Camp Προετοιμασίας

Αλγόριθμοι

Νικόλαος Σ. Παπασπύρου

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχ. & Μηχ. Υπολογιστών

Ιούνιος 2007

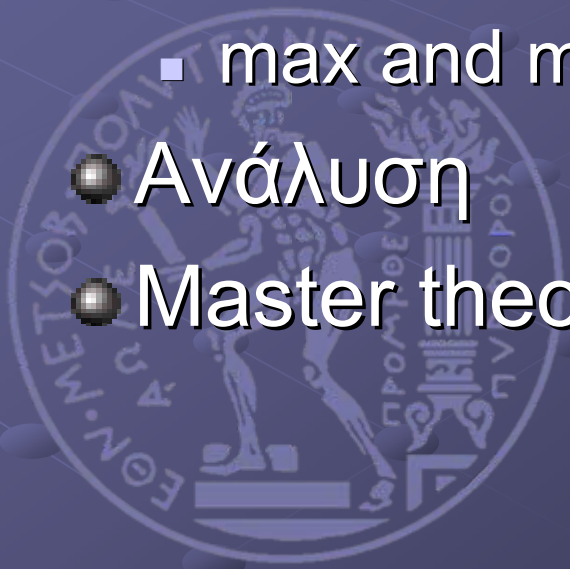
Σύνοψη

- Διαίρει και κυρίευε
  Divide and conquer
- Άπληστοι αλγόριθμοι
  Greedy algorithms
- Δυναμικός προγραμματισμός
  Dynamic programming

Divide and conquer

- Ιδέα
- Παραδείγματα
  - merge sort
  - binary search
  - max and min
- Ανάλυση
- Master theorem

- Προβλήματα
  - count inversions
  - closest pair
  - integer arithmetic
  - matrix multiplication
  - k-th element



Εύρεση Μικρότερου-Μεγαλύτερου

- Χώρισε σε 2 υποακολουθίες
- Βρες μεγαλύτερο και μικρότερο
- Σύγκρινε μεγαλύτερα και μικρότερα

$$T(n) = \begin{cases} 0 & , \text{για } n = 1 \\ 1 & , \text{για } n = 2 \\ 2T(\frac{n}{2}) + 2 & , \text{για } n > 2 \end{cases}$$

$$T(n) = \frac{n}{2}T(2) + 2^{k+1} - 2 = \frac{3}{2}n - 2 \qquad (\frac{n}{2} = 2^k)$$

Analysis of Recurrence

$$T(N) \le \begin{cases} 0 & \text{if } N = 1 \\ \underbrace{T(\lceil N/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor N/2 \rfloor)}_{\text{solve right half}} + \underbrace{cN}_{\text{combine}} & \text{otherwise} \end{cases}$$

$$\Rightarrow T(N) \le cN \lceil \log_2 N \rceil$$

Proof by induction on N.

- **Base case: N = 1.**
- **Define $n_1 = \lfloor n / 2 \rfloor$, $n_2 = \lceil n / 2 \rceil$.**
- **Induction step: assume true for 1, 2, . . . , N – 1.**

$$\begin{aligned} T(N) &\le T(n_1) + T(n_2) + cn \\ &\le cn_1 \lceil \log_2 n_1 \rceil + cn_2 \lceil \log_2 n_2 \rceil + cn \\ &\le cn_1 \lceil \log_2 n_2 \rceil + cn_2 \lceil \log_2 n_2 \rceil + cn \\ &= cn \lceil \log_2 n_2 \rceil + cn \\ &\le cn(\lceil \log_2 n \rceil - 1) + cn \\ &= cn \lceil \log_2 n \rceil \end{aligned}$$

$$\begin{aligned} n_2 &= \lceil n/2 \rceil \\ &\le \left\lceil 2^{\lceil \log_2 n \rceil}/2 \right\rceil \\ \Rightarrow \log_2 n_2 &\le \lceil \log_2 n \rceil - 1 \end{aligned}$$

Το Κύριο Θεώρημα (Master Theorem)

Θεώρημα 6.9.1. Master Theorem *Εστω* $a \geq$ *και* $b > 1$ *σταθερές,* $f(n)$ *μια συνάρτηση, και η* $T(n)$ *ορίζεται στους μη αρνητικούς ακεραίους από την αναδρομή*

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

(το n/b *σημαίνει είτε* $\lfloor n/b \rfloor$ *είτε* $\lceil n/b \rceil$*). Τότε η* $T(n)$ *μπορεί να φραχτεί ασυμπτωτικά ως εξής:*

- $T(n) = \Theta(f(n))$, *αν* $f(n) = \Omega(n^{\log_b a+\epsilon})$ *για κάποια σταθερά* $\epsilon > 0$, *και αν* $af(n/b) \leq cf(n)$ *για κάποια σταθερά* $c > 1$ *και όλα τα αρκετά μεγάλα* n
- $T(n) = \Theta(f(n)log_2 n)$, *αν* $f(n) = \Theta(n^{\log_b a})$
- $T(n) = \Theta(n^{\log_b a})$, *αν* $f(n) = O(n^{\log_b a-\epsilon})$ *για κάποια σταθερά* $\epsilon > 0$

Counting Inversions

Web site tries to match your preferences with others on Internet.

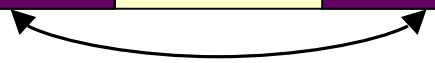
- You rank N songs.
- Web site consults database to find people with similar rankings.

Closeness metric.

- My rank = { 1, 2, . . . , N }.
- Your rank = { a_1, a_2, . . . , a_N }.
- Number of inversions between two preference lists.
- Songs i and j inverted if i < j, but $a_i > a_j$

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5



Inversion

Inversions

{3, 2}, {4, 2}

Counting Inversions

Brute-force solution.

- **Check all pairs i and j such that i < j.**
- **Θ (N^2) comparisons.**

Note: there can be a quadratic number of inversions.

- **Asymptotically faster algorithm must compute total number without even looking at each inversion individually.**

Counting Inversions: Divide-and-Conquer

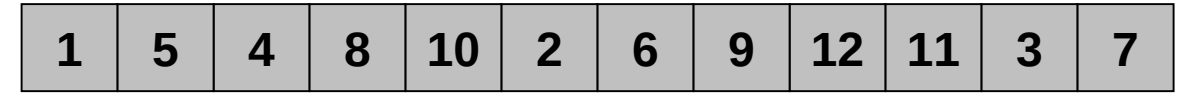
Divide-and-conquer.

1	5	4	8	10	2	6	9	12	11	3	7

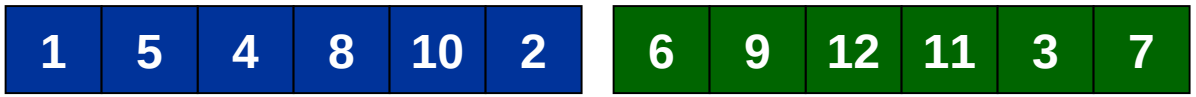
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide: separate list into two pieces.**



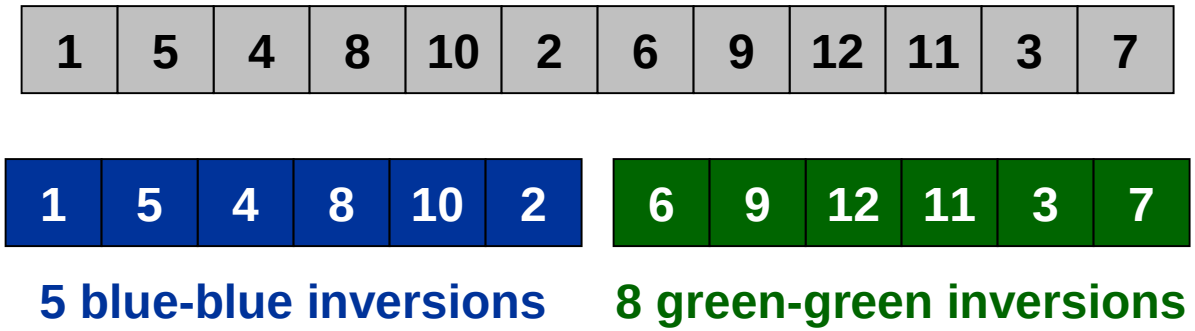
O(1)



Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

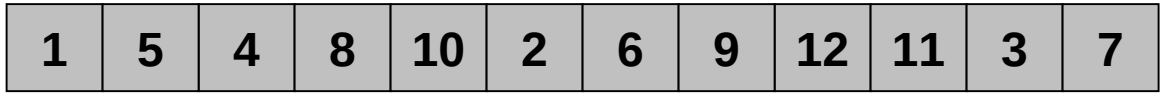
- **Divide: separate list into two pieces.**
- **Conquer: recursively count inversions in each half separately.**



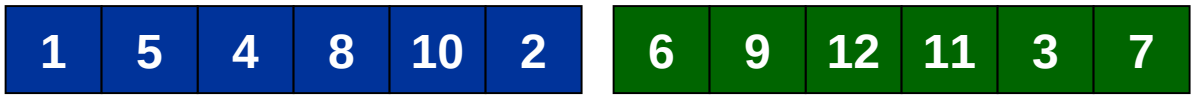
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide: separate list into two pieces.**
- **Conquer: recursively count inversions in each half.**
- **Combine: count inversions where a_i and a_j are in different halves.**



O(1)



2T(N / 2)

5 blue-blue inversions **8 green-green inversions**

9 blue-green inversions:
{5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7}

$O(N^2)$

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide: separate list into two pieces.**
- **Conquer: recursively count inversions in each half.**
- **Combine: count inversions where a_i and a_j are in different halves.**
- **Return sum of three quantities.**

1	5	4	8	10	2	6	9	12	11	3	7

O(1)

1	5	4	8	10	2

6	9	12	11	3	7

2T(N / 2)

5 blue-blue inversions **8 green-green inversions**

9 blue-green inversions:
{5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7}

O(N^2)

Total = 5 + 8 + 9 = 22.

O(1)

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide: separate list into two pieces.**
- **Conquer: recursively count inversions in each half.**
- **Combine: count inversions where a_i and a_j are in different halves.**
- **Return sum of three quantities.**

1	5	4	8	10	2	6	9	12	11	3	7

O(1)

1	5	4	8	10	2

6	9	12	11	3	7

2T(N / 2)

5 blue-blue inversions **8 green-green inversions**

9 blue-green inversions:
{5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7}

Can we do this step in sub-quadratic time?

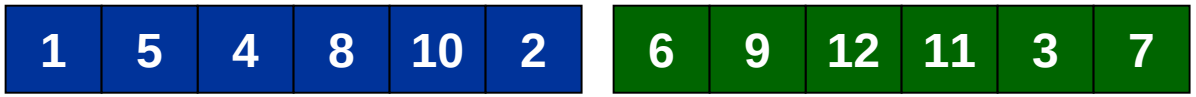
Total = 5 + 8 + 9 = 22.

O(1)

Counting Inversions: Good Combine

Combine: count inversions where a_i and a_j are in different halves.

- **Key idea: easy if each half is sorted.**
- **Sort each half.**
- **Count inversions.**



O(N log N)

9 blue-green inversions: 4 + 2 + 2 + 1 + 0 + 0

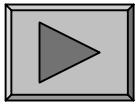
O(N)

$$T(N) = T(\lfloor N/2 \rfloor) + T(\lceil N/2 \rceil) + O(N \log N) \Rightarrow T(N) = O(N \log^2 N)$$

Counting Inversions: Better Combine

Combine: count inversions where a_i and a_j are in different halves.

- **Assume each half is pre-sorted.**
- **Count inversions.**
- **Merge two sorted halves into sorted whole.**



3	7	10	14	18	19

2	11	16	17	23	25

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

O(N)

2	3	7	10	11	14	16	17	18	19	23	25

O(N)

$$T(N) = T(\lfloor N/2 \rfloor) + T(\lceil N/2 \rceil) + O(N) \Rightarrow T(N) = O(N \log N)$$

Closest Pair

Given N points in the plane, find a pair that is closest together.

- **For concreteness, we assume Euclidean distances.**
- **Foundation of then-fledgling field of computational geometry.**
- **Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.**

Brute force solution.

- **Check all pairs of points p and q.**
- **Θ (N^2) comparisons.**

One dimensional version (points on a line).

- **O(N log N) easy.**

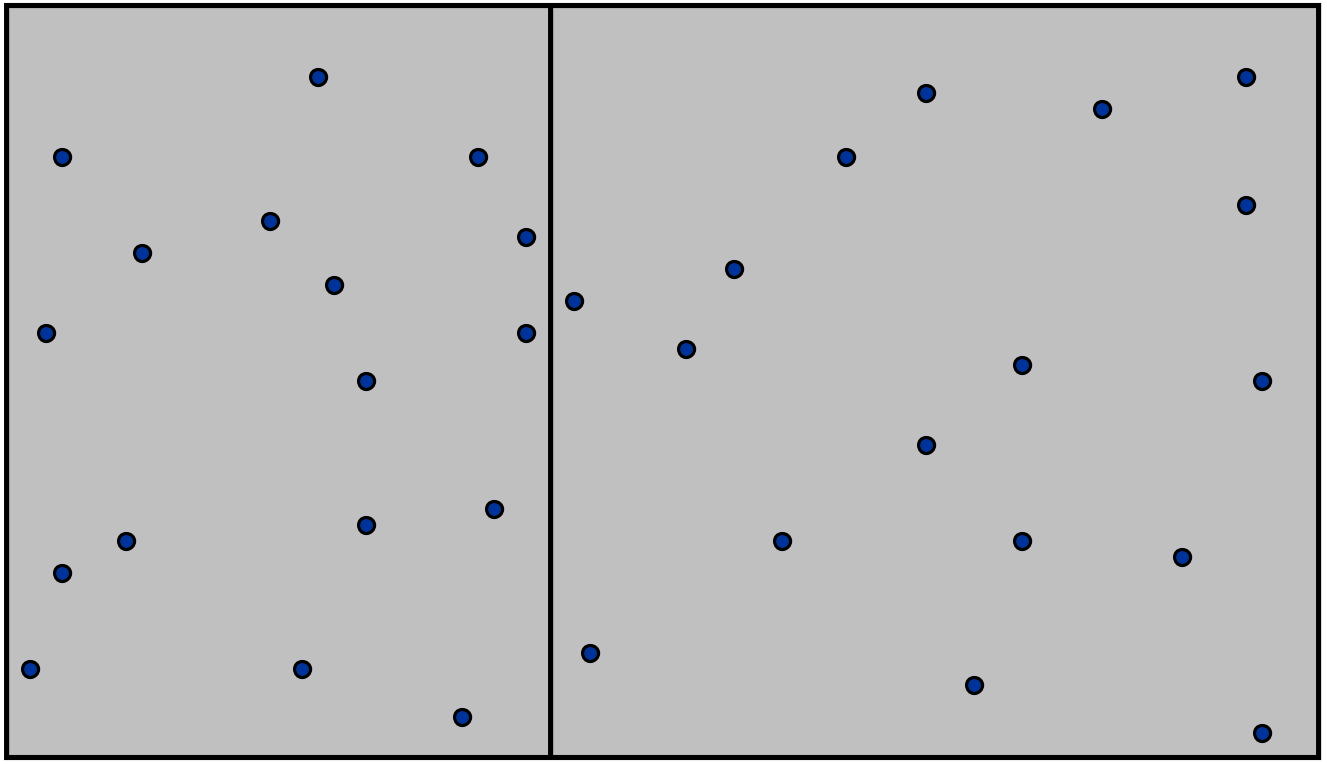
Assumption to make presentation cleaner.

- **No two points have same x coordinate.**

Closest Pair

Algorithm.

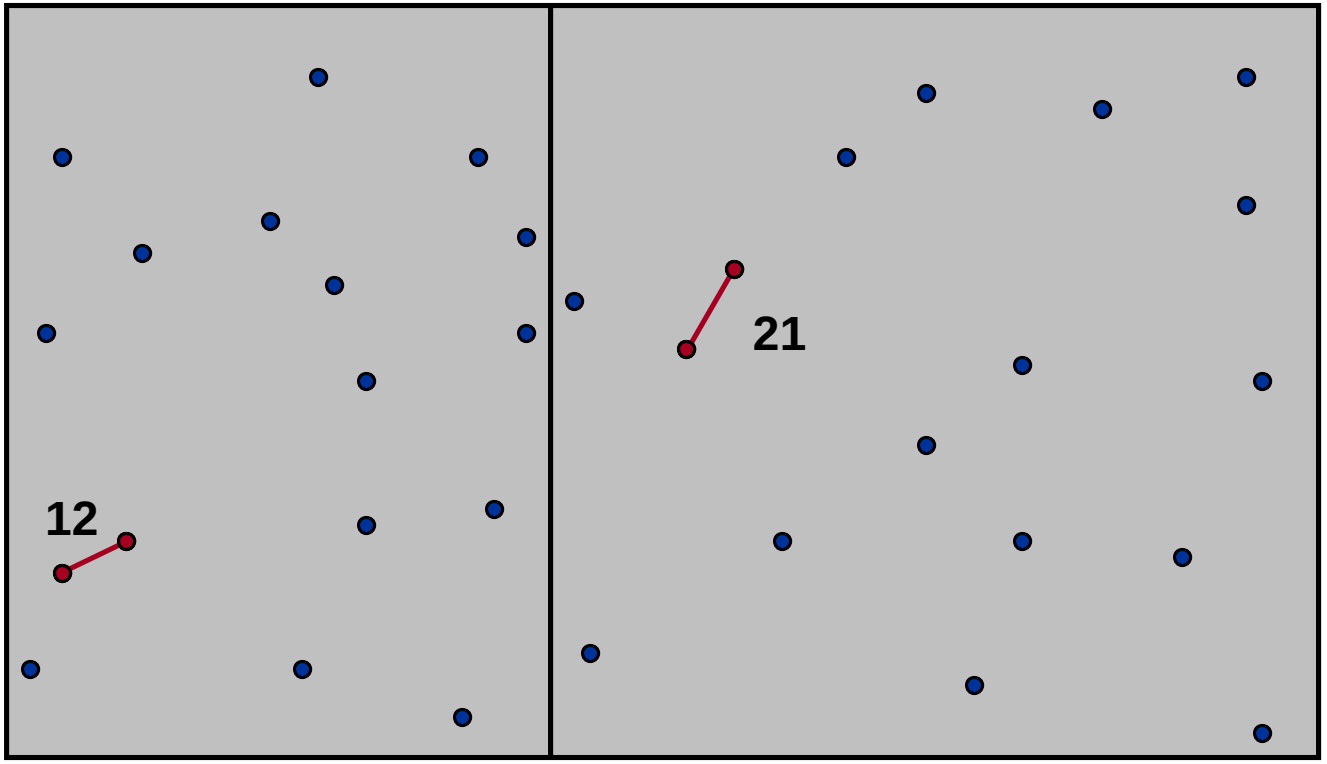
- **Divide: draw vertical line so that roughly N / 2 points on each side.**



Closest Pair

Algorithm.

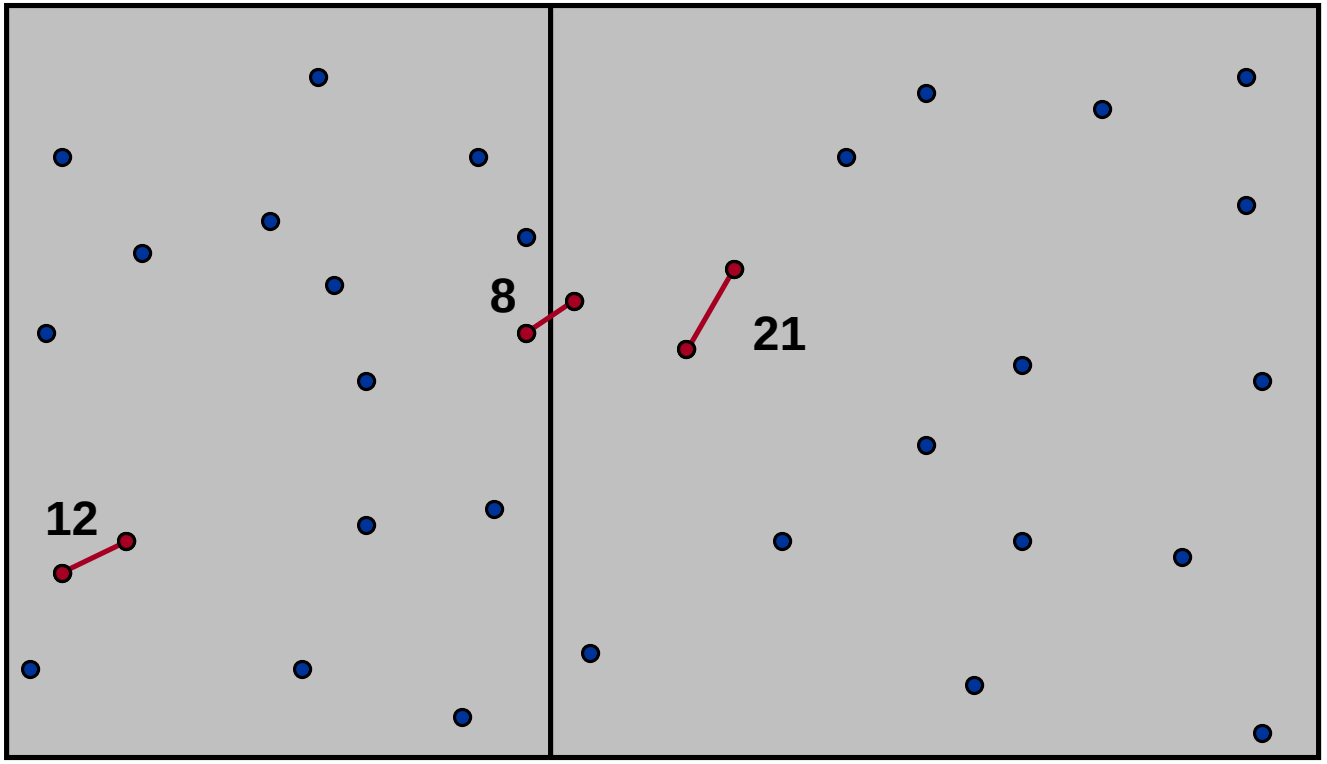
- **Divide: draw vertical line so that roughly N / 2 points on each side.**
- **Conquer: find closest pair in each side recursively.**



Closest Pair

Algorithm.

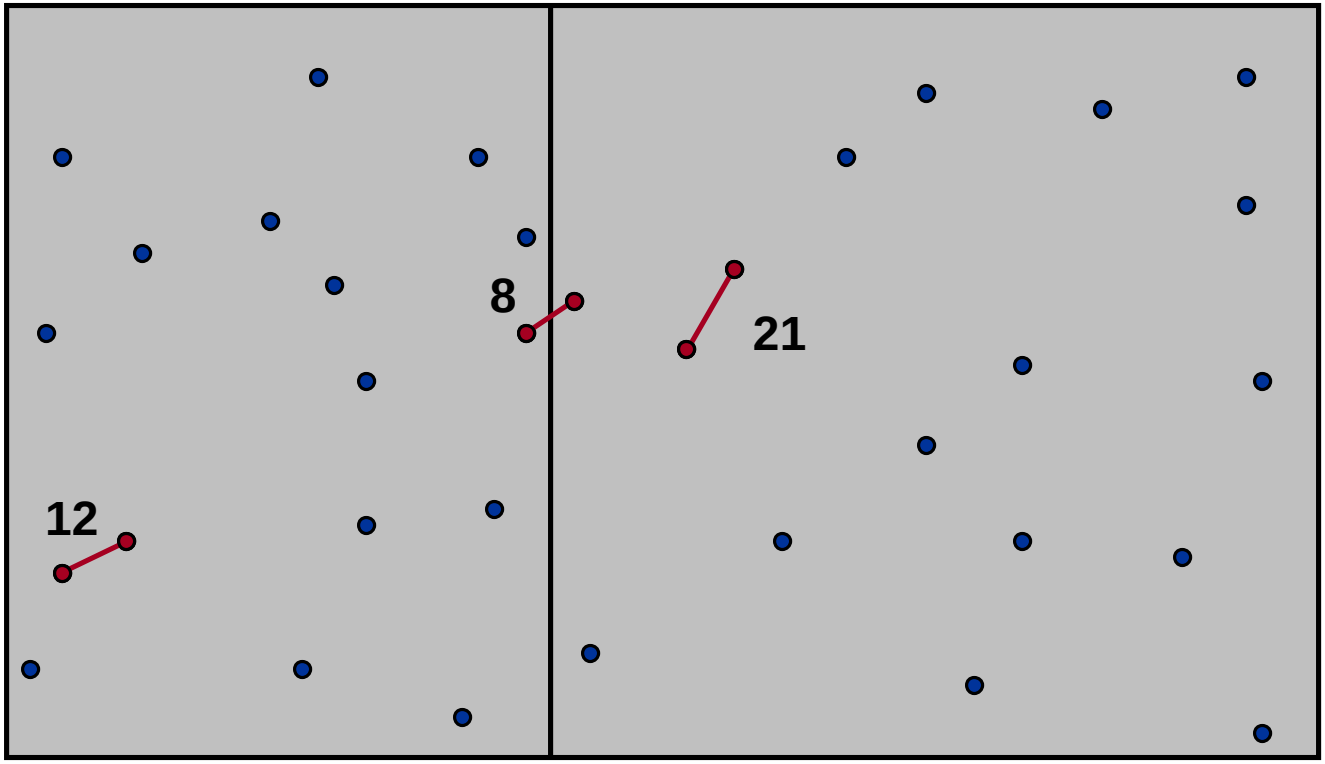
- **Divide: draw vertical line so that roughly N / 2 points on each side.**
- **Conquer: find closest pair in each side recursively.**
- **Combine: find closest pair with one point in each side.**



Closest Pair

Algorithm.

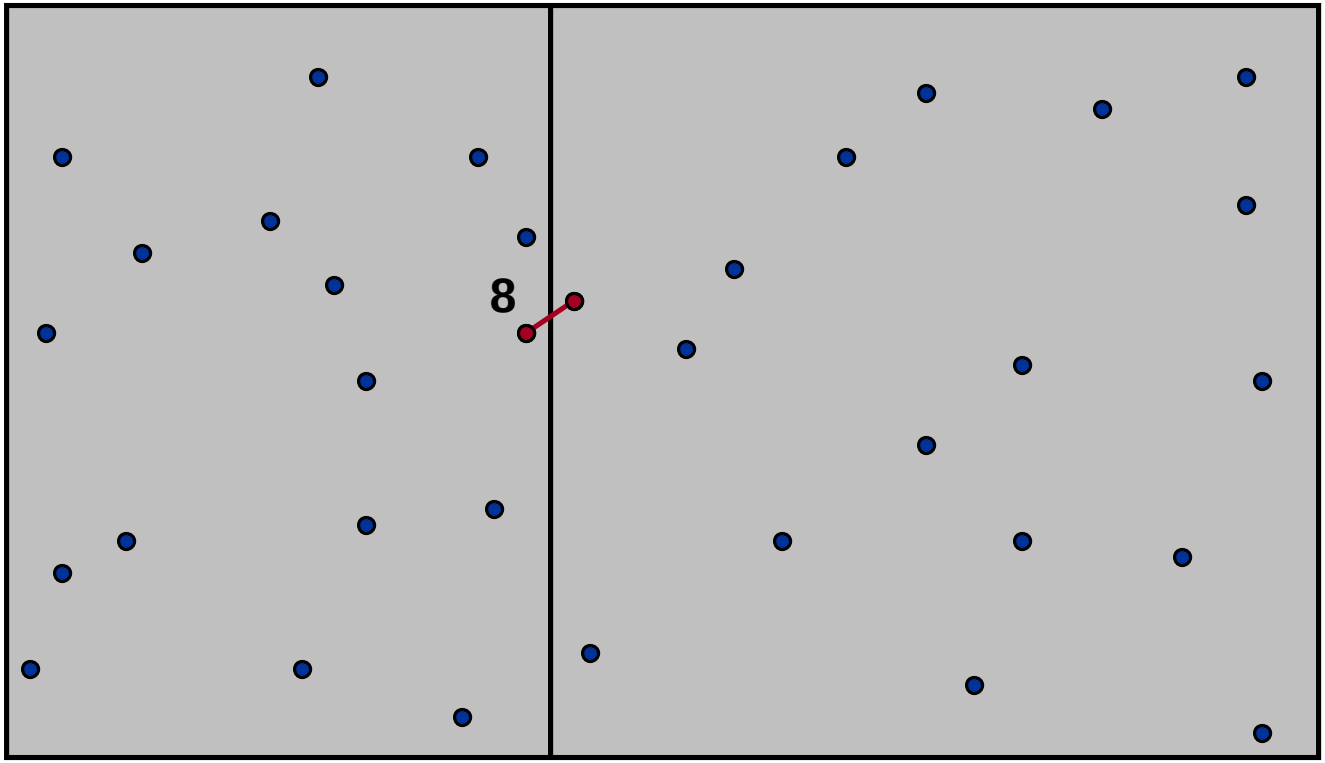
- **Divide: draw vertical line so that roughly N / 2 points on each side.**
- **Conquer: find closest pair in each side recursively.**
- **Combine: find closest pair with one point in each side.**
- **Return best of 3 solutions.**



Closest Pair

Algorithm.

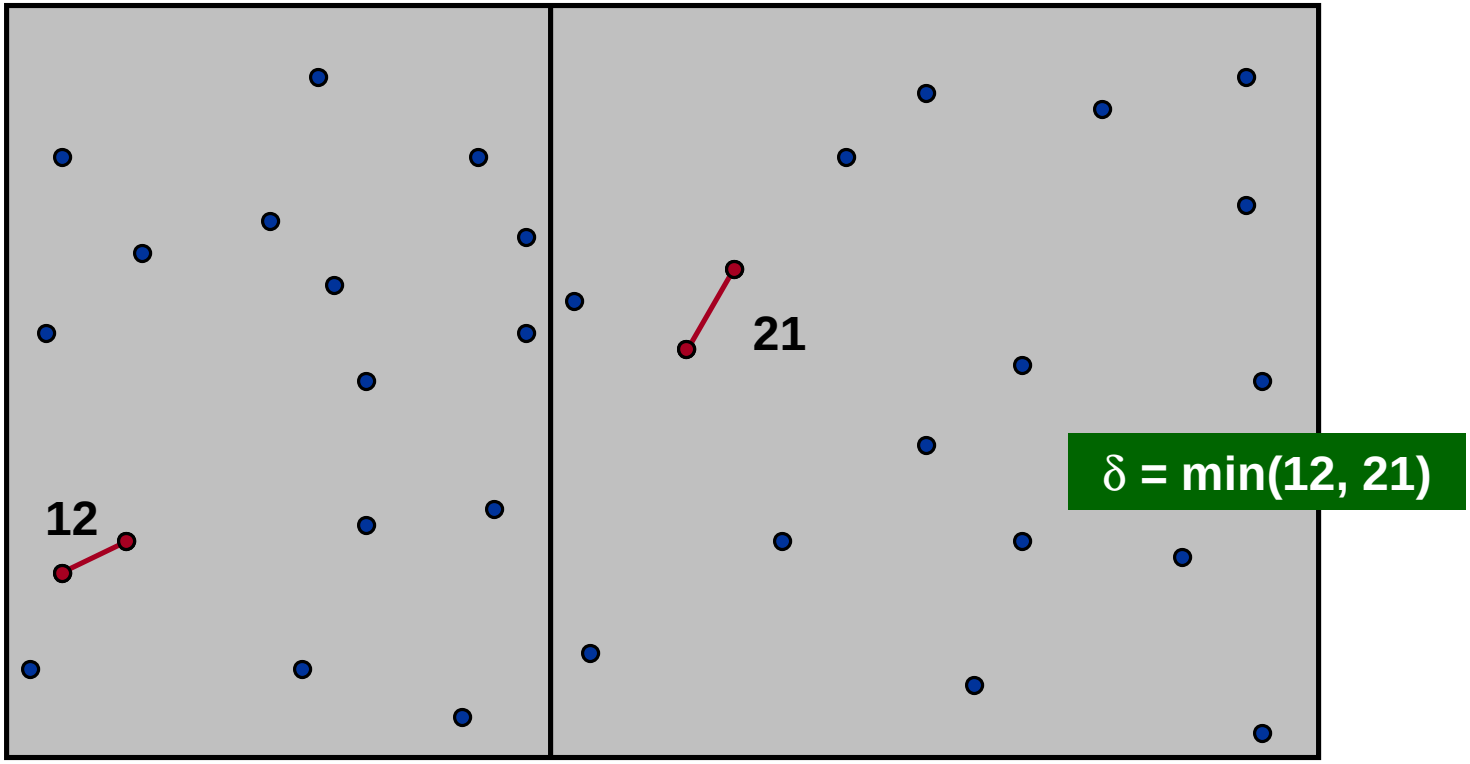
- **Divide: draw vertical line so that roughly N / 2 points on each side.**
- **Conquer: find closest pair in each side recursively.**
- **Combine: find closest pair with one point in each side.**
- **Return best of 3 solutions.**



Closest Pair

Key step: find closest pair with one point in each side.

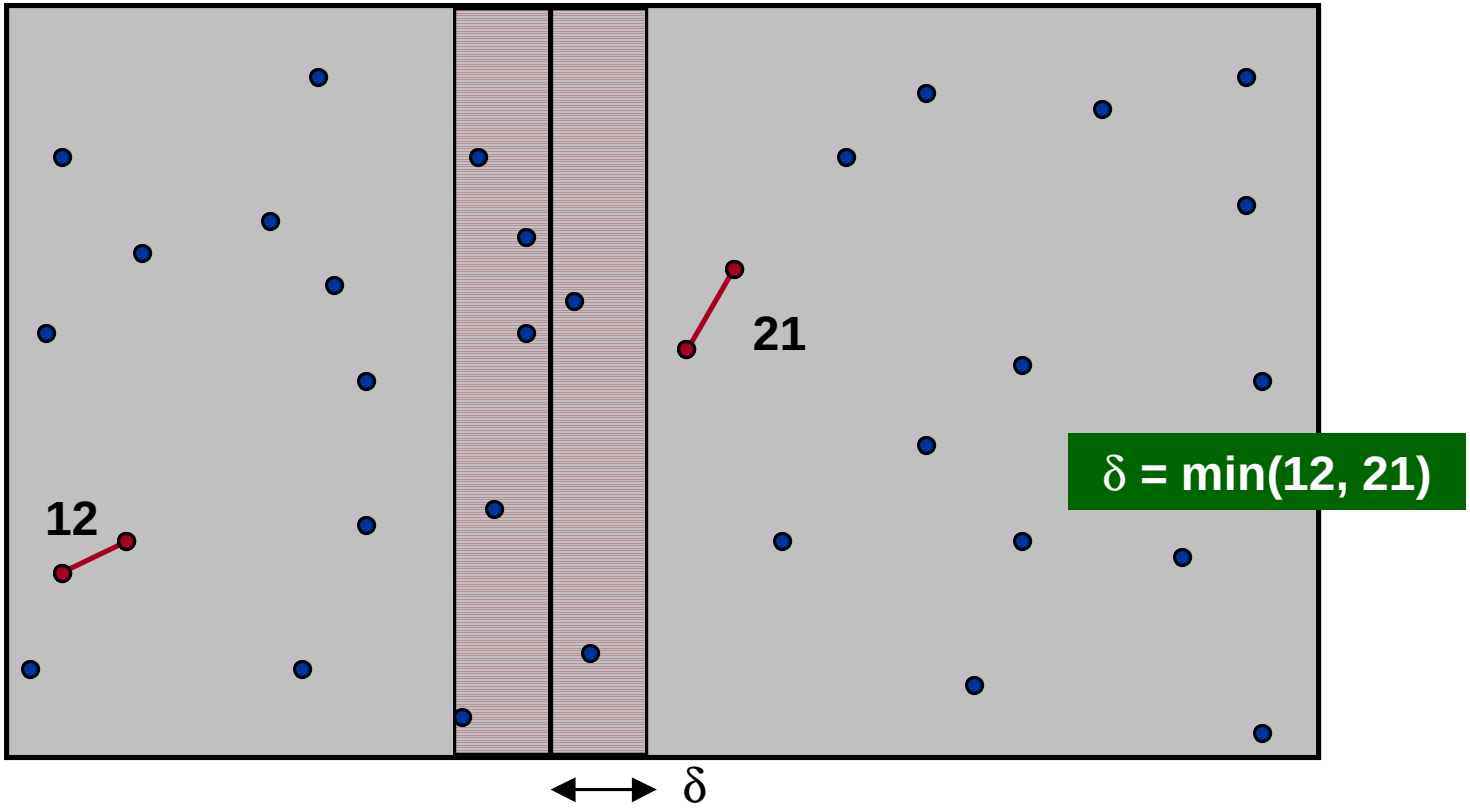
- **Extra information: closest pair entirely in one side had distance δ.**



Closest Pair

Key step: find closest pair with one point in each side.

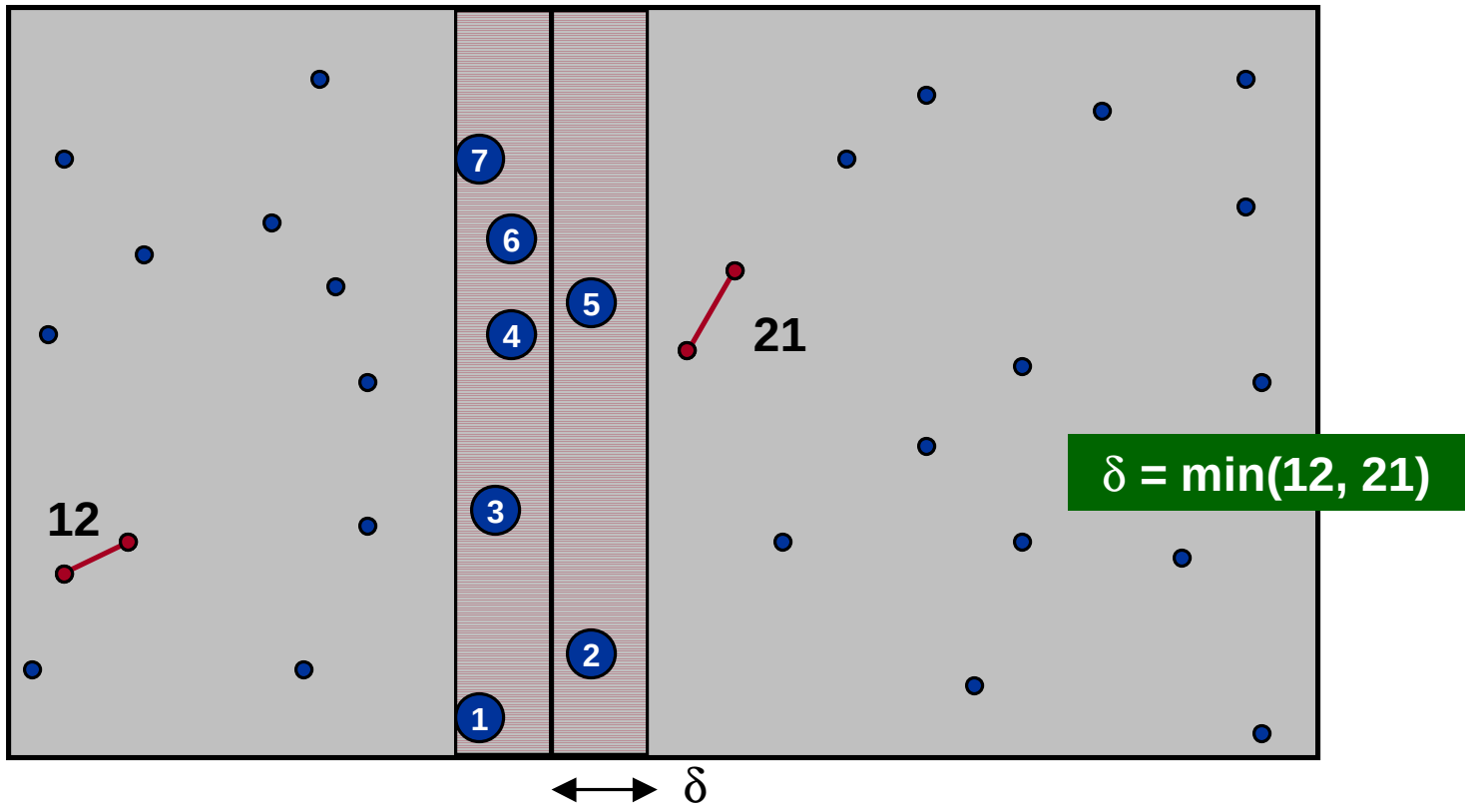
- **Extra information: closest pair entirely in one side had distance δ.**
- **Observation: only need to consider points S within δ of line.**



Closest Pair

Key step: find closest pair with one point in each side.

- **Extra information: closest pair entirely in one side had distance δ.**
- **Observation: only need to consider points S within δ of line.**
- **Sort points in strip S by their y coordinate.**
  - **suffices to compute distances for pairs within constant number of positions of each other in sorted list!**

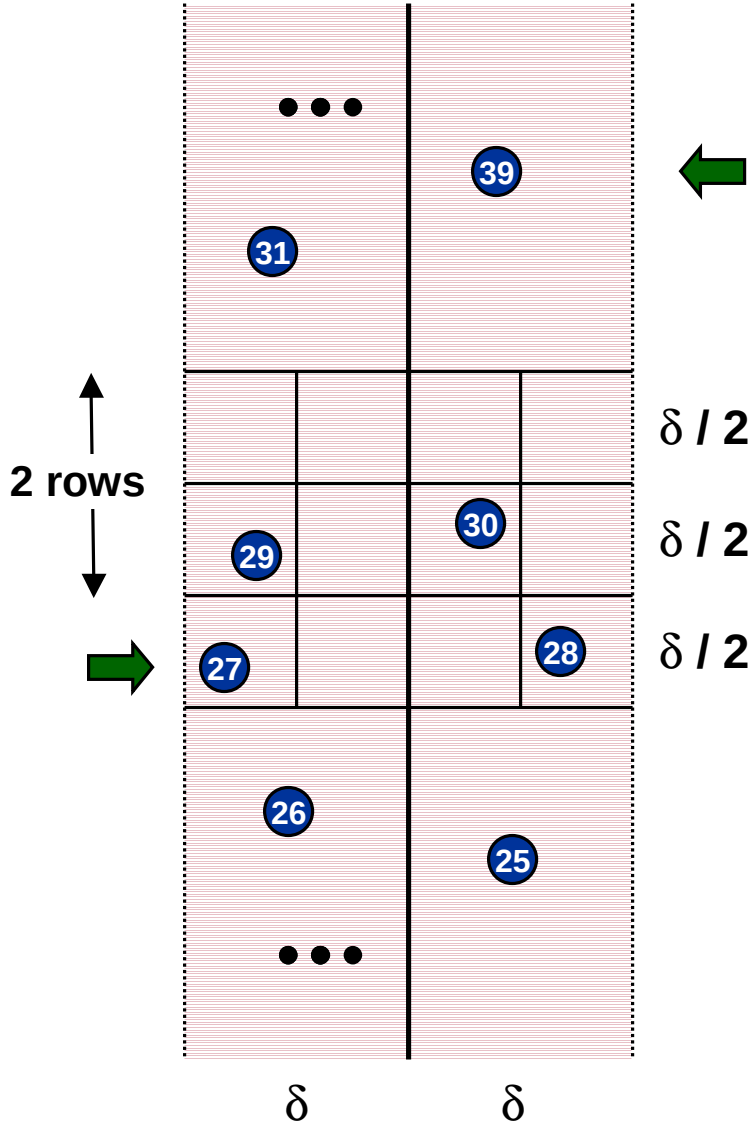


Closest Pair

S = list of points in the strip sorted by their y coordinate.

Crucial fact: if p and q are in S, and if d(p, q) < δ , then they are within 11 positions of each other in S.

- **No two points lie in same box.**
- **Two points at least 2 rows apart have distance ≥ 2δ / 2.**



Closest Pair

δ = ClosestPair (p_1, p_2 , . . . , p_N)

O(N log N) ⇨ **Compute separation line x = x_{med} such that half the points have x coordinate less than x_{med}, and half are greater.**

2T(N / 2) ⇨ **δ_1 = ClosestPair(left half)**
δ_2 = ClosestPair(right half)
δ = min (δ_1 , δ_2)

O(N) ⇨ **Delete all points further than δ from separation line.**

O(N log N) ⇨ **Sort remaining points in strip by y coordinate.**

O(N) ⇨ **Scan in y order, and compute distance between each point and next 11 neighbors.**

O(N) ⇨ **If any of these distances is less than δ , update δ .**

$$T(N) = T(\lfloor N/2 \rfloor) + T(\lceil N/2 \rceil) + O(N \log N) \Rightarrow T(N) = O(N \log^2 N)$$

Closest Pair

Can we achieve O(N log N)?

- **Yes. Don't sort points in strip from scratch each time.**
- **Each recursive call should return two lists: all points sorted by y coordinate, and all points sorted by x coordinate.**
- **Sorting is accomplished by merging two already sorted lists.**

$$T(N) = T(\lfloor N/2 \rfloor) + T(\lceil N/2 \rceil) + O(N) \Rightarrow T(N) = O(N \log N)$$

Integer Arithmetic

Given two N-digit integers a and b, compute a + b.

- **O(N) bit operations.**

Multiplication: given two N-digit integers a and b, compute ab.

- **Brute force solution: $\Theta(N^2)$ bit operations.**

Application.

- **Cryptography.**

1	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

					1	1	0	1	0	1	0	1			
					*	0	1	1	1	1	1	0	1		
					1	1	0	1	0	1	0	1	0		
				0	0	0	0	0	0	0	0	0			
			1	1	0	1	0	1	0	1	0				
		1	1	0	1	0	1	0	1	0					
	1	1	0	1	0	1	0	1	0						
1	1	0	1	0	1	0	1	0							
0	0	0	0	0	0	0	0	0							
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	0

Divide-and-Conquer Multiplication: First Attempt

To multiply two N-digit integers:

- **Multiply four N/2-digit integers.**
- **Add two N/2-digit integers, and shift to obtain result.**

$$\begin{aligned} 123{,}456 \times 987{,}654 &= (10^3 w + x) \times (10^3 y + z) \\ &= 10^6 (wy) + 10^3 (wz + xy) + 10^0 (xz) \\ &= 10^6 (121{,}401) + 10^3 (80{,}442 + 450{,}072) + 10^0 (298{,}224) \\ &= 121{,}401{,}299{,}224 \end{aligned}$$

$w = 123$
$x = 456$
$y = 987$
$z = 654$

N is a power of 2 ⇒

$$ab = (10^{N/2} w + x)(10^{N/2} y + z)$$

$$T(N) = \underbrace{4T(N/2)}_{\text{recursive calls}} + \underbrace{\Theta(N)}_{\text{add, shift}} \Rightarrow T(N) = \Theta(N^2)$$

Karatsuba Multiplication

To multiply two N-digit integers:

- **Add two N/2 digit integers.**
- **Multiply three N/2-digit integers.**
- **Subtract two N/2-digit integers, and shift to obtain result.**

$$
\begin{aligned}
123{,}456 \times 987{,}654 &= (10^3 w + x) \times (10^3 y + z) \\
&= 10^6 (wy) + 10^3 (wz + xy) + 10^0 (xz) \\
&= 10^6 (p) + 10^3 (r - p - q) + 10^0 (q) \\
&= 10^6 (121{,}401) + 10^3 (950{,}139 - 121{,}401 - 298{,}224) + 10^0 (298{,}224) \\
&= 121{,}401{,}299{,}224
\end{aligned}
$$

$w = 123$
$x = 456$
$y = 987$
$z = 654$

$p = wy$
$q = xz$
$r = (w + x)(y + z)$

$(wz + xy) = r - p - q$

Karatsuba Multiplication: Analysis

To multiply two N-digit integers:

- Add two N/2 digit integers.
- Multiply three N/2-digit integers.
- Subtract two N/2-digit integers, and shift to obtain result.

Karatsuba-Ofman (1962).

- $O(N^{1.585})$ bit operations.

$$p = wy$$
$$q = xz$$
$$r = (w + x)(y + z)$$

$$(wz + xy) = r - p - q$$

$$ab = (10^{N/2} w + x)(10^{N/2} y + z)$$

$$T(N) \le \underbrace{T(\lfloor N/2 \rfloor) + T(\lceil N/2 \rceil) + T(1 + \lceil N/2 \rceil)}_{\text{recursive calls}} + \underbrace{\Theta(N)}_{\text{add, subtract, shift}}$$

$$\Rightarrow T(N) = O(N^{\log_2 3})$$

Matrix Multiplication

Given two N x N matrices A and B, compute C = AB.

$$c_{ij} = \sum_{k=1}^{N} a_{ik}\, b_{kj}$$

- **Brute force: $\Theta(N^3)$ time.**

$$\begin{pmatrix} 26 & 62 & 98 \\ 80 & 224 & 368 \\ 134 & 386 & 638 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 4 \\ 6 & 8 & 10 \\ 12 & 14 & 16 \end{pmatrix} \times \begin{pmatrix} 1 & 7 & 13 \\ 3 & 9 & 15 \\ 5 & 11 & 17 \end{pmatrix}$$

$$\begin{pmatrix} c_{11} & c_{12} & c_{13} & \cdots & c_{1N} \\ c_{21} & c_{22} & c_{23} & \cdots & c_{2N} \\ c_{31} & c_{32} & c_{33} & \cdots & c_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{N1} & c_{N2} & c_{N3} & \cdots & c_{NN} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1N} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2N} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & \cdots & a_{NN} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} & \cdots & b_{1N} \\ b_{21} & b_{22} & b_{23} & \cdots & b_{2N} \\ b_{31} & b_{32} & b_{33} & \cdots & b_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{N1} & b_{N2} & b_{N3} & \cdots & b_{NN} \end{pmatrix}$$

Hard to imagine naïve algorithm can be improved upon.

Matrix Multiplication: Warmup

Warmup: divide-and-conquer.

- **Divide: partition A and B into N/2 x N/2 blocks.**
- **Conquer: multiply 8 N/2 x N/2 recursively.**
- **Combine: add appropriate products using 4 matrix additions.**

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

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

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

Matrix Multiplication: Idea

Idea: multiply 2 x 2 matrices with only 7 scalar multiplications.

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

$$\begin{aligned} P_1 &= a \times (g - h) \\ P_2 &= (a + b) \times h \\ P_3 &= (c + d) \times e \\ P_4 &= d \times (f - e) \\ P_5 &= (a + d) \times (e + h) \\ P_6 &= (b - d) \times (f + h) \\ P_7 &= (a - c) \times (e + g) \end{aligned}$$

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

- **7 multiplications.**
- **18 = 10 + 8 additions and subtractions.**

Note: did not rely on commutativity of scalar multiplication.

Matrix Multiplication: Strassen

Generalize to matrices.

- Divide: partition A and B into N/2 x N/2 blocks.
- Compute: 14 N/2 x N/2 matrices via 10 matrix add/subtract.
- Conquer: multiply 7 N/2 x N/2 recursively.
- Combine: 7 products into 4 terms using 8 matrix add/subtract.

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

Analysis.

- Assume N is a power of 2.
- T(N) = # arithmetic operations.

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

Beyond Strassen

Can you multiply two 2 x 2 matrices with only 7 scalar multiplications?

- **Yes! Strassen (1969).**

$$\Theta(N^{\log_2 7}) = O(N^{2.81})$$

Can you multiply two 2 x 2 matrix with only 6 scalar multiplications?

- **Impossible (Hopcroft and Kerr, 1971).**

$$\Theta(N^{\log_2 6}) = O(N^{2.59})$$

Two 3 x 3 matrices with only 21 scalar multiplications?

- **Also impossible.**

$$\Theta(N^{\log_3 21}) = O(N^{2.77})$$

Two 70 x 70 matrices with only 143,640 scalar multiplications?

- **Yes! (Pan, 1980).**

$$\Theta(N^{\log_{70} 143640}) = O(N^{2.80})$$

Decimal wars.

- **December, 1979: $O(N^{2.521813})$.**
- **January, 1980: $O(N^{2.521801})$.**

Coppersmith-Winograd (1987): $O(N^{2.376})$.

Εύρεση k-οστού Μικρότερου

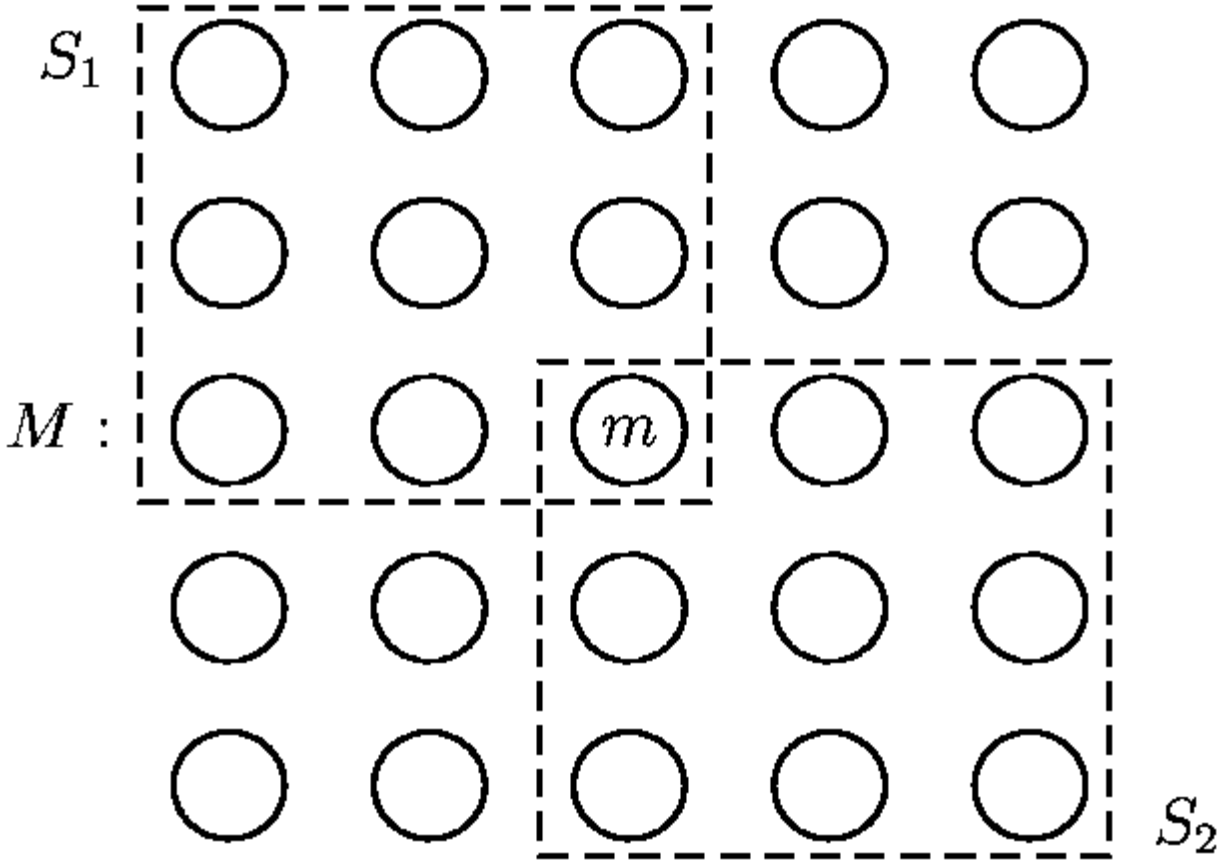
```
procedure Selection (p, q, k);
begin
  while p<q do
  begin
    choose t from [p..q]; partition (p, q, t);
    if k<t then q:=t−1 else begin p:=t; k:=k−p+1 end
  end
end
```

Πολυπλοκότητα: $O(n^2)$ χειρότερη, $O(n)$ μέση

Βελτίωση Πολυπλοκότητας

- Με επιλογή κατάλληλου σημείου partition
- Χωρισμός σε n/5 ακολουθίες 5 στοιχείων
- Ταξινόμηση 5άδων και σχηματισμός *M* από τα μεσαία στοιχεία: *O*(n)
- Εύρεση του μεσαίου *m* της *M*: T(n/5)
- Τουλάχιστον $\frac{1}{4}$ στοιχεία μεγαλύτερα του m και $\frac{1}{4}$ στοιχεία μικρότερα του m

Η μέθοδος σχηματικά



Πολυπλοκότητα Βελτίωσης

$$T(n) = \left\{ \begin{array}{ll} d & , \text{για } n \leq i \\ T(\frac{n}{5}) + T(\frac{3n}{4}) + cn & , \text{για } n > i \end{array} \right\} = O(n)$$

Θεώρημα 6.8.1. *Αν ισχύει ότι* $T(1) = d$ *και* $T(n) = T(an) + T(bn) + cn$ *με* $a + b < 1$ *τότε η συνάρτηση* $T(n)$ *είναι γραμμική, ισχύει δηλαδή ότι* $T(n) = O(n)$.

Greedy algorithms

- Ιδέα
- Παραδείγματα
  - breakpoint selection
  - activity selection
  - coin change
  - minimizing lateness
- Γνωστοί αλγόριθμοι
  - Dijkstra (single source shortest paths)
  - Prim, Kruskal (MST)

Η Τεχνική Greedy (Άπληστη)

```
function Greedy (a:set of elements) : solution;
var
  x:item;
begin
  solution:=empty;
  for all elements of a do
  begin
    (* χρήση των κανόνων βελτιστοποίησης *)
    x:=OptSelectRemove(a);
    (* έλεγχος αν πληρούνται οι περιορισμοί *)
    if feasible(solution+{x}) then solution:=solution+{x}
    (* η λύση μεγαλώνει και ενημερώνεται η αντικειμενική συνάρτηση *)
  end;
  greedy:=solution
End
```

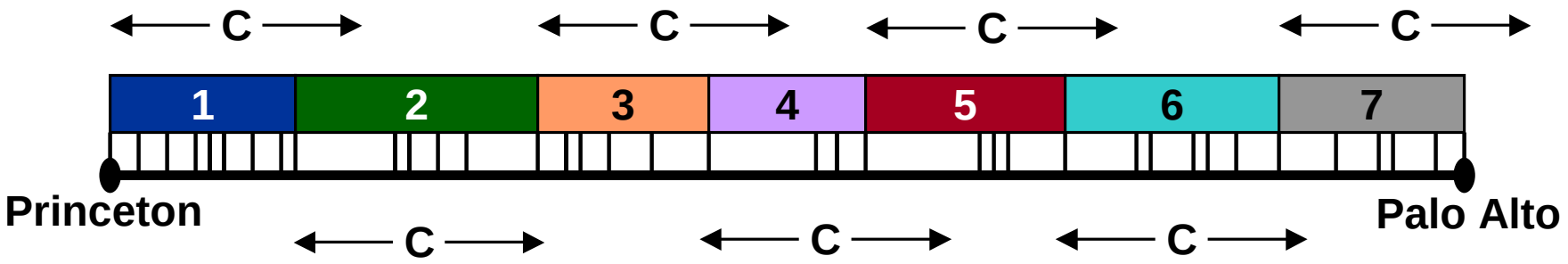
Selecting Breakpoints

Minimizing breakpoints.

- Truck driver going from Princeton to Palo Alto along predetermined route.
- Refueling stations at certain points along the way.
- Truck fuel capacity = C.

Greedy algorithm.

- Go as far as you can before refueling.



Selecting Breakpoints: Greedy Algorithm

Greedy Breakpoint Selection Algorithm

Sort breakpoints by increasing value:
$0 = b_0 < b_1 < b_2 < \ldots < b_n$.

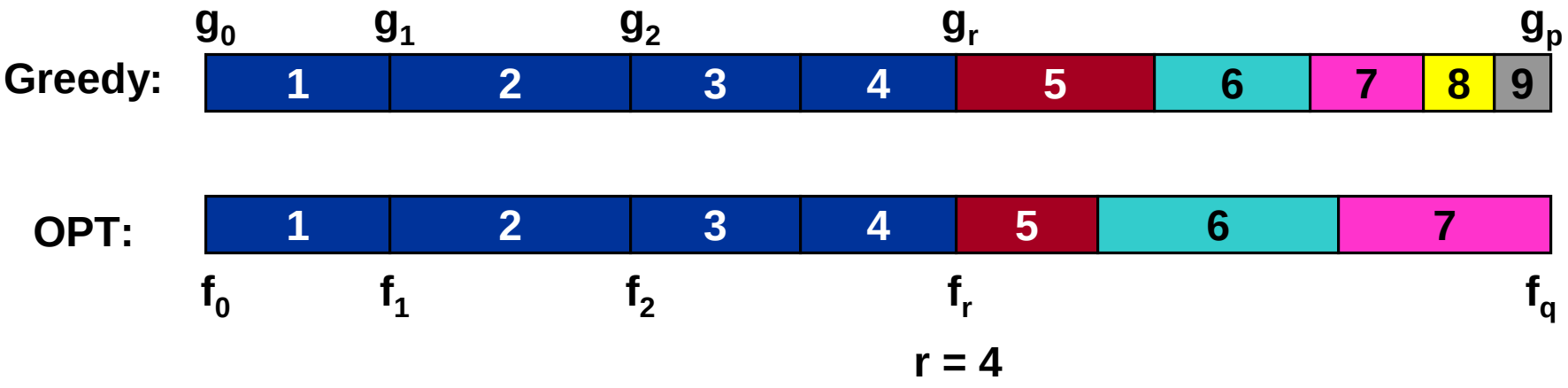
$S \leftarrow \{0\}$ ← S = breakpoints selected.
$x = 0$
while $(x \neq b_n)$
  let p be largest integer such that $b_p \leq x + C$
  if $(b_p = x)$
    return "no solution"
  $x \leftarrow b_p$
  $S \leftarrow S \cup \{p\}$
return S

Selecting Breakpoints

Theorem: greedy algorithm is optimal.

Proof (by contradiction):

- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy and assume it is not optimal.
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in optimal solution with $f_0 = g_0, f_1 = g_1, \ldots, f_r = g_r$ for largest possible value of r.
- Note: q < p.

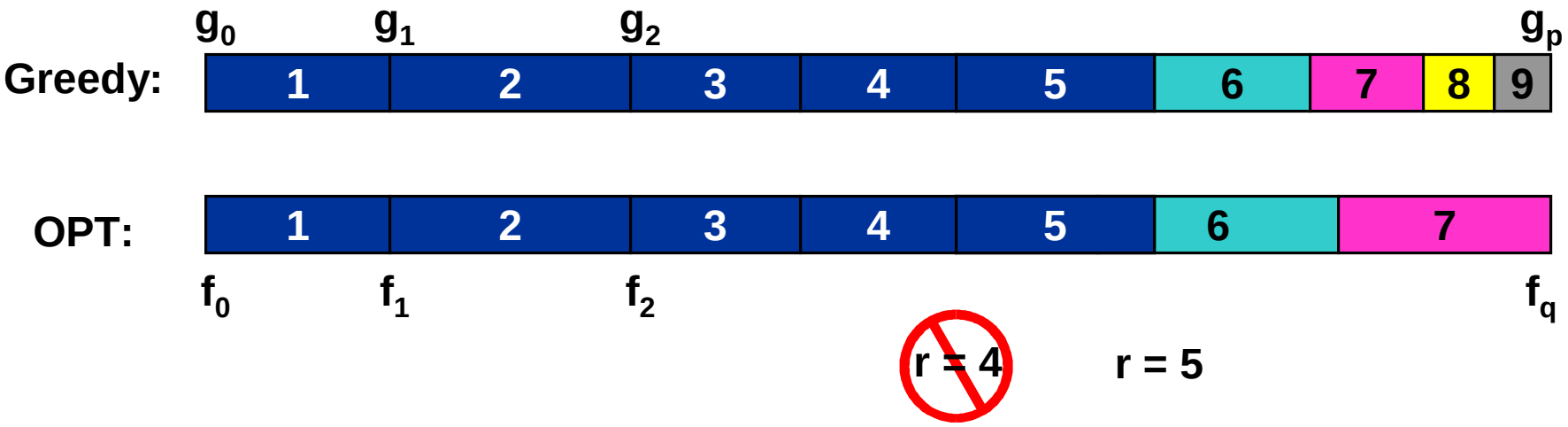


Selecting Breakpoints

Theorem: greedy algorithm is optimal.

Proof (by contradiction):

- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy and assume it is not optimal.
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in optimal solution with $f_0 = g_0, f_1 = g_1, \ldots, f_r = g_r$ for largest possible value of r.
- Note: q < p.

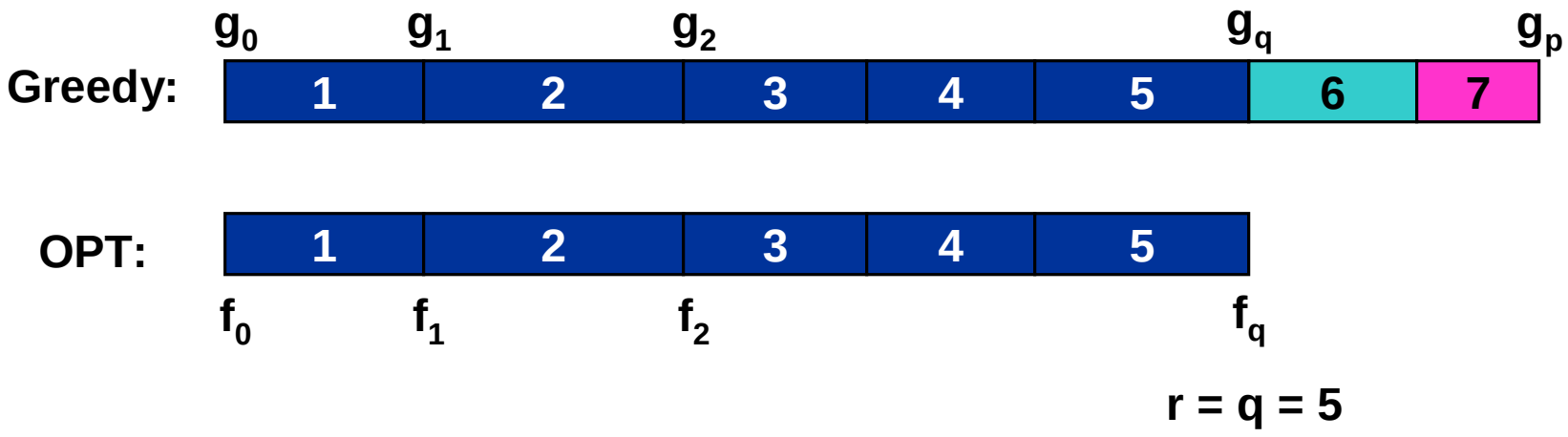


Selecting Breakpoints

Theorem: greedy algorithm is optimal.

Proof (by contradiction):

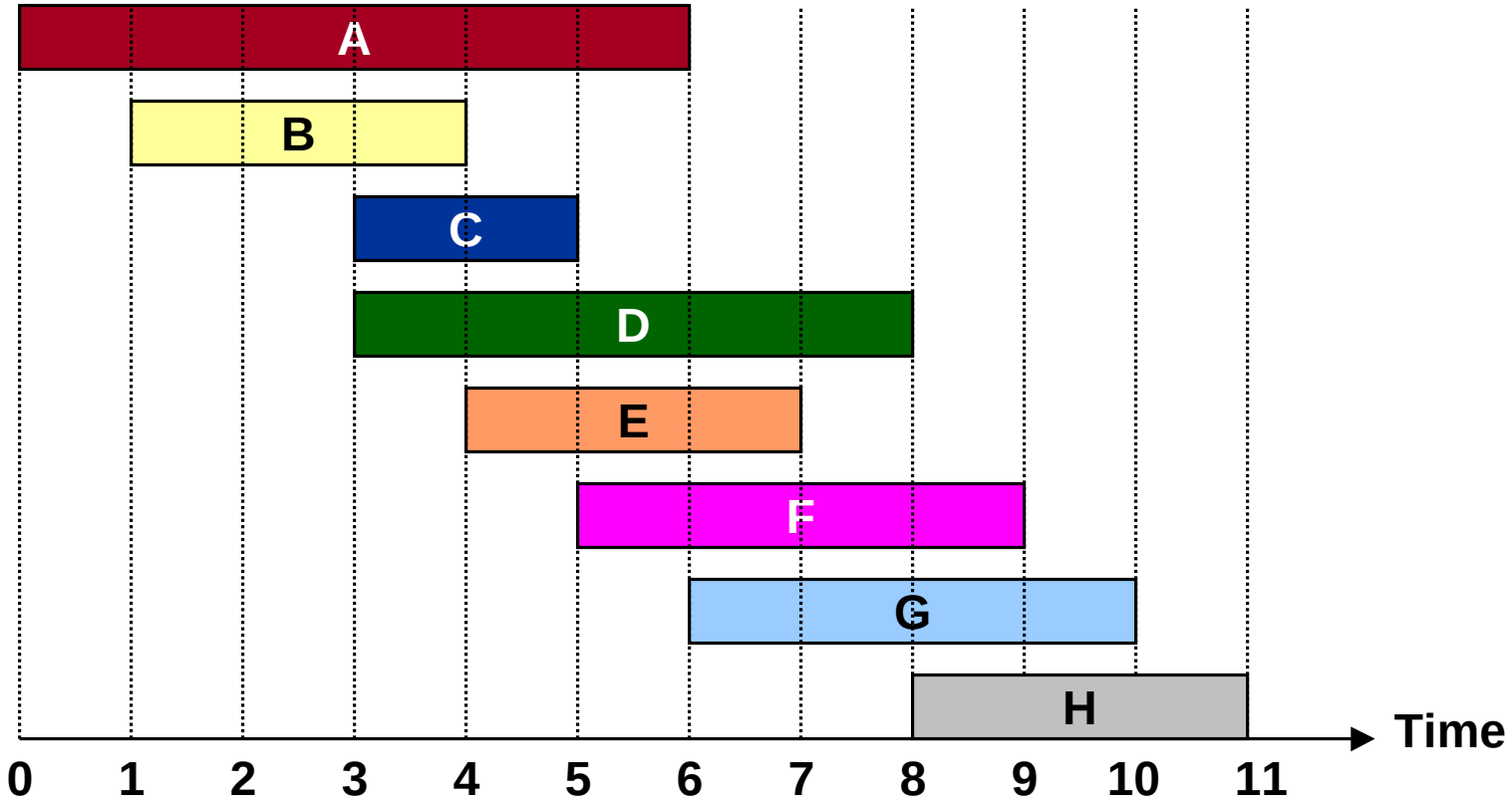
- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy and assume it is not optimal.
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in optimal solution with $f_0 = g_0, f_1 = g_1, \ldots, f_r = g_r$ for largest possible value of r.
- Note: $q < p$.
- Thus, $f_0 = g_0, f_1 = g_1, \ldots, f_q = g_q$



Activity Selection

Activity selection problem (CLR 17.1).

- Job requests 1, 2, … , n.
- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

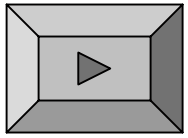


Activity Selection: Greedy Algorithm

Greedy Activity Selection Algorithm

```
Sort jobs by increasing finish times so that
f1 ≤ f2 ≤ ... ≤ fn.

S = φ                          ← S = jobs selected.
for j = 1 to n
   if (job j compatible with A)
      S ← S ∪ {j}
return S
```

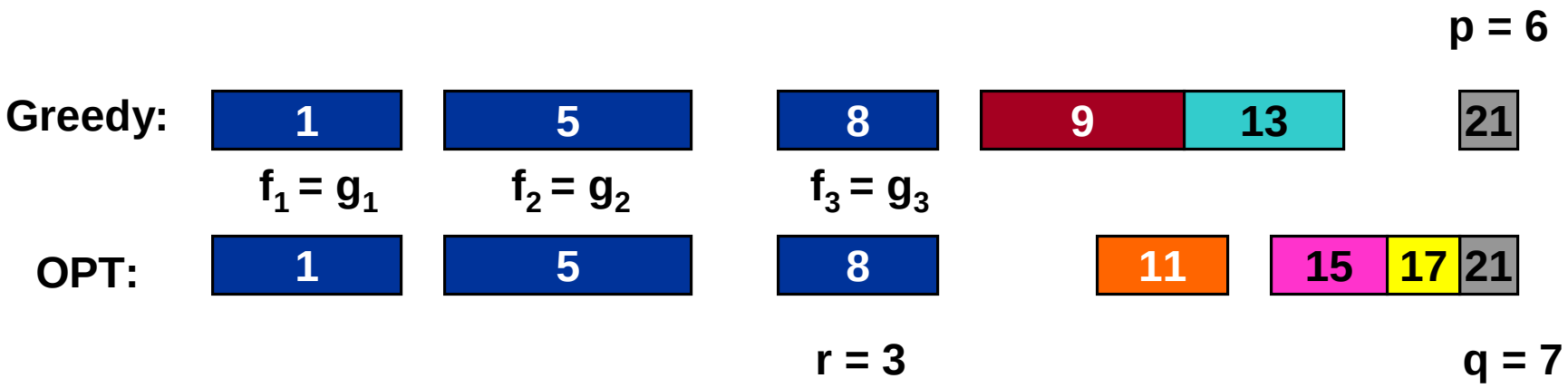


Activity Selection

Theorem: greedy algorithm is optimal.

Proof (by contradiction):

- Let $g_1, g_2, \ldots g_p$ denote set of jobs selected by greedy and assume it is not optimal.
- Let $f_1, f_2, \ldots f_q$ denote set of jobs selected by optimal solution with $f_1 = g_1, f_2 = g_2, \ldots, f_r = g_r$ for largest possible value of r.
- Note: r < q.

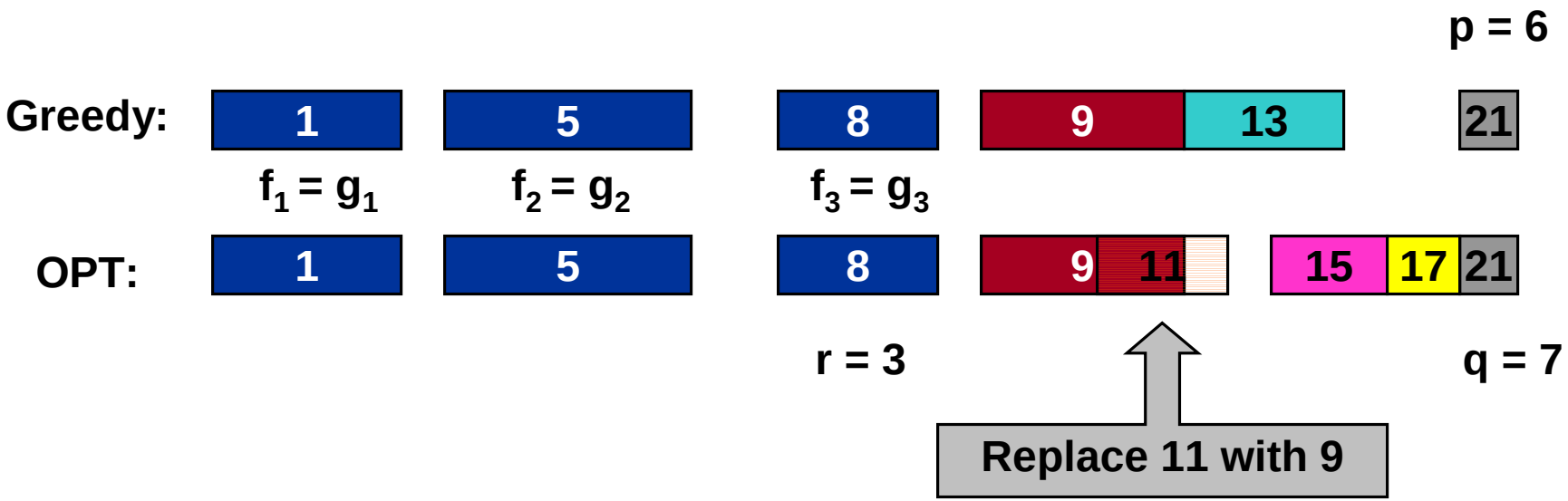


Activity Selection

Theorem: greedy algorithm is optimal.

Proof (by contradiction):

- Let $g_1, g_2, \ldots g_p$ denote set of jobs selected by greedy and assume it is not optimal.
- Let $f_1, f_2, \ldots f_q$ denote set of jobs selected by optimal solution with $f_1 = g_1, f_2 = g_2, \ldots, f_r = g_r$ for largest possible value of r.
- Note: r < q.

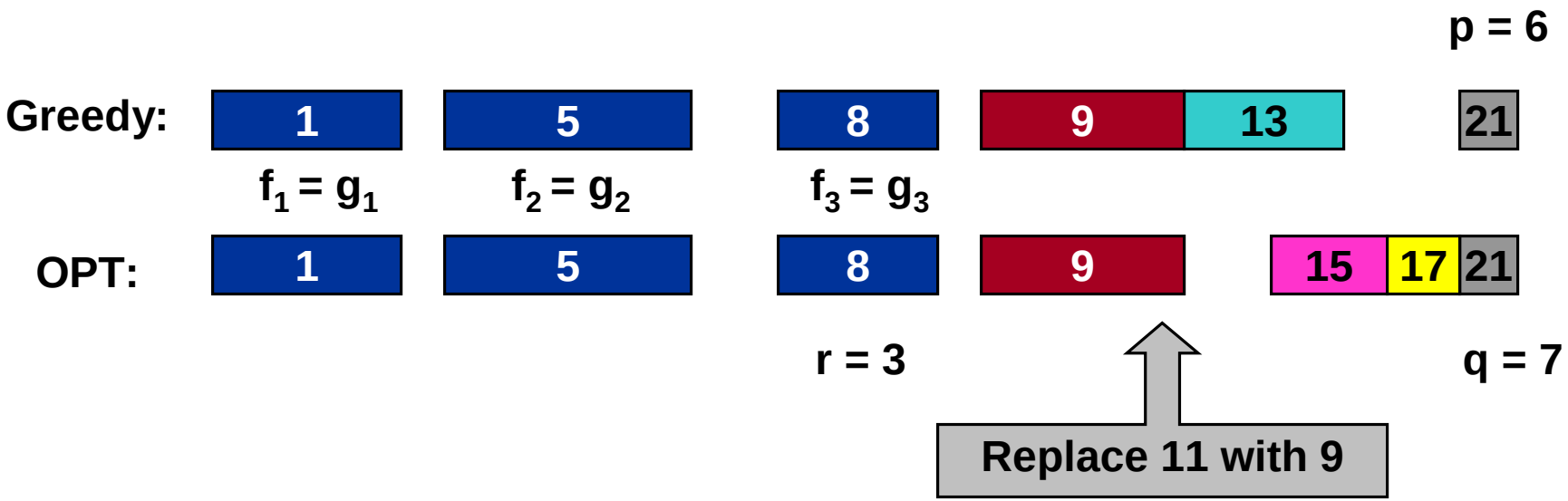


Activity Selection

Theorem: greedy algorithm is optimal.

Proof (by contradiction):

- Let $g_1, g_2, \ldots g_p$ denote set of jobs selected by greedy and assume it is not optimal.
- Let $f_1, f_2, \ldots f_q$ denote set of jobs selected by optimal solution with $f_1 = g_1, f_2 = g_2, \ldots, f_r = g_r$ for largest possible value of r.
- Note: r < q.

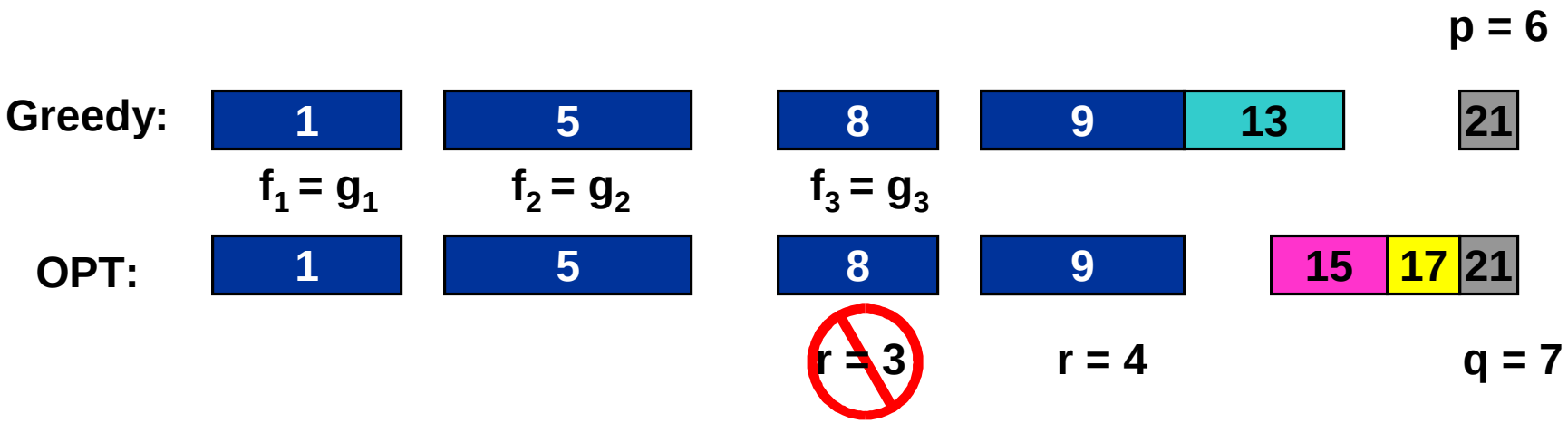


Activity Selection

Theorem: greedy algorithm is optimal.

Proof (by contradiction):

- Let $g_1, g_2, \ldots g_p$ denote set of jobs selected by greedy and assume it is not optimal.
- Let $f_1, f_2, \ldots f_q$ denote set of jobs selected by optimal solution with $f_1 = g_1, f_2 = g_2, \ldots, f_r = g_r$ for largest possible value of r.
- Note: r < q.



Making Change

Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

- **Ex. 34¢.**

Greedy algorithm.

- **At each iteration, add coin of the largest value that does not take us past the amount to be paid.**
- **Ex. $2.89.**

Coin-Changing: Greedy Algorithm

Greedy Coin-Changing Algorithm

Sort coins denominations by increasing value: $c_1 < c_2 < \ldots < c_n$.

S = coins selected.

$S \leftarrow \phi$
while ($x \neq 0$)
    let p be largest integer such that $c_p \leq x$
    if (p = 0)
        return "no solution found"
    $x \leftarrow x - c_p$
    $S \leftarrow S \cup \{p\}$
return S

Is Greedy Optimal for Coin-Changing Problem?

Yes, for U.S. coinage: $\{c_1, c_2, c_3, c_4, c_5\} = \{1, 5, 10, 25, 100\}$.

Ad hoc proof.

- **Consider optimal way to change amount $c_k \le x < c_{k+1}$.**
- **Greedy takes coin k.**
- **Suppose optimal solution does not take coin k.**
  - **it must take enough coins of type $c_1, c_2, \ldots, c_{k-1}$ to add up to x.**

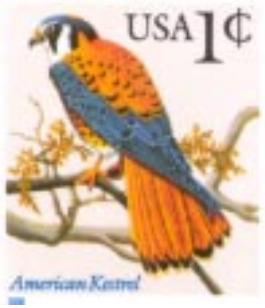
k	c_k	Max # taken by optimal solution	Max value of coins 1, 2, . . . , k in any OPT
1	1	4	4
2	5	1	4 + 5 = 9
3	10	2	20 + 4 = 24
4	25	3	75 + 24 = 99
5	100	no limit	no limit

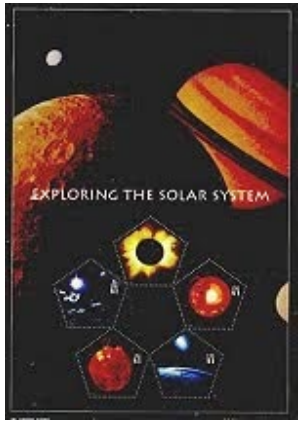
2 dimes ⇒ no nickels

Does Greedy Always Work?

US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

- **Ex. 140¢.**
- **Greedy: 100, 34, 1, 1, 1, 1, 1, 1.**
- **Optimal: 70, 70.**





Characteristics of Greedy Algorithms

Greedy choice property.

- Globally optimal solution can be arrived at by making locally optimal (greedy) choice.
- At each step, choose most "promising" candidate, without worrying whether it will prove to be a sound decision in long run.

Optimal substructure property.

- Optimal solution to the problem contains optimal solutions to sub-problems.
  - if best way to change 34¢ is {25, 5, 1, 1, 1, 1} then best way to change 29¢ is {25, 1, 1, 1, 1}.

Objective function does not explicitly appear in greedy algorithm!

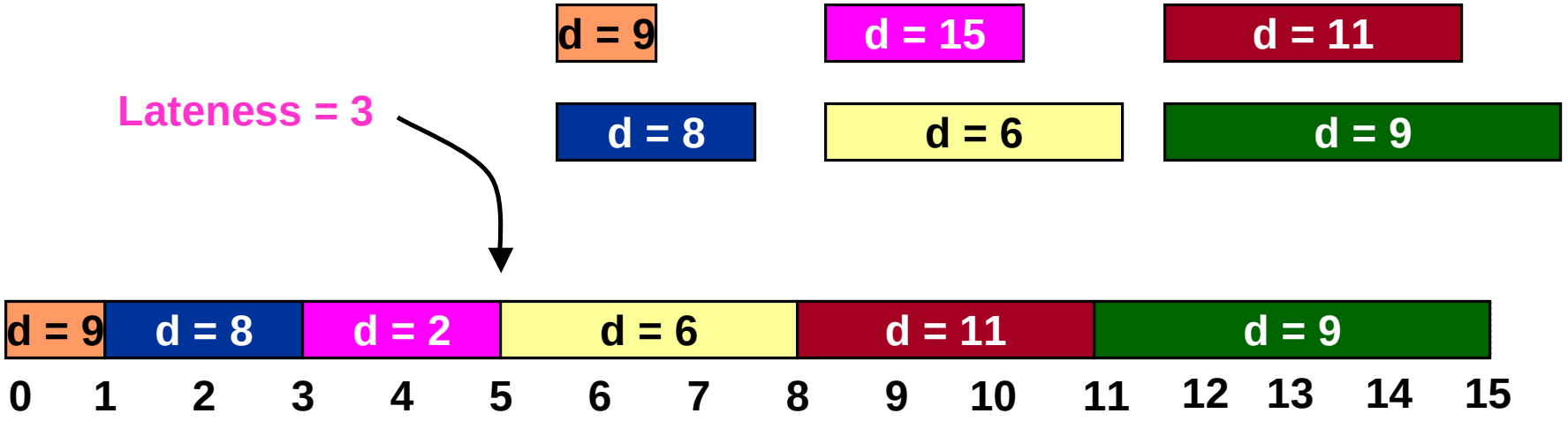
Hard, if not impossible, to precisely define "greedy algorithm."

- See matroids (CLR 17.4), greedoids for very general frameworks.

Minimizing Lateness

Minimizing lateness problem.

- Single resource can process one job at a time.
- n jobs to be processed.
  - job j requires p_j units of processing time.
  - job j has due date d_j.
- If we assign job j to start at time s_j, it finishes at time $f_j = s_j + p_j$.
- Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

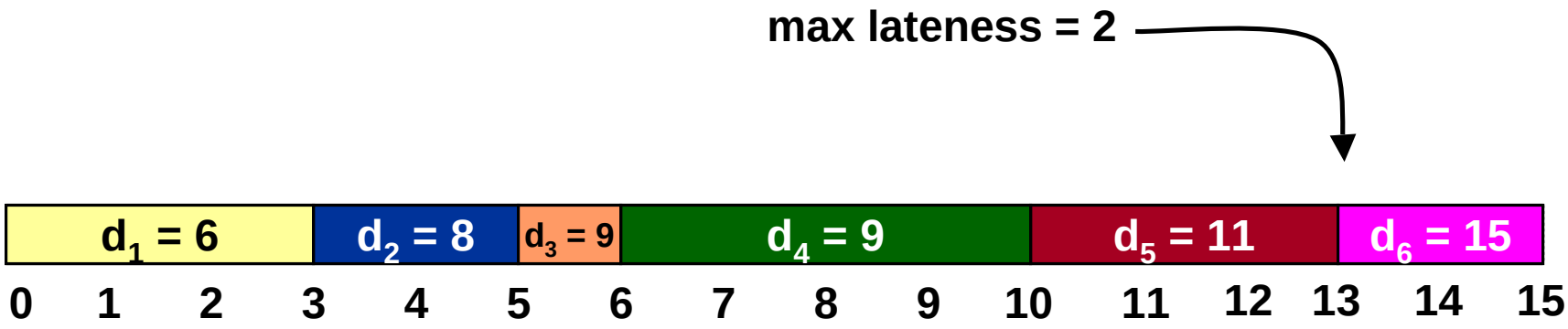


Minimizing Lateness: Greedy Algorithm

Greedy Activity Selection Algorithm

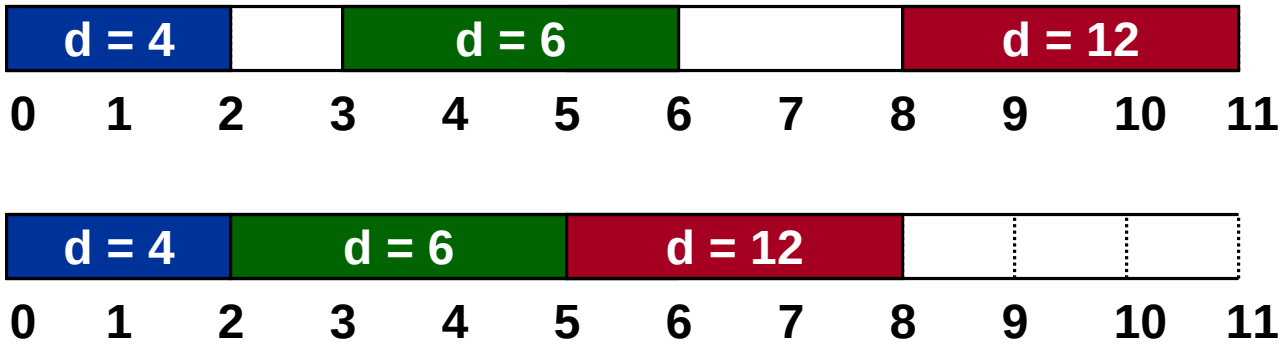
```
Sort jobs by increasing deadline so that
d1 ≤ d2 ≤ … ≤ dn.

t = 0
for j = 1 to n
   Assign job j to interval [t, t + pj]
   sj ← t, fj ← t + pj
   t ← t + pj
output intervals [sj, fj]
```



Minimizing Lateness: No Idle Time

Fact 1: there exists an optimal schedule with no idle time.

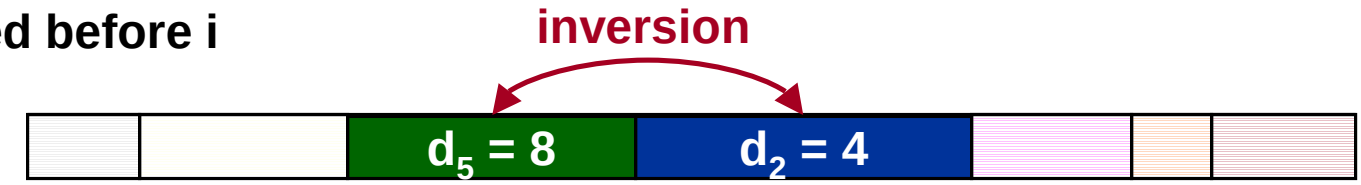


Fact 2: the greedy schedule has no idle time.

Minimizing Lateness: Inversions

An inversion in schedule S is a pair of jobs i and j such that:

- **i < j**
- **j scheduled before i**



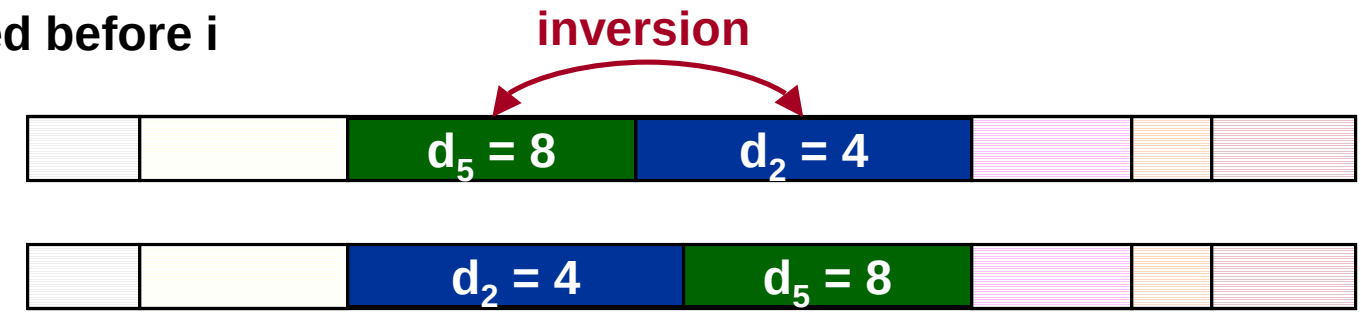
Fact 3: greedy schedule $\Leftrightarrow$ no inversions.

Fact 4: if a schedule (with no idle time) has an inversion, it has one whose with a pair of inverted jobs scheduled consecutively.

Minimizing Lateness: Inversions

An inversion in schedule S is a pair of jobs i and j such that:

- **i < j**
- **j scheduled before i**



Fact 3: greedy schedule $\Leftrightarrow$ no inversions.

Fact 4: if a schedule (with no idle time) has an inversion, it has one whose with a pair of inverted jobs scheduled consecutively.

Fact 5: swapping two adjacent, inverted jobs:

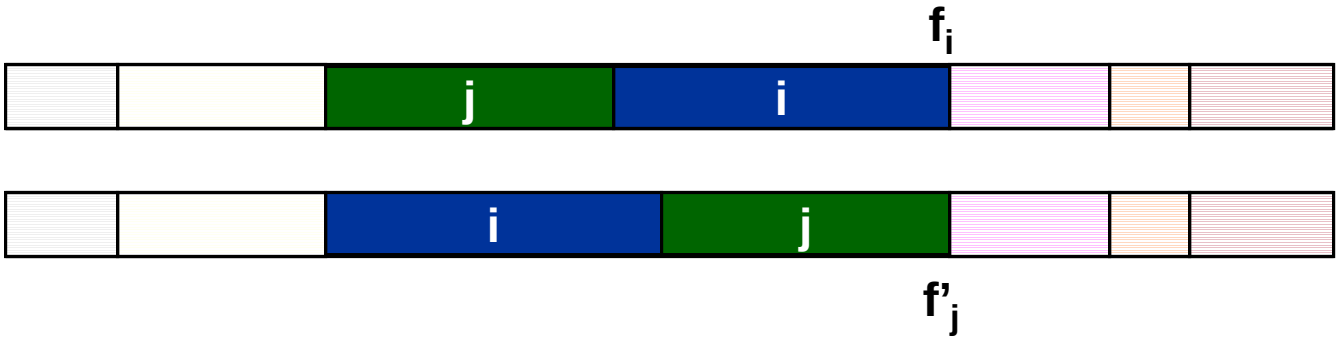
- **Reduces the number of inversions by one.**
- **Does not increase the maximum lateness.**

Theorem: greedy schedule is optimal.

Minimizing Lateness: Proof of Fact 5

An inversion in schedule S is a pair of jobs i and j such that:

- **i < j**
- **j scheduled before i**

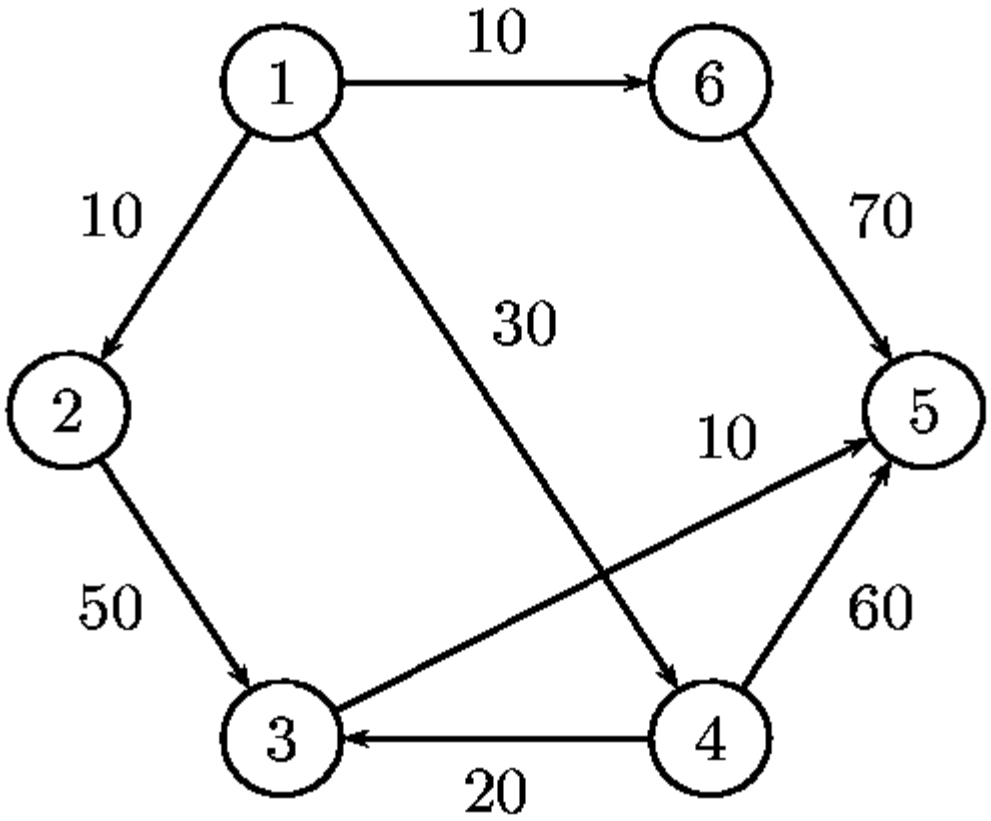


Swapping two adjacent, inverted jobs does not increase max lateness.

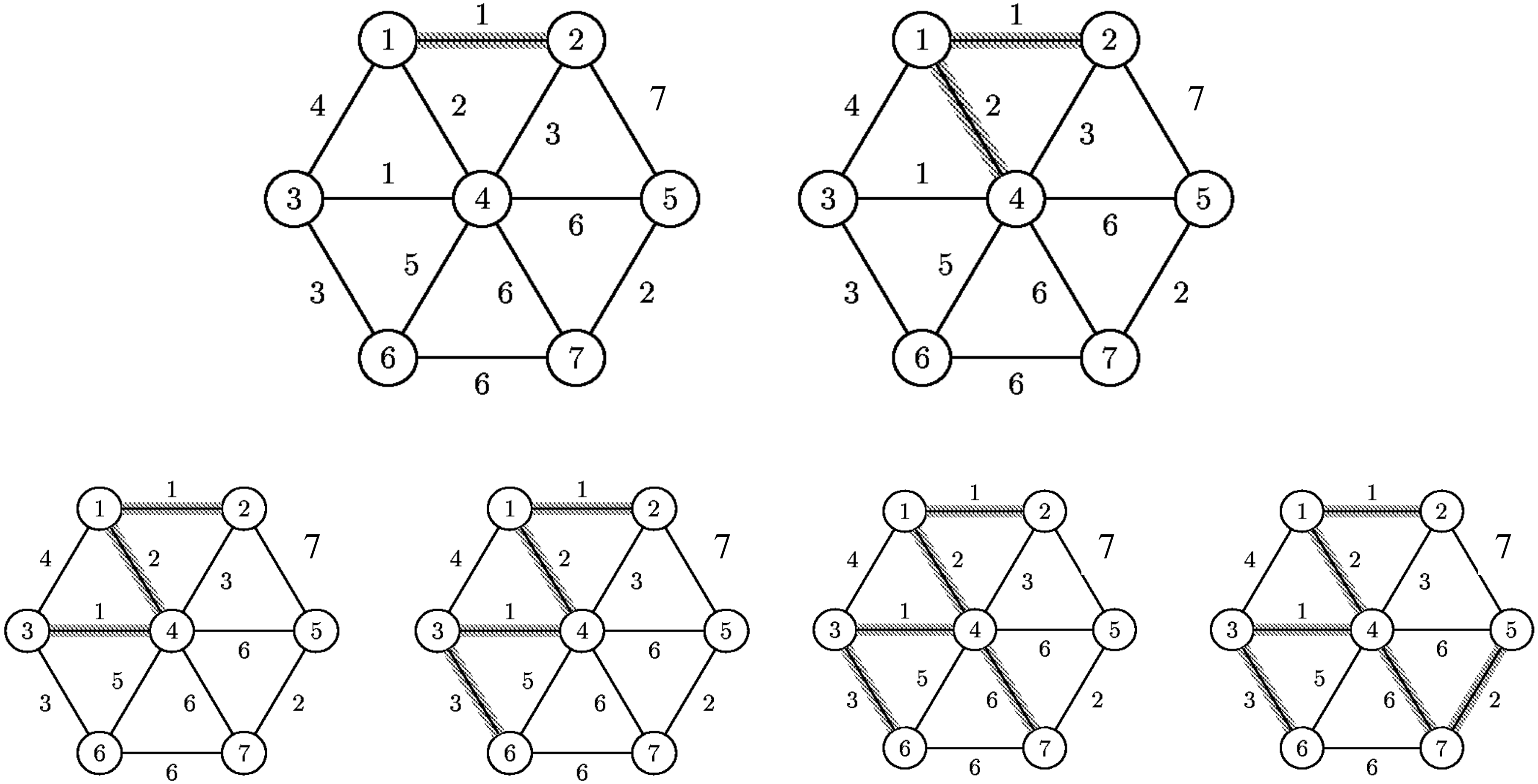
- $\ell'_k = \ell_k$ **for all** $k \neq i, j$
- $\ell'_i \leq l_i$
- **If job j is late:**

$$
\begin{aligned}
\ell'_j &= f'_j - d_j && \text{(definition)} \\
&= f_i - d_j && (j \text{ finishes at time } f_i) \\
&\leq f_i - d_i && (i < j) \\
&\leq \ell_i && \text{(definition)}
\end{aligned}
$$

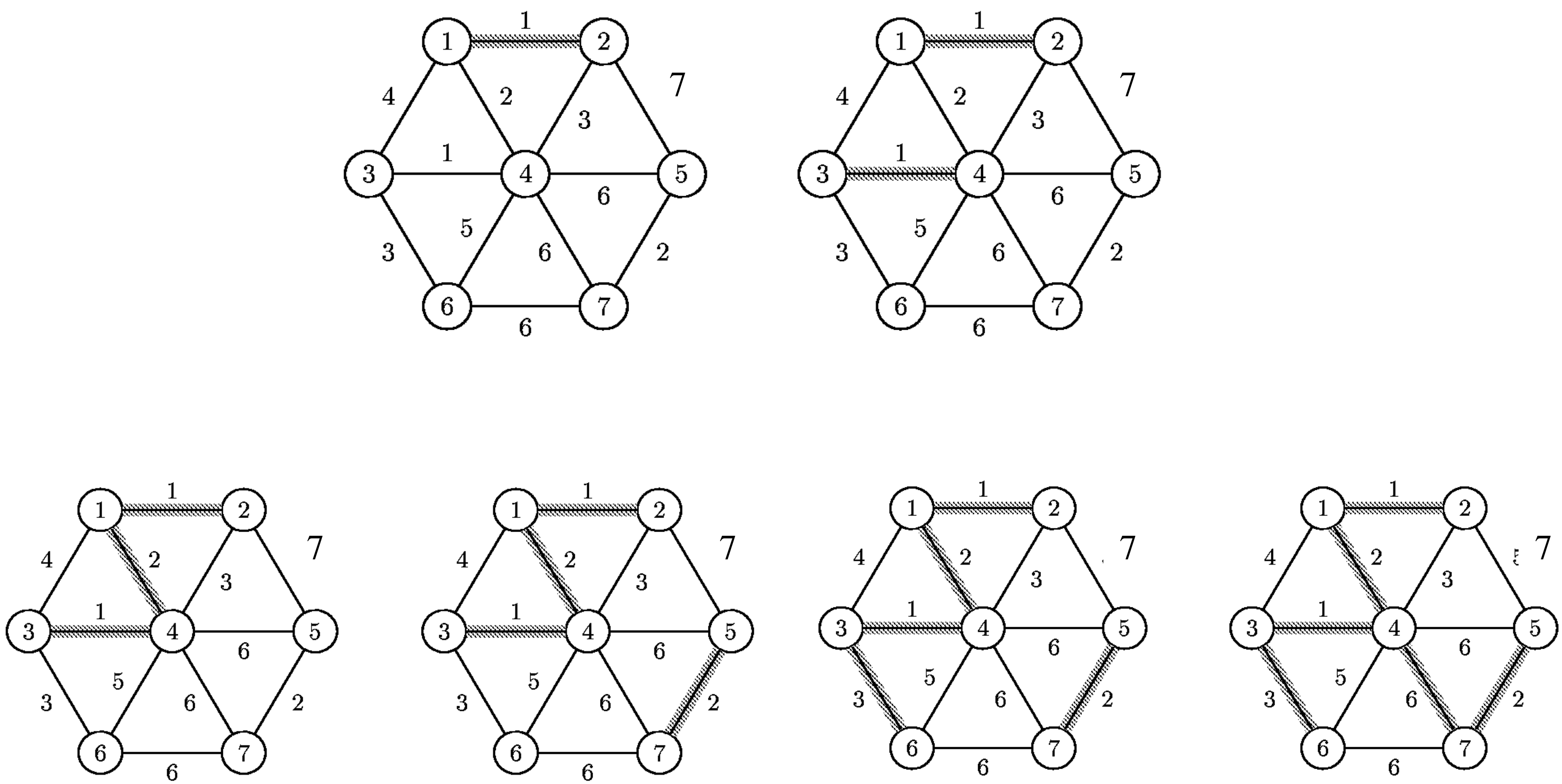
Πρόβλημα Συντομότερων Μονοπατιών (Single Source Shortest Paths)



Παράδειγμα (με Prim)

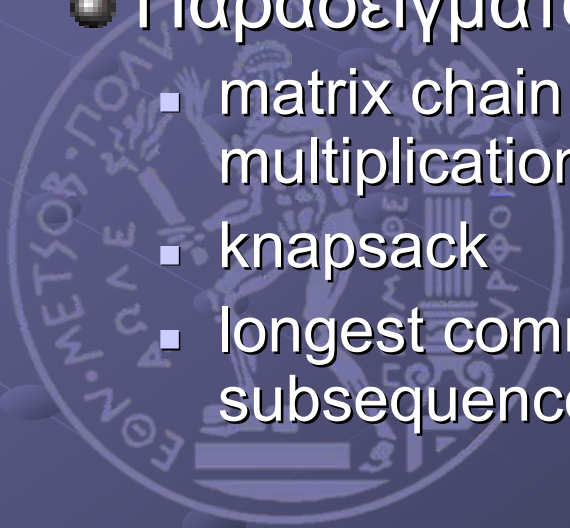


Παράδειγμα (με Kruskal)



Dynamic programming

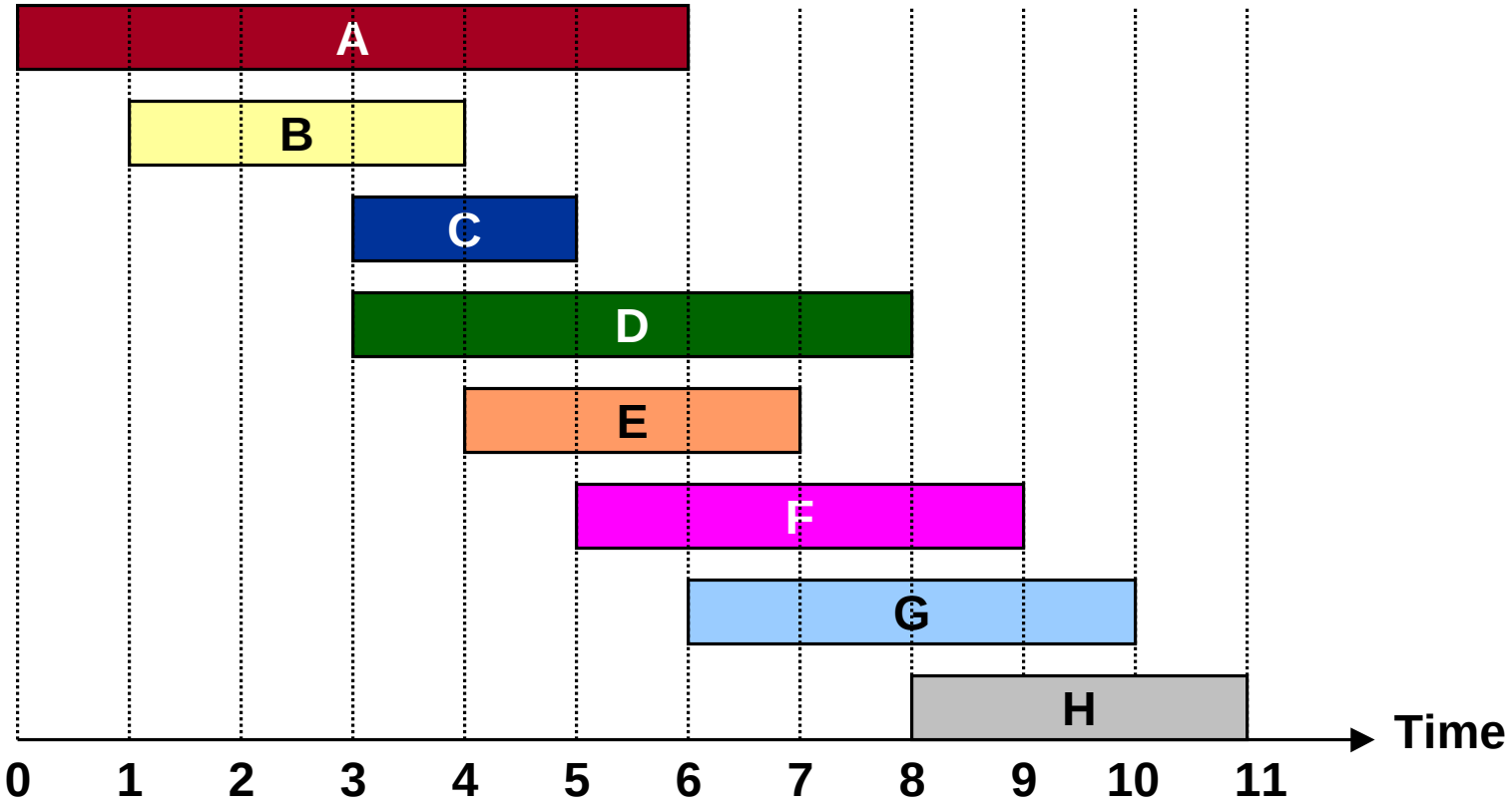
- Ιδέα
- Παραδείγματα (ι)
  - weighted activity selection
- Optimal substructure
- Παραδείγματα (ιι)
  - matrix chain multiplication
  - knapsack
  - longest common subsequence
- Γνωστοί αλγόριθμοι και προβλήματα
  - Floyd (all pairs shortest paths)
  - Travelling salesman



Weighted Activity Selection

Weighted activity selection problem (generalization of CLR 17.1).

- Job requests 1, 2, … , N.
- Job j starts at s_j, finishes at f_j , and has weight w_j .
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.



Activity Selection: Greedy Algorithm

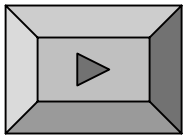
Recall greedy algorithm works if all weights are 1.

Greedy Activity Selection Algorithm

Sort jobs by increasing finish times so that $f_1 \leq f_2 \leq \ldots \leq f_N$.

S = jobs selected.

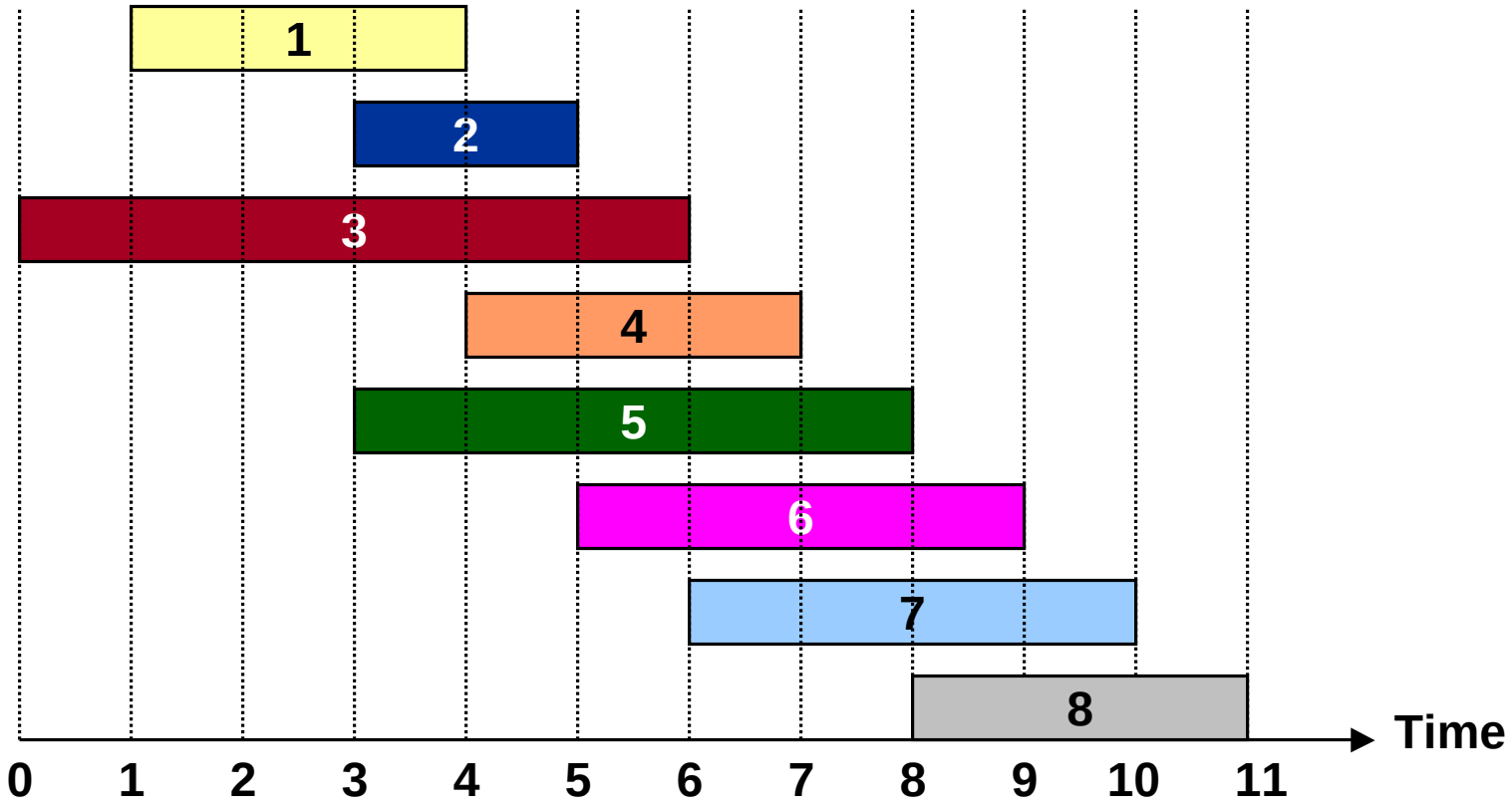
```
S = φ
FOR j = 1 to N
   IF (job j compatible with A)
      S ← S ∪ {j}
RETURN S
```



Weighted Activity Selection

Notation.

- Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_N$.
- Define q_j = largest index i < j such that job i is compatible with j.
  - $q_7 = 3$, $q_2 = 0$



Weighted Activity Selection: Structure

Let OPT(j) = value of optimal solution to the problem consisting of job requests {1, 2, . . . , j }.

- **Case 1: OPT selects job j.**
  - **can't use incompatible jobs { q_j + 1, q_j + 2, . . . , j-1 }**
  - **must include optimal solution to problem consisting of remaining compatible jobs { 1, 2, . . . , q_j }**
- **Case 2: OPT does not select job j.**
  - **must include optimal solution to problem consisting of remaining compatible jobs { 1, 2, . . . , j - 1 }**

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{w_j + OPT(q_j),\ OPT(j-1)\} & \text{otherwise} \end{cases}$$

Weighted Activity Selection: Brute Force

Recursive Activity Selection

INPUT: $N, s_1,\ldots,s_N , f_1,\ldots,f_N , w_1,\ldots,w_N$

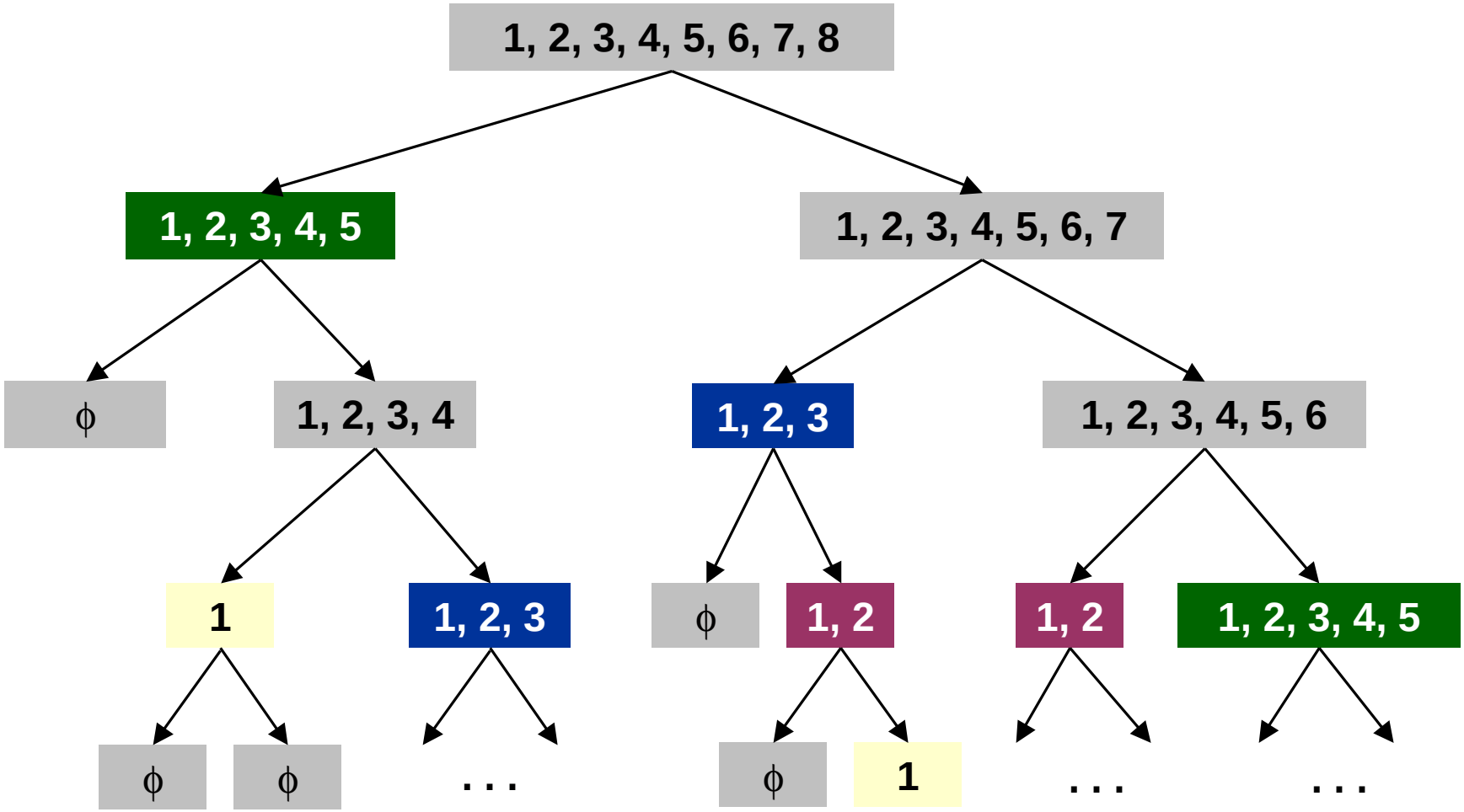
Sort jobs by increasing finish times so that $f_1 \le f_2 \le \ldots \le f_N$.

Compute $q_1, q_2 , \ldots , q_N$

```
r-compute(j) {
   IF (j = 0)
      RETURN 0
   ELSE
      return max(w_j + r-compute(q_j), r-compute(j-1))
}
```

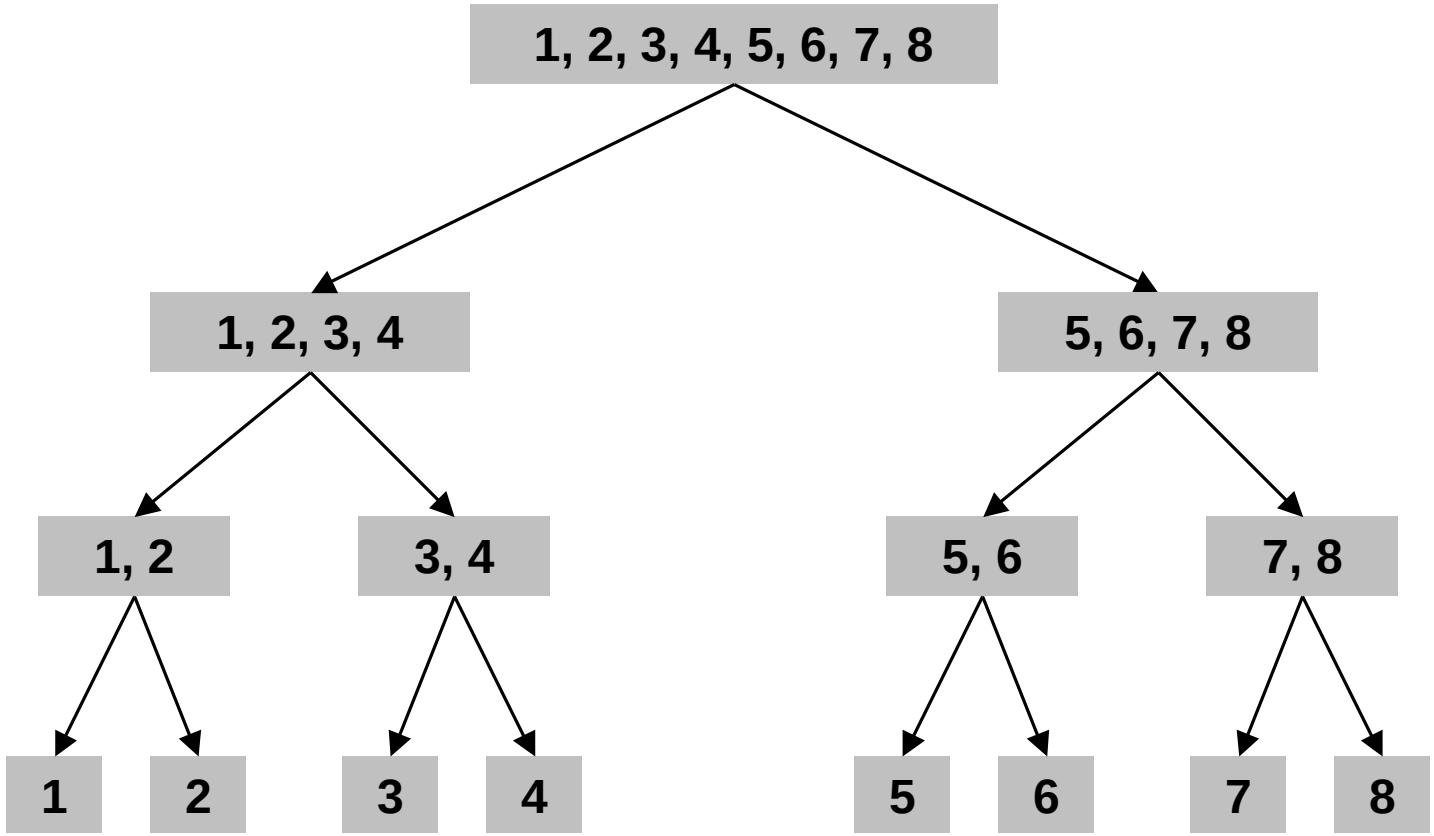
Dynamic Programming Subproblems

Spectacularly redundant subproblems $\Rightarrow$ exponential algorithms.



Divide-and-Conquer Subproblems

Independent subproblems $\Rightarrow$ efficient algorithms.



Weighted Activity Selection: Memoization

Memoized Activity Selection

```
INPUT: N, s1,…,sN , f1,…,fN , w1,…,wN

Sort jobs by increasing finish times so that
f1 ≤ f2 ≤ ... ≤ fN.

Compute q1, q2 , ... , qN

Global array OPT[0..N]
FOR j = 0 to N
   OPT[j] = "empty"

m-compute(j) {
   IF (j = 0)
      OPT[0] = 0
   ELSE IF (OPT[j] = "empty")
      OPT[j] = max(wj + m-compute(qj), m-compute(j-1))
   RETURN OPT[j]
}
```

Weighted Activity Selection: Running Time

Claim: memoized version of algorithm takes O(N log N) time.

- **Ordering by finish time: O(N log N).**
- **Computing q_j: O(N log N) via binary search.**

- **`m-compute(j)`: each invocation takes O(1) time and either**
  - **(i) returns an existing value of `OPT[]`**
  - **(ii) fills in one new entry of `OPT[]` and makes two recursive calls**

- **Progress measure Φ = # nonempty entries of `OPT[]`.**
  - **Initially $\Phi = 0$, throughout $\Phi \leq N$.**
  - **(ii) increases Φ by 1 $\Rightarrow$ at most 2N recursive calls.**

- **Overall running time of `m-compute(N)` is O(N).**

Weighted Activity Selection: Finding a Solution

`m-compute(N)` determines value of optimal solution.

- **Modify to obtain optimal solution itself.**

Finding an Optimal Set of Activities

```
ARRAY: OPT[0..N]
Run m-compute(N)

find-sol(j) {
   IF (j = 0)
      output nothing
   ELSE IF (w_j + OPT[q_j] > OPT[j-1])
      print j
      find-sol(q_j)
   ELSE
      find-sol(j-1)
}
```

- **# of recursive calls $\leq$ N $\Rightarrow$ O(N).**

Weighted Activity Selection: Bottom-Up

Unwind recursion in memoized algorithm.

Bottom-Up Activity Selection

INPUT: N, $s_1, \ldots, s_N$, $f_1, \ldots, f_N$, $w_1, \ldots, w_N$

Sort jobs by increasing finish times so that $f_1 \leq f_2 \leq \ldots \leq f_N$.

Compute $q_1, q_2, \ldots, q_N$

ARRAY: OPT[0..N]
OPT[0] = 0

FOR j = 1 to N
  OPT[j] = max(w_j + OPT[q_j], OPT[j-1])

Dynamic Programming Overview

Dynamic programming.

- Similar to divide-and-conquer.
  - solves problem by combining solution to sub-problems
- Different from divide-and-conquer.
  - sub-problems are not independent
  - save solutions to repeated sub-problems in table

Recipe.

- Characterize structure of problem.
  - optimal substructure property
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Top-down vs. bottom-up.

- Different people have different intuitions.

Πότε μπορούμε να χρησιμοποιήσουμε την τεχνική του Δυναμικού Προγραμματισμού;

- Το πρόβλημα πρέπει να έχει δύο ιδιότητες:
  - **Optimal substructure:** Οποιαδήποτε υπακολουθία της βέλτιστης ακολουθίας αποφάσεων είναι βέλτιστη για το αντίστοιχο υποπρόβλημα.
  - **Overlapping subproblems:** Κάποια μικρότερα στιγμιότυπα του προβλήματος επιλύονται από μια αναδρομική λύση του προβλήματος πολλές φορές..

Παράδειγμα

Συντομότερα μονοπάτια:

Αν v->w: v, … , u, …, w συντομότερο

τότε και u->w: u,…, w συντομότερο

Αναδρομική σχέση «προς τα εμπρός» (forward):

$$MinPathCost_{u \to w} = \min_{\forall k \ adjacent \ to \ u} (Cost(u,k) + MinPathCost_{k \to w})$$

Αναδρομική σχέση «προς τα πίσω» (backward):

$$MinPathCost_{u \to w} = \min_{\forall k \ adjacent \ to \ w} (MinPathCost_{u \to k} + Cost(k,w))$$

Matrix-Chain Multiplication

Δίνεται μια ακολουθία πινάκων $A_1, A_2, \ldots, A_n$, θέλουμε να υπολογίσουμε το γινόμενο τους
$A_1 \cdot A_2 \cdot \ldots \cdot A_n$

Βάζουμε παρενθέσεις και υπολογίζουμε τα γινόμενα από ζευγάρια πινάκων:

$$A_1 \cdot A_2 \cdot A_3 \cdot A_4 = (A_1 \cdot (A_2 \cdot A_3)) \cdot A_4$$

Κόστος του πολλαπλασιασμού δύο πινάκων

Δίνεται ένας πίνακας $\boldsymbol{p} \times \boldsymbol{q}$ A και ένας πίνακας $\boldsymbol{q} \times \boldsymbol{r}$ B, το γινόμενο τους είναι ένας πίνακας $\boldsymbol{p} \times \boldsymbol{r}$ C που προκύπτει ως εξής:

$$C_{i,j} = \sum_{k=1}^{q} A_{i,k} B_{k,j}$$

Το κόστος υπολογισμού του πίνακα C είναι $\boldsymbol{p} \cdot \boldsymbol{q} \cdot \boldsymbol{r}$.

Κόστος του πολλαπλασιασμού τριών πινάκων

Δίνονται τρεις πίνακες:

$A \quad p \times q$

$B \quad q \times r$

$C \quad r \times s$

Υπάρχουν 2 τρόποι για το $A \cdot B \cdot C$:

$(A \cdot B) \cdot C$

Κόστος: $p \cdot q \cdot r + p \cdot r \cdot s = p \cdot r \cdot (q + s)$.

$A \cdot (B \cdot C)$

Κόστος: $q \cdot r \cdot s + p \cdot q \cdot s = (p + r) \cdot q \cdot s$.

Παράδειγμα

Δίνονται τρεις πίνακες:

A 10×100

B 100×5

C 5×50

$(A \cdot B) \cdot C$ κόστος: ***7500***

$10 \cdot 100 \cdot 5 = 5000$

$10 \cdot 5 \cdot 50 = 2500$

$A \cdot (B \cdot C)$ κόστος: ***75000***

$100 \cdot 5 \cdot 50 = 25000$

$10 \cdot 100 \cdot 50 = 50000$

Περισσότεροι πίνακες, περισσότεροι τρόποι να βάλεις παρενθέσεις

Για παράδειγμα για 4 πίνακες υπάρχουν οι εξής τρόποι να βάλεις παρενθέσεις:

$(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$
$(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$
$((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$
$((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$
$(((A_1 \cdot A_2) \cdot A_3) \cdot A_4)$

Το κόστος του υπολογισμού του γινομένου των πινάκων εξαρτάται από τον τρόπο που βάζουμε τις παρενθέσεις.

Εύρεση βέλτιστη θέση παρενθέσεων

Ποίες επιλογές θα κάνουμε για να βρούμε τη θέση των παρενθέσεων

Ποία είναι η αρχική επιλογή που πρέπει να κάνουμε πριν κάνουμε άλλες επιλογές;

Έχει το Matrix-Chain Multiplication Optimal Substructure;

Έστω πως δίνεται μια βέλτιστη τοποθέτηση των παρενθέσεων για το $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ και ότι ισχύει

$$(A_1 \cdot A_2 \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot A_{k+2} \cdot \ldots \cdot A_n),$$

τι μπορούμε να πού για τον τρόπο που έχουν τοποθετηθεί η παρενθέσεις των γινομένων

$A_1 \cdot A_2 \cdot \ldots \cdot A_k$ και $A_{k+1} \cdot A_{k+2} \cdot \ldots \cdot A_n$

Αναδρομική Σχέση

Έστω $d_0, d_1, \ldots, d_n$ οι διαστάσεις των πινάκων; δηλαδή

ο πίνακας A_i έχει διαστάσεις $d_{i-1} \times d_i$.

Έστω $C(i, j)$ το κόστος μια βέλτιστης τοποθέτησης παρενθέσεων του γινομένου $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.

$$C(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} (d_{i-1} d_k d_j + C(i,k) + C(k+1,j)) & \text{if } i < j \end{cases}$$

The Algorithm

Optimal-Parenthesization(*d*, *n*)

```
1     for i = 1..n
2     do C[i, i] ← 0
3     for i = 1..n – 1
4         do for j = 1..n – i
5                do C[j, j + i] ← ∞
6                for k = j..j + i – 1
7                    do c ← d[i – 1] · d[k] · d[j + i] + C[j, k] + C[k + 1, j + i]
8                       if c < C[j, j + i]
9                       then C[j, j + i] ← c
10                           return C[1, n]
```

Computing a Solution, Not Only Its Cost

Optimal-Parenthesization(*d*, *n*)

```
1     for i = 1..n
2         do C[i, i] ← 0
3     for i = 1..n – 1
4         do for j = 1..n – i
5                do C[j, j + i] ← ∞
6                for k = j..j + i – 1
7                    do c ← d[i – 1] · d[k] · d[j + i] + C[j, k] + C[k + 1, j + i]
8                       if c < C[j, j + i]
9                       then C[j, j + i] ← c
10                           S[j, j + i] ← k
```

Computing a Solution, Not Only Its Cost

```
Extract-Parenthesization(i, j)
1       if  i = j
2       then  print "A_i"
3       else  print "("
4             Extract-Parenthesization(i, S[i, j])
5             print " · "
6             Extract-Parenthesization(S[i, j] + 1, j)
7             print ")"
```

Complexity of the Algorithm

Ο αλγόριθμος για το πρόβλημα matrix-chain multiplication απλά γεμίζει έναν πίνακα.

Χρονική πολυπλοκότητα $O(n^3)$.

Στην πραγματικότητα $\Omega(n^3)$.

Χωρική πολυπλοκότητα: $\Theta(n^2)$ για αποθήκευση των C και s.

Knapsack Problem

Knapsack problem.

- Given N objects and a "knapsack."
- Item i weighs $w_i > 0$ Newtons and has value $v_i > 0$.
- Knapsack can carry weight up to W Newtons.
- Goal: fill knapsack so as to maximize total value.

v_i / w_i

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

Greedy = 35: { 5, 2, 1 }

OPT value = 40: { 3, 4 }

Knapsack Problem: Structure

OPT(n, w) = max profit subset of items {1, . . . , n} with weight limit w.

- **Case 1: OPT selects item n.**
  - **new weight limit = $w - w_n$**
  - **OPT selects best of {1, 2, . . . , n – 1} using this new weight limit**
- **Case 2: OPT does not select item n.**
  - **OPT selects best of {1, 2, . . . , n – 1} using weight limit w**

$$OPT(n,w) = \begin{cases} 0 & \text{if } n = 0 \\ OPT(n-1,w) & \text{if } w_n > w \\ \max\{OPT(n-1,w),\ \ v_n + OPT(n-1,w-w_n)\} & \text{otherwise} \end{cases}$$

New dynamic programming technique.

- **Weighted activity selection: binary choice.**
- **Segmented least squares: multi-way choice.**
- **Knapsack: adding a new variable.**

Knapsack Problem: Bottom-Up

Bottom-Up Knapsack

```
INPUT: N, W, w_1,…,w_N, v_1,…,v_N

ARRAY: OPT[0..N, 0..W]

FOR w = 0 to W
   OPT[0, w] = 0

FOR n = 1 to N
   FOR w = 1 to W
      IF (w_n > w)
         OPT[n, w] = OPT[n-1, w]
      ELSE
         OPT[n, w] = max {OPT[n-1, w], v_n + OPT[n-1, w-w_n ]}

RETURN OPT[N, W]
```

Knapsack Algorithm

W + 1 →

N + 1 ↓

Weight Limit	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	35	40

Item	Value	Weight
1	1	1
2	6	2
3	8	5
4	22	6
5	28	7

Knapsack Problem: Running Time

Knapsack algorithm runs in time O(NW).

- **Not polynomial in input size!**
- **"Pseudo-polynomial."**
- **Decision version of Knapsack is "NP-complete."**
- **Optimization version is "NP-hard."**

Knapsack approximation algorithm.

- **There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum.**
- **Stay tuned.**

Longest Common Subsequence

Monkey DNA = ACCGGAATTAAACCCAATT = S_1

Your DNA = TAGCATTCACATTTAATATC = S_2

G ATT ACA = S_3

$$LCS(X_i, Y_j) = \begin{cases} i = 0 \text{ or } j = 0: & ? \\ i = j: & LCS(X_{i-1}, Y_{j-1}) \\ i \neq j: & \max(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) \end{cases}$$

Longest Common Subsequence

S[j][k] = the length of the longest common substring of strings u[1…j] and v[1…k]

```
LONGEST-COMMON-SUBSEQUENCE(u,v)
1. init S[j][k] to 0 for every j=0,…,|u|
                        and every k=0,…,|v|
2. for j=1 to |u| do
3.     for k=1 to |v| do
4.         S[j][k] = max{ S[j-1][k], S[j][k-1] }
5.         if (u[j] = v[k]) then
6.             S[j][k] = S[j-1][k-1]+1
7. RETURN S[|u|][|v|]
```

All Pairs Shortest Paths

Είσοδος: Γράφος G(V,E) με n κόμβους και βάρη σε κάθε πλευρά u->w, C[u,w].

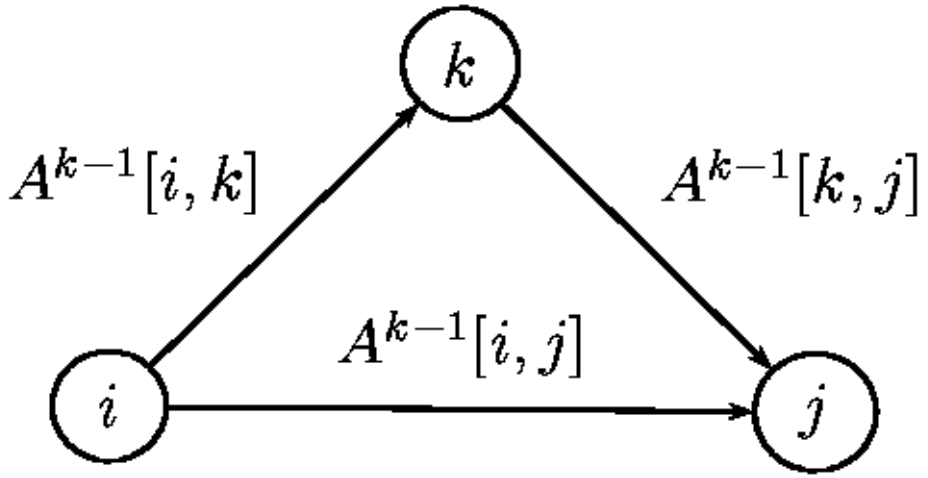
Εξοδος: συντομότερο μονοπάτι για κάθε ζεύγος κόμβων.

Αρχικοποίηση:

$$A[i,j] = \begin{cases} 0 & ,i = j \\ C[i,j] & ,i \neq j \& (i,j) \in E \\ \infty & ,\text{αλλιώς} \end{cases}$$

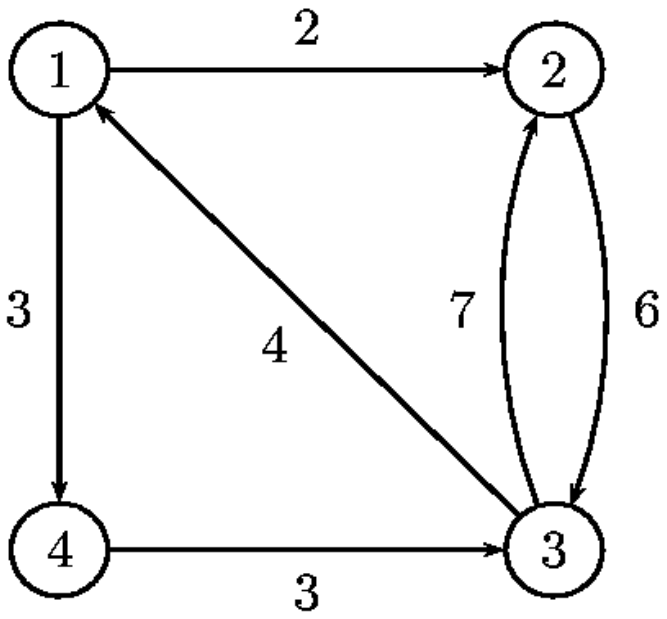
Αναδρομική σχέση

$$A^k[i,j] = \min(A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j])$$



Αλγόριθμος Floyd

```
procedure AllShortestPaths (...);
begin
  for i:=1 to n do
    for j:=1 to n do A[i,j]:=Cost[i,j];
  for k:=1 to n do
    for i:=1 to n do
      for j:=1 to n do
        A[i,j]:= min(A[i,j], A[i,k]+A[k,j])
end
```



$$C = \begin{bmatrix} 0 & 2 & \infty & 3 \\ \infty & 0 & 6 & \infty \\ 4 & 7 & 0 & \infty \\ \infty & \infty & 3 & 0 \end{bmatrix}$$

$$A^1 = \begin{bmatrix} 0 & 2 & \infty & 3 \\ \infty & 0 & 6 & \infty \\ 4 & 6 & 0 & 7 \\ \infty & \infty & 3 & 0 \end{bmatrix}, A^2 = \begin{bmatrix} 0 & 2 & 8 & 3 \\ \infty & 0 & 6 & \infty \\ 4 & 6 & 0 & 7 \\ \infty & \infty & 3 & 0 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 2 & 8 & 3 \\ 10 & 0 & 6 & 13 \\ 4 & 6 & 0 & 7 \\ 7 & 9 & 3 & 0 \end{bmatrix}, A^4 = \begin{bmatrix} 0 & 2 & 6 & 3 \\ 10 & 0 & 6 & 13 \\ 4 & 6 & 0 & 7 \\ 7 & 9 & 3 & 0 \end{bmatrix}$$

Πρόβλημα Πλανόδιου Πωλητή (Traveling Salesman Problem - TSP)

Είσοδος: Πλήρης γράφος με βάρη.

Εξοδος: Κύκλος ελαχίστου κόστους που περνάει από όλους τους κόμβους 1 φορά.

Αρχή βελτιστότητας:

$$MinCostTour_{1\to 1} = \min_{k\neq 1}\{cost[1,k] + MinCostTour_{k\to 1}\}$$

Αναδρομικές Σχέσεις

g(i,S) : κόστος συντομότερου μονοπατιού από τον i στον 1 που περνά από όλους τους κόμβους του S.

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{cost[1,k] + g(k, V - \{1,k\})\}$$

$$g(i, S) = \min_{j \in S} \{cost[i,j] + g(j, S - \{j\})\}$$

$$g(i, \emptyset) = cost[i, 1].$$