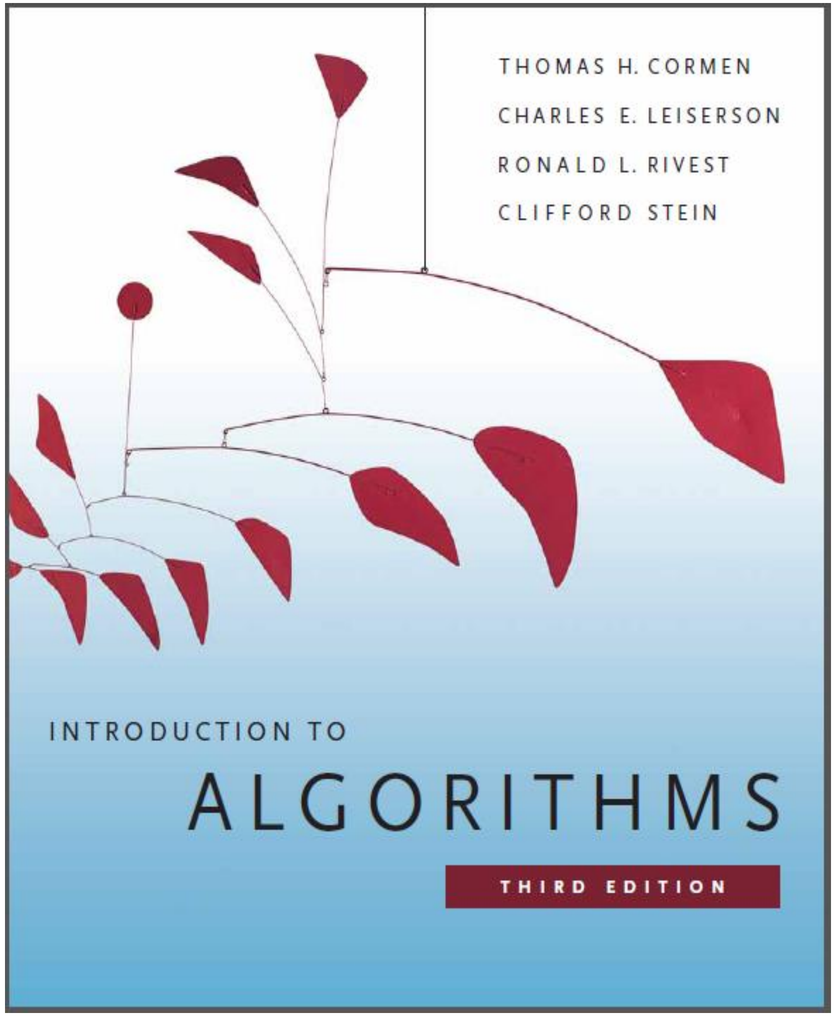
Lecture 1
Greedy, Divide-and-conquer

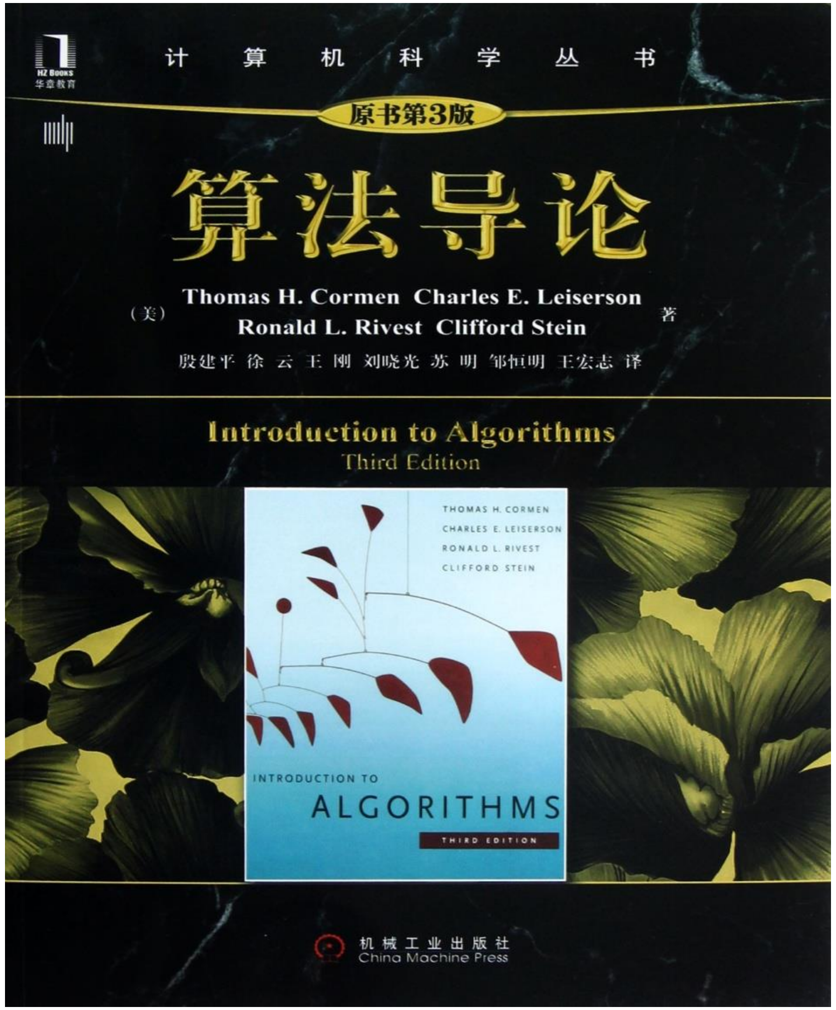
Algorithm

张子臻，中山大学计算机学院

zhangzizhen@gmail.com

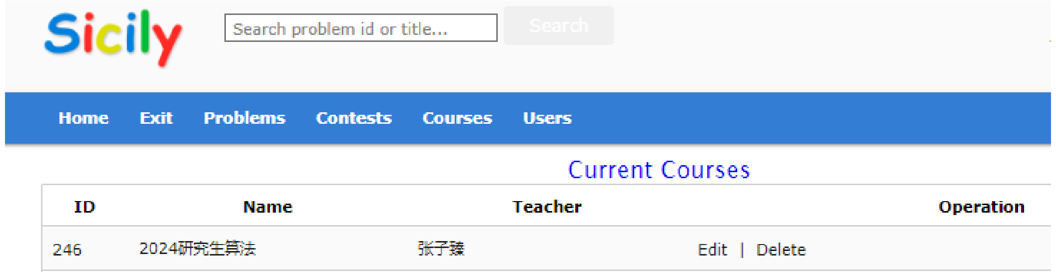
Textbook


THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
INTRODUCTION TO
ALGORITHMS
THIRD EDITION


计 算 机 科 学 丛 书
原书第3版
算法导论
（美） Thomas H. Cormen Charles E. Leiserson
Ronald L. Rivest Clifford Stein 著
殷建平 徐 云 王 刚 刘晓光 苏 明 邹恒明 王宏志 译
Introduction to Algorithms
Third Edition
机械工业出版社
China Machine Press

Online Platforms

- http://soj.acmm.club/



- https://leetcode.com/
- http://codeforces.com/

Greedy Algorithm

- Definition:

  A *greedy algorithm* is an algorithm in which at each stage a locally optimal choice is made.

- Characteristics:

  1. Greedy-choice property: A global optimum can be arrived at by selecting a local optimum.

  2. Optimal substructure: An optimal solution to the problem contains an optimal solution to subproblems.

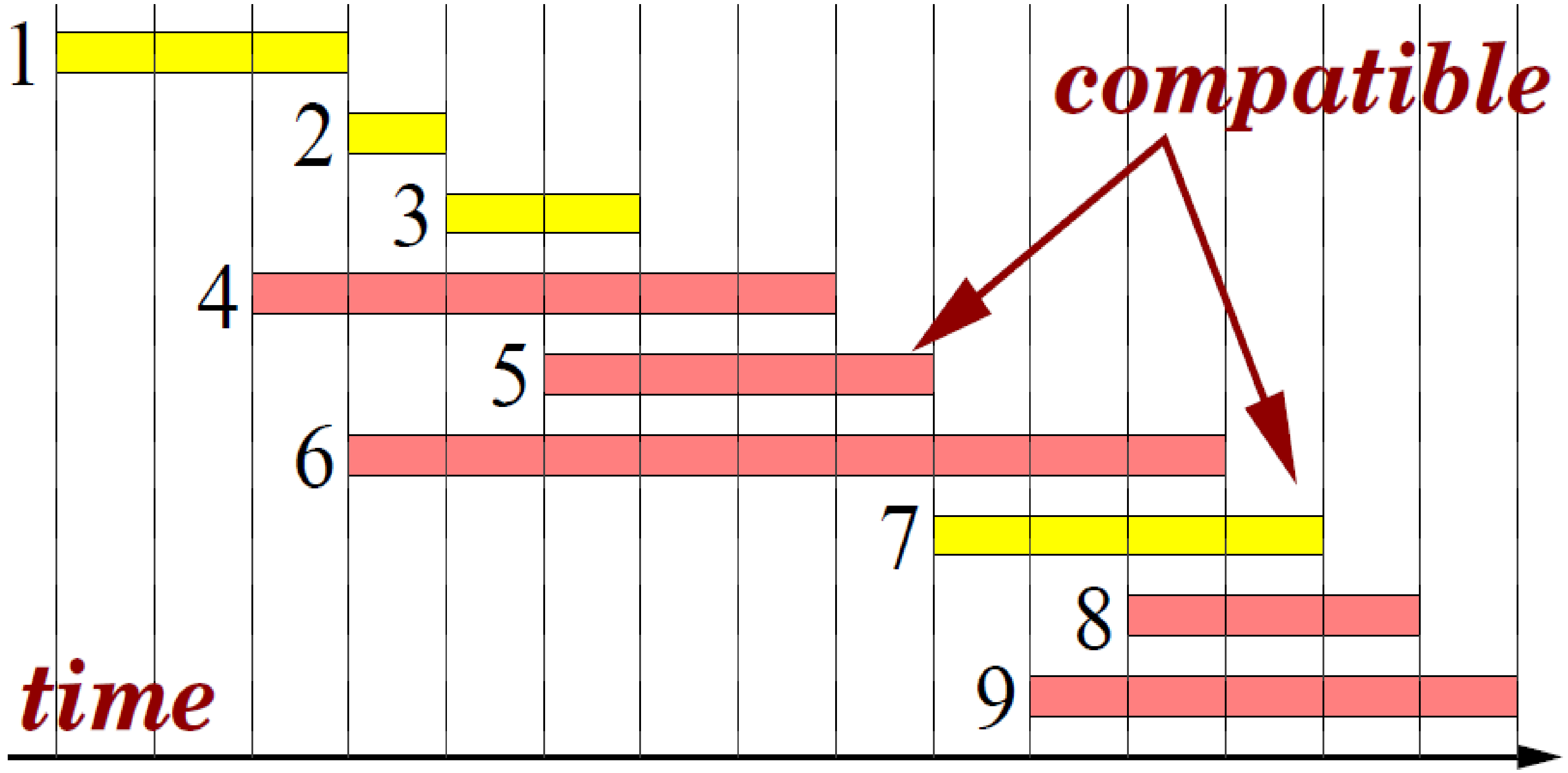
- Greedy algorithms are usually extremely efficient, but they can only be applied to a small number of problems.

Activity Selection Problem

- Let $S = \{1, 2, . . . , n\}$ be the set of activities that compete for a resource. Each activity i has its starting time s_i and finish time f_i with $s_i \leq f_i$, namely, if selected, i takes place during time $[s_i, f_i)$. No two activities can share the resource at any time point. We say that activities i and j are compatible if their time periods are disjoint.

- The *activity selection problem* is the problem of selecting the largest set of mutually compatible activities.

Activity Selection Problem



Activity Selection Problem

- **Greedy template**. Consider activities in some natural order. Take each activity provided it's compatible with the ones already taken.

  [Earliest start time] Consider jobs in ascending order of s_i.

  [Earliest finish time] Consider jobs in ascending order of f_i.

  [Shortest interval] Consider jobs in ascending order of f_i-s_i.

  [Fewest conflicts] For each job i, count the number of conflicting jobs c_i. Schedule in ascending order of c_i.

Activity Selection Problem

- Counterexamples:

counterexample for earliest start time

counterexample for shortest interval

counterexample for fewest conflicts

Activity Selection Problem

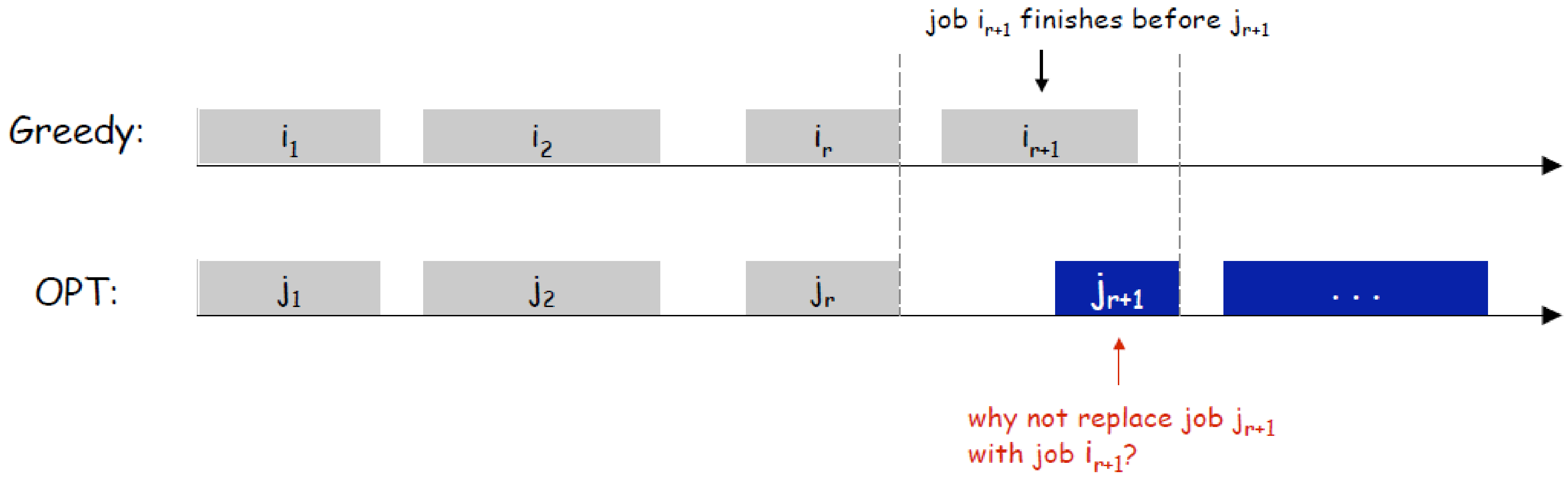
- Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort activities by finish times so that f1 <= f2 <= ... <= fn
A = Φ
for j = 1 to n {
    if (activity j compatible with A) A = A ∪ {j}
}
return A
```

- Implementation. O(nlogn)+O(n)

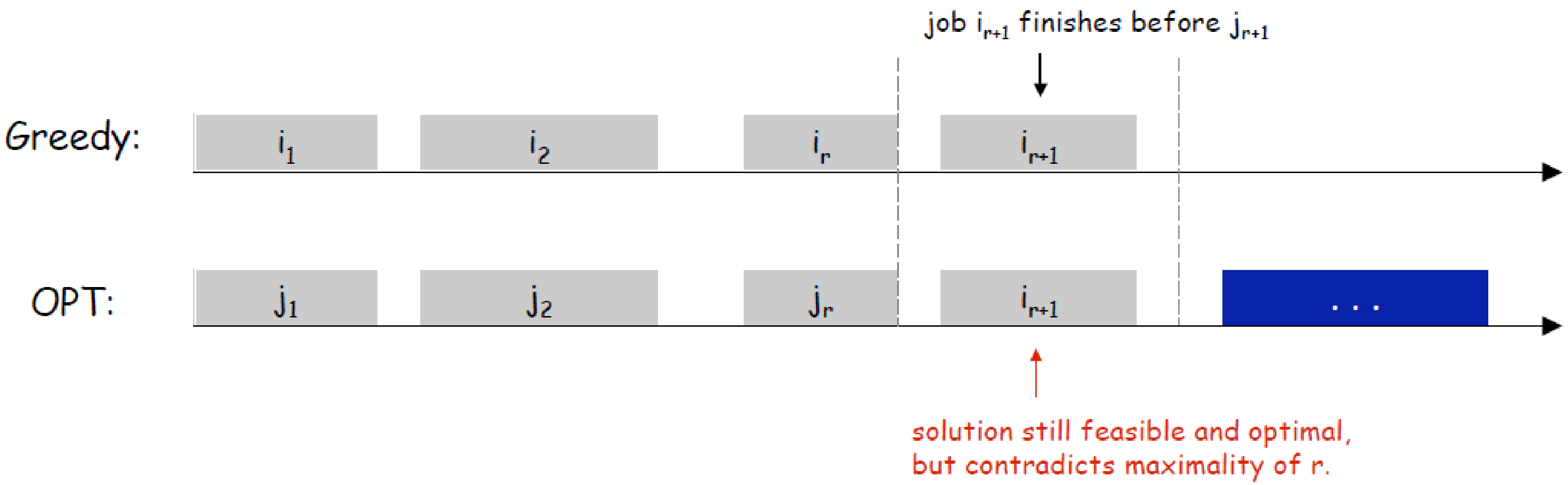
Activity Selection Problem

- **Theorem**: Greedy algorithm is optimal for the activity selection problem.
- **Proof: (by contradiction)**
  - Assume greedy is not optimal.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



Activity Selection Problem

- **Theorem**: Greedy algorithm is optimal for the activity selection problem.
- **Proof: (by contradiction)**
  - Assume greedy is not optimal.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1$, $i_2 = j_2$, ..., $i_r = j_r$ for the largest possible value of r.



Fractional Knapsack Problem

- We have n objects and a knapsack. The i-th object has positive weight w_i and positive value v_i. The knapsack capacity is C. We wish to select a set of proportions of objects to put in the knapsack so that the total values is maximum and without breaking the knapsack.

- Example:
- $n = 5$, $C = 100$

$$\max . \sum_{i=1}^{n} v_i x_i$$

$$s.t. \sum_{i=1}^{n} w_i x_i \leq W$$

$$0 \leq x_i \leq 1$$

w	10	20	30	40	50
v	20	30	66	40	60

Fractional Knapsack Problem

- **Greedy template**.

  [Select always the lighter object] Total selected weight 100 and total value 156.

object	1	2	3	4	5
selected	1	1	1	1	0

  [Select always the most valuable object] Total selected weight 100 and total value 146.

object	1	2	3	4	5
selected	0	0	1	0.5	1

  [Select always the object with highest ratio value/weight] Total selected weight 100 and total value 164.

object	1	2	3	4	5
ratio	2.0	1.5	2.2	1.0	1.2
selected	1	1	1	0	0.8

Fractional Knapsack Problem

- **Theorem:** The greedy algorithm that always selects the object with better ratio value/weight always finds an optimal solution to the Fractional Knapsack problem.
- **Proof:**

Assume that the objects are {1, ... , n} and that

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

Let $X = (x_1, \ldots, x_n)$ be the solution computed by the greedy algorithm.

If $x_i = 1$ for all i, the solution is optimal. Otherwise, let j be the smallest value for which $x_j < 1$. According to the algorithm we have: If $i < j$ then $x_i = 1$, and if $i > j$ then $x_i = 0$.

Furthermore,

$$\sum_{i=1}^{n} w_i x_i = W$$

Fractional Knapsack Problem

- Let $Y = (y_1, \ldots , y_n)$ be any feasible solution, we have

$$\sum_{i=1}^{n} w_i y_i \le W = \sum_{i=1}^{n} w_i x_i$$

so, $\sum_{i=1}^{n} w_i (x_i - y_i) \ge 0$

Let $V(.)$ denotes the total value of a feasible solution.

$$V(X) - V(Y) = \sum_{i=1}^{n} v_i (x_i - y_i) = \sum_{i=1}^{n} w_i \frac{v_i}{w_i} (x_i - y_i)$$

If $i<j$, $x_i=1$, then $x_i-y_i>=0$ and $v_i/w_i>=v_j/w_j$, we have

$$(x_i - y_i)\frac{v_i}{w_i} \ge (x_i - y_i)\frac{v_j}{w_j}$$

If $i>j$, $x_i=0$, then $x_i-y_i<=0$ but $v_i/w_i<=v_j/w_j$, we also have

$$(x_i - y_i)\frac{v_i}{w_i} \ge (x_i - y_i)\frac{v_j}{w_j}$$

Fractional Knapsack Problem

- Plugging the inequality we have,

$$V(X)-V(Y)=\sum_{i=1}^{n} w_i \frac{v_i}{w_i}(x_i-y_i) \ge \sum_{i=1}^{n} w_i \frac{v_j}{w_j}(x_i-y_i)$$

$$=\frac{v_j}{w_j}\sum_{i=1}^{n} w_i(x_i-y_i) \ge 0$$

- Therefore, *X* is an optimal solution.

0-1 Knapsack Problem

$$\max . \sum_{i=1}^{n} v_i x_i$$

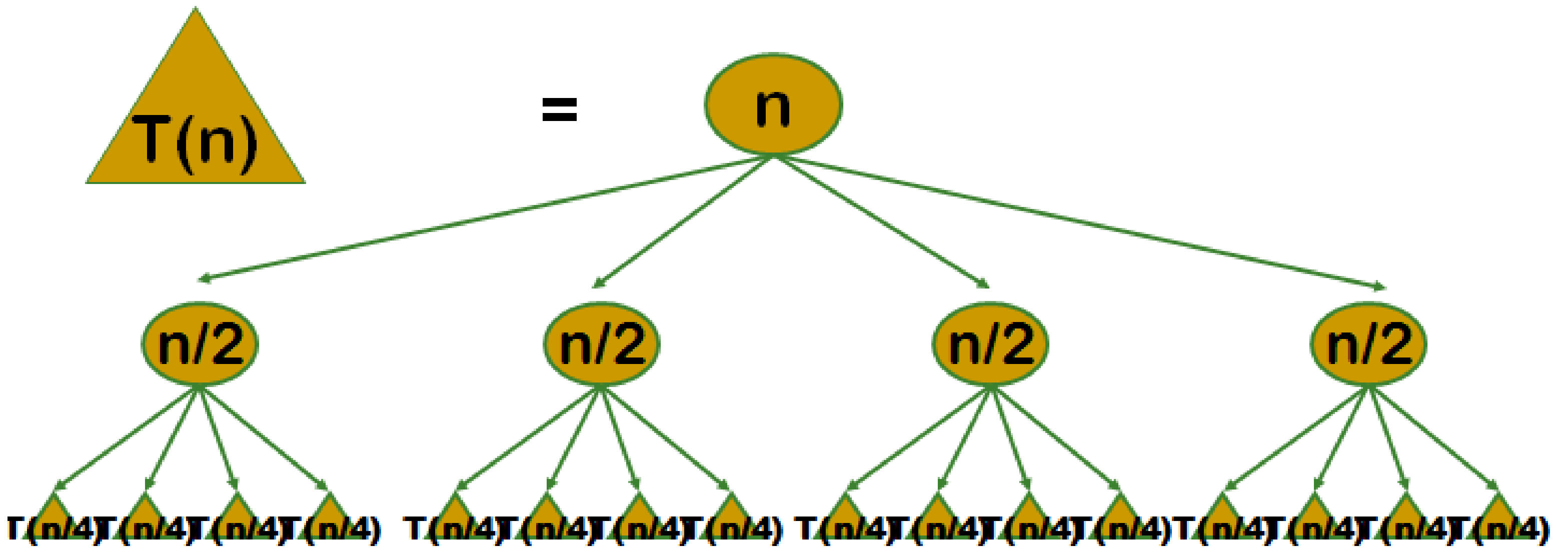
$$s.t. \sum_{i=1}^{n} w_i x_i \le W$$

$$x_i \in \{0,1\}$$

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ and } w \geqslant w_i \end{cases}$$

Divide-and-conquer

- A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.



Merge-sort

```
procedure MERGE-SORT(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
            MERGE-SORT(A, p, q)
            MERGE-SORT(A, q + 1, r)
            MERGE(A, p, q, r)
```

```
procedure MERGE(A, p, q, r)
    n1 ← q − p + 1; n2 ← r − q
    allocate arrays L[1 . . . n1 + 1] and R[1 . . . n2 + 1]
    for i ← 1 to n1
        do L[i] ← A[p + i − 1]
    for j ← 1 to n2
        do R[j] ← A[q + j]
    L[n1 + 1] ← ∞; R[n2 + 1] ← ∞
    i ← 1; j ← 1
    for k ← p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
                i ← i + 1
            else A[k] ← R[j]
                j ← j + 1
```

Analysis of the Merge-sort algorithm

- Described by recursive equation
- Suppose $T(n)$ is the running time on a problem of size n.
- $T(n) = \begin{cases} \Theta(1) & \text{if } n \le n_c \\ aT(n/b)+D(n)+C(n) & \text{if } n > n_c \end{cases}$

where a: number of subproblems

n/b: size of each subproblem

$D(n)$: cost of divide operation

$C(n)$: cost of combination operation

Analysis of the Merge-sort algorithm

- **Divide**: $D(n) = \Theta(1)$
- **Conquer**: $a=2, b=2$, so $2T(n/2)$
- **Combine**: $C(n) = \Theta(n)$
- $T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2)+\Theta(n) & \text{if } n>1 \end{cases}$
- $T(n) = \begin{cases} c & \text{if } n=1 \\ 2T(n/2)+cn & \text{if } n>1 \end{cases}$

Analysis of the Merge-sort algorithm

- The recursive equation can be solved by recursive tree.
- $T(n) = 2T(n/2)+ cn,$
- lg n+1 levels, cn at each level, thus
- Total cost for merge sort is:

  $T(n) = cn\lg n + cn = \Theta(n\lg n)$.
- Question: best, worst, average?

Analysis of the Merge-sort algorithm

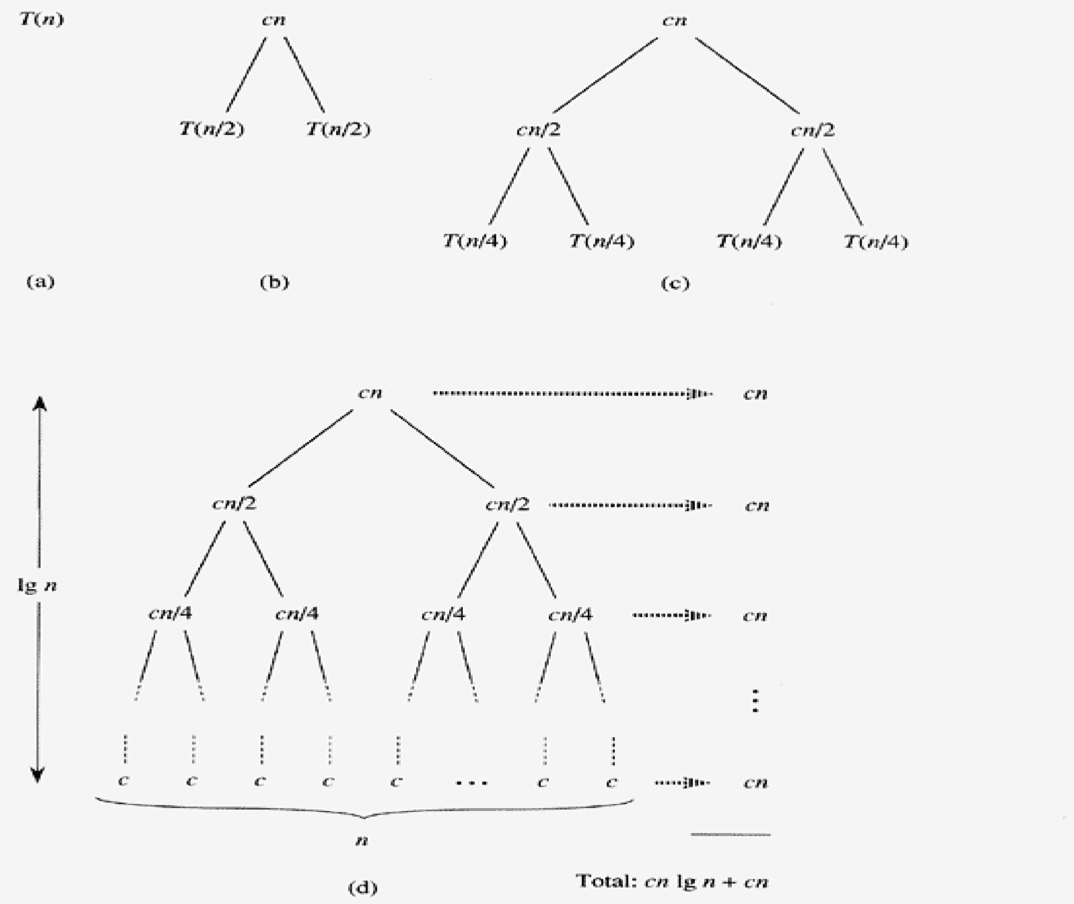


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

Master theorem

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ∎

Exercise

- Use the master method to give tight asymptotic bounds for the following recurrences.
  - (a) $T(n)=2T(n/4)+1$
  - (b) $T(n)=2T(n/4)+\sqrt{n}$
  - (c) $T(n)=2T(n/4)+n$
  - (d) $T(n)=2T(n/4)+n^2$

Multiplication of Large Integers

Consider the problem of multiplying two (large) *n*-digit integers represented by arrays of their digits such as:

A = 12345678901357986429 B = 87654321284820912836

The grade-school algorithm:

$$
\begin{array}{rl}
 & a_1 \; a_2 \ldots \; a_n \\
 & \underline{b_1 \; b_2 \ldots \; b_n} \\
 & (d_{10})\, d_{11} d_{12} \ldots d_{1n} \\
 & (d_{20})\, d_{21} d_{22} \ldots d_{2n} \\
 & \ldots \ldots \ldots \ldots \ldots \ldots \ldots \\
 & \underline{(d_{n0})\, d_{n1} d_{n2} \ldots d_{nn}}
\end{array}
$$

Efficiency: $\Theta(n^2)$ single-digit multiplications

First Divide-and-Conquer Algorithm

A small example: $A * B$ where $A = 2135$ and $B = 4014$

$A = (21 \cdot 10^2 + 35)$, $B = (40 \cdot 10^2 + 14)$

So, $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are n-digit, A_1, A_2, B_1, B_2 are $n/2$-digit numbers),

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

Recurrence for the number of one-digit multiplications $M(n)$:

$$M(n) = 4M(n/2), \quad M(1) = 1$$

Solution: $M(n) = n^2$

Second Divide-and-Conquer Algorithm

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

The idea is to decrease the number of multiplications from 4 to 3:

$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$

i.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$ which requires only 3 multiplications at the expense of 3 extra add/sub.

Recurrence for the number of multiplications M(n):

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

Karatsuba Multiplication Algorithm

KARATSUBA-MULTIPLY(x, y, n)

IF $(n = 1)$

  RETURN $x \times y$.

ELSE

  $m \leftarrow \lceil n / 2 \rceil$.

  $a \leftarrow \lfloor x / 2^m \rfloor$; $b \leftarrow x \bmod 2^m$.

  $c \leftarrow \lfloor y / 2^m \rfloor$; $d \leftarrow y \bmod 2^m$.

  $e \leftarrow$ KARATSUBA-MULTIPLY(a, c, m).

  $f \leftarrow$ KARATSUBA-MULTIPLY(b, d, m).

  $g \leftarrow$ KARATSUBA-MULTIPLY($a - b$, $c - d$, m).

  RETURN $2^{2m}\, e + 2^m\, (e + f - g) + f$.

Example of Large-Integer Multiplication

2135 * 4014

= (21*10^2 + 35) * (40*10^2 + 14)

= (21*40)*10^4 + c1*10^2 + 35*14

where c1 = (21+35) *(40+14) - 21*40 - 35*14, and

21*40 = (2*10 + 1) * (4*10 + 0)

= (2*4)*10^2 + c2*10 + 1*0

where c2 = (2+1) *(4+0) - 2*4 - 1*0, etc.

- This process requires 9 digit multiplications as opposed to 16.

Matrix multiplication

- Given two *n-by-n* matrices *A* and *B*, compute *C* = *AB.*
- Grade-school. $\Theta(n^3)$ arithmetic operations.

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$\begin{bmatrix} .59 & .32 & .41 \\ .31 & .36 & .25 \\ .45 & .31 & .42 \end{bmatrix} = \begin{bmatrix} .70 & .20 & .10 \\ .30 & .60 & .10 \\ .50 & .10 & .40 \end{bmatrix} \times \begin{bmatrix} .80 & .30 & .50 \\ .10 & .40 & .10 \\ .10 & .30 & .40 \end{bmatrix}$$

Block matrix multiplication

$$\overset{C_{11}}{\begin{bmatrix} 152 & 158 & 164 & 170 \\ 504 & 526 & 548 & 570 \\ 856 & 894 & 932 & 970 \\ 1208 & 1262 & 1316 & 1370 \end{bmatrix}} = \overset{A_{11} \qquad A_{12}}{\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}} \times \overset{B_{11}}{\underset{B_{11}}{\begin{bmatrix} 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix}}}$$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}$$

Matrix multiplication

- To multiply two *n-by-n matrices A and B:*
  - Divide: partition *A and B into 1/2n-by-1/2n* blocks.
  - Conquer: multiply 8 pairs of 1/2*n-by-1/2n* matrices, recursively.
  - Combine: add appropriate products using 4 matrix additions.

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

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

- Running time

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow T(n) = \Theta(n^3)$$

Strassen's method

- Key idea: multiply 2-by-2 blocks with only 7 multiplications. (plus 11 additions and 7 subtractions)

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

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

$$\begin{aligned} P_1 &\leftarrow A_{11} \times (B_{12} - B_{22}) \\ P_2 &\leftarrow (A_{11} + A_{12}) \times B_{22} \\ P_3 &\leftarrow (A_{21} + A_{22}) \times B_{11} \\ P_4 &\leftarrow A_{22} \times (B_{21} - B_{11}) \\ P_5 &\leftarrow (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &\leftarrow (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &\leftarrow (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

Pf. $$\begin{aligned} C_{12} &= P_1 + P_2 \\ &= A_{11} \times (B_{12} - B_{22}) + (A_{11} + A_{12}) \times B_{22} \\ &= A_{11} \times B_{12} + A_{12} \times B_{22}. \ \checkmark \end{aligned}$$

Strassen's algorithm

STRASSEN (n, A, B)

---

assume n is a power of 2

IF $(n = 1)$ RETURN $A \times B$.

Partition A and B into 2-by-2 block matrices.

keep track of indices of submatrices (don't copy matrix entries)

$P_1 \leftarrow$ STRASSEN $(n / 2, A_{11}, (B_{12} - B_{22}))$.

$P_2 \leftarrow$ STRASSEN $(n / 2, (A_{11} + A_{12}), B_{22})$.

$P_3 \leftarrow$ STRASSEN $(n / 2, (A_{21} + A_{22}), B_{11})$.

$P_4 \leftarrow$ STRASSEN $(n / 2, A_{22}, (B_{21} - B_{11}))$.

$P_5 \leftarrow$ STRASSEN $(n / 2, (A_{11} + A_{22}) \times (B_{11} + B_{22}))$.

$P_6 \leftarrow$ STRASSEN $(n / 2, (A_{12} - A_{22}) \times (B_{21} + B_{22}))$.

$P_7 \leftarrow$ STRASSEN $(n / 2, (A_{11} - A_{21}) \times (B_{11} + B_{12}))$.

$C_{11} = P_5 + P_4 - P_2 + P_6$.

$C_{12} = P_1 + P_2$.

$C_{21} = P_3 + P_4$.

$C_{22} = P_1 + P_5 - P_3 - P_7$.

RETURN C.

Analysis of Strassen's algorithm

- **Theorem**. Strassen's algorithm requires *O(n^2.81^)* arithmetic operations to multiply two *n-by-n* matrices.

$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

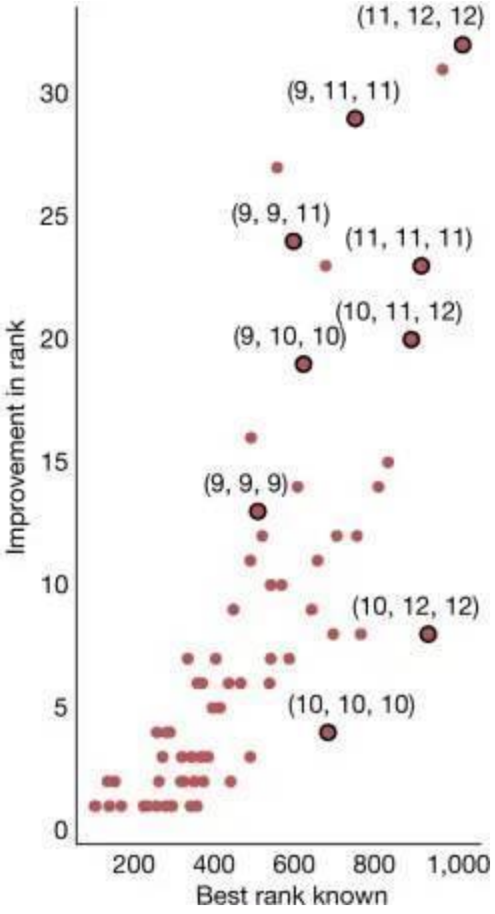
- Q. What if *n* is not a power of *2* ?
- A. Could pad matrices with zeros.

$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 10 & 11 & 12 & 0 \\ 13 & 14 & 15 & 0 \\ 16 & 17 & 18 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 84 & 90 & 96 & 0 \\ 201 & 216 & 231 & 0 \\ 318 & 342 & 366 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

AlphaTensor

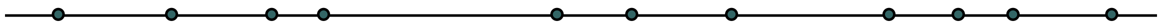
- https://www.nature.com/articles/s41586-022-05172-4

Size (n, m, p)	Best method known	Best rank known	AlphaTensor rank Modular	AlphaTensor rank Standard
(2, 2, 2)	(Strassen, 1969)[2]	7	7	7
(3, 3, 3)	(Laderman, 1976)[15]	23	23	23
(4, 4, 4)	(Strassen, 1969)[2] (2, 2, 2) ⊗ (2, 2, 2)	49	47	49
(5, 5, 5)	(3, 5, 5) + (2, 5, 5)	98	96	98
(2, 2, 3)	(2, 2, 2) + (2, 2, 1)	11	11	11
(2, 2, 4)	(2, 2, 2) + (2, 2, 2)	14	14	14
(2, 2, 5)	(2, 2, 2) + (2, 2, 3)	18	18	18
(2, 3, 3)	(Hopcroft and Kerr, 1971)[16]	15	15	15
(2, 3, 4)	(Hopcroft and Kerr, 1971)[16]	20	20	20
(2, 3, 5)	(Hopcroft and Kerr, 1971)[16]	25	25	25
(2, 4, 4)	(Hopcroft and Kerr, 1971)[16]	26	26	26
(2, 4, 5)	(Hopcroft and Kerr, 1971)[16]	33	33	33
(2, 5, 5)	(Hopcroft and Kerr, 1971)[16]	40	40	40
(3, 3, 4)	(Smirnov, 2013)[18]	29	29	29
(3, 3, 5)	(Smirnov, 2013)[18]	36	36	36
(3, 4, 4)	(Smirnov, 2013)[18]	38	38	38
(3, 4, 5)	(Smirnov, 2013)[18]	48	47	47
(3, 5, 5)	(Sedoglavic and Smirnov, 2021)[19]	58	58	58
(4, 4, 5)	(4, 4, 2) + (4, 4, 3)	64	63	63
(4, 5, 5)	(2, 5, 5) ⊗ (2, 1, 1)	80	76	76

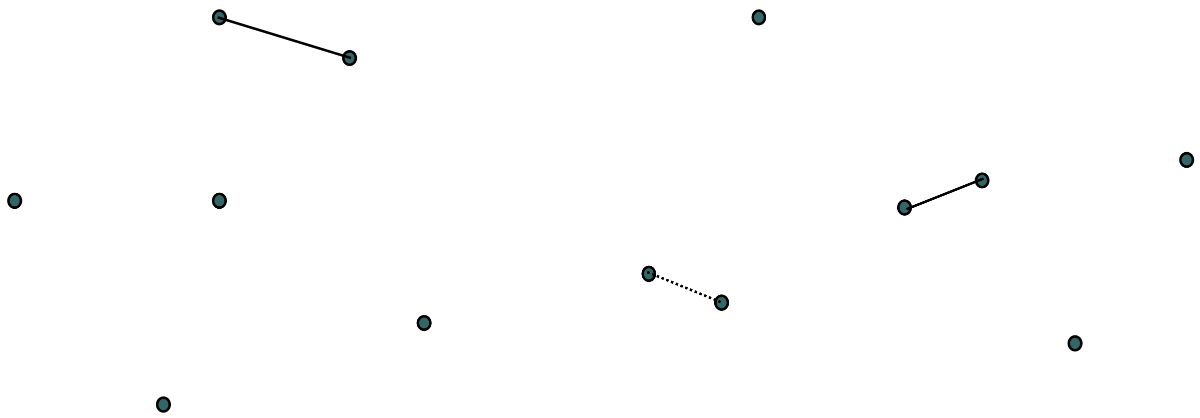


Closest Pair

- Given a set S = {p_1, p_2, ..., p_n} of *n* points in the plane find the two points of *S* whose distance is the smallest.
- 1-D



- 2-D



Closest Pair – Naïve Algorithm

Pseudo code

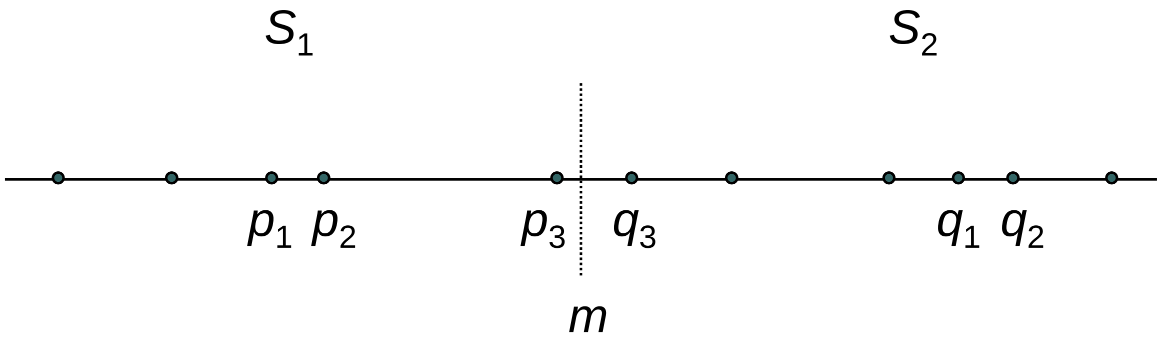
for each pt i∈S
    for each pt j∈S and i<>j
    {
      compute distance of i, j
      if distance of i, j < min_dist
          min_dist = distance i, j
    }
return min_dist

- Time Complexity– $O(n^2)$
- Can we do better?

1-D Closest Pair – Divide & Conquer

- We consider a divide-and-conquer algorithm for CLOSEST-PAIR in 1 dimension (d = 1).
- Partition S, a set of points on a line, into two sets S_1 and S_2 at some point m such that for every point $p \in S_1$ and $q \in S_2$, $p < q$.
- Solving CLOSEST-PAIR recursively on S_1 and S_2 separately produces $\{p_1, p_2\}$, the closest pair in S_1, and $\{q_1, q_2\}$, the closest pair in S_2.
- Let δ be the smallest distance found so far:

  $\delta = \min(|p_2 - p_1|, |q_2 - q_1|)$
- The closest pair in S is either $\{p_1, p_2\}$ or $\{q_1, q_2\}$ or some $\{p_3, q_3\}$ with $p_3 \in S_1$ and $q_3 \in S_2$.



1-D Closest Pair – Divide & Conquer

- To check for such a point $\{p_3, q_3\}$, is it necessary to test every possible pair of points in S_1 and S_2?
- Note that if $\{p_3, q_3\}$ is to be closer than δ (i.e., $|q_3 - p_3| < \delta$), then both p_3 and q_3 must be within δ of m.
- Because δ is the distance between the closest pair in either S_1 or S_2, a semi-closed interval of length δ can contain at most 1 point.
- For the same reason, there can be at most 1 point of S_2 within δ of m
- So, the number of distance computations needed to check for a closest pair $\{p_3, q_3\}$ with $p_3 \in S_1$ and $q_3 \in S_2$ is 1, not $O(N^2)$.
- Thus a divide-and-conquer algorithm can solve 1-dimensional CLOSEST-PAIR in $O(N \log N)$ time.

1-D Closest Pair – Divide & Conquer

Divide-and-conquer for *d* = 1

procedure CPAIR1(S)

Input: $X[1:N]$, N points of S in one dimension.

Output: δ, the distance between the two closest points.

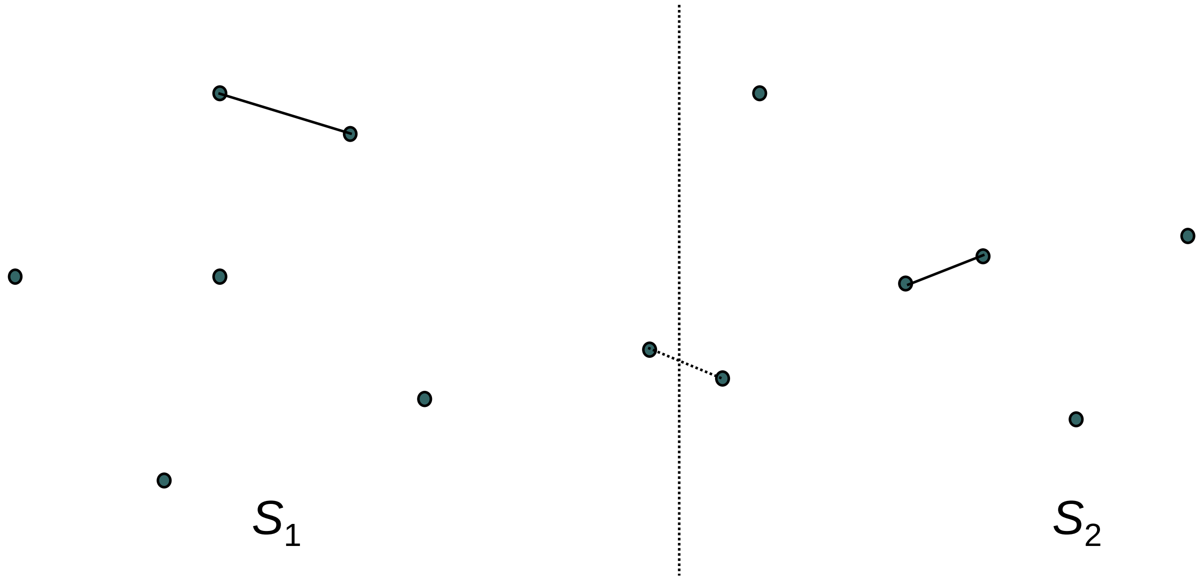
```
begin
    if   (|S| = 2) then
        δ = |X[2] - X[1]|
    else if (|S| = 1) then
        δ = ∞
    else
        begin
            Construct(S1, S2)  /* S1 = {p: p ≤ m}, S2 = {p: p > m} */
            δ1 = CPAIR1(S1)
            δ2 = CPAIR1(S2)
            p = max(S1)
            q = min(S2)
            δ = min(δ1, δ2, q - p)
        end
    endif
    return δ
end
```

Closest Pair – Divide & Conquer

- Divide the problem into two equal-sized sub problems
- Solve those sub problems recursively
- Merge the sub problem solutions into an overall solution

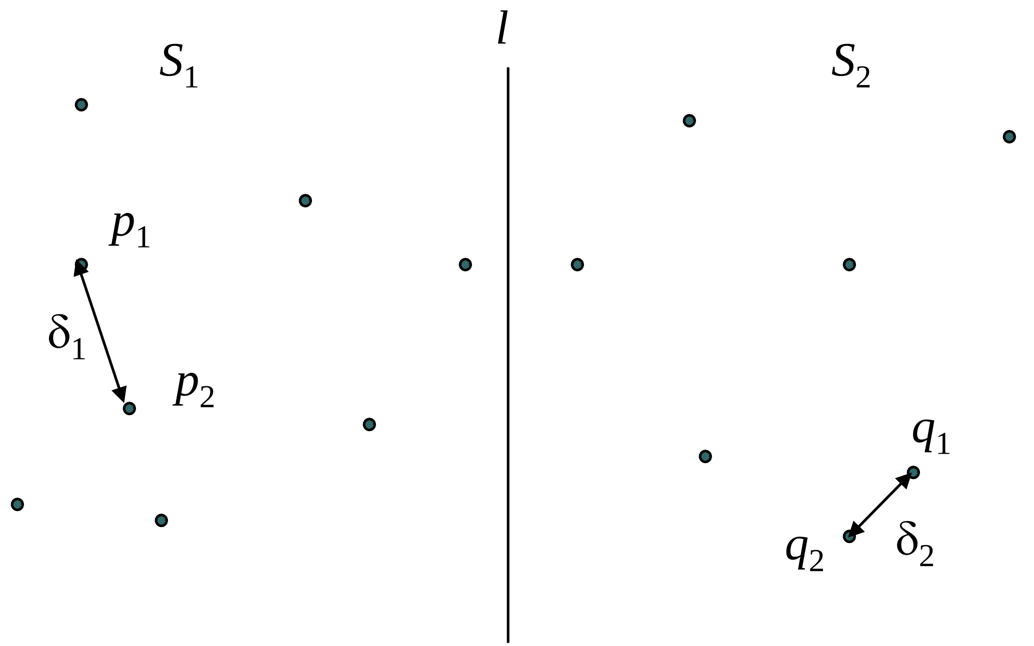
Closest Pair – Divide & Conquer

- Assume that we have solutions for sub problems S1, S2.
- How can we merge in a time-efficient way?
  - The closest pair can consist of one point from S_1 and another from S_2
  - Testing all possibilities requires: $O(n/2) \cdot O(n/2) \in O(n^2)$
  - Not good enough



Closest Pair – Divide & Conquer

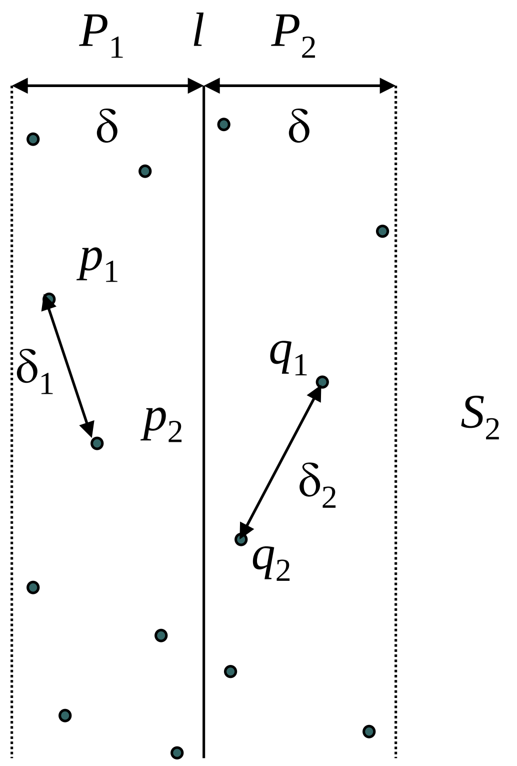
- Partition two dimensional set S into subsets S_1 and S_2 by a vertical line l at the median x coordinate of S.
- Solve the problem recursively on S_1 and S_2.
- Let {$p1$, $p2$} be the closest pair in S_1 and {$q1$, $q2$} in S_2.
- Let δ_1 = distance($p1$,$p2$) and δ_2 = distance($q1$,$q2$)
- Let $\delta = \min(\delta_1, \delta_2)$



Closest Pair – Divide & Conquer

- In order to merge we have to determine if exists a pair of points $\{p, q\}$ where $p \in S_1$, $q \in S_2$ and distance$(p, q) < \delta$.
- If so, p and q must both be within δ of l.
- Let P_1 and P_2 be vertical regions of the plane of width δ on either side of l.
- If $\{p, q\}$ exists, p must be within P_1 and q within P_2.
- However, every point in S_1 and S_2 may be a candidate, as long as each is within δ of l, which implies: $O(n/2) \cdot O(n/2) = O(n^2)$

- Can we do better ?

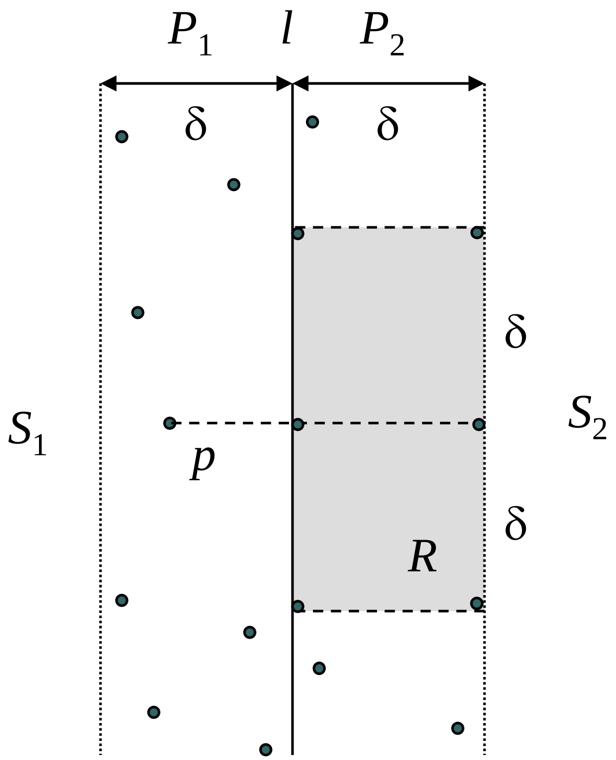


Closest Pair – Divide & Conquer

How many points are there in rectangle *R*?

- Since no two points can be closer than δ, there can only be at most 6 points
- Therefore, $6 \cdot O(n/2) \in O(n)$
- Thus, the time complexity is
  - $O(n \log n)$

- How do we know which 6 points to check?



Closest Pair – Divide & Conquer

How do we know which 6 points to check?

- Project *p* and all the points of S_2 within P_2 onto *l*.
- Only the points within δ of *p* in the *y* projection need to be considered (max of 6 points).
- After sorting the points on y coordinate we can find the points by scanning the sorted lists. Points are sorted by y coordinates.
- To prevent resorting in O(n log n) in each merge, two previously sorted lists are merged in O(n).

Time Complexity: O(n log n)

Thank you!