Analysis of Algorithms: Introduction

- What is Algorithm?
  - a clearly specified set of simple instructions to be followed to solve a problem
    - Takes a set of values, as input and
    - produces a value, or set of values, as output
  - May be specified
    - In English
    - As a computer program
    - As a pseudo-code
- Data structures
  - Methods of organizing data
- Program = algorithms + data structures

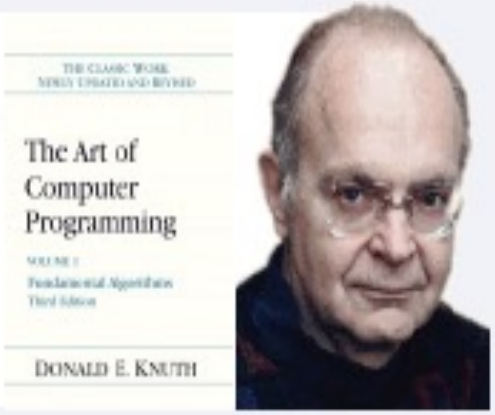
Analysis of Algorithms: Introduction

" A procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation. " — *webster.com*

Merriam-Webster

" An algorithm is a finite, definite, effective procedure, with some input and some output. "

— *Donald Knuth*



Analysis of Algorithms: Introduction

Etymology. [Knuth, TAOCP]

- *Algorism* = process of doing arithmetic using Arabic numerals.
- A misperception: *algiros* [painful] + *arithmos* [number].
- True origin: Abu 'Abd Allah Muhammad ibn Musa al-Khwarizm was a famous 9th century Persian textbook author who wrote *Kitāb al-jabr wa'l-muqābala*, which evolved into today's high school algebra text.

Introduction to Algorithms

- Stacks and queues.
- Sorting.
- Searching.
- Graph algorithms.
- String processing.

```
private static void sort(double[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int lt = lo, gt = hi;
   int i = lo;
   while (i <= gt)
   {
      if      (a[i] < a[lo]) exch(a, lt++, i++);
      else if (a[i] > a[lo]) exch(a, i, gt--);
      else                   i++;
   }

   sort(a, lo, lt - 1);
   sort(a, gt + 1, hi);
}
```

Emphasizes critical thinking, problem-solving, and code.

Design and Analysis of Algorithms

- Greedy.
- Divide-and-conquer.
- Dynamic programming.
- Network flow.
- Randomized algorithms.
- Intractability.
- Coping with intractability.
- Data structures.

$$\begin{aligned}\sum_{i=1}^{N}\sum_{j=i+1}^{N}\frac{2}{j-i+1} &= 2\sum_{i=1}^{N}\sum_{j=2}^{N-i+1}\frac{1}{j} \\ &\leq 2N\sum_{j=1}^{N}\frac{1}{j} \\ &\sim 2N\int_{x=1}^{N}\frac{1}{x}\,dx \\ &= 2N\ln N\end{aligned}$$

Emphasizes critical thinking, problem-solving, and rigorous analysis.

Applications of Algorithms

Internet. Web search, packet routing, distributed file sharing, ...
Biology. Human genome project, protein folding, ...
Computers. Circuit layout, databases, caching, networking, compilers, ...
Computer graphics. Movies, video games, virtual reality, ...
Security. Cell phones, e-commerce, voting machines, ...
Multimedia. MP3, JPG, DivX, HDTV, face recognition, ...
Social networks. Recommendations, news feeds, advertisements, ...
Physics. N-body simulation, particle collision simulation, ...
⋮

We emphasize algorithms and techniques that are useful in practice.

Introduction

- Why need algorithm analysis ?
  - writing a working program is not good enough
  - The program may be inefficient!
  - If the program is run on a large data set, then the running time becomes an issue

Example: Selection Problem

- Given a list of N numbers, determine the *k*th largest, where k ≤ N.
- Algorithm 1:
  (1) Read N numbers into an array
  (2) Sort the array in decreasing order by some simple algorithm
  (3) Return the element in position k

Example: Selection Problem…

- Algorithm 2:
  (1) Read the first k elements into an array and sort them in decreasing order
  (2) Each remaining element is read one by one
    - If smaller than the kth element, then it is ignored
    - Otherwise, it is placed in its correct spot in the array, bumping one element out of the array.
  (3) The element in the kth position is returned as the answer.

Example: Selection Problem…

- Which algorithm is better when
  - N =100 and k = 100?
  - N =100 and k = 1?
- What happens when N = 1,000,000 and k = 500,000?
- There exist better algorithms

Algorithm Analysis

- We only analyze *correct* algorithms
- An algorithm is correct
  - If, for every input instance, it halts with the correct output
- Incorrect algorithms
  - Might not halt at all on some input instances
  - Might halt with other than the desired answer
- Analyzing an algorithm
  - Predicting the resources that the algorithm requires
  - Resources include
    - Memory
    - Communication bandwidth
    - Computational time (usually most important)

Algorithm Analysis…

- Factors affecting the running time
  - computer
  - compiler
  - algorithm used
  - input to the algorithm
    - The content of the input affects the running time
    - typically, the *input size* (number of items in the input) is the main consideration
      - E.g. sorting problem ⇒ the number of items to be sorted
      - E.g. multiply two matrices together ⇒ the total number of elements in the two matrices
- Machine model assumed
  - Instructions are executed one after another, with no concurrent operations ⇒ Not parallel computers

Example

- Calculate $\sum_{i=1}^{N} i^3$

```
int sum(int n)
{
      int partialSum;

1     partialSum=0;
2     for (int i=1;i<=n;i++)
3        partialSum += i*i*i;
4     return partialSum;
}
```

1
2N+2
4N
1

- Lines 1 and 4 count for one unit each
- Line 3: executed N times, each time four units
- Line 2: (1 for initialization, N+1 for all the tests, N for all the increments) total 2N + 2
- total cost: 6N + 4 $\Rightarrow$ O(N)

Worst- / average- / best-case

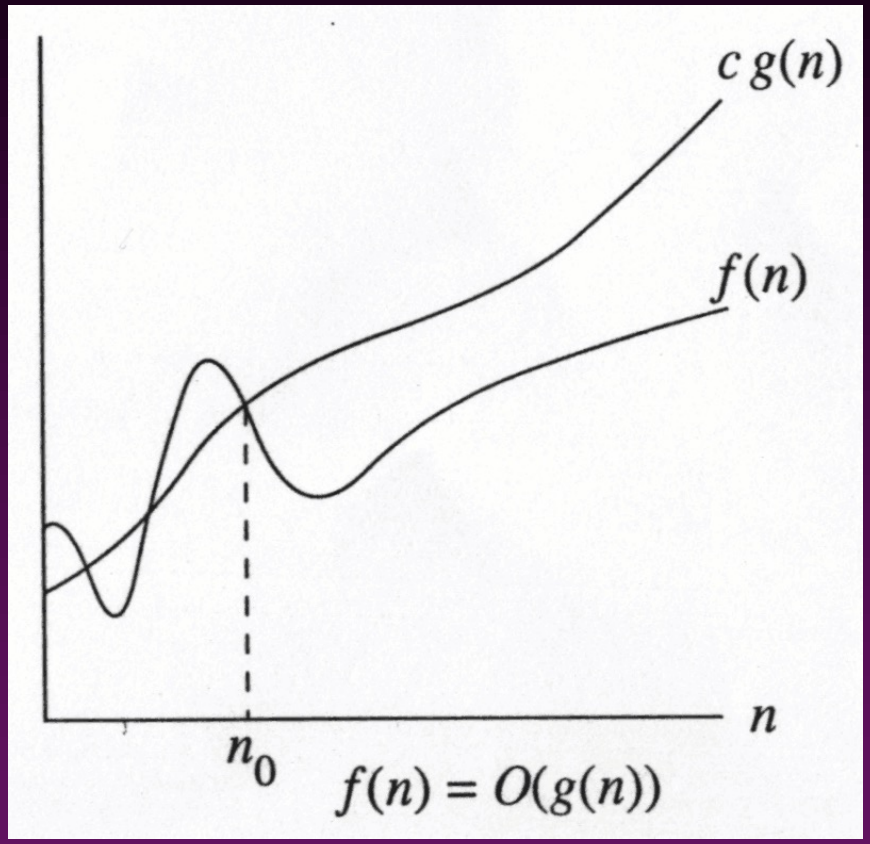
- Worst-case running time of an algorithm
  - The longest running time for **any** input of size n
  - An upper bound on the running time for any input

    ⇒ guarantee that the algorithm will never take longer
  - Example: Sort a set of numbers in increasing order; and the data is in decreasing order
  - The worst case can occur fairly often
    - E.g. in searching a database for a particular piece of information
- Best-case running time
  - sort a set of numbers in increasing order; and the data is already in increasing order
- Average-case running time
  - May be difficult to define what “average” means

Running-time of algorithms

- Bounds are for the algorithms, rather than programs
  - programs are just implementations of an algorithm, and almost always the details of the program do not affect the bounds
- Bounds are for algorithms, rather than problems
  - A problem can be solved with several algorithms, some are more efficient than others

Growth Rate



- The idea is to establish a relative order among functions for large n
- $\exists$ c , $n_0 > 0$ such that $f(N) \leq c\, g(N)$ when $N \geq n_0$
- f(N) grows no faster than g(N) for "large" N

Asymptotic notation: Big-Oh

- f(N) = O(g(N))
- There are positive constants c and n_0 such that

$$f(N) \leq c\, g(N) \text{ when } N \geq n_0$$

- The growth rate of f(N) is *less than or equal to* the growth rate of g(N)
- g(N) is an upper bound on f(N)

Big-Oh: example

- Let $f(N) = 2N^2$. Then
  - $f(N) = O(N^4)$
  - $f(N) = O(N^3)$
  - $f(N) = O(N^2)$ (best answer, asymptotically tight)

- $O(N^2)$: reads “order N-squared” or “Big-Oh N-squared”

Big Oh: more examples

- $N^2 / 2 – 3N = O(N^2)$
- $1 + 4N = O(N)$
- $7N^2 + 10N + 3 = O(N^2) = O(N^3)$
- $\log_{10} N = \log_2 N / \log_2 10 = O(\log_2 N) = O(\log N)$
- $\sin N = O(1)$; $10 = O(1)$, $10^{10} = O(1)$
- $$\sum_{i=1}^{N} i \leq N \cdot N = O(N^2)$$
  $$\sum_{i=1}^{N} i^2 \leq N \cdot N^2 = O(N^3)$$
- $\log N + N = O(N)$
- $\log^k N = O(N)$ for any constant k
- $N = O(2^N)$, but 2^N is not $O(N)$
- 2^{10N} is not $O(2^N)$

Math Review: logarithmic functions

$$x^a = b \quad iff \quad \log_x b = a$$

$$\log ab = \log a + \log b$$

$$\log_a b = \frac{\log_m b}{\log_m a}$$

$$\log a^b = b \log a$$

$$a^{\log n} = n^{\log a}$$

$$\log^b a = (\log a)^b \neq \log a^b$$

$$\frac{d \log_e x}{dx} = \frac{1}{x}$$

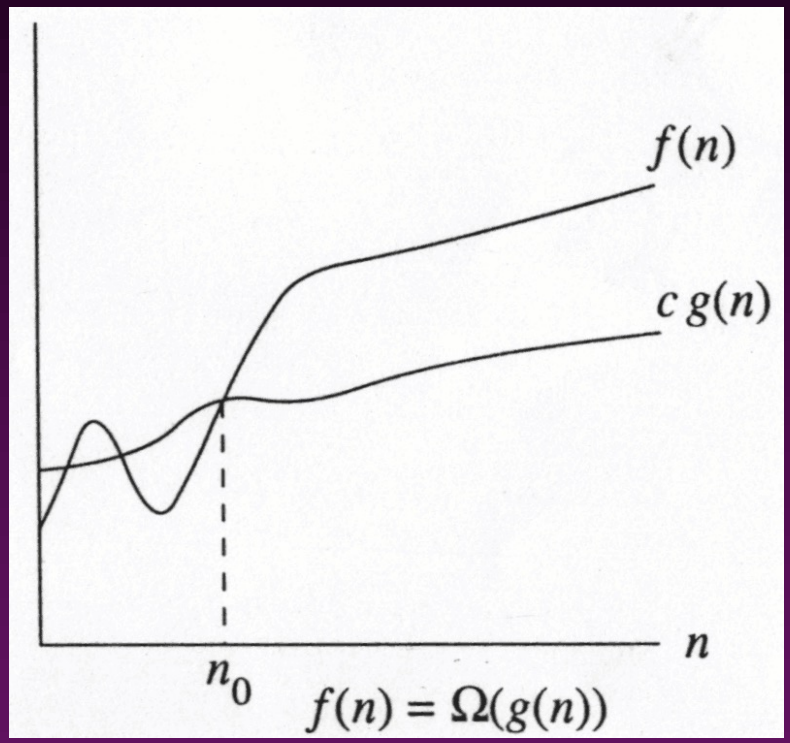
Some rules

When considering the growth rate of a function using Big-Oh

- Ignore the lower order terms and the coefficients of the highest-order term
- No need to specify the base of logarithm
  - Changing the base from one constant to another changes the value of the logarithm by only a constant factor

- If $T_1(N) = O(f(N)$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$,
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$

Big-Omega



- ∃ c , n_0 > 0 such that f(N) ≥ c g(N) when N ≥ n_0
- f(N) grows no slower than g(N) for “large” N

Big-Omega

- $f(N) = \Omega(g(N))$
- There are positive constants c and n_0 such that

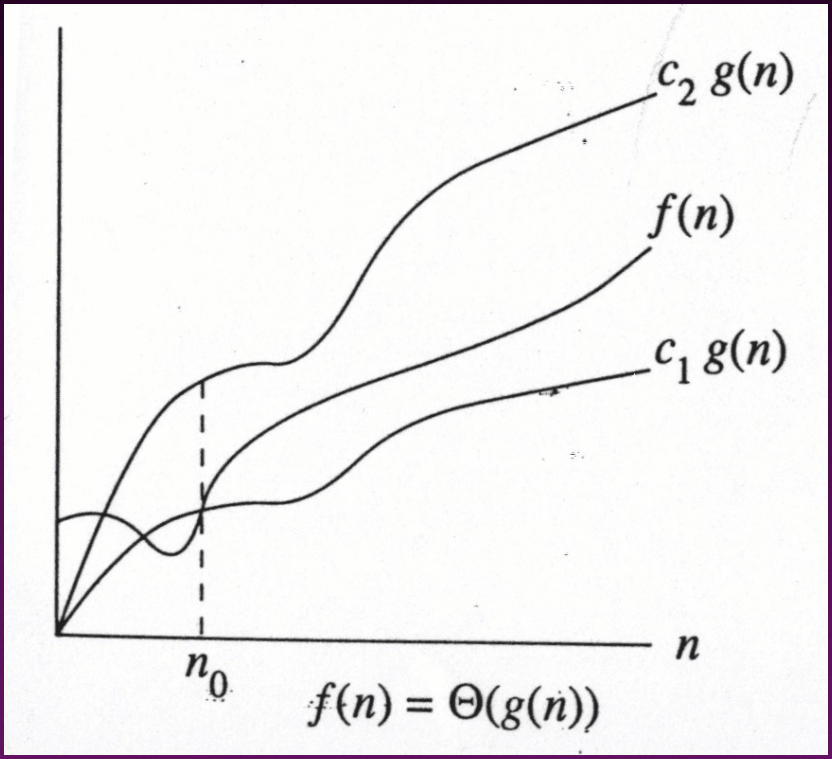
$$f(N) \geq c\, g(N) \text{ when } N \geq n_0$$

- The growth rate of f(N) is *greater than or equal to* the growth rate of g(N).

Big-Omega: examples

- Let $f(N) = 2N^2$. Then
  - $f(N) = \Omega(N)$
  - $f(N) = \Omega(N^2)$ (best answer)

f(N) = Θ(g(N))



- the growth rate of f(N) *is the same as* the growth rate of g(N)

Big-Theta

- $f(N) = \Theta(g(N))$ iff
  $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$
- The growth rate of f(N) *equals* the growth rate of g(N)
- Example: Let $f(N)=N^2$, $g(N)=2N^2$
  - Since $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$, thus $f(N) = \Theta(g(N))$.
- Big-Theta means the bound is the tightest possible.

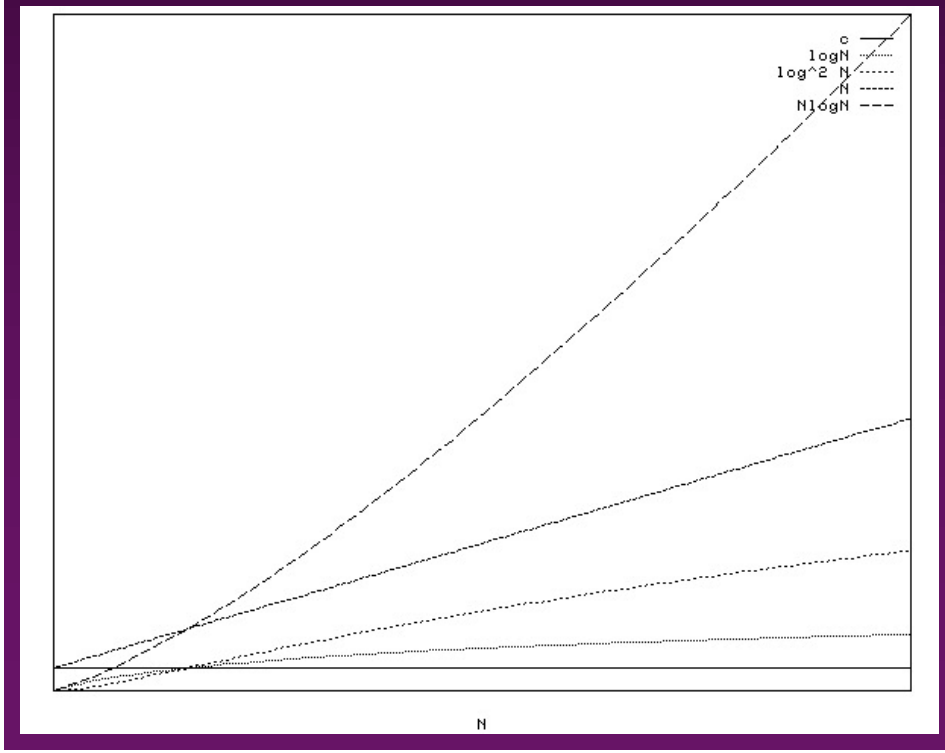
Some rules

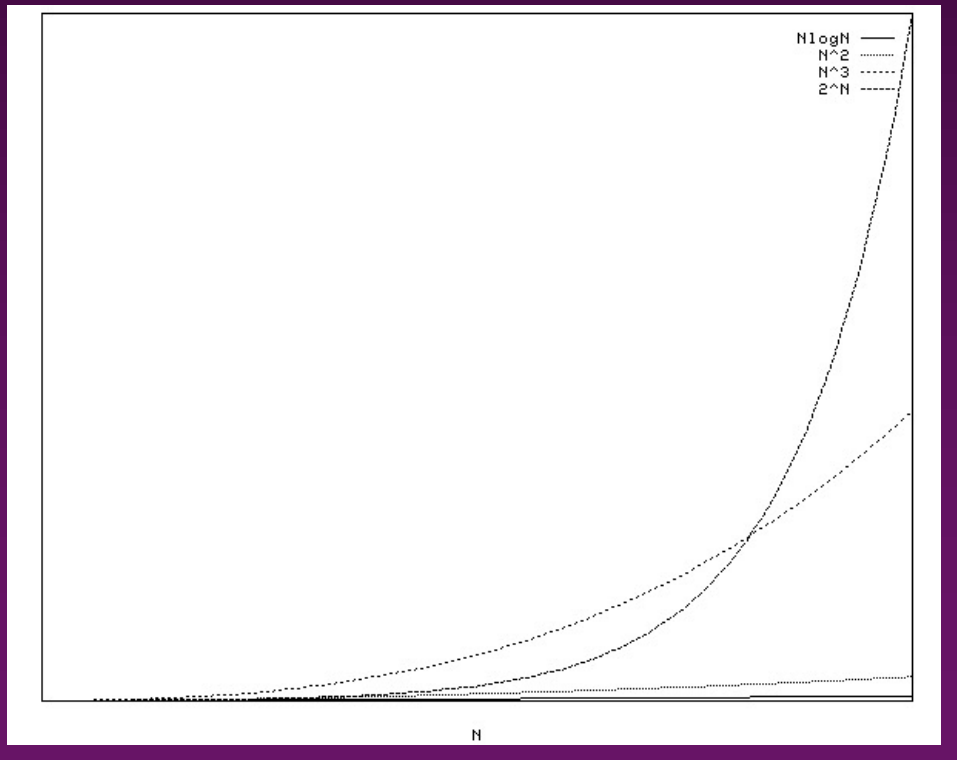
- If T(N) is a polynomial of degree k, then $T(N) = \Theta(N^k)$.
- For logarithmic functions, $T(\log_m N) = \Theta(\log N)$.

Typical Growth Rates

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Figure 2.1 Typical growth rates





Growth rates …

- Doubling the input size
  - $f(N) = c \quad \Rightarrow f(2N) = f(N) = c$
  - $f(N) = \log N \quad \Rightarrow f(2N) = f(N) + \log 2$
  - $f(N) = N \quad \Rightarrow f(2N) = 2\, f(N)$
  - $f(N) = N^2 \quad \Rightarrow f(2N) = 4\, f(N)$
  - $f(N) = N^3 \quad \Rightarrow f(2N) = 8\, f(N)$
  - $f(N) = 2^N \quad \Rightarrow f(2N) = f^2(N)$
- Advantages of algorithm analysis
  - To eliminate bad algorithms early
  - pinpoints the bottlenecks, which are worth coding carefully

Using L' Hopital's rule

- L' Hopital's rule
  - If $\lim_{n\to\infty} f(N) = \infty$ and $\lim_{n\to\infty} g(N) = \infty$

    then $\lim_{n\to\infty} \frac{f(N)}{g(N)} = \lim_{n\to\infty} \frac{f'(N)}{g'(N)}$

- Determine the relative growth rates (using L' Hopital's rule if necessary)
  - compute $\lim_{n\to\infty} \frac{f(N)}{g(N)}$
  - if 0: f(N) = O(g(N)) and f(N) is not Θ(g(N))
  - if constant $\neq 0$: f(N) = Θ(g(N))
  - if ∞: f(N) = Ω(f(N)) and f(N) is not Θ(g(N))
  - limit oscillates: no relation

General Rules

- For loops
  - at most the running time of the statements inside the for-loop (including tests) times the number of iterations.
- Nested for loops

```
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        k++;
```

  - the running time of the statement multiplied by the product of the sizes of all the for-loops.
  - $O(N^2)$

General rules (cont'd)

- Consecutive statements

```
for (i=0;i<n;i++)
        a[i]=0;
for (i=0;i<n;i++)
        for (j=0;j<n;j++)
                a[i] += a[j]+i+j;
```

  - These just add
  - $O(N) + O(N^2) = O(N^2)$

- `If S1`

  `Else S2`

  - never more than the running time of the test plus the larger of the running times of S1 and S2.

Another Example

- Maximum Subsequence Sum Problem
- Given (possibly negative) integers A_1, A_2, ..., A_n, find the maximum value of $\sum_{k=i}^{j} A_k$
  - For convenience, the maximum subsequence sum is 0 if all the integers are negative
- E.g. for input –2, 11, -4, 13, -5, -2
  - Answer: 20 (A_2 through A_4)

Algorithm 1: Simple

- Exhaustively tries all possibilities (brute force)

```
int maxSubSum1 (const vector<int> &a)
{
        int maxSum=0;

        for (int i=0;i<a.size();i++)
                for (int j=i;j<a.size();j++)
                {
                        int thisSum=0;

                        for (int k=i;k<=j;k++)
                                thisSum += a[k];

                        if (thisSum > maxSum)
                                maxSum = thisSum;
                }
        return maxSum;
}
```

- $O(N^3)$

Algorithm 2: Divide-and-conquer

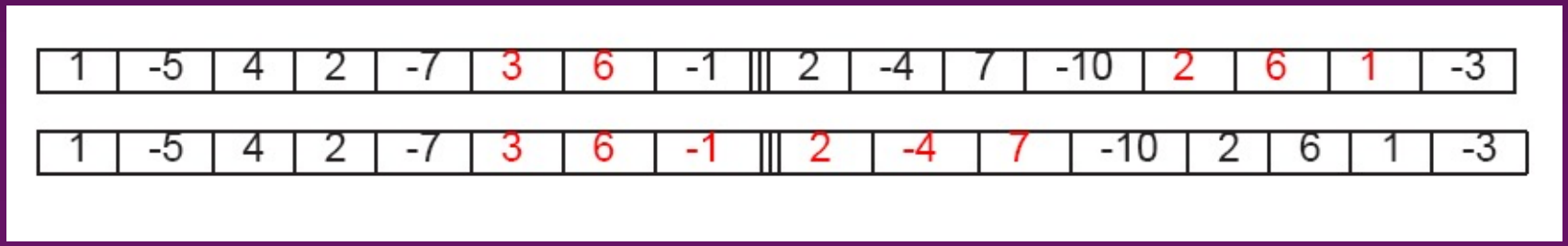
- Divide-and-conquer
  - split the problem into two roughly equal subproblems, which are then solved **recursively**
  - patch together the two solutions of the subproblems to arrive at a solution for the whole problem

First half				Second half			
4	-3	5	-2	-1	2	6	-2

- The maximum subsequence sum can be
  - Entirely in the left half of the input
  - Entirely in the right half of the input
  - It crosses the middle and is in both halves

Algorithm 2 (cont'd)

- The first two cases can be solved recursively
- For the last case:
  - find the largest sum in the first half that includes the last element in the first half
  - the largest sum in the second half that includes the first element in the second half
  - add these two sums together



Algorithm 2 …

// Input : $A[i \ldots j]$ with $i \leq j$

// Output : the MCS of $A[i \ldots j]$

$MCS(A, i, j)$

1. If $i == j$ return $A[i]$ — **O(1)**
2. Else
3. Find $MCS(A, i, \lfloor \frac{i+j}{2} \rfloor)$; — **T(m/2)**
4. Find $MCS(A, \lfloor \frac{i+j}{2} \rfloor + 1, j)$; — **T(m/2)**
5. Find MCS that contains — **O(m)**
   both $A\left[\left|\frac{i+j}{2}\right|\right]$ and $A\left[\left|\frac{i+j}{2}\right| + 1\right]$;
6. Return Maximum of the three sequences found — **O(1)**

Algorithm 2 (cont'd)

- Recurrence equation

$$T(1) = 1$$

$$T(N) = 2T(\frac{N}{2}) + N$$

  - 2 T(N/2): two subproblems, each of size N/2
  - N: for "patching" two solutions to find solution to whole problem

Algorithm 2 (cont'd)

- Solving the recurrence:

$$T(N) = 2T(\frac{N}{2}) + N$$
$$= 4T(\frac{N}{4}) + 2N$$
$$= 8T(\frac{N}{8}) + 3N$$
$$= \cdots$$
$$= 2^k T(\frac{N}{2^k}) + kN$$

- With k=log N (i.e. $2^k = N$), we have

$$T(N) = NT(1) + N\log N$$
$$= N\log N + N$$

- Thus, the running time is O(N log N)
  - faster than Algorithm 1 for large data sets

Complexities of Data Structures and Algorithms

This page covers the Space and Time Big-O complexities of common algorithms widely used in Computer Science and Information Technology. When preparing for technical interviews in the past, one can find spending hours crawling the internet putting together the best, average, and worst case complexities for search and sorting algorithms. Over the last few years, IT experts interviewed at several Silicon Valley start-ups, and also some bigger companies, like Yahoo, eBay, LinkedIn, and Google, and each preparation time of interview, one can think surely that "Why someone hasn't created a nice Big-O summery sheet of complexity analysis so that one can save a lot of preparation time.

Data Structure Operations

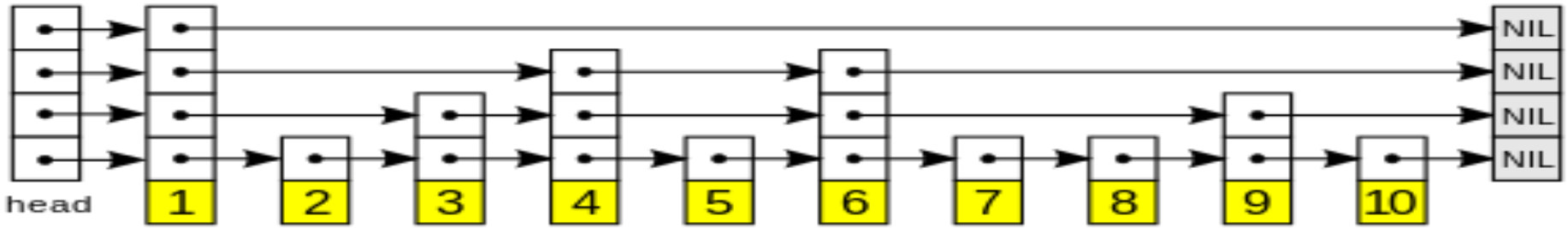
Data Structure	**Time Complexity**								**Space Complexity**
	Average				**Worst**				**Worst**
	Access	**Search**	**Insertion**	**Deletion**	**Access**	**Search**	**Insertion**	**Deletion**	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Skip lists *are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications. Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.*

Skip List *is a data structure that allows fast search within an ordered sequence of elements. Fast search is made possible by maintaining a linked hierarchy of subsequences, each skipping over fewer elements. Searching starts in the sparsest subsequence until two consecutive elements have been found, one smaller and one larger than the element searched for. Via the linked hierarchy these two elements link to elements of the next sparsest subsequence where searching is continued until finally we are searching in the full sequence. The elements that are skipped over may be chosen probabilistically or deterministically with the former being more common.*



Above schematic picture shows the skip list data structure. Each box with an arrow represents a pointer and a row is a linked list giving a sparse subsequence; the numbered boxes at the bottom represent the ordered data sequence. Searching proceeds downwards from the sparsest subsequence at the top until consecutive elements bracketing the search element are found.

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	**Average**	**Worst**	**Worst**
Quick Sort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Tim Sort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)

Algorithm	Time Complexity			Space Complexity
	Best	**Average**	**Worst**	**Worst**
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((n.log(n))^2)	O((n.log(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

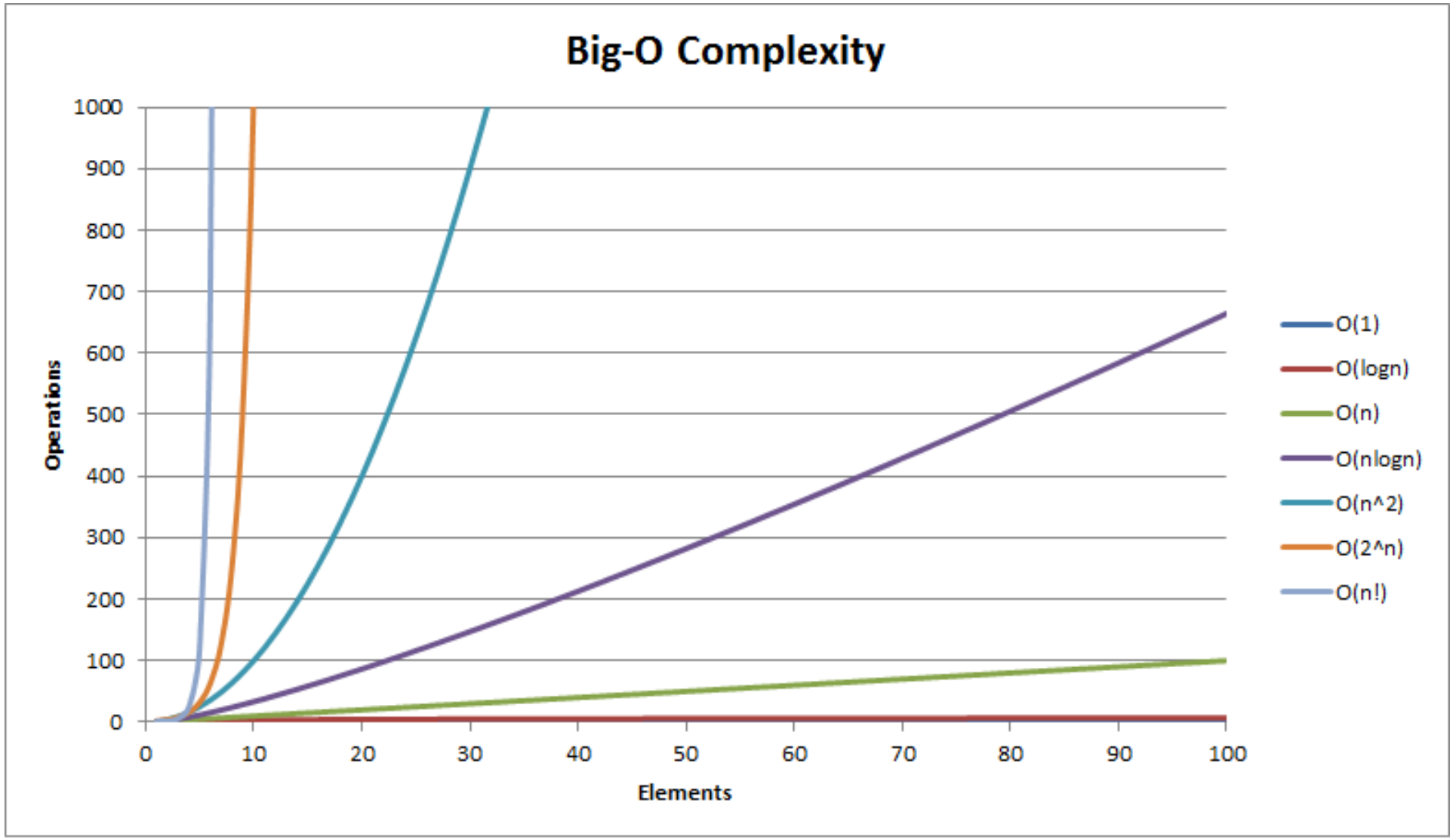
Graph Operations

Node / Edge Management	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency List	O(|V|+|E|)	O(1)	O(1)	O(|V| + |E|)	O(|E|)	O(|V|)
Incidence List	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Adjacency Matrix	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Incidence Matrix	O(|V|·|E|)	O(|V| · |E|)	O(|V|· |E|)	O(|V|· |E|)	O(|V|· |E|)	O(|E|)

Heap Operations

Type	Time Complexity						
	Heapify	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Linked List (Sorted)	-	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Linked List (Unsorted)	-	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Heap	O(n)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Binomial Heap	-	O(1)	O(log(n))	O(log(n))	O(1)	O(log(n))	O(log(n))
Fibonacci Heap	-	O(1)	O(log(n))	O(1)	O(1)	O(log(n))	O(1)

Big-O Complexity Chart



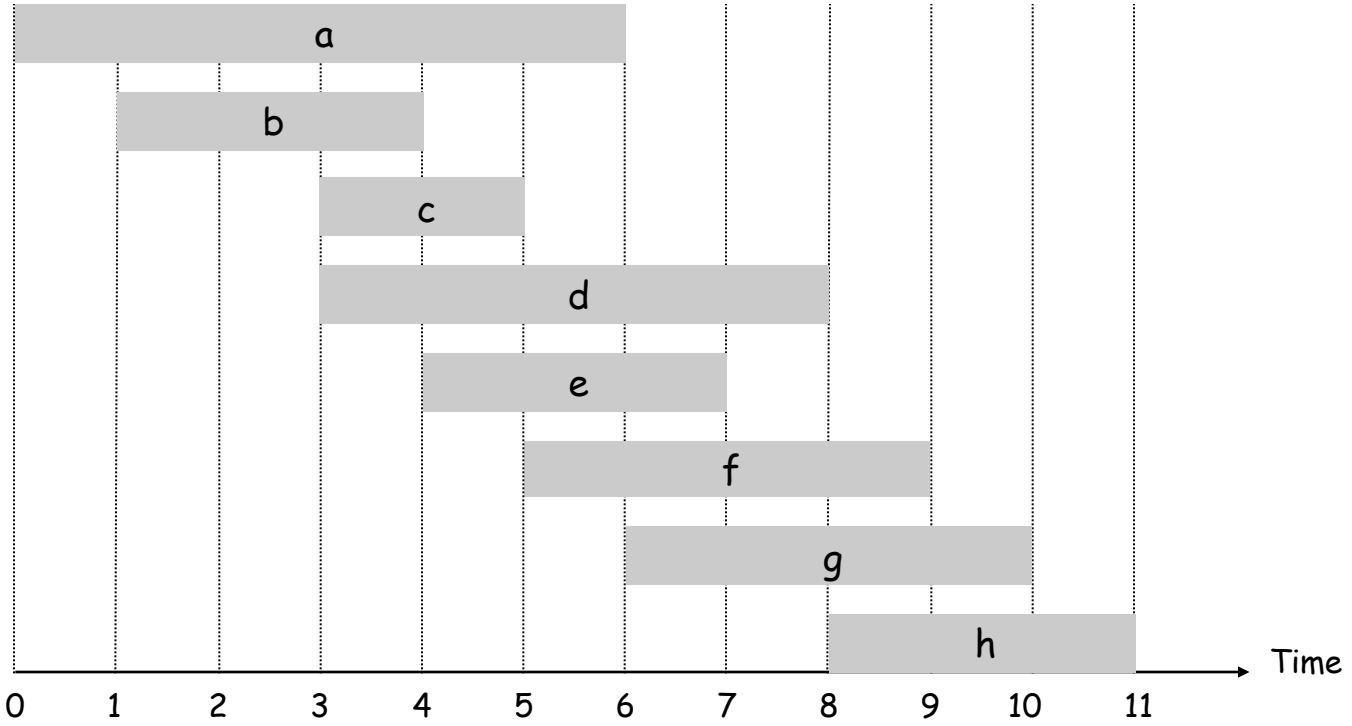
1.2 Five Representative Problems

Interval Scheduling

Input. Set of jobs with start times and finish times.

Goal. Find maximum cardinality subset of mutually compatible jobs.

jobs don't overlap

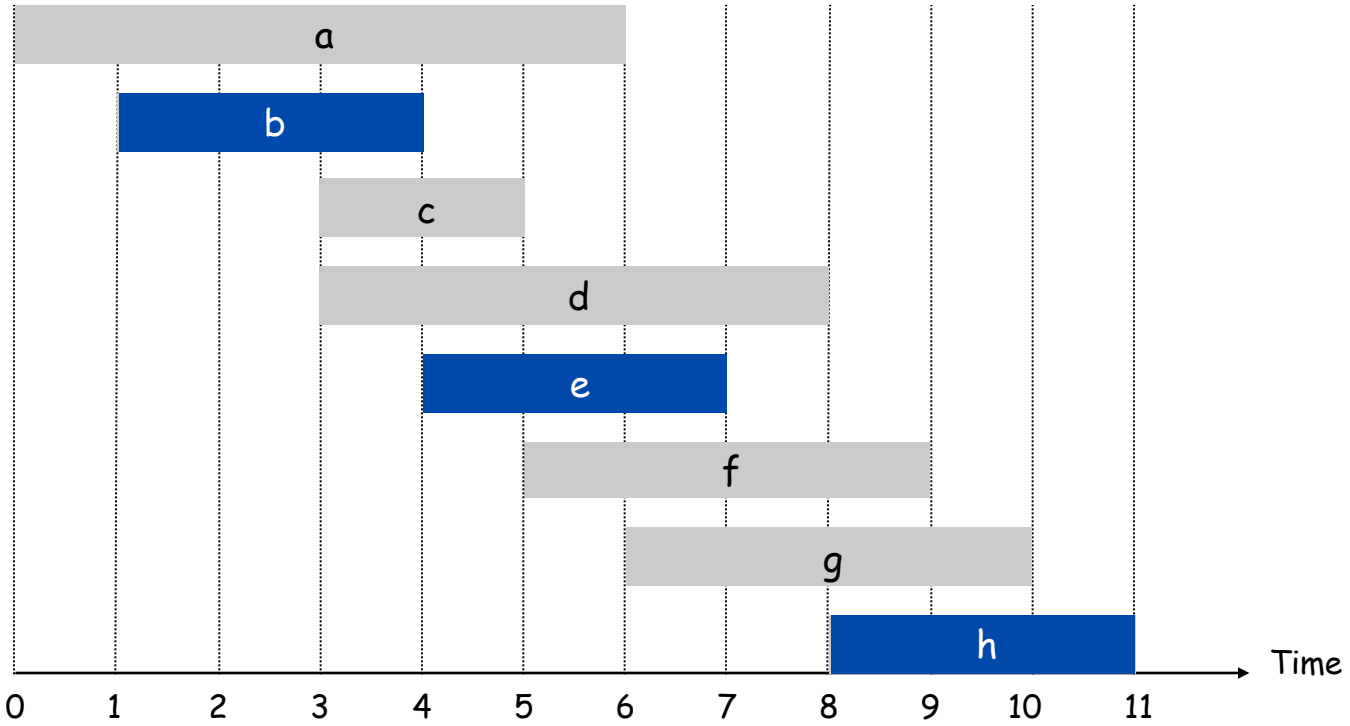


Interval Scheduling

Input. Set of jobs with start times and finish times.

Goal. Find maximum cardinality subset of mutually compatible jobs.

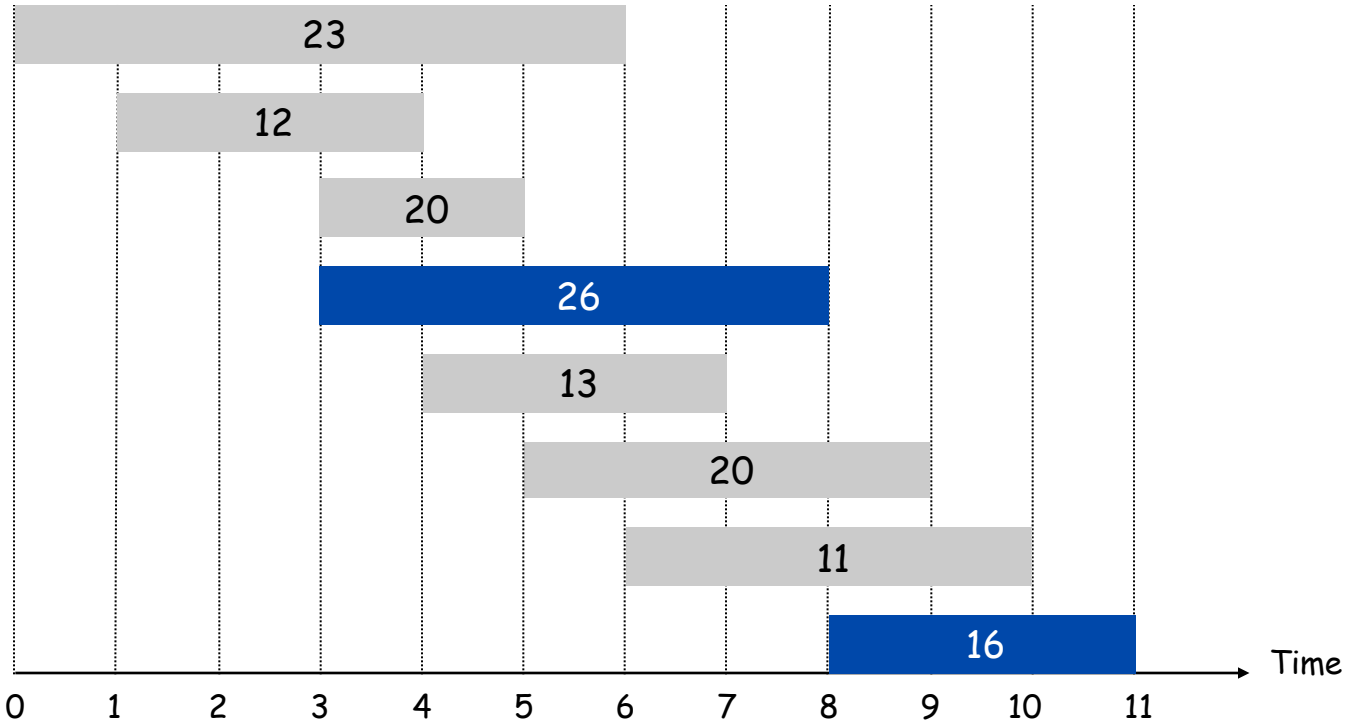
jobs don't overlap



Weighted Interval Scheduling

Input. Set of jobs with start times, finish times, and weights.

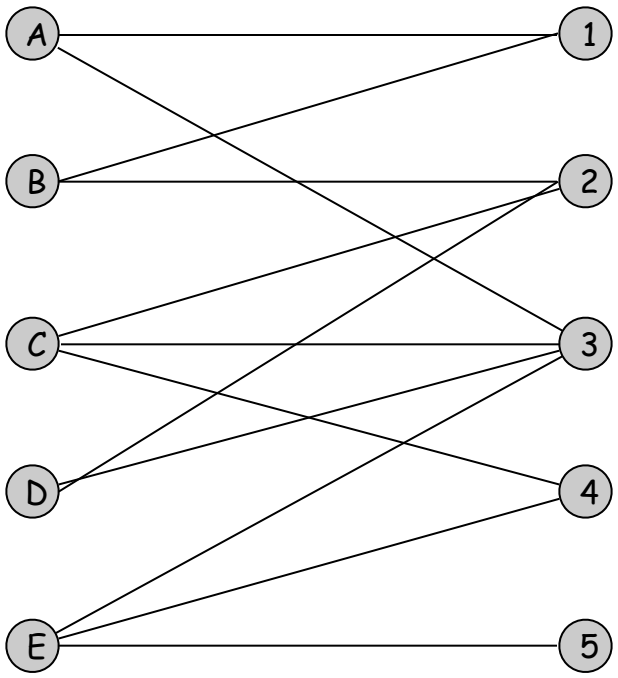
Goal. Find maximum weight subset of mutually compatible jobs.



Bipartite Matching

Input. Bipartite graph.

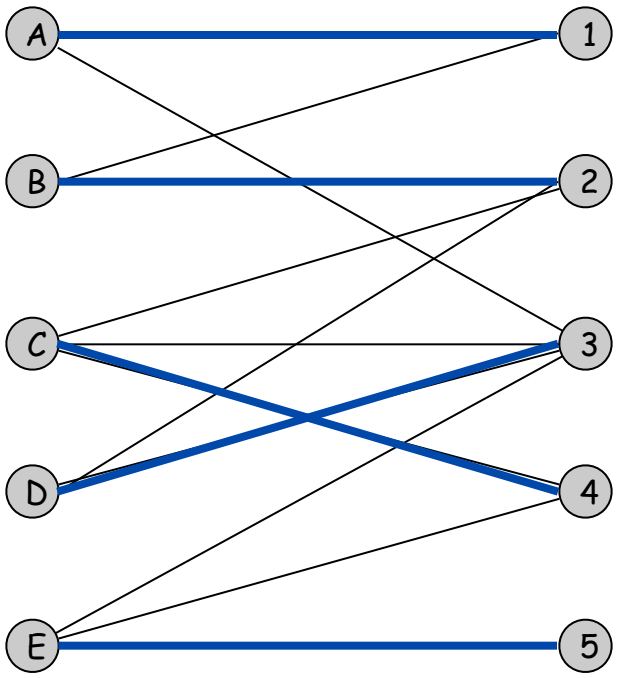
Goal. Find maximum cardinality matching.



Bipartite Matching

Input. Bipartite graph.

Goal. Find maximum cardinality matching.

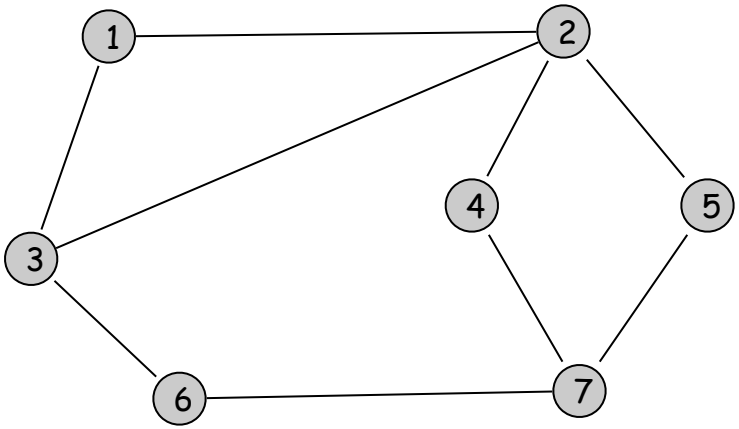


Independent Set

Input. Graph.

Goal. Find maximum cardinality independent set.

subset of nodes such that no two joined by an edge

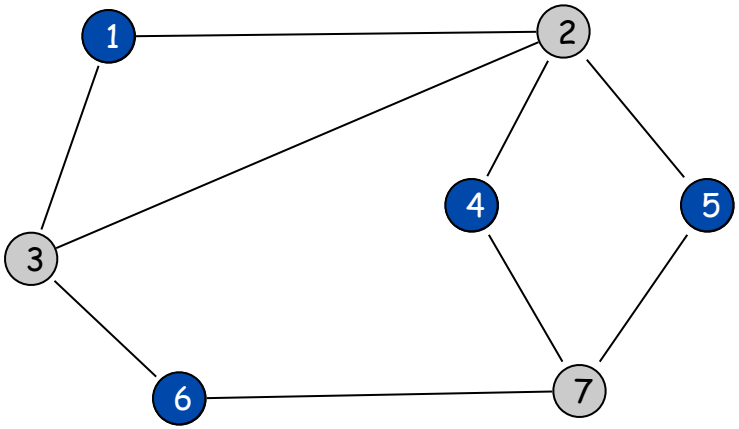


Independent Set

Input. Graph.

Goal. Find maximum cardinality independent set.

subset of nodes such that no two joined by an edge

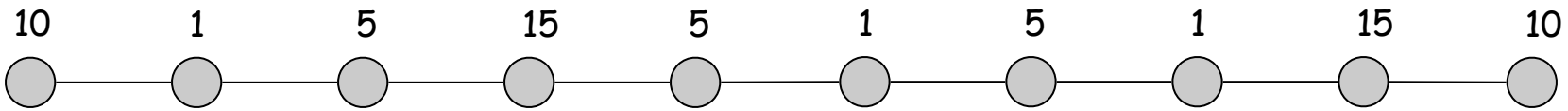


Competitive Facility Location

Input. Graph with weight on each node.

Game. Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbours has been selected.

Goal. Select a maximum weight subset of nodes.



Second player can guarantee 20, but not 25.

Five Representative Problems

Variations on a theme: independent set.

Interval scheduling: n log n greedy algorithm.

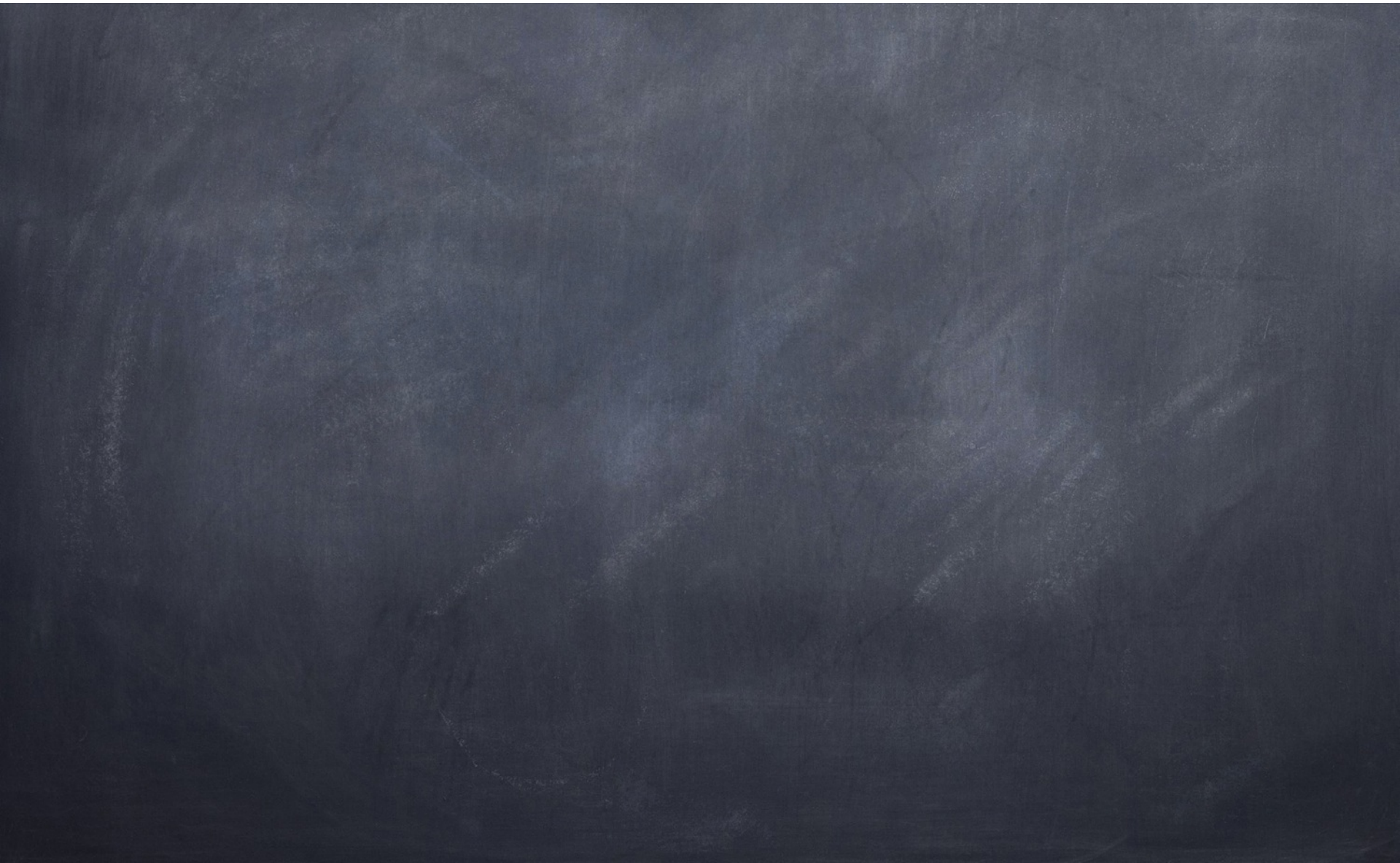
Weighted interval scheduling: n log n dynamic programming algorithm.

Bipartite matching: n^k max-flow based algorithm.

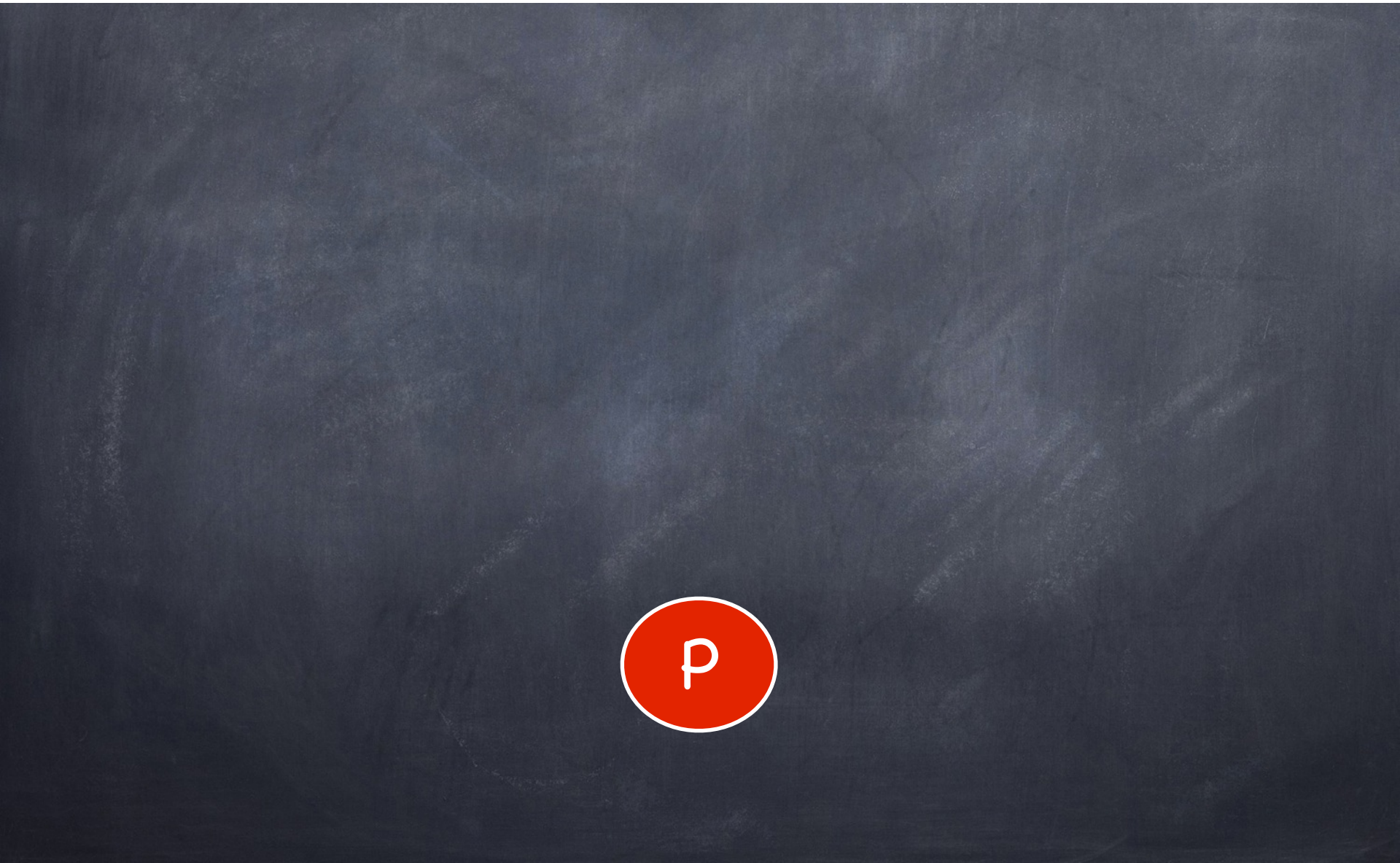
Independent set: NP-complete.

Competitive facility location: PSPACE-complete.

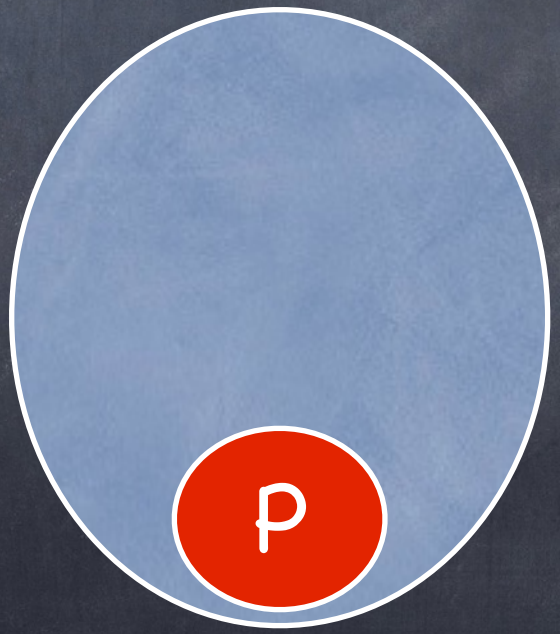
A few low order Complexity Classes



A few low order Complexity Classes



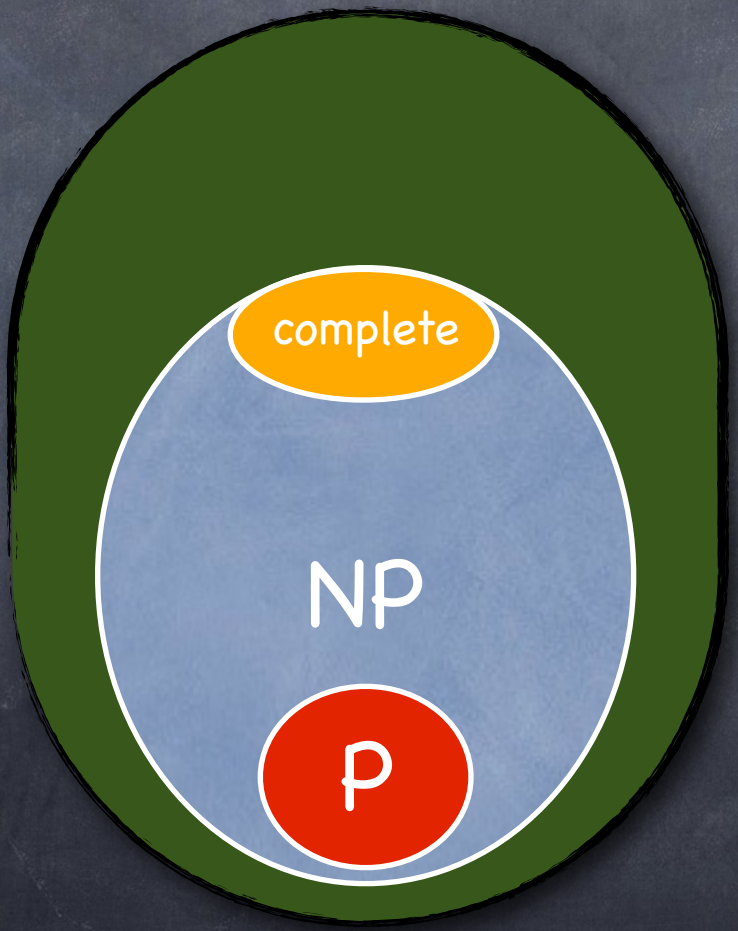
A few low order Complexity Classes



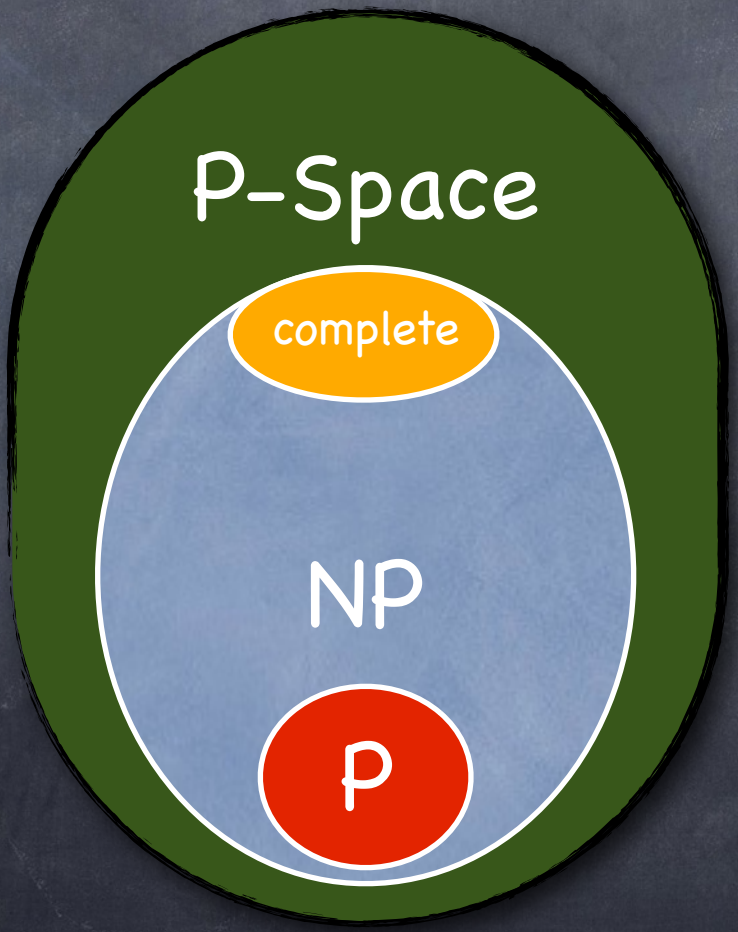
A few low order Complexity Classes

A few low order Complexity Classes

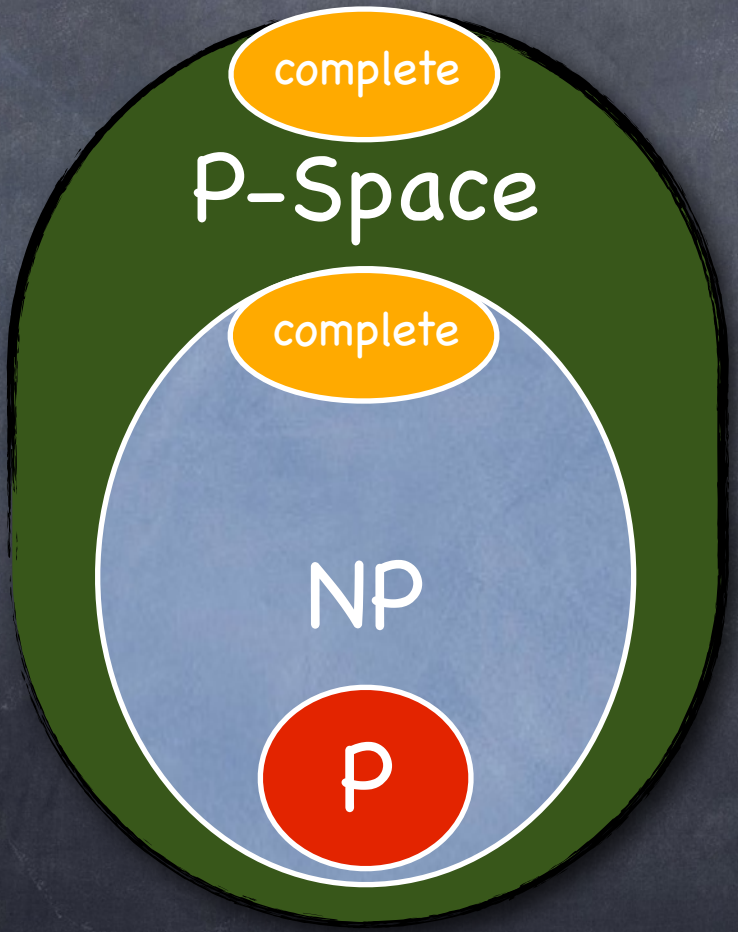
A few low order Complexity Classes



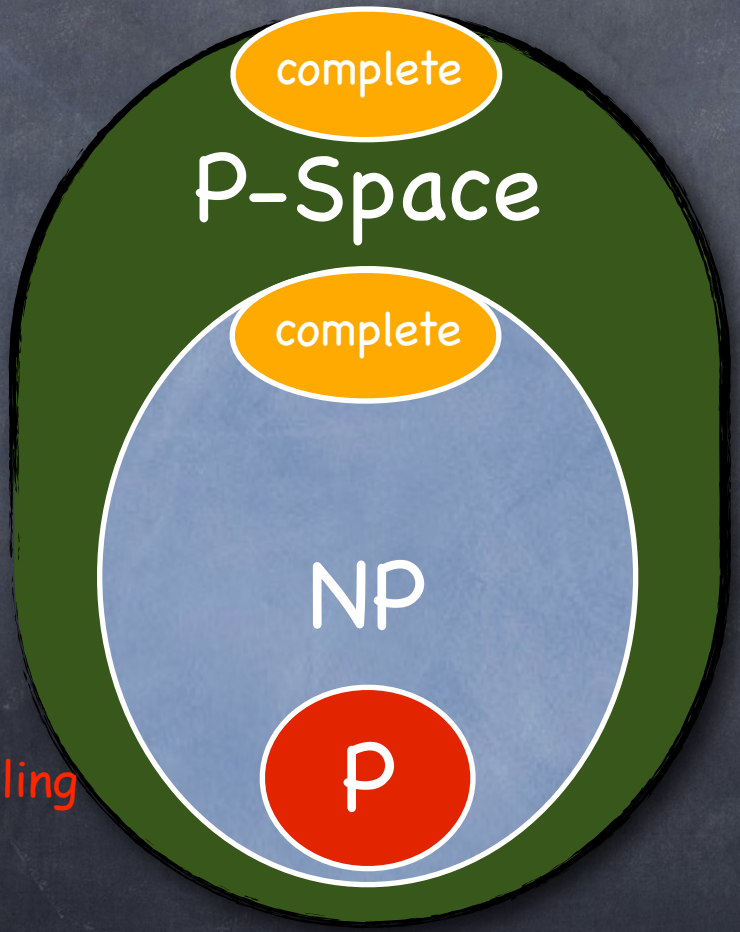
A few low order Complexity Classes



A few low order Complexity Classes



A few low order Complexity Classes

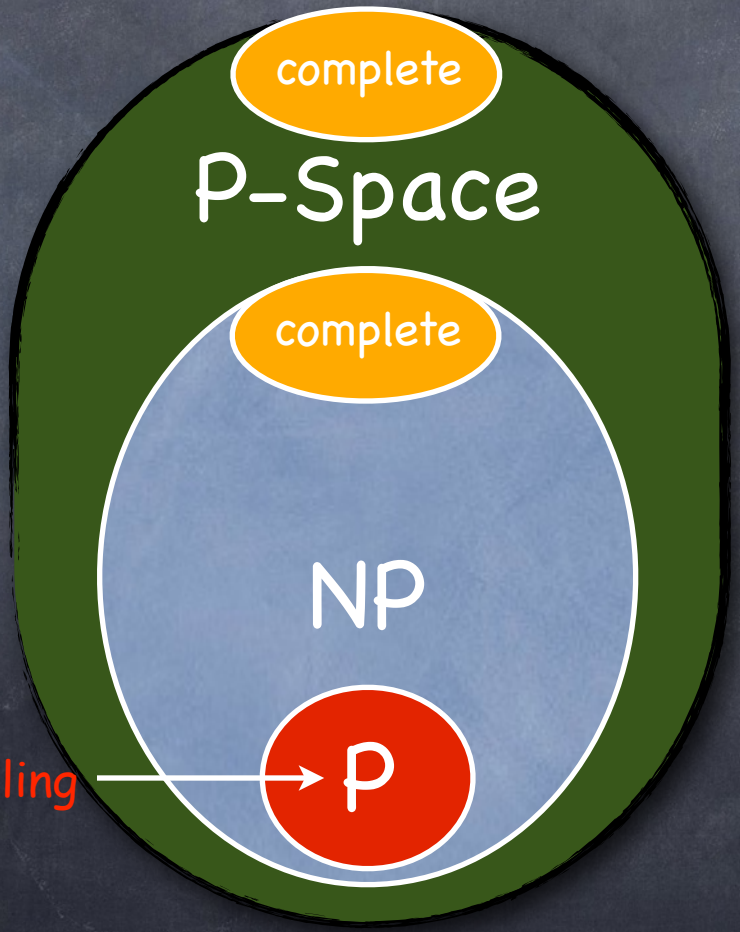


Interval scheduling

Weighted interval scheduling

Bipartite matching

A few low order Complexity Classes



Interval scheduling

Weighted interval scheduling

Bipartite matching

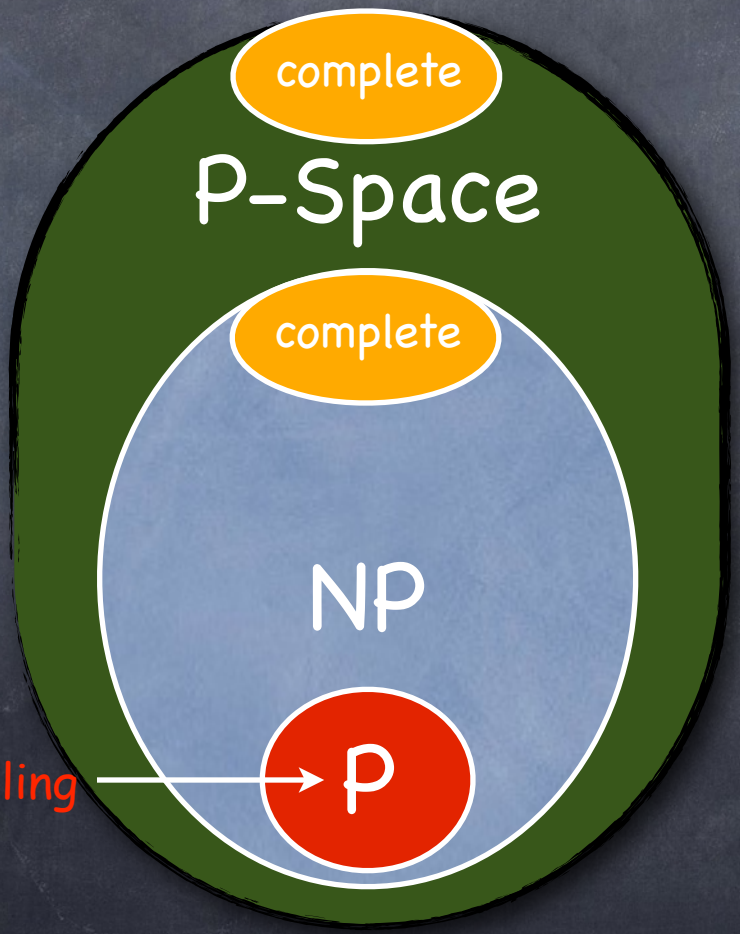
A few low order Complexity Classes

Independent set

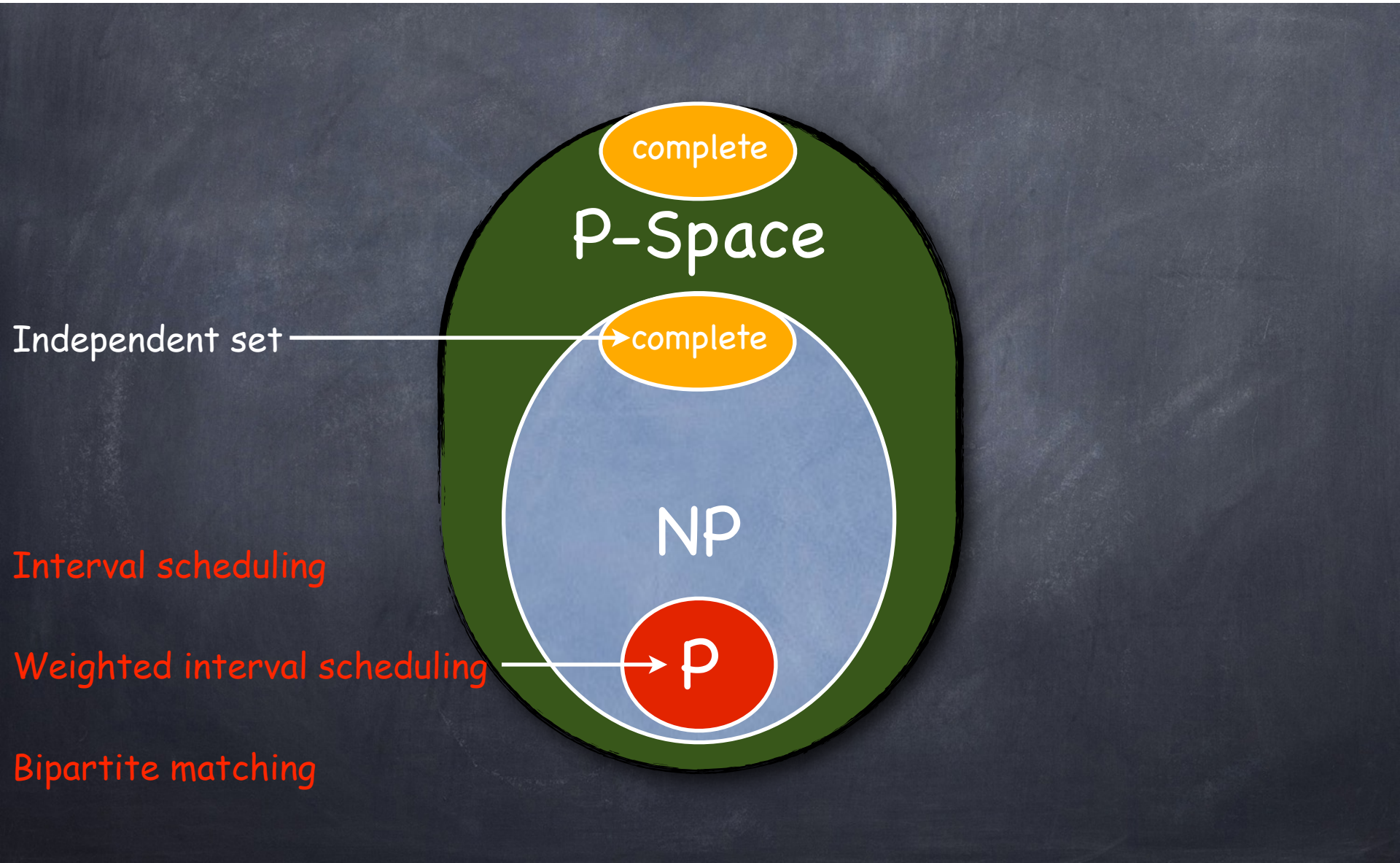
Interval scheduling

Weighted interval scheduling

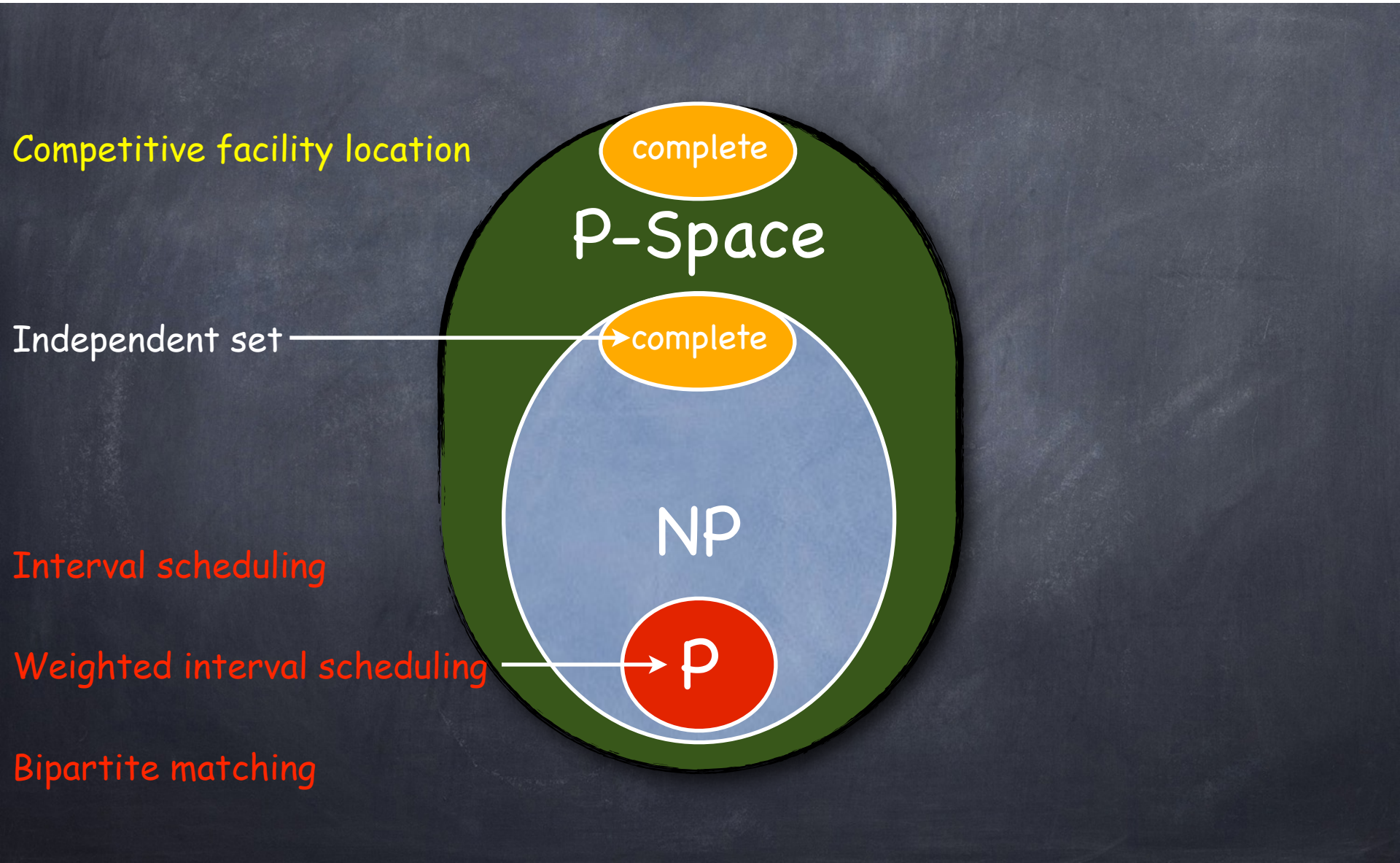
Bipartite matching



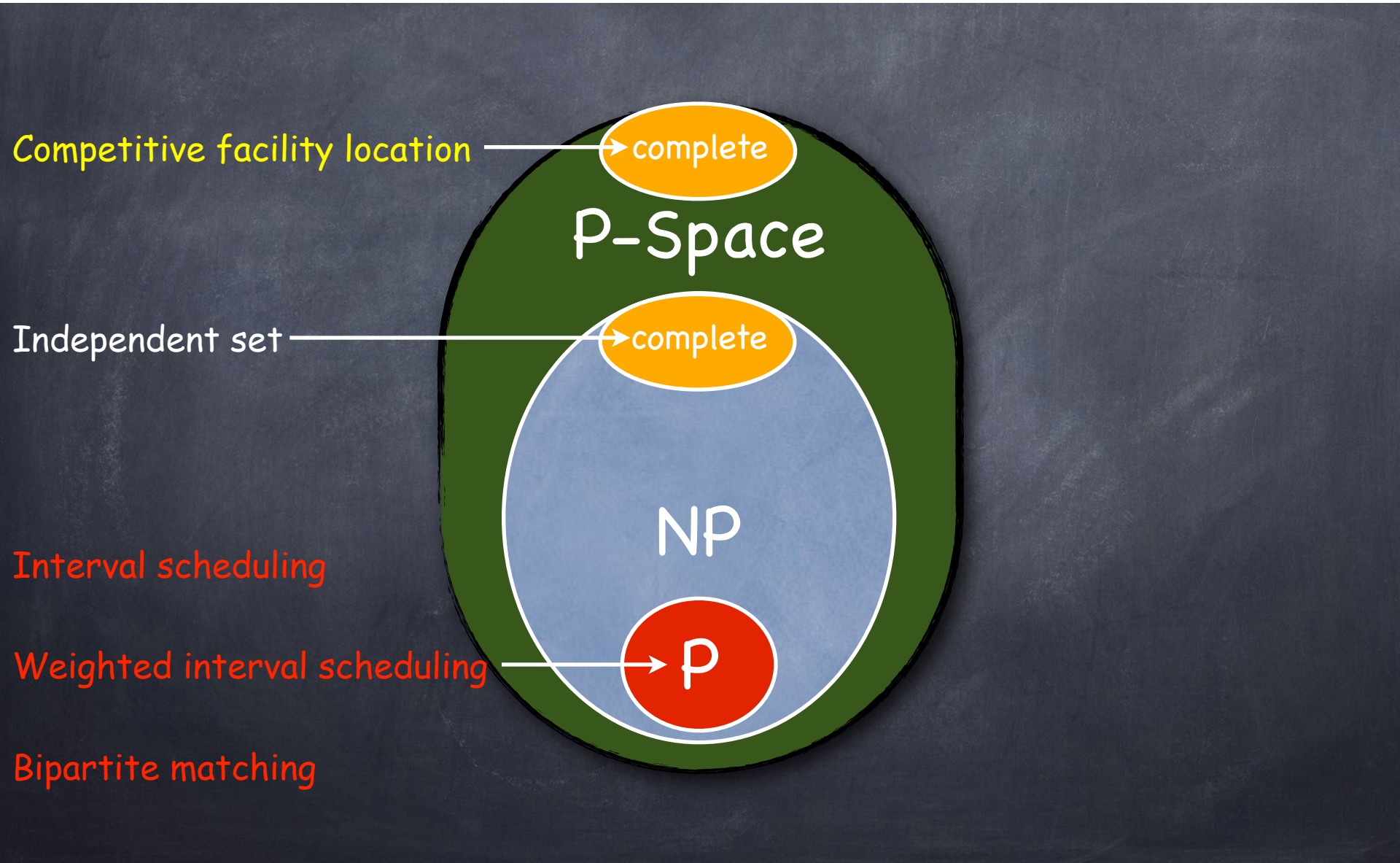
A few low order Complexity Classes



A few low order Complexity Classes



A few low order Complexity Classes



COMP 482: Design and Analysis of Algorithms

Spring 2013

Lecture 2

Prof. Swarat Chaudhuri

Recap: Stable Matching Problem

Goal. Given n men and n women, find a stable matching.

Perfect matching: Everyone is matched monogamously.

Stable matching: perfect matching with no unstable pairs.

favorite ↓ least favorite ↓

	1st	2nd	3rd
Xavier	Amy	Bertha	Clare
Yancey	Bertha	Amy	Clare
Zeus	Amy	Bertha	Clare

Men's Preference Profile

favorite ↓ least favorite ↓

	1st	2nd	3rd
Amy	Yancey	Xavier	Zeus
Bertha	Xavier	Yancey	Zeus
Clare	Xavier	Yancey	Zeus

Women's Preference Profile

Propose-And-Reject Algorithm

Propose-and-reject algorithm. [Gale-Shapley 1962] Intuitive method that guarantees to find a stable matching.

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every woman) {
    Choose such a man m
    w = 1st woman on m's list to whom m has not yet proposed
    if (w is free)
        assign m and w to be engaged
    else if (w prefers m to her fiancé m')
        assign m and w to be engaged, and m' to be free
    else
        w rejects m
}
```

Woman-Pessimality of GS matching

Def. Man m is a valid partner of woman w if there exists some stable matching in which they are matched.

Man-optimal assignment. Each man receives best valid partner.

Woman-pessimal assignment. Each woman receives worst valid partner.

Claim. GS finds woman-pessimal stable matching S*.

We will prove this using contradiction.

Woman Pessimality

Claim. GS finds woman-pessimal stable matching S*.

Pf.

- Suppose A-Z matched in S*, but Z is not worst valid partner for A.
- There exists stable matching S in which A is paired with a man, say Y, whom she likes less than Z.
- Let B be Z's partner in S.
- Z prefers A to B. ← man-optimality
- Thus, A-Z is an unstable in S. ▪

S

Amy-Yancey
Bertha-Zeus
. . .

Proving termination

Claim. Algorithm terminates.

General proof method for termination:

- A certain expression (known as the *progress measure*)
  - Decreases strictly in value in every loop iteration
  - Cannot ever be less than 0

For stable matching problem, progress measure in $(n^2 - m)$, where m is the number of proposals already made.

To show that this is a progress measure, must show that no proposal is revisited.

Termination proofs

```
while (j < N) {
  j = j + 1
  y = x + y
  x = y - x
}
```

Progress measure: (N – j)? What if this is negative?

Proof principle:

- Consider a loop while (B) {P}.
- There is a loop invariant I and a progress measure M such that under assumption (I ∧ B),
  - M is nonnegative
  - P causes the value of M to strictly decrease

Q1: Proving termination

Give a formal termination termination argument for the following algorithm (what does it do, by the way?)

```
int bot = -1;
int top = size;
while (top - bot > 1) {
   int mid = (top + bot)/2;
   if (array[mid] <  seek) bot = mid;
   else top = mid;
}
return top;
```

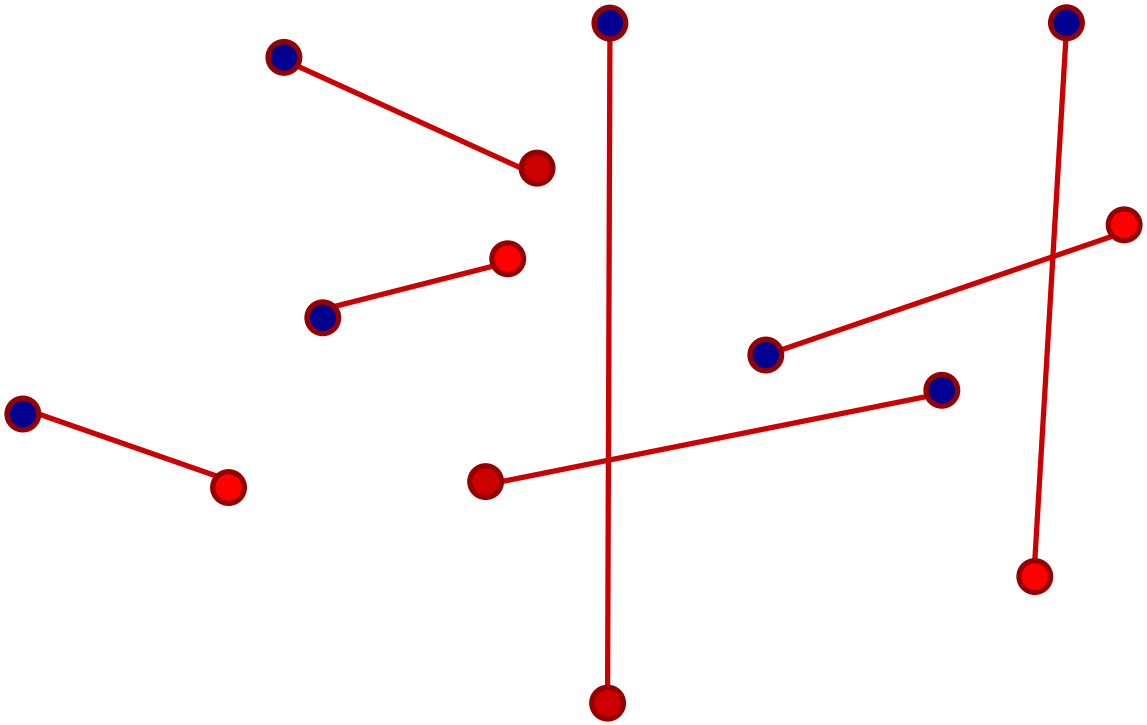
A puzzle for the adventurous: Dijkstra's map problem

Given

- two sets of points in R^2 of equal cardinality

Find

- A one-to-one mapping such that mapping lines do not cross in R^2



Proposed algorithm

```
choose any one-to-one mapping
while (exists crossing)
       uncross a pair of crossing lines
```

Prove that this algorithm terminates.

Q2: A few good men (and women)

Consider a town with n men and n women, where each man/woman has a preference list that ranks all members of the opposite sex.

Of the n men, k are considered "good"; the rest are considered "bad." Similarly, we have k good women and (n – k) bad women.

The preference lists here have the property that everyone would marry a good person rather than a bad person.

Show that in every stable matching, every good man is married to a good woman.

Q3: True or false?

Consider an instance of Stable Matching in which there is a man m and a woman w such that m is ranked first on the preference list of w and w is ranked first on the preference list of m. Then for every stable matching, (m, w) belongs to S.

Q4: Stable matching with indifference

Consider a version of the problem where men and women can be *indifferent* about certain options. Each man and woman ranks the members of the opposite sex as before, but now, there can be *ties* in the ranking.

We will say w prefers m to m' if m' is ranked higher than m in the preference list of w (i.e., m and m' are not tied).

Define a *strong instability* in a perfect matching S to be a pair (m, w) where each of m and w prefers the other over their partner in S. Does there always exist a perfect matching with no strong instability?

Answer: Yes

1. Break the ties in some fashion (for example in lexicographic order).
2. Run the GS algorithm on this new input.
3. Return the output

Does this algorithm

- Terminate in n^2 steps? Yes, because GS does
- Return a perfect matching? Yes, because GS does
- Produce a strong instability? No, because then GS would be generating this instability, which cannot be true!

…stable matching with indifference (continued)

Let a *weak instability* in a perfect matching S be a pair (m.w) such that their partners in in S are m' and w', and one of the following holds:

- m prefers w to w', and w either prefers m to m' or is indifferent between these two choices
- w prefers m to m', and m either prefers w to w' or is indifferent between these two choices

Is there an algorithm that's guaranteed to find a perfect matching with no weak instability? If not, show why not.

Answer: there isn't such an algorithm

Here is an input on which a weak instability must always exist

	1st	2nd
Xavier	Amy, Bertha	
Yancey	…	…

	1st	2nd
Amy	Xavier	Yancey
Bertha	Xavier	Yancey

Deceit: Machiavelli Meets Gale-Shapley

Q. Can there be an incentive for a woman to misrepresent your preference profile?

- Assume you know men's propose-and-reject algorithm will be run.
- Assume that you know the preference profiles of all other participants.

	1st	2nd	3rd
Xavier	A	B	C
Yancey	B	A	C
Zeus	A	B	C

Men's Preference List

	1st	2nd	3rd
Amy	Y	X	Z
Bertha	X	Y	Z
Clare	X	Y	Z

Women's True Preference Profile

	1st	2nd	3rd
Amy	Y	Z	X
Bertha	X	Y	Z
Clare	X	Y	Z

Amy Lies

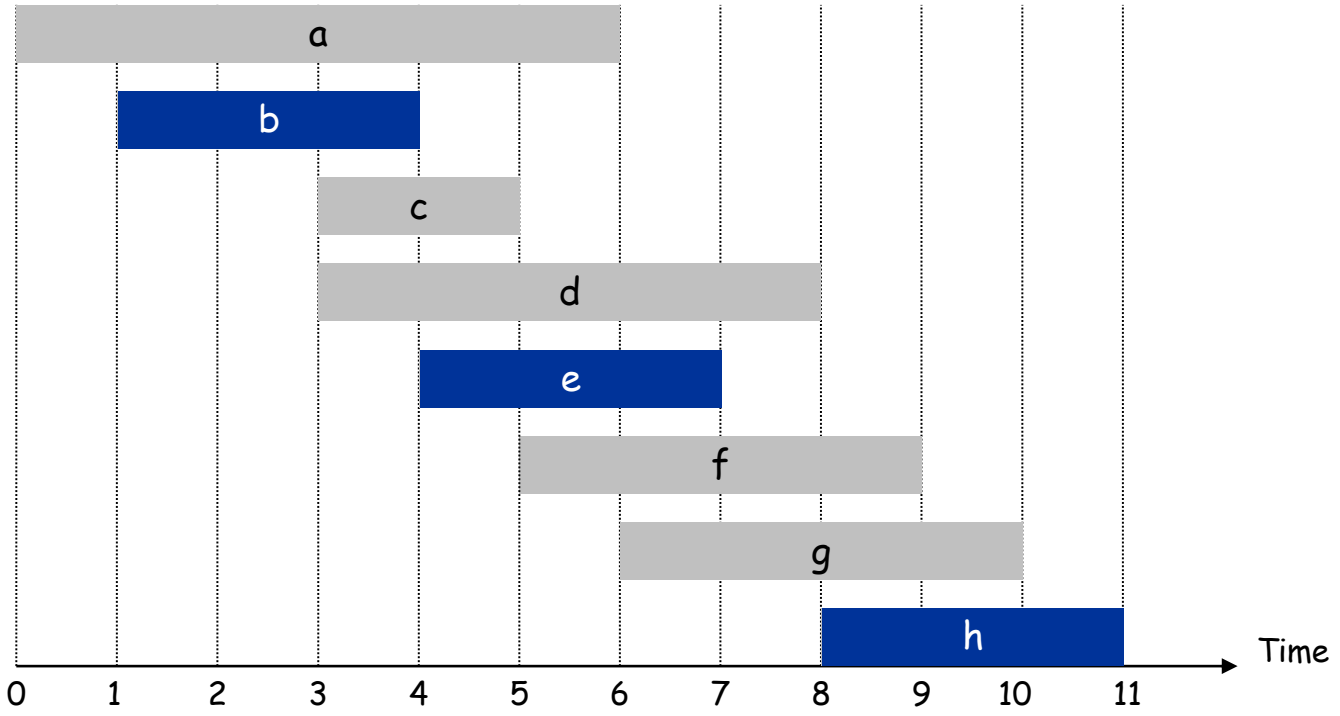
1.2 Five Representative Problems

Interval Scheduling

Input. Set of jobs with start times and finish times.

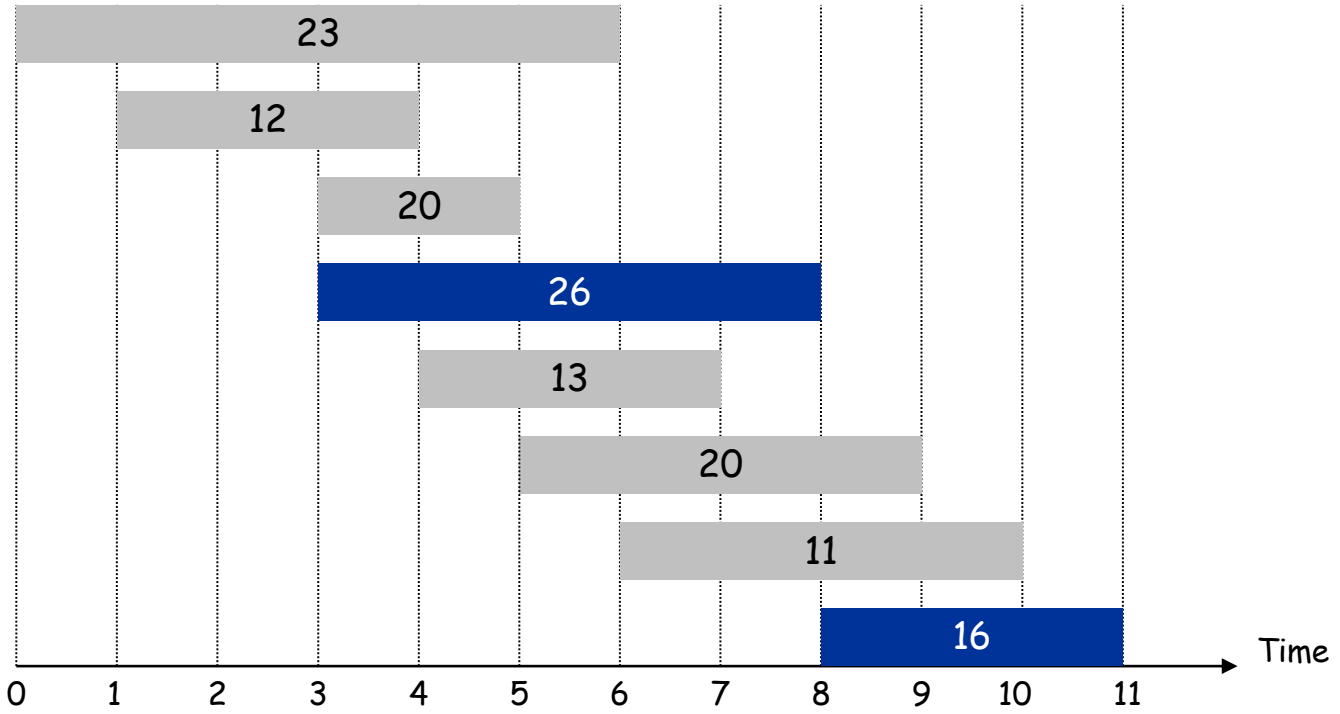
Goal. Find maximum cardinality subset of mutually compatible jobs.

jobs don't overlap



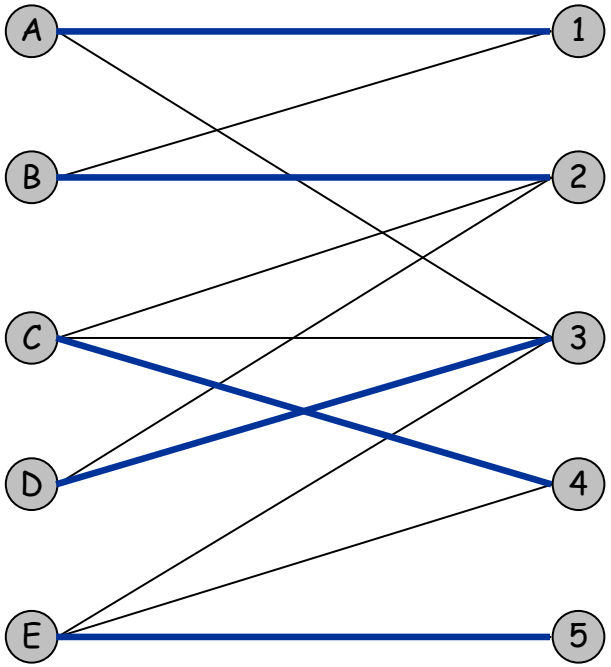
Weighted Interval Scheduling

Input. Set of jobs with start times, finish times, and weights.
Goal. Find maximum weight subset of mutually compatible jobs.



Bipartite Matching

Input. Bipartite graph.
Goal. Find maximum cardinality matching.

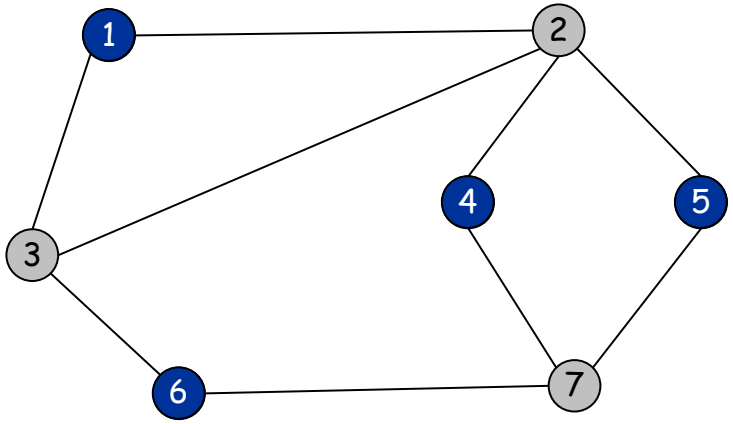


Independent Set

Input. Graph.

Goal. Find maximum cardinality independent set.

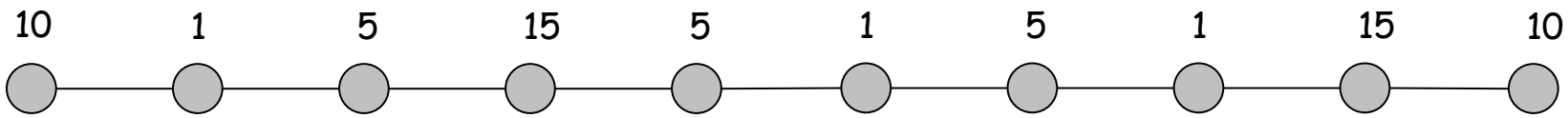
subset of nodes such that no two joined by an edge



Competitive Facility Location

Input. Graph with weight on each node.
Game. Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

Goal. Strategy for Player 2 to select a subset of nodes of weight > B.



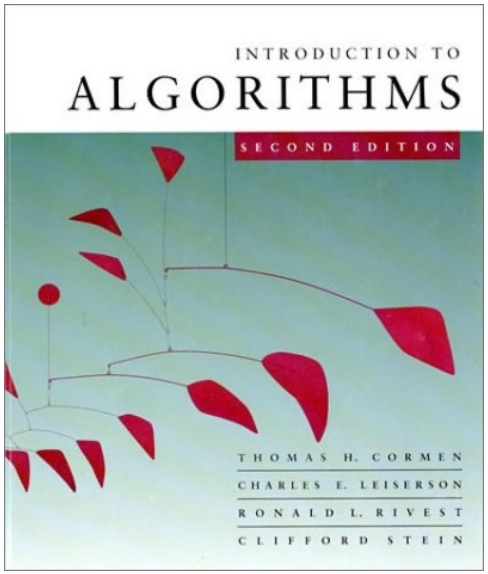
Second player can guarantee 20, but not 25.

Five Representative Problems

Variations on a theme: independent set.

Interval scheduling: n log n greedy algorithm.
Weighted interval scheduling: n log n dynamic programming algorithm.
Bipartite matching: n^k max-flow based algorithm.
Independent set: NP-complete.
Competitive facility location: PSPACE-complete.

Divide-and-Conquer Algorithm: An Idea of Master Theorem



Integer Multiplication

- Let X = $\boxed{A}\boxed{B}$ and Y = $\boxed{C}\boxed{D}$ be the n bit integers, where A, B, C and D are n/2 bit integers
- Simple Method: $XY = (A.2^{n/2}+B)(C.2^{n/2}+D)$
- Running Time Recurrence

$$T(n) < 4T(n/2) + 100n$$

How do we solve it?

Substitution Method

The most general method:

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.
3. ***Solve*** for constants.

Example: $T(n) = 4T(n/2) + 100n$

- [Assume that $T(1) = \Theta(1)$.]
- Guess $O(n^3)$. (Prove O and Ω separately.)
- Assume that $T(k) \le ck^3$ for $k < n$.
- Prove $T(n) \le cn^3$ by induction.

Example of Substitution

$$
\begin{aligned}
T(n) &= 4T(n/2) + 100n \\
&\le 4c(n/2)^3 + 100n \\
&= (c/2)n^3 + 100n \\
&= cn^3 - ((c/2)n^3 - 100n) \quad \leftarrow \textit{Desired} - \textit{Residual} \\
&\le cn^3 \quad \leftarrow \textit{Desired}
\end{aligned}
$$

whenever $(c/2)n^3 - 100n \ge 0$, for example, if $c \ge 200$ and $n \ge 1$.

Residual

Example (continued)

- We must also handle the initial conditions, that is, ground the induction with base cases.
- ***Base:*** $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant.
- For $1 \le n < n_0$, we have "$\Theta(1)$" $\le cn^3$, if we pick c big enough.

This bound is not tight!

A Tighter Upper Bound?

We shall prove that $T(n) = O(n^2)$.

Assume that $T(k) \le ck^2$ for $k < n$:

$$\begin{aligned} T(n) &= 4T(n/2) + 100n \\ &\le \quad cn^2 + 100n \\ &\le cn^2 \end{aligned}$$

for ***no*** choice of $c > 0$. Lose!

A Tighter Upper Bound!

IDEA: Strengthen the inductive hypothesis.

- ***Subtract*** a low-order term.

Inductive Hypothesis: $T(k) \le c_1 k^2 - c_2 k$ for $k < n$.

$$
\begin{aligned}
T(n) &= 4T(n/2) + 100n \\
&\le 4(c_1(n/2)^2 - c_2(n/2)) + 100n \\
&= c_1 n^2 - 2c_2 n + 100n \\
&= c_1 n^2 - c_2 n - (c_2 n - 100n) \\
&\le c_1 n^2 - c_2 n \quad \text{if } c_2 > 100.
\end{aligned}
$$

Pick c_1 big enough to handle the initial conditions.

Recursion-Tree Method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion tree method is good for generating guesses for the substitution method.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (…).
- The recursion-tree method promotes intuition (common sense, instinct, perception).

Example of Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

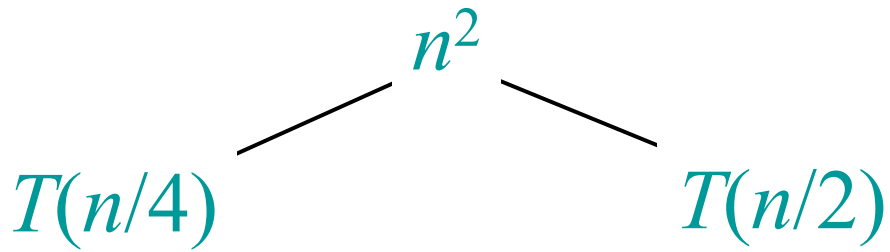
Example of Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$T(n)$

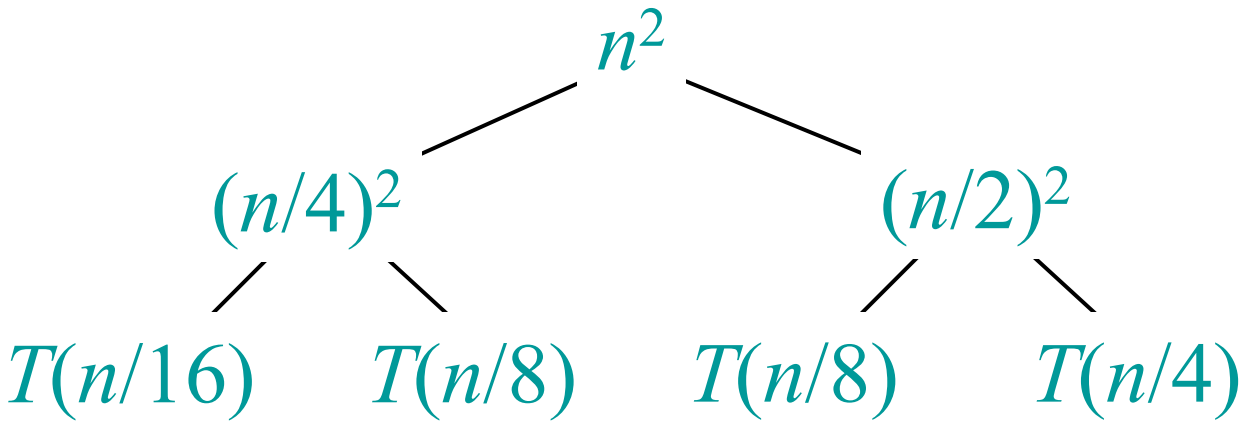
Example of Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



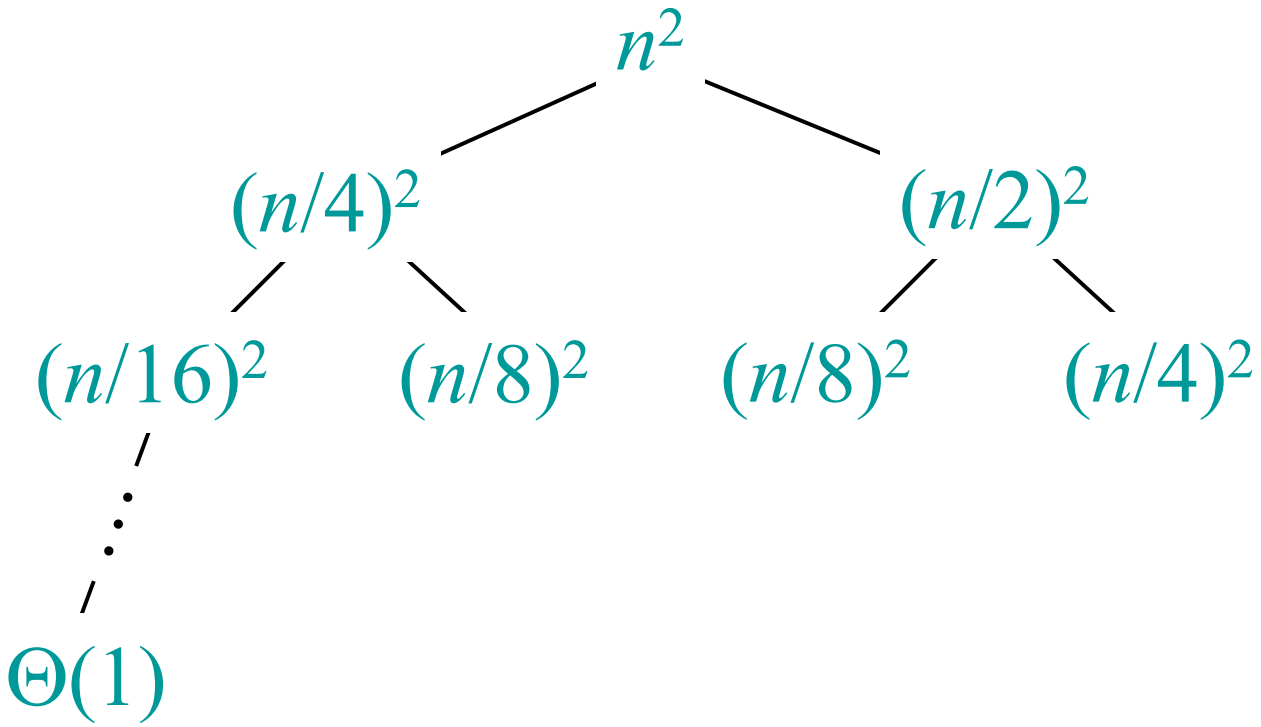
Example of Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



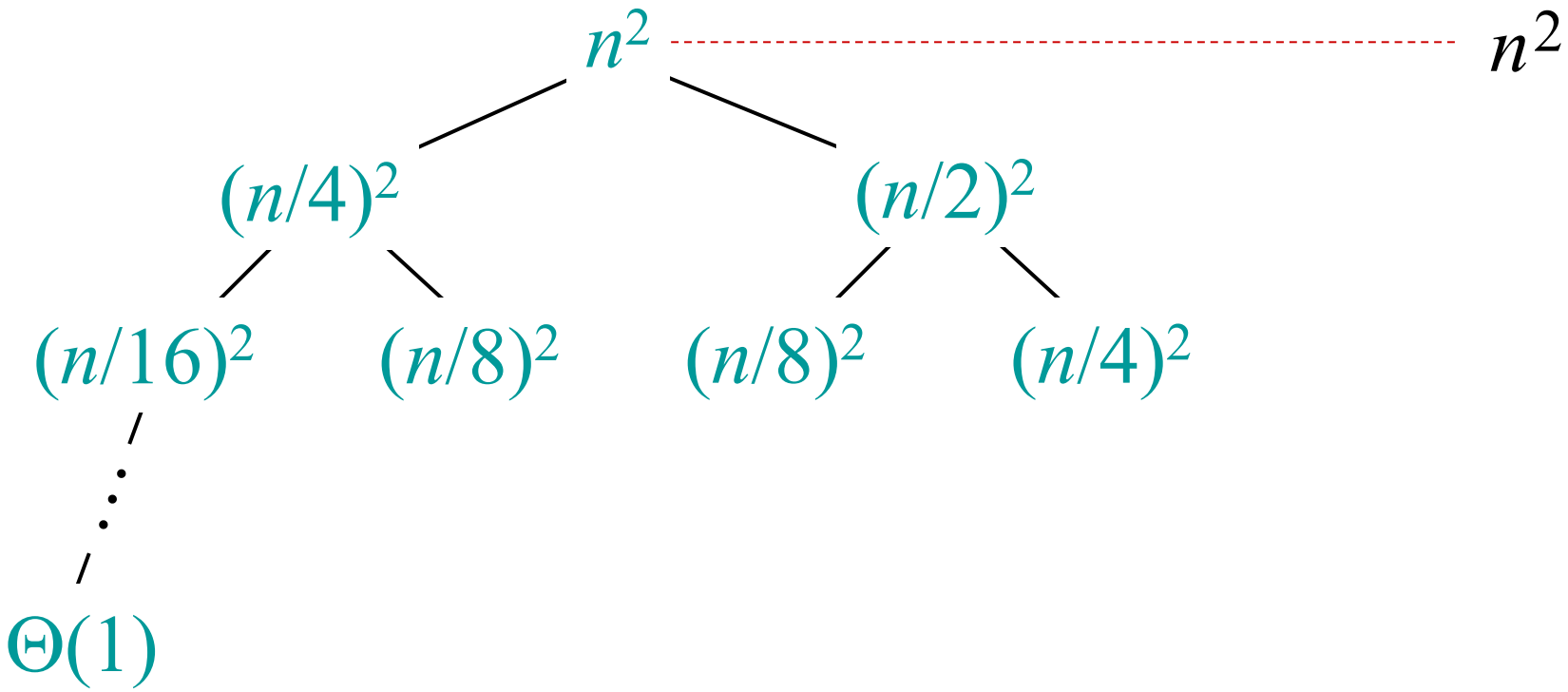
Example of Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



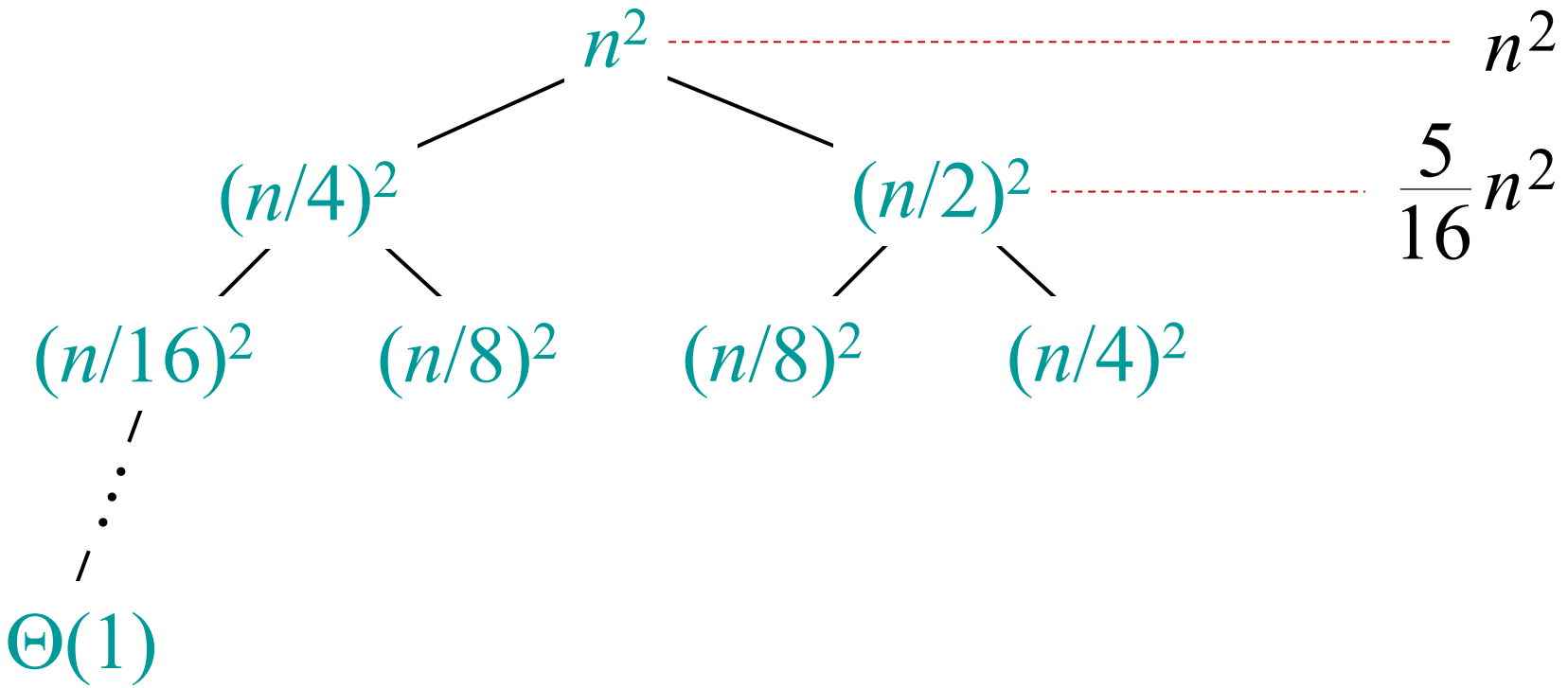
Example of Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



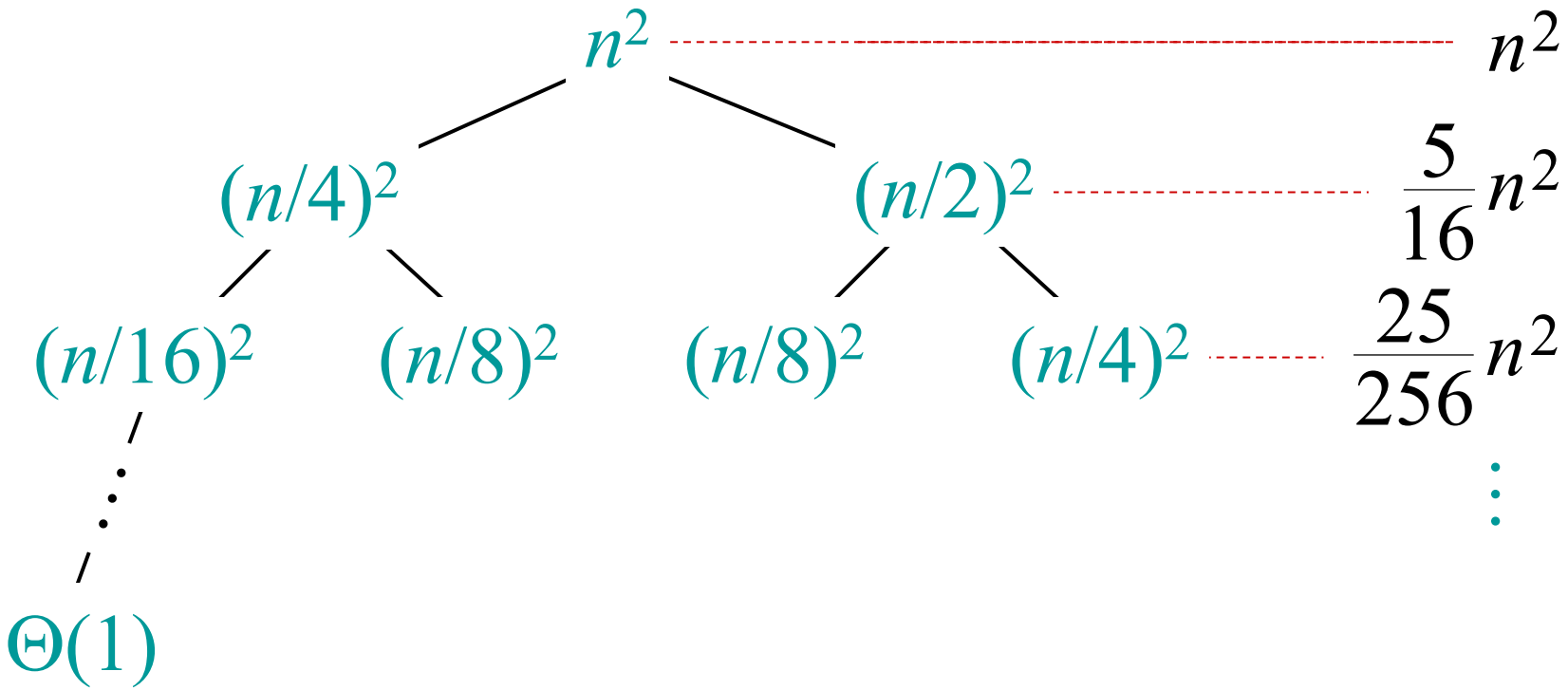
Example of Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



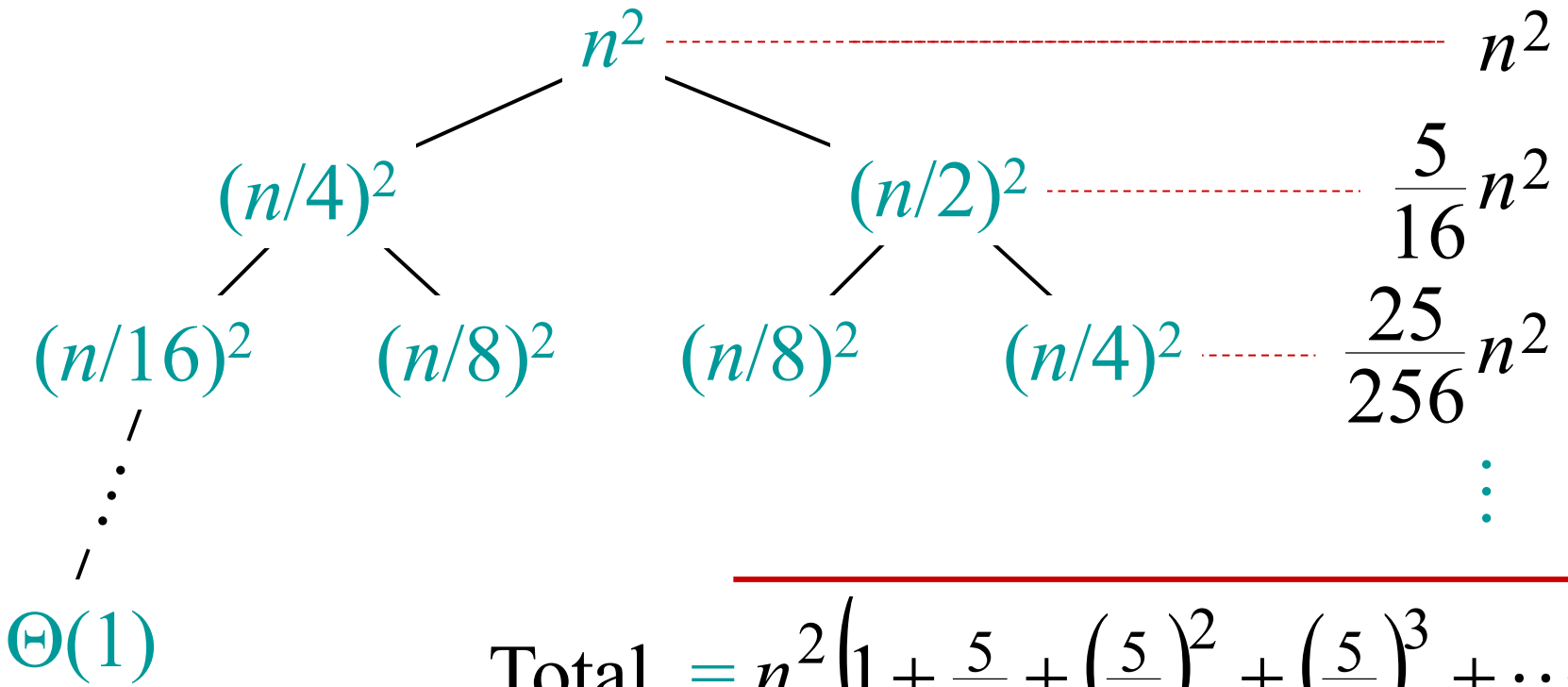
Example of Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

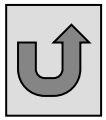
$$= \Theta(n^2) \quad \textit{Geometric Series}$$

Appendix: Geometric Series

$$1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x} \quad \text{for } x \neq 1$$

$$1 + x + x^2 + \cdots = \frac{1}{1 - x} \quad \text{for } |x| < 1$$

Return to last slide viewed.



The Master Method

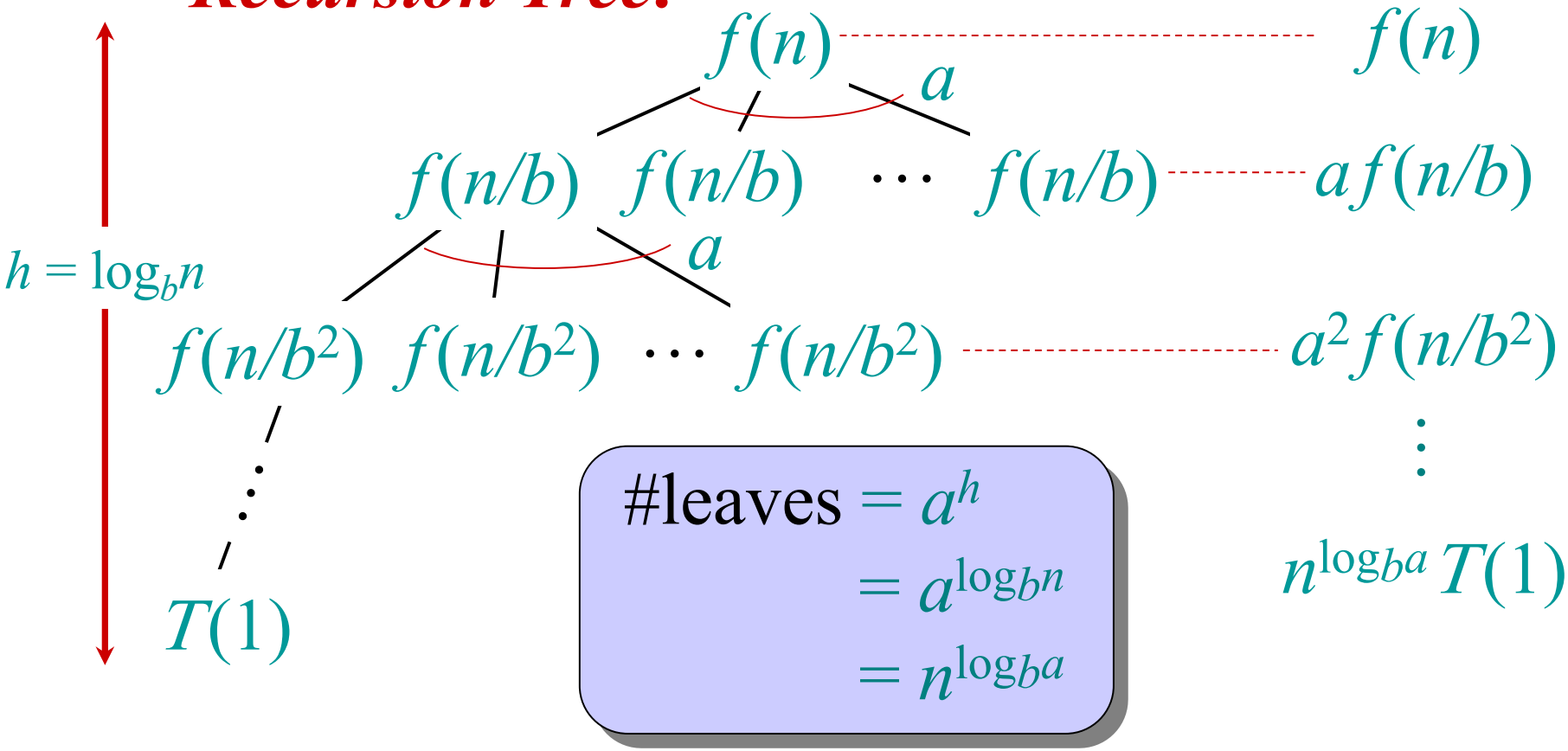
The master method applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\ ,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

Idea of Master Theorem

Recursion Tree:



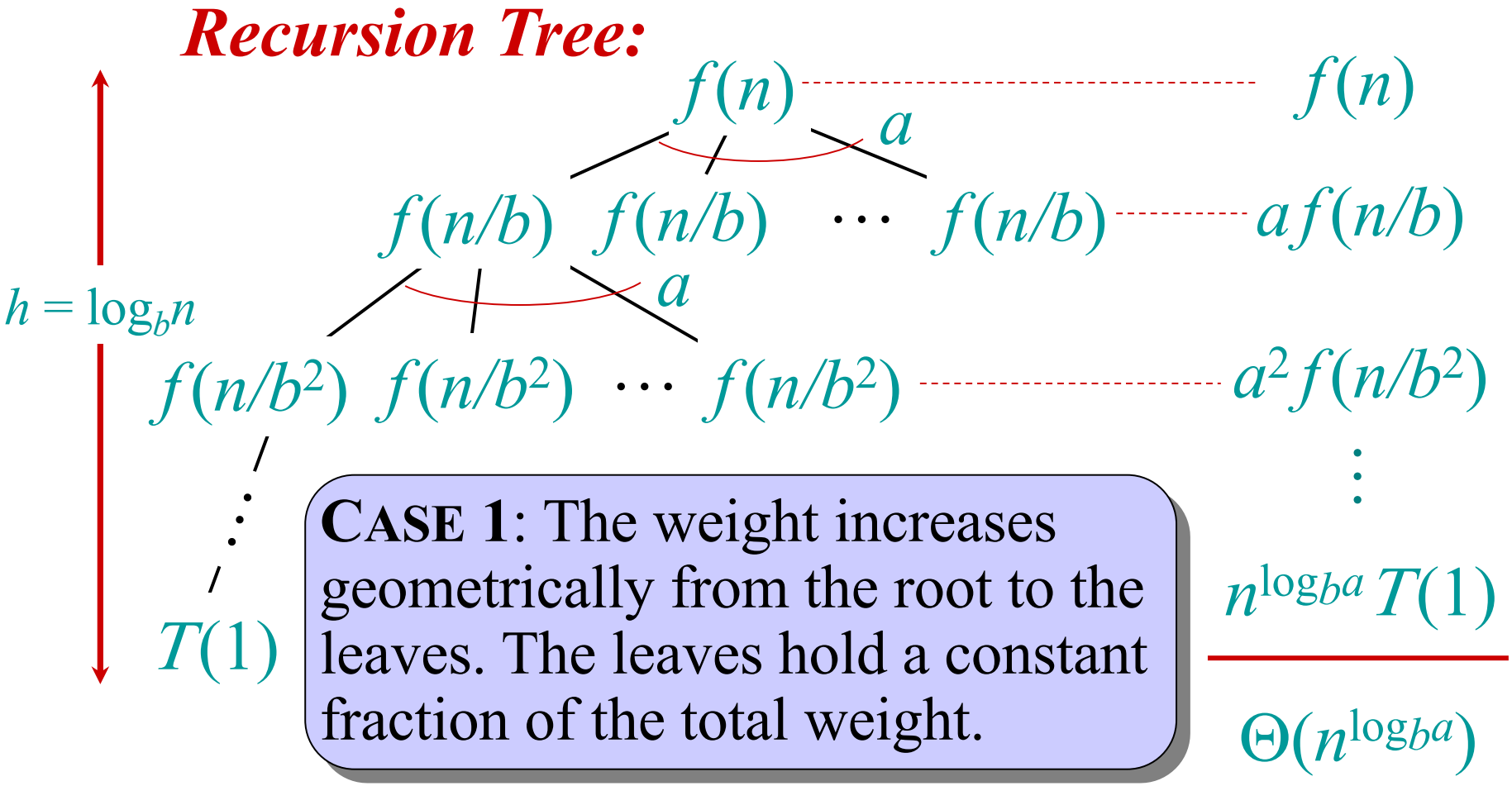
Three Common Cases

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

   Solution: $T(n) = \Theta(n^{\log_b a})$.

Idea of Master Theorem



Three Common Cases

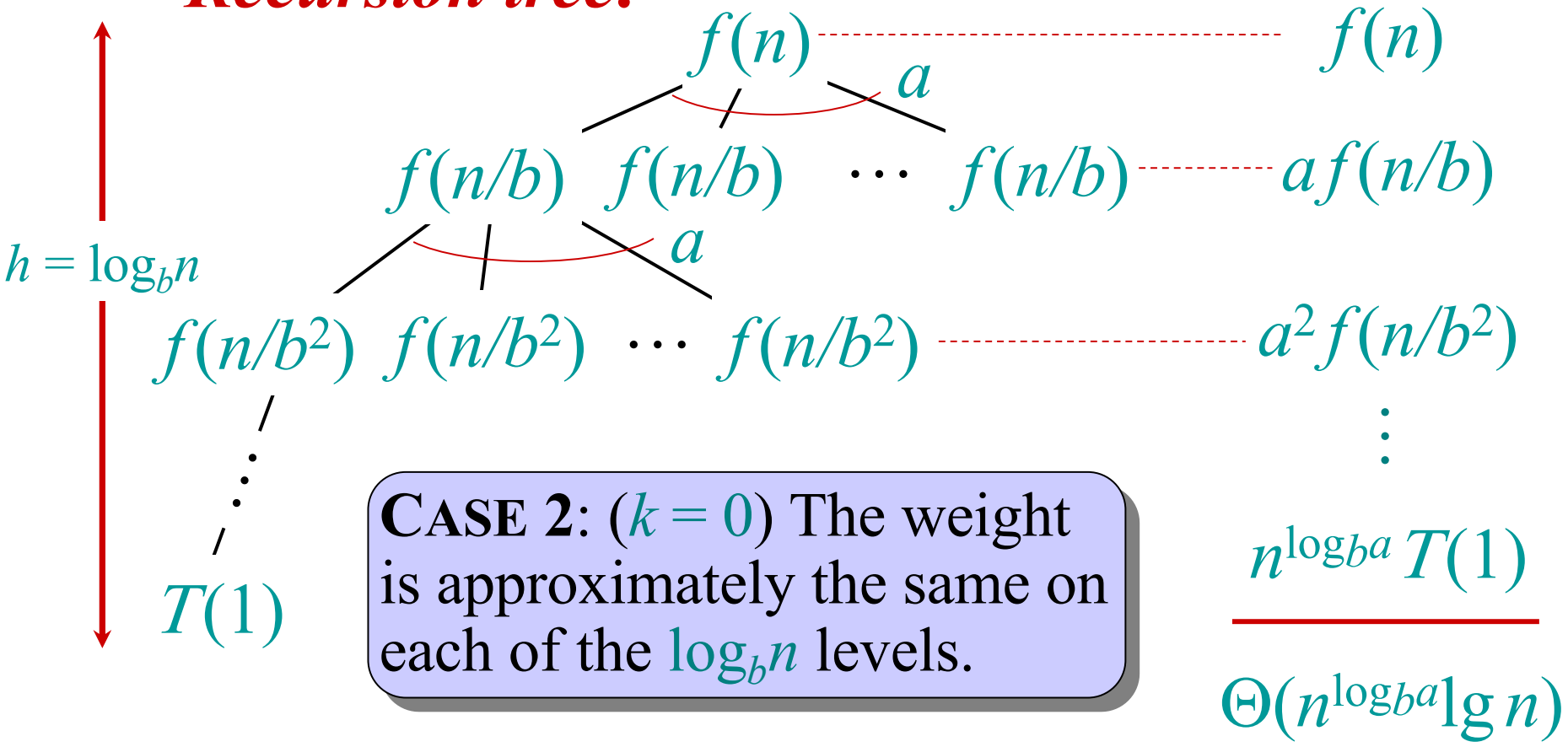
Compare $f(n)$ with $n^{\log_b a}$:

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Idea of Master Theorem

Recursion tree:



CASE 2: ($k = 0$) The weight is approximately the same on each of the $\log_b n$ levels.

Three Common Cases (cont.)

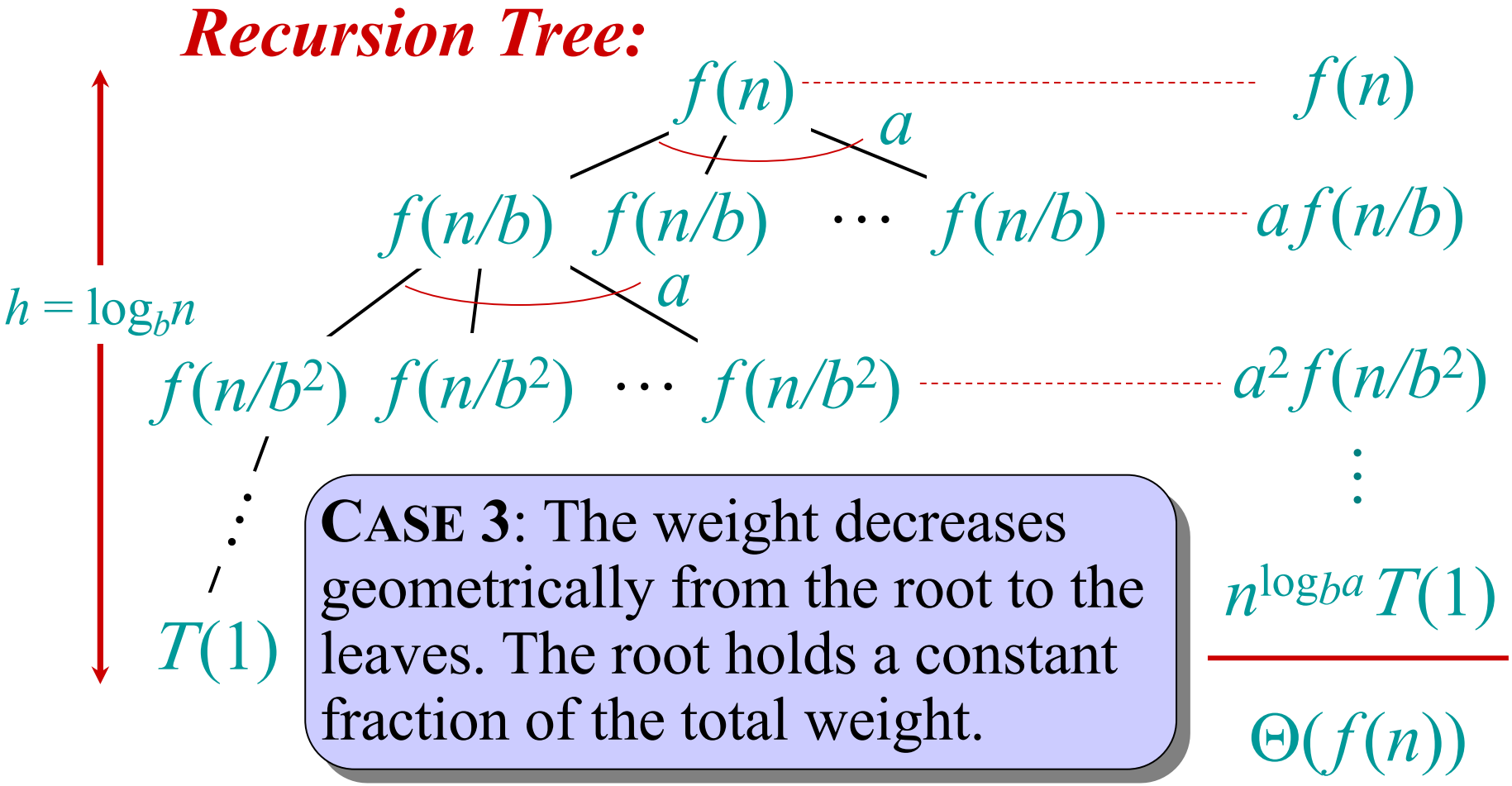
Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

   and $f(n)$ satisfies the ***regularity condition*** that $a f(n/b) \leq c f(n)$ for some constant $c < 1$.

   Solution: $T(n) = \Theta(f(n))$.

Idea of Master Theorem



Examples

Ex. $T(n) = 4T(n/2) + n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n.$

CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

$\therefore\ T(n) = \Theta(n^2).$

Ex. $T(n) = 4T(n/2) + n^2$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2.$

CASE 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.

$\therefore\ T(n) = \Theta(n^2 \lg n).$

Examples

Ex. $T(n) = 4T(n/2) + n^3$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^3.$

CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(cn/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore\ T(n) = \Theta(n^3)$.

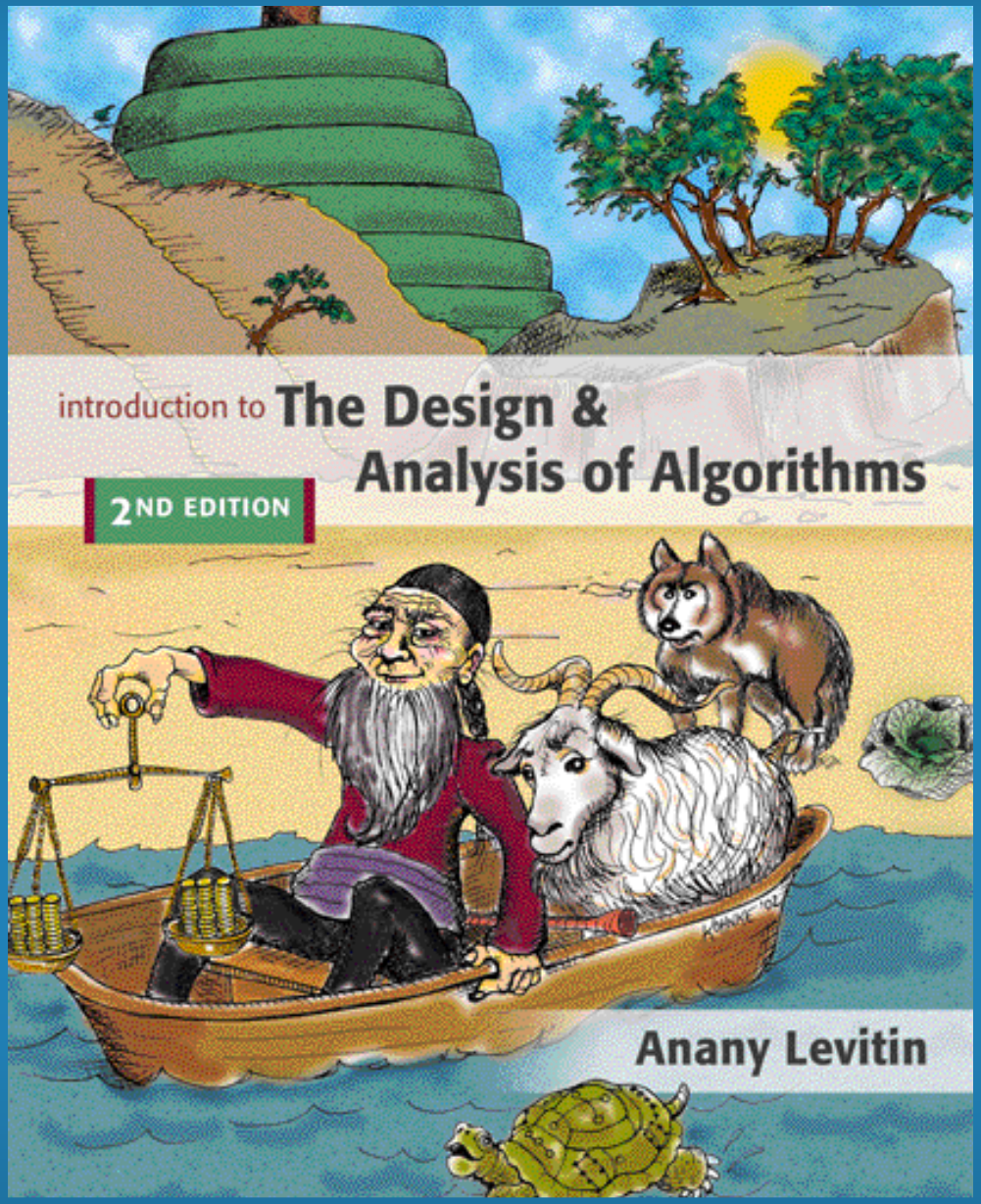
Ex. $T(n) = 4T(n/2) + n^2/\lg n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2/\lg n.$

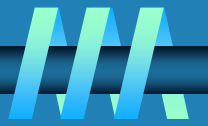
Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^\varepsilon = \omega(\lg n)$.

Chapter 4

Divide-and-Conquer

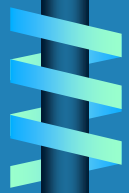


Divide-and-Conquer

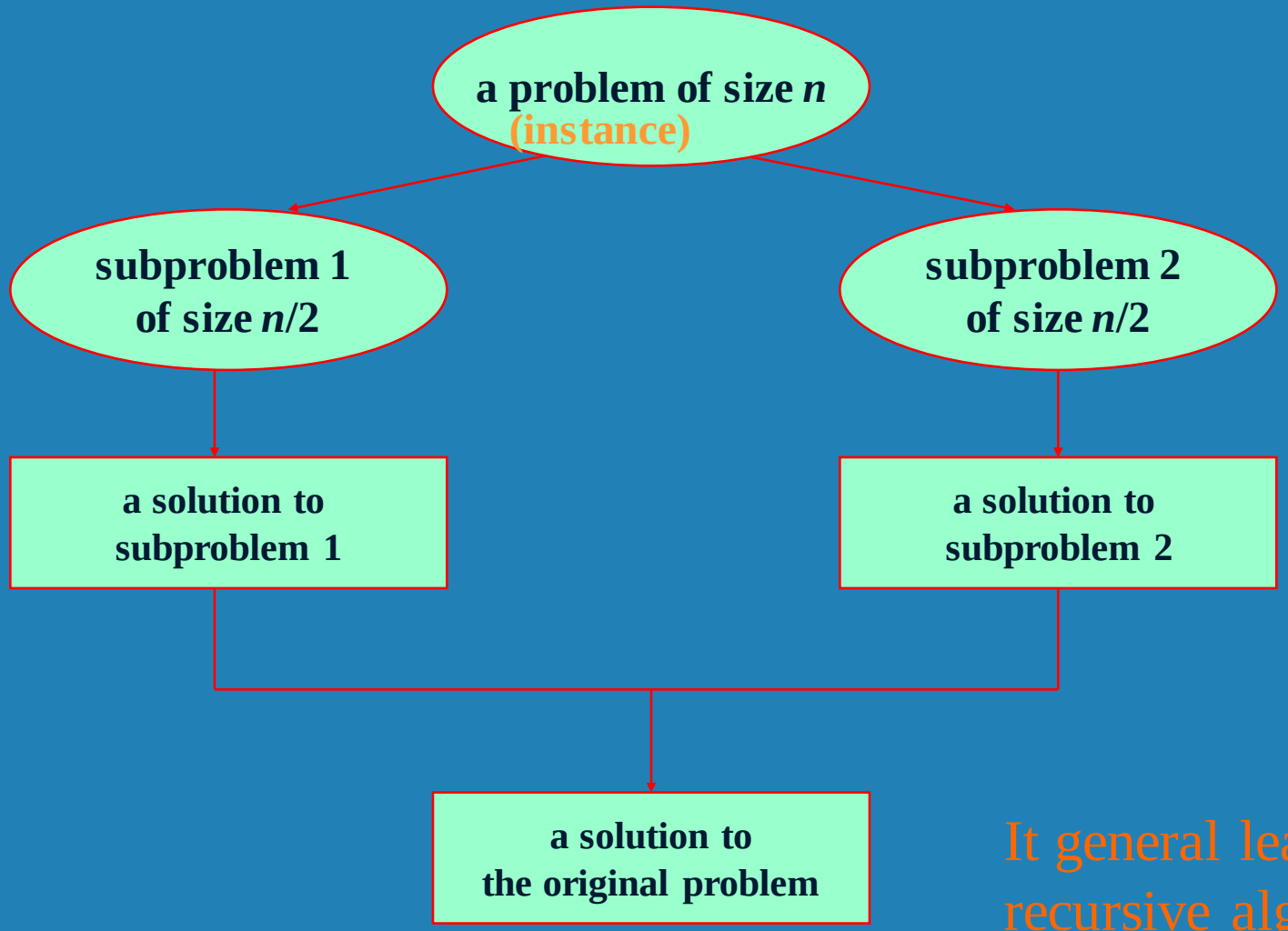


The most-well known algorithm design strategy:

1. **Divide instance of problem into two or more smaller instances**
2. **Solve smaller instances recursively**
3. **Obtain solution to original (larger) instance by combining these solutions**

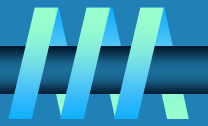


Divide-and-Conquer Technique (cont.)



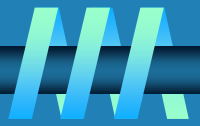
It general leads to a recursive algorithm!

Divide-and-Conquer Examples



- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search (?)
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair and convex-hull algorithms

General Divide-and-Conquer Recurrence



$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

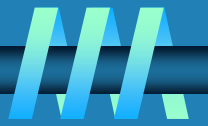
Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ $\Theta(n^2)$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ $\Theta(n^2 \log n)$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ $\Theta(n^3)$

Mergesort



- Split array A[0..*n*-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

Pseudocode of Merge

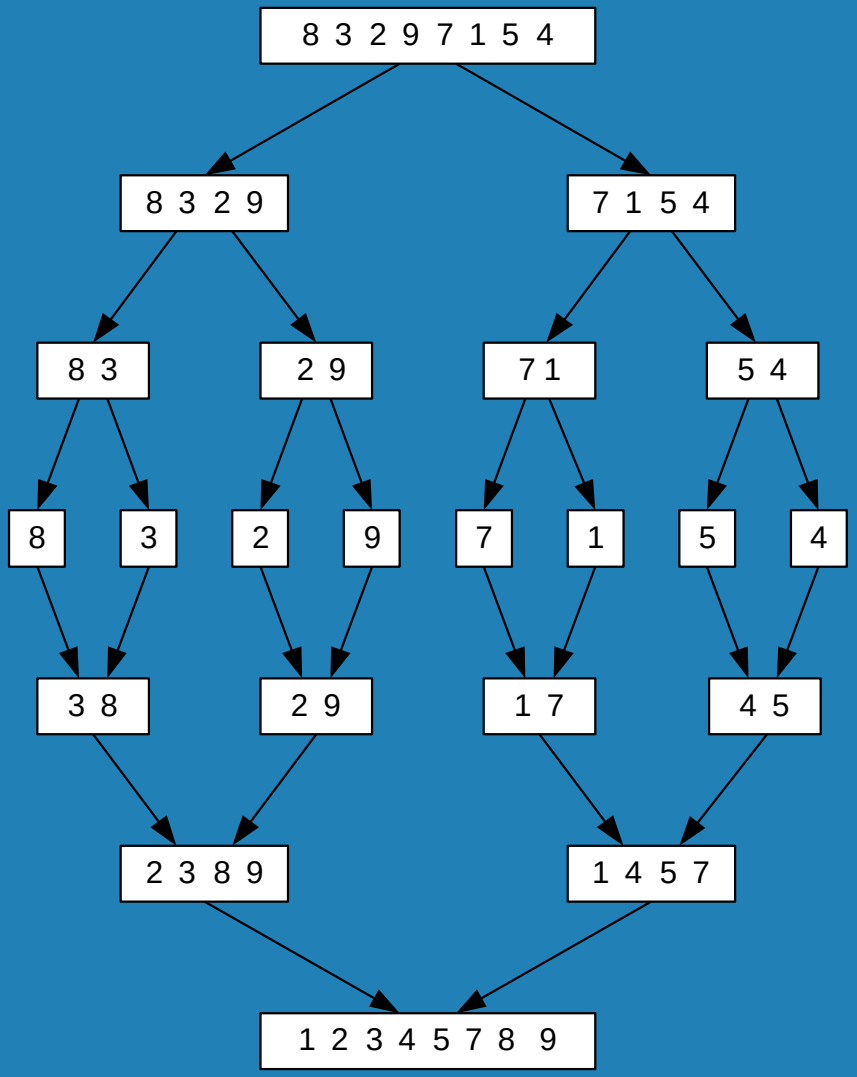
ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C

$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i+1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

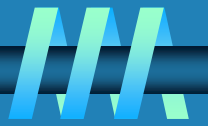
Time complexity: $\Theta(p+q) = \Theta(n)$ comparisons

Mergesort Example



The non-recursive version of Mergesort starts from merging single elements into sorted pairs.

Analysis of Mergesort



- All cases have same efficiency: $\Theta(n \log n)$

$$T(n) = 2T(n/2) + \Theta(n),\ T(1) = 0$$

- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

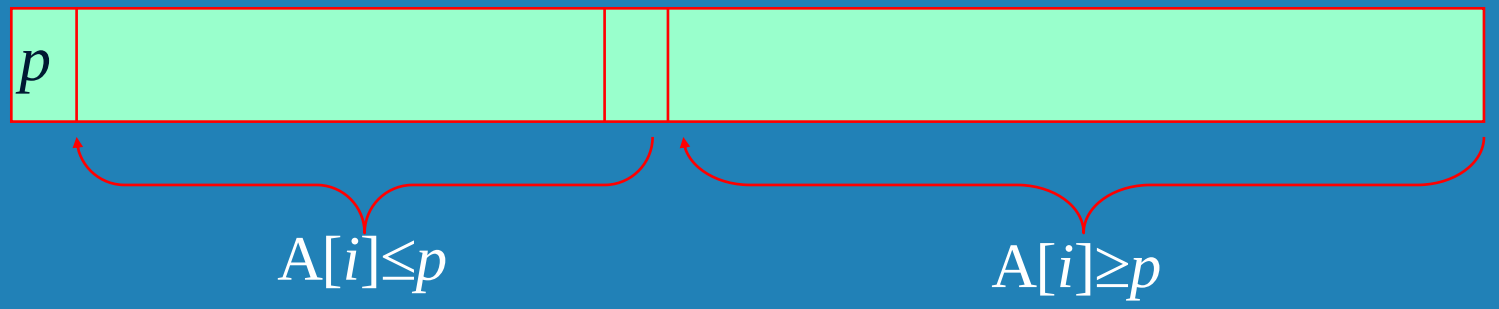
$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

- Space requirement: $\Theta(n)$ (not in-place)

- Can be implemented without recursion (bottom-up)

Quicksort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Partitioning Algorithm

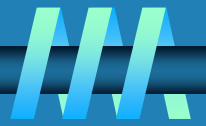
Algorithm *Partition*$(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] < p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

or $i > r$

or $j = l$

Time complexity: $\Theta(r-l)$ comparisons

Quicksort Example



5 3 1 9 8 2 4 7

2 3 1 4 5 8 9 7

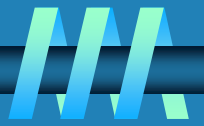
1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

Analysis of Quicksort



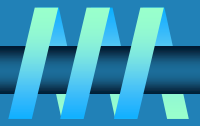
- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$ $\quad T(n) = T(n-1) + \Theta(n)$
- Average case: random arrays — $\Theta(n \log n)$

- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion

  These combine to 20-25% improvement

- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Binary Search



Very efficient algorithm for searching in sorted array:

K

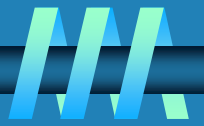
vs

A[0] . . . A[m] . . . A[n-1]

If K = A[m], stop (successful search); otherwise, continue searching by the same method in A[0..m-1] if K < A[m] and in A[m+1..n-1] if K > A[m]

$l \leftarrow 0$; $r \leftarrow n-1$
while $l \leq r$ **do**
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$ **return** m
    else if $K < A[m]$ $r \leftarrow m-1$
    else $l \leftarrow m+1$
return -1

Analysis of Binary Search



- Time efficiency
  - worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$, $C_w(1) = 1$
    solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

    This is VERY fast: e.g., $C_w(10^6) = 20$

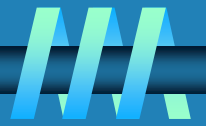
- Optimal for searching a sorted array

- Limitations: must be a sorted array (not linked list)

- Bad (degenerate) example of divide-and-conquer because only one of the sub-instances is solved

- Has a continuous counterpart called *bisection method* for solving equations in one unknown $f(x) = 0$ (see Sec. 12.4)

Binary Tree Algorithms



Binary tree is a divide-and-conquer ready structure!

Ex. 1: Classic traversals (preorder, inorder, postorder)

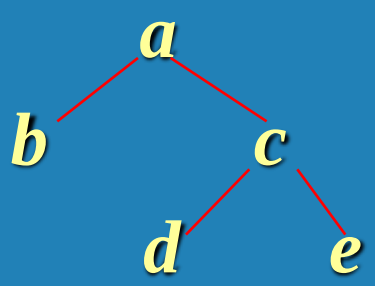
Algorithm *Inorder*(*T*)

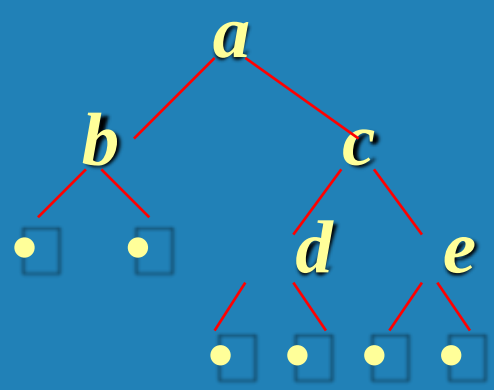
if $T \neq \varnothing$

  Inorder(T_{left})

  print(root of *T*)

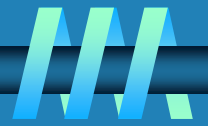
  Inorder(T_{right})



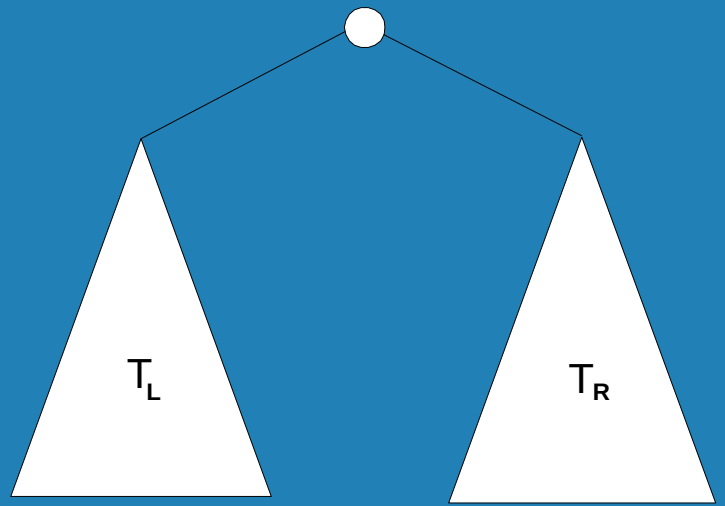


Efficiency: $\Theta(n)$. Why? Each node is visited/printed once.

Binary Tree Algorithms (cont.)



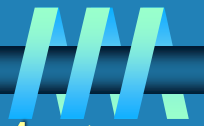
Ex. 2: Computing the height of a binary tree



$h(T) = \max\{h(T_L), h(T_R)\} + 1$ if $T \neq \varnothing$ and $h(\varnothing) = -1$

Efficiency: $\Theta(n)$. Why?

Multiplication of Large Integers



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

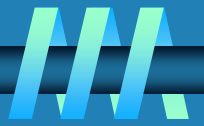
A = 12345678901357986429 B = 87654321284820912836

The grade-school algorithm:

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

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 (4-1) extra add/sub.**

Recurrence for the number of multiplications M(n):

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

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

What if we count both multiplications and additions?

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

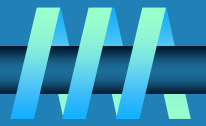
21*40 = (2*10 + 1) * (4*10 + 0)

= (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.

Conventional Matrix Multiplication



- **Brute-force algorithm**

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

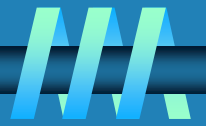
$$= \begin{pmatrix} a_{00} * b_{00} + a_{01} * b_{10} & a_{00} * b_{01} + a_{01} * b_{11} \\ a_{10} * b_{00} + a_{11} * b_{10} & a_{10} * b_{01} + a_{11} * b_{11} \end{pmatrix}$$

8 multiplications

4 additions

Efficiency class in general: $\Theta(n^3)$

Strassen's Matrix Multiplication



- **Strassen's algorithm for two 2x2 matrices (1969):**

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

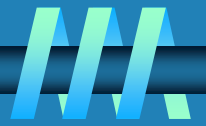
$$= \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

- $m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
- $m_2 = (a_{10} + a_{11}) * b_{00}$
- $m_3 = a_{00} * (b_{01} - b_{11})$
- $m_4 = a_{11} * (b_{10} - b_{00})$
- $m_5 = (a_{00} + a_{01}) * b_{11}$
- $m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
- $m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

7 multiplications

18 additions

Strassen's Matrix Multiplication

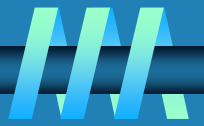


Strassen observed [1969] that the product of two matrices can be computed in general as follows:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

Formulas for Strassen's Algorithm



$$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$$

$$M_2 = (A_{10} + A_{11}) * B_{00}$$

$$M_3 = A_{00} * (B_{01} - B_{11})$$

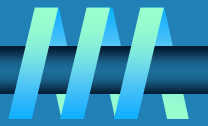
$$M_4 = A_{11} * (B_{10} - B_{00})$$

$$M_5 = (A_{00} + A_{01}) * B_{11}$$

$$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$$

$$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$$

Analysis of Strassen's Algorithm



If n is not a power of 2, matrices can be padded with zeros.

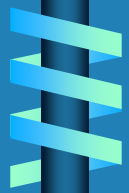
What if we count both multiplications and additions?

Number of multiplications:

$$M(n) = 7M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$ **vs.** n^3 **of brute-force alg.**

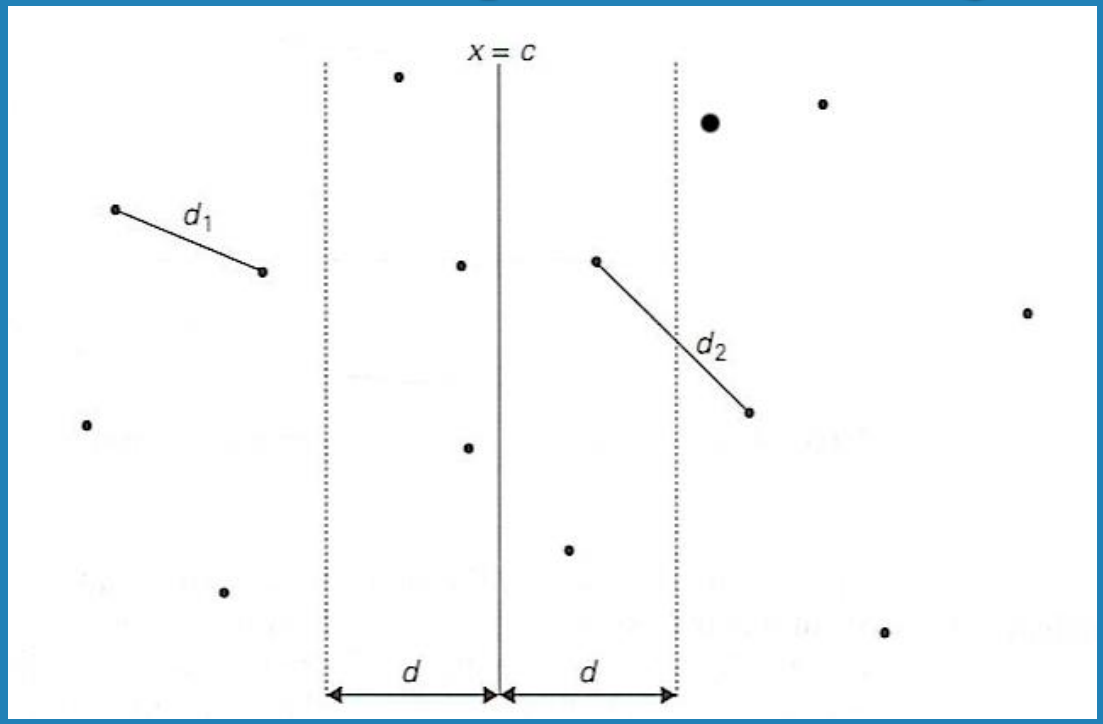
Algorithms with better asymptotic efficiency are known but they are even more complex and not used in practice.



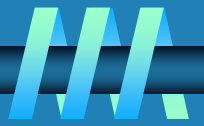
Closest-Pair Problem by Divide-and-Conquer

Step 0 Sort the points by x (list one) and then by y (list two).

Step 1 Divide the points given into two subsets S_1 and S_2 by a vertical line $x = c$ so that half the points lie to the left or on the line and half the points lie to the right or on the line.



Closest Pair by Divide-and-Conquer (cont.)



Step 2 Find recursively the closest pairs for the left and right subsets.

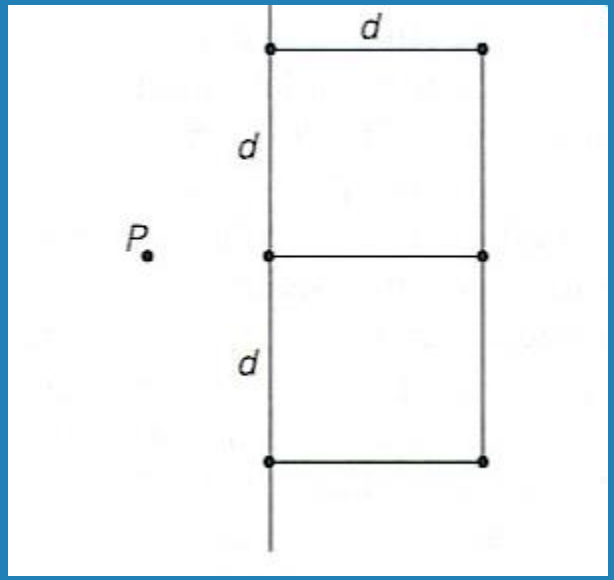
Step 3 Set $d = \min\{d_1, d_2\}$

We can limit our attention to the points in the symmetric vertical strip of width $2d$ as possible closest pair. Let C_1 and C_2 be the subsets of points in the left subset S_1 and of the right subset S_2, respectively, that lie in this vertical strip. The points in C_1 and C_2 are stored in increasing order of their y coordinates, taken from the second list.

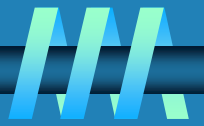
Step 4 For every point $P(x,y)$ in C_1, we inspect points in C_2 that may be closer to P than d. There can be no more than **6 such points** (because $d \le d_2$)!

Closest Pair by Divide-and-Conquer: Worst Case

The worst case scenario is depicted below:



Efficiency of the Closest-Pair Algorithm



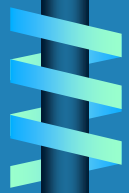
Running time of the algorithm (without sorting) is:

$$T(n) = 2T(n/2) + M(n), \text{ where } M(n) \in \Theta(n)$$

By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)

$$T(n) \in \Theta(n \log n)$$

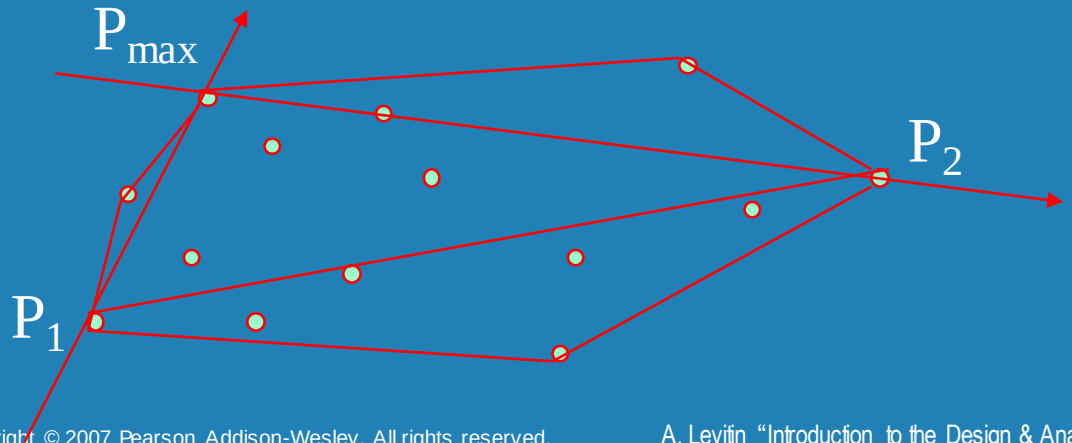
So the total time is $\Theta(n \log n)$.



Quickhull Algorithm

***Convex hull*: smallest convex set that includes given points. An O(n^3) bruteforce time is given in Levitin, Ch 3.**

- **Assume points are sorted by *x*-coordinate values**
- **Identify *extreme points* P_1 and P_2 (leftmost and rightmost)**
- **Compute *upper hull* recursively:**
  - **find point P_{max} that is farthest away from line P_1P_2**
  - **compute the upper hull of the points to the left of line P_1P_{max}**
  - **compute the upper hull of the points to the left of line $P_{max}P_2$**
- **Compute *lower hull* in a similar manner**



Efficiency of Quickhull Algorithm

- Finding point farthest away from line P_1P_2 can be done in linear time
- Time efficiency: $T(n) = T(x) + T(y) + T(z) + T(v) + O(n)$, where $x + y + z + v <= n$.
  - worst case: $\Theta(n^2)$ $T(n) = T(n-1) + O(n)$
  - average case: $\Theta(n)$ (under reasonable assumptions about distribution of points given)
- If points are not initially sorted by x-coordinate value, this can be accomplished in $O(n \log n)$ time
- Several $O(n \log n)$ algorithms for convex hull are known

Divide and Conquer

- *Divide-and-Conquer* Method for Algo. Design:
  - If the problem size is small enough to solve it in a straightforward manner, solve it. Else:
    - **Divide**: Divide the problem into two or more disjoint *subproblems (smaller problems)*
    - **Conquer**: Use divide-and-conquer approach recursively to solve these subproblems
    - **Combine**: Combine the solutions of the subproblems towards finding a final solution for the original problem

Divide and Conquer

- Traditionally
  - Algorithms which contain at least 2 recursive calls are called *divide and conquer* algorithms, while algorithms with one recursive call are not called *divide and conquer* algorithms.
- Classic Examples
  - Merge-sort and Quick-sort
    - The problem is divided into smaller sub-problems.
- Examples of Recursive Algorithms that are not following the Divide and Conquer Strategy:
  - **Find-set** in a Disjoint Set implementation.
    - Mostly because it does not "divide" the problem into the smaller sub-problems since it only has one recursive call.
  - Even though the recursive method to compute the Fibonacci numbers has 2 recursive calls
    - It's really not divide and conquer because it doesn't divide the problem.

This Lecture

- *Divide-and-conquer* technique for algorithm design. Example problems:
  - Integer Multiplication
  - Subset Sum Recursive Problem
  - Closest Points Problem
  - Skyline Problem
  - Strassen's Algorithm
  - Tromino Tiling

Integer Multiplication

- The standard integer multiplication routine of 2 n-digit numbers
  - Involves n multiplications of an n-digit number by a single digit
  - Plus the addition of n numbers, which have at most 2 n digits

```
  1011
 x1111
  1011
 10110
101100
+1011000
10100101
```

		quantity	time
1)	Multiplication n-digit by 1-digit	n	O(n)
2)	Additions 2n-digit by n-digit max	n	O(n)

Total time = $n*O(n) + n*O(n) = 2n*O(n) = O(n^2)$

Integer Multiplication

- Let's consider a Divide and Conquer Solution
  - Imagine multiplying an n-bit number by another n-bit number.

```
  1011
 x1111
  1011
 10110
101100
+1011000
10100101
```

    - We can split up each of these numbers into 2 halves.
    - Let the 1st number be I, and the 2nd number J
    - Let the "left half" of the 1st number by I_h and the "right half" be I_l.

  - So in this example: I is 1011 and J is 1111
    - I becomes $10*2^2 + 11$ where $I_h = 10*2^2$ and $I_l = 11$.
    - and $J_h = 11*2^2$ and $J_l = 11$

Integer Multiplication

- So for multiplying any n-bit integers I and J
  - We can split up I into $(I_h * 2^{n/2}) + I_l$
  - And J into $(J_h * 2^{n/2}) + J_l$
- Then we get
  - $I \times J = [(I_h \times 2^{n/2}) + I_l] \times [(J_h \times 2^{n/2}) + J_l]$
  - $I \times J = I_h \times J_h \times 2^n + (I_l \times J_h + I_h \times J_l) \times 2^{n/2} + I_l \times J_l$

- So what have we done?
  - We've broken down the problem of multiplying 2 n-bit numbers into
    - 4 multiplications of n/2-bit numbers plus 3 additions.
  - $T(n) = 4T(n/2) + \theta(n)$
  - Solving this using the Master Theorem gives us…
    - $T(n) = \theta(n^2)$

Integer Multiplication

- So we haven't really improved that much,
  - Since we went from a $O(n^2)$ solution to a $O(n^2)$ solution
- Can we optimize this in any way?
  - We can re-work this formula using some clever choices…
  - Some clever choices of:

    $P_1 = (I_h + I_l) \times (J_h + J_l) = I_h x J_h + I_h x J_l + I_l x J_h + I_l x J_l$

    $P_2 = I_h \times J_h$, and

    $P_3 = I_l \times J_l$
  - Now, note that

    $P_1 - P_2 - P_3 = I_h x J_h + I_h x J_l + I_l x J_h + I_l x J_l - I_h x J_h - I_l x J_l$

    $= I_h x J_l + I_l x J_h$
  - Then we can substitute these in our original equation:

    $I \times J = P_2 \times 2^n + [P_1 - P_2 - P_3] \times 2^{n/2} + P_3.$

Integer Multiplication

$$I \times J = P_2 \times 2^n + [P_1 - P_2 - P_3] \times 2^{n/2} + P_3.$$

- Have we reduced the work?
  - Calculating P2 and P3 – take n/2-bit multiplications.
  - P1 takes two n/2-bit additions and then one n/2-bit multiplication.
  - Then, 2 subtractions and another 2 additions, which take O(n) time.

- This gives us : $T(n) = 3T(n/2) + \theta(n)$
  - Solving gives us $T(n) = \theta(n^{(\log_2 3)})$, which is approximately $T(n) = \theta(n^{1.585})$, a solid improvement.

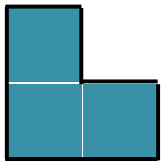
Integer Multiplication

- Although this seems it would be slower initially because of some extra pre-computing before doing the multiplications, ***for very large integers***, this will save time.

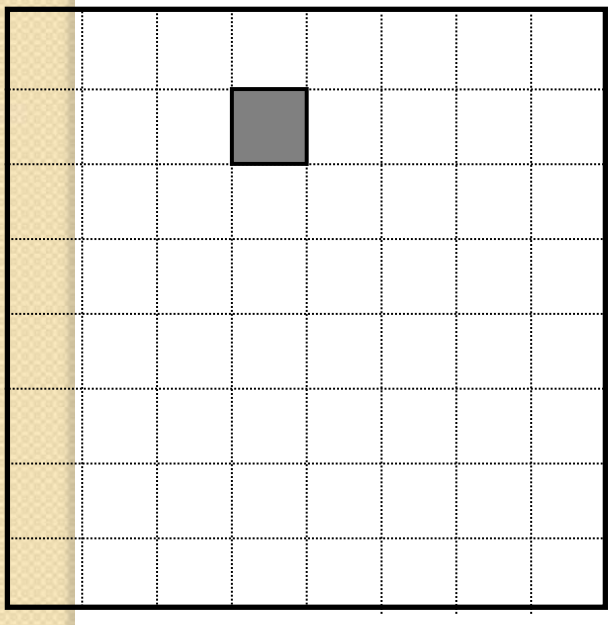
- Q: Why won't this save time for small multiplications?
  - A: The hidden constant in the $\theta(n)$ in the second recurrence is much larger. It consists of 6 additions/subtractions whereas the $\theta(n)$ in the first recurrence consists of 3 additions/subtractions.

Tromino Tiling

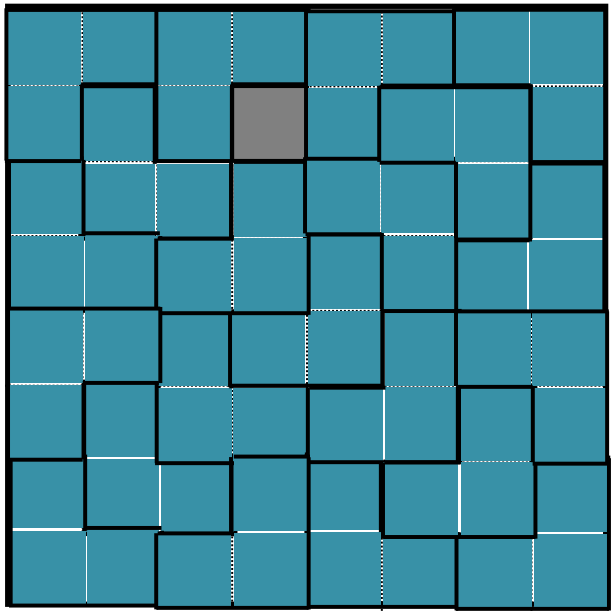
A tromino tile:



And a $2^n x 2^n$ board with a hole:

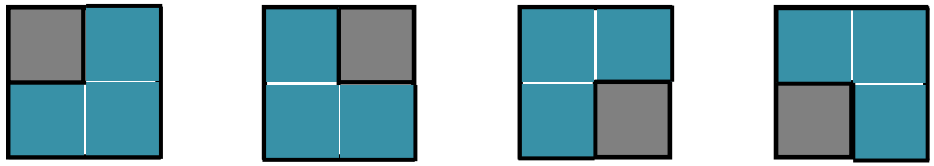


A tiling of the board with trominos:



Tiling: Trivial Case (n = 1)

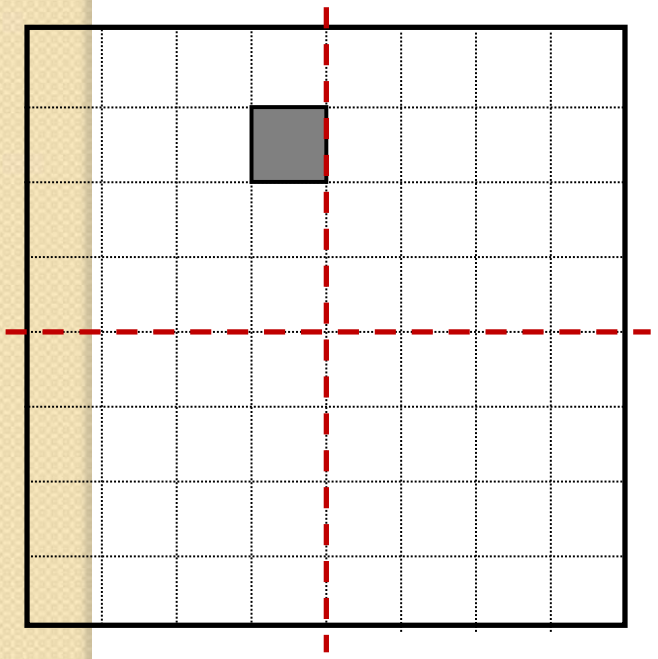
- Trivial case (n = 1): tiling a 2x2 board with a hole:



- Idea – try somehow to reduce the size of the original problem, so that we eventually get to the 2x2 boards which we know how to solve…

Tiling: Dividing the Problem

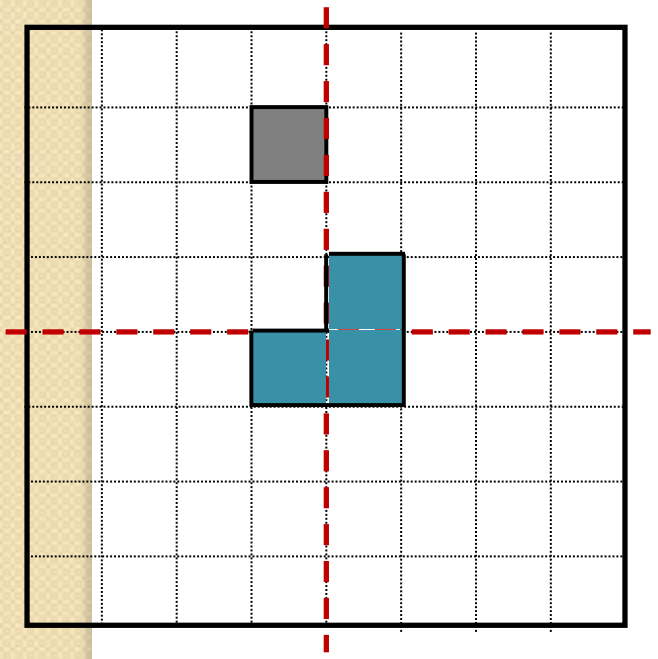
- To get smaller square boards let's divide the original board into for boards



- Great! We have one problem of the size $2^{n-1}x2^{n-1}$!
- But: The other three problems are not similar to the original problems – they do not have holes!

Tiling: Dividing the Problem

- Idea: insert one tromino at the center to get three holes in each of the three smaller boards



- Now we have four boards with holes of the size $2^{n-1}x2^{n-1}$.
- Keep doing this division, until we get the 2x2 boards with holes – we know how to tile those

Tiling: Algorithm

INPUT: *n* – the board size (2^nx2^n board), *L* – location of the hole.
OUTPUT: tiling of the board

```
Tile(n, L)
   if n = 1 then
      Trivial case
      Tile with one tromino
      return
   Divide the board into four equal-sized boards
   Place one tromino at the center to cut out 3 additional
  holes
   Let L1, L2, L3, L4 denote the positions of the 4 holes
   Tile(n-1, L1)
   Tile(n-1, L2)
   Tile(n-1, L3)
   Tile(n-1, L4)
```

Divide and Conquer

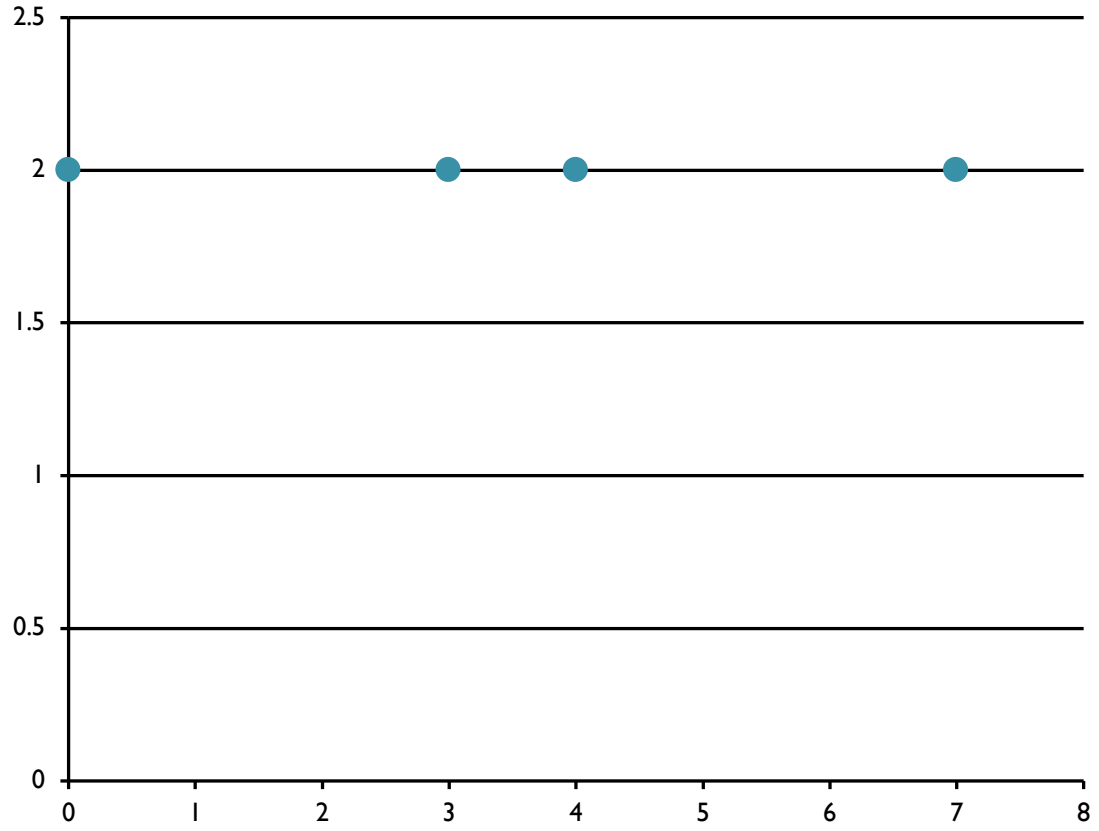
- *Divide-and-Conquer* Method for Algo. Design:
  - If the problem size is small enough to solve it in a straightforward manner, solve it. Else:
    - **Divide**: Divide the problem into two or more disjoint *subproblems (smaller problems)*
    - **Conquer**: Use divide-and-conquer approach recursively to solve these subproblems
    - **Combine**: Combine the solutions of the subproblems towards finding a final solution for the original problem

Tiling: Divide-and-Conquer

- Tiling is a divide-and-conquer algorithm:
  - Just do it trivially if the board is 2x2, else:
  - **Divide** the board into four smaller boards (introduce holes at the corners of the three smaller boards to make them look like original problems)
  - **Conquer** using the same algorithm recursively
  - **Combine** by placing a single tromino in the center to cover the three introduced holes

Finding the Closest Pair of Points

- Problem:
  - Given n ordered pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, find the distance between the two points in the set that are closest together.



Closest-Points Problem

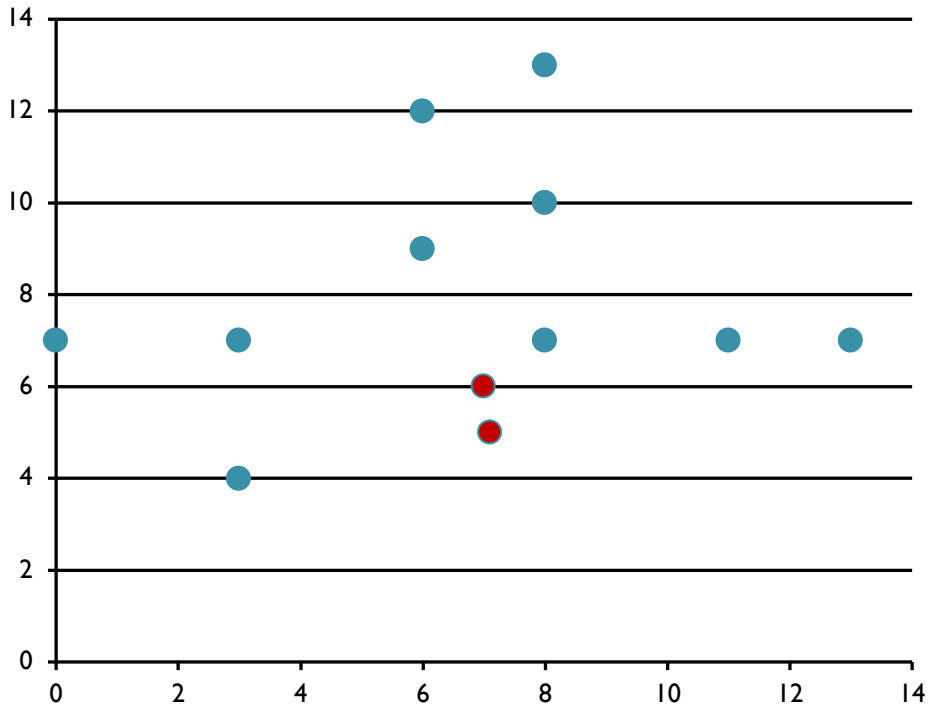
- Brute Force Algorithm
  - Iterate through all possible pairs of points, calculating the distance between each of these pairs. Any time you see a distance shorter than the shortest distance seen, update the shortest distance seen.

Since computing the distance between two points takes O(1) time,

And there are a total of $n(n-1)/2 = \theta(n^2)$ distinct pairs of points,

It follows that the running time of this algorithm is $\theta(n^2)$.

Can we do better?

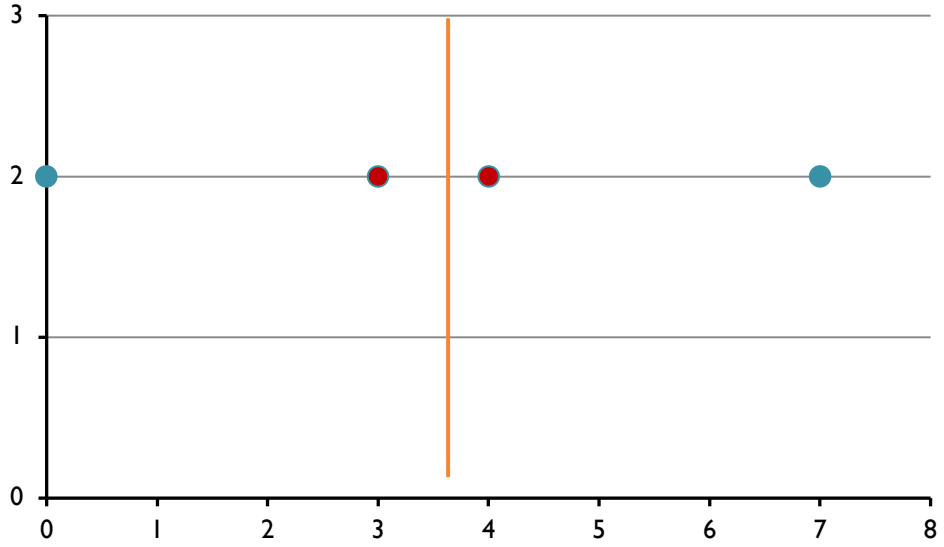


Closest-Points Problem

- Here's the idea:

1) Split the set of n points into 2 halves by a vertical line.
   - Do this by sorting all the points by their x-coordinate and then pick the middle point and draw a vertical line just to the right of it.
2) Recursively solve the problem on both sets of points.
3) Return the smaller of the two values.

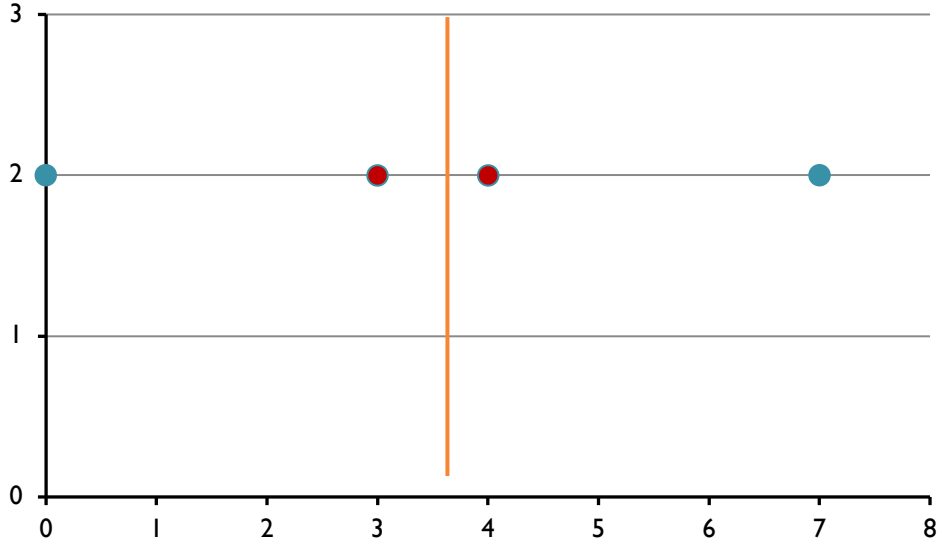
- ***What's the problem with this idea?***



Closest Points Problem

- The problem is that the actual shortest distance between any 2 of the original points MIGHT BE between a point in the 1st set and a point in the 2nd set! Like in this situation:

- So we would get a shortest distance of 3, instead of 1.

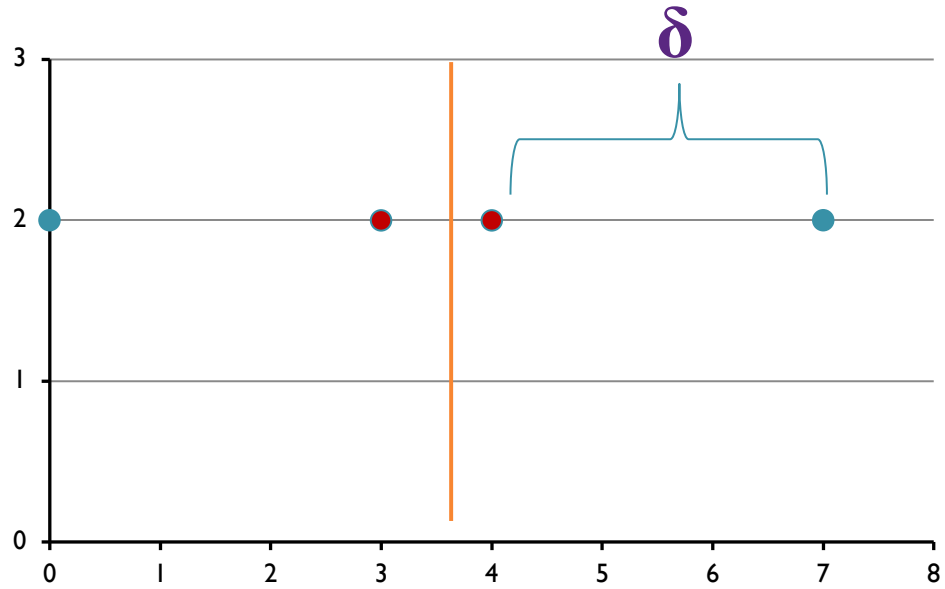


- Original idea:
  1) Split the set of n points into 2 halves by a vertical line.
     - Do this by sorting all the points by their x-coordinate and then picking the middle point and drawing a vertical line just to the right of it.
  2) Recursively solve the problem on both sets of points.
  3) Return the smaller of the two values.

We must adapt our approach:

- In step 3, we can "save" the smaller of the two values (called δ), then we have to check to see if there are points that are closer than δ apart.

- Do we need to search thru all possible pairs of points from the 2 different sides?
  - NO, we can only consider points that are within δ of dividing line.

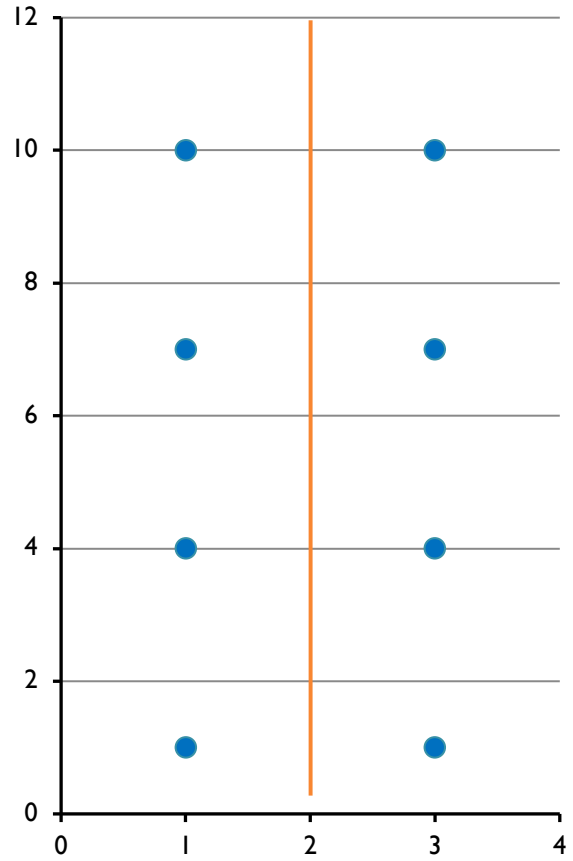


Closest Points Problem

- However, one could construct a case where ALL the points on each side are within δ of the vertical line:

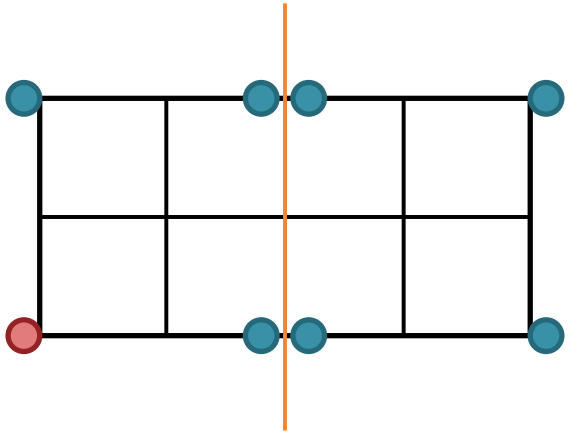
- So, this case is as bad as our original idea where we'd have to compare each pair of points to one another from the different groups.

  - But, wait!! Is it really necessary to compare each point on one side with every other point on every other side???



Closest Points Problem

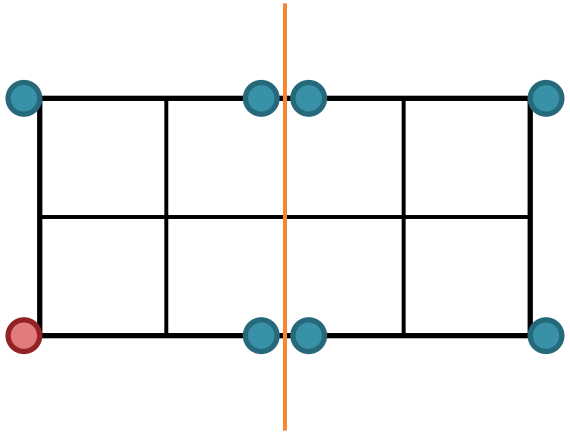
- Consider the rectangle around the dividing line that is constructed by eight $\delta/2$ x $\delta/2$ squares.



  - Note that the diagonal of each square is $\delta/\sqrt{2}$, which is less than δ.
  - Since each square lies on a single side of the dividing line, at most one point lies in each box
    - Because if 2 points were within a single box the distance between those 2 points would be less than δ.
  - Therefore, there are at MOST 7 other points that could possibly be a distance of less than δ apart from a given point, that have a greater y coordinate than that point.
    - (We assume that our point is on the bottom row of this grid; we draw the grid that way.)

Closest Points Problem

- Now we have the issue of how do we know ***which 7 points*** to compare a given point with?



- The idea is:
  - As you are processing the points recursively, ***SORT*** them based on the **y-coordinate**.
- Then for a given point within the strip, you only need to compare with the next 7 points.

Closest Points Problem

- Now the Recurrence relation for the runtime of this problem is:
  - $T(n) = T(n/2) + O(n)$
  - Which is the same as Mergesort, which we've shown to be $O(n \log n)$.

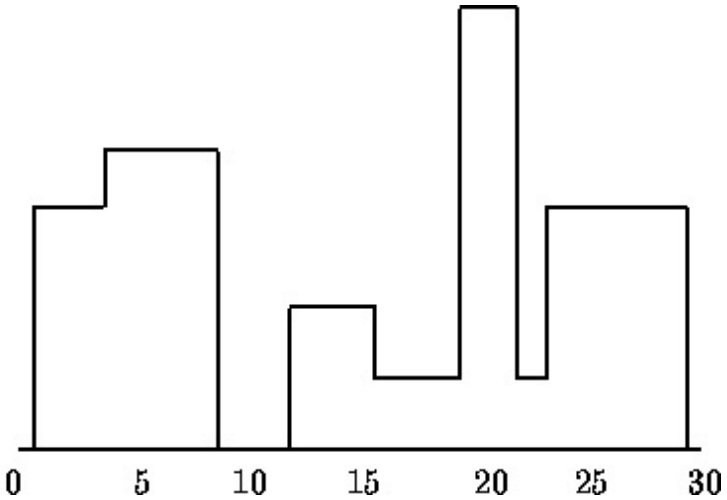
ClosestPair(ptsByX, ptsByY, n)

```
if (n = 1) return 1
if (n = 2) return distance(ptsByX[0], ptsByX[1])

// Divide into two subproblems
mid ← n/2 -1
copy ptsByX[0 . . . mid] into new array XL in x order.
copy ptsByX[mid+1 . . . n − 1] into new array XR

copy ptsByY into arrays Y L and Y R in y order, s.t.
        XL and Y L refer to same points, as do XR,Y R.

// Conquer
distL ← ClosestPair(XL, Y L, n/2)
distR ← ClosestPair(XR, Y R, n/2)

// Combine
midPoint ptsByX[mid]
lrDist min(distL, distR)
Construct array yStrip, in increasing y order,
    of all points p in ptsByY s.t.
    |p.x − midPoint.x| < lrDist

// Check yStrip
minDist ← lrDist
for (j← 0; j ≤ yStrip.length − 2; j++) {
    k ← j + 1
    while (k yStrip.length − 1 and
            yStrip[k].y − yStrip[j].y < lrDist) {
    d ← distance(yStrip[j], yStrip[k])
    minDist ← min(minDist, d)
    k++
    }
}
return minDist
```

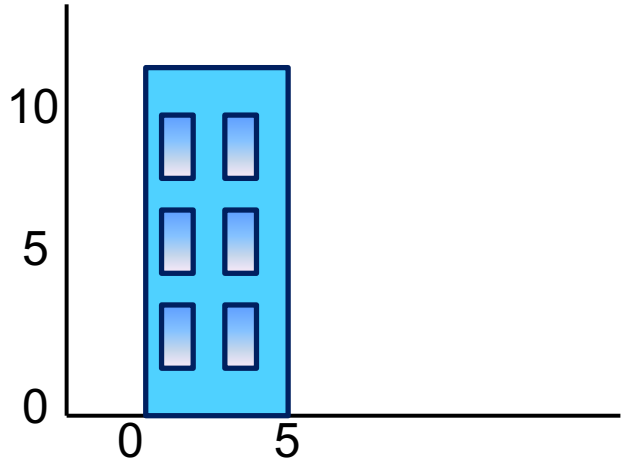
```
closest_pair(p) {
  mergesort(p, 1, n) // n is number of points
  return rec_cl_pair(p, 1, 2)
}

rec_cl_pair(p, i, j) {
  if (j - i < 3)  { \\ If there are three points or less...
      mergesort(p, i, j)  // based on y coordinate
      return shortest_distance(p[i], p[i+1], p[i+2])
  }

  xval = p[(i+j)/2].x
  deltaL = rec_cl_pair(p, i, (i+j)/2)
  deltaR = rec_cl_pair(p, (i+j)/2+1, j)
  delta = min(deltaL, deltaR)
  merge(p, i, j)  // merge points based on y coordinate

  v = vert_strip(p, xval, delta)

  for k=1 to size(v)-1
    for s = (k+1) to min(t, k+7)
      delta = min(delta, dist(v[k], v[s]))
  return delta
}
```

Skyline Problem

- You are to design a program to assist an architect in drawing the skyline of a city given the locations of the buildings in the city.
  - To make the problem tractable, all buildings are rectangular in shape and they share a common bottom (the city they are built in is very flat).

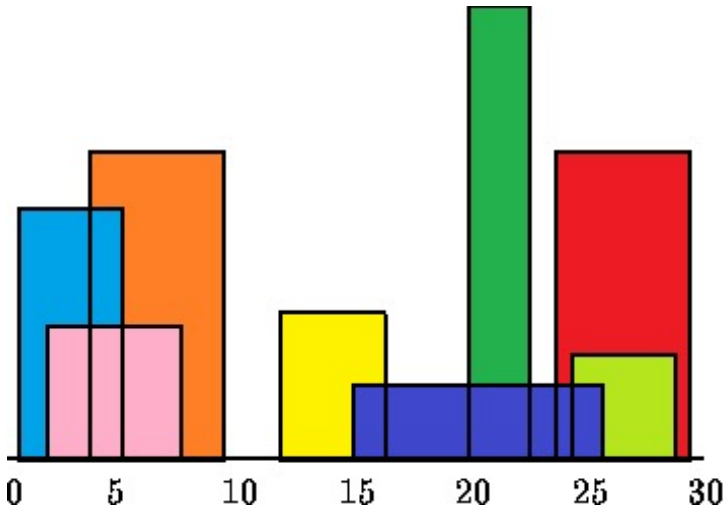


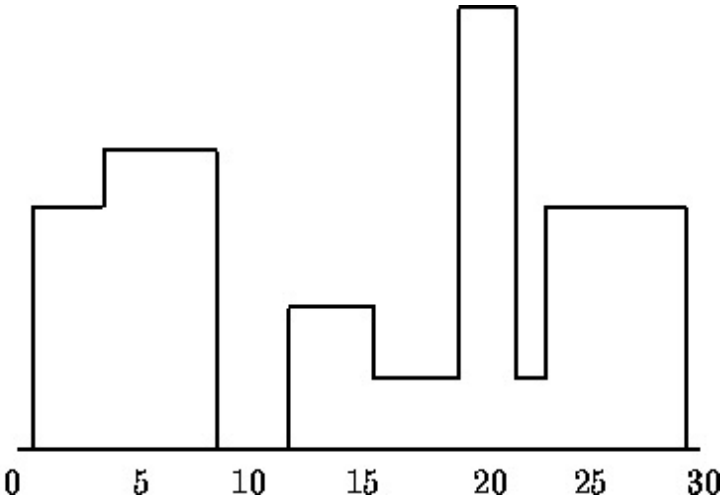
- A building is specified by an ordered triple (Li, Hi, Ri)
  - where Li and Ri are left and right coordinates, respectively, of building *i* and Hi is the height of the building.
  - Below the single building is specified by (1,11,5)



Skyline Problem

- In the diagram below buildings are shown on the left with triples :
  - (1,11,5), (2,6,7), (3,13,9), (12,7,16), (14,3,25), (19,18,22), (23,13,29), (24,4,28)
- The skyline of those buildings is shown on the right, represented by the sequence:
  - (1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19, 18, 22, 3, 23, 13, 29, 0)





Skyline Problem

- We can solve this problem by separating the buildings into two halves and solving those recursively and then Merging the 2 skylines.
  - Similar to merge sort.
  - Requires that we have a way to merge 2 skylines.

- Consider two skylines:
  - Skyline A: $a_1, h_{11}, a_2, h_{12}, a_3, h_{13}, \ldots, a_n, 0$
  - Skyline B: $b_1, h_{21}, b_2, h_{22}, b_3, h_{23}, \ldots, b_m, 0$

- Merge(list of a's, list of b's)
  - $(c_1, h_{11}, c_2, h_{21}, c_3, \ldots, c_{n+m}, 0)$

Skyline Problem

- Clearly, we merge the list of a's and b's just like in the standard Merge algorithm.
  - But, it addition to that, we properly decide on the correct height in between each set of these boundary values.
  - We can keep two variables, one to store the current height in the first set of buildings and the other to keep the current height in the second set of buildings.
  - We simply pick the greater of the two to put in the gap.
- After we are done, (or while we are processing), we have to eliminate redundant "gaps", such as 8, 15, 9, 15, 12, where there is the same height between the x-coordinates 8 and 9 as there is between the x-coordinates 9 and 12.
  - (Similarly, we will eliminate or never form gaps such as 8, 15, 8, where the x-coordinate doesn't change.)

Skyline Problem - Runtime

- Since merging two skylines of size n/2 should take O(n), letting T(n) be the running time of the skyline problem for n buildings, we find that T(n) satisfies the following recurrence:
  - $T(n) = 2T(n/2) + O(n)$
- Thus, just like Merge Sort, for the Skyline problem **$\underline{T(n) = O(n.\log n)}$**

Subset Sum Recursive Problem

- Given n items and a target value, T, determine whether there is a subset of the items such that their sum equals T.
  - Determine whether there is a subset S of {1, …, n} such that the elements of S add up to T.
- Two cases:
  - Either there is a subset S in items {1, …, n-1} that adds up to T.
  - Or there is a subset S in items {1,…, n-1} that adds up to T – n, where S U {n} is the solution.

```
public static boolean SS(int[] vals, int target, int length,
                         String numbers)
```

- The divide-and-conquer algorithm based on this recursive solution has a running time given by the recurrence:
  - $T(n) = 2T(n-1) + O(1)$

Subset Sum Recursive Problem

```
public class subsetsumrec {

public static boolean SS(int[] vals, int target, int length,
                         String numbers) {
  // Empty set satisfies this target.
  if (target == 0) {
       System.out.println(numbers);
       return true;
  }

  // An empty set can't add up to a non-zero value.
  if (length == 0)
       return false;

  return SS(vals, target - vals[length-1], length-1, numbers+",
  "+vals[length-1]) ||
  SS(vals, target, length-1, numbers ); } }
```

Strassen's Algorithm

- A fundamental numerical operation is the multiplication of 2 matrices.
  - The standard method of matrix multiplication of two n x n matrices takes T(n) = O(n3).

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \text{X} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} c_{11} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

The following algorithm multiples n x n matrices A and B:

```
// Initialize C.
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
          C [i, j] += A[i, k] * B[k, j];
```

Strassen's Algorithm

- We can use a Divide and Conquer solution to solve matrix multiplication by separating a matrix into 4 quadrants:

$$\left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right] \times \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right] = \left[\begin{array}{cc|cc} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ \hline c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{array}\right]$$

- Then we know have:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \quad C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

if $C = AB$ **, then we have the following:**

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$
$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$
$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

8 n/2 * n/2 matrix multiples + 4 n/2 * n/2 matrix additions

$T(n) = 8T(n/2) + O(n^2)$

If we solve using the master theorem we still have $O(n^3)$

Strassen's Algorithm

- Strassen showed how two matrices can be multiplied using only 7 multiplications and 18 additions:
  - Consider calculating the following 7 products:
    - $q_1 = (a_{11} + a_{22}) * (b_{11} + b_{22})$
    - $q_2 = (a_{21} + a_{22}) * b_{11}$
    - $q_3 = a_{11} * (b_{12} - b_{22})$
    - $q_4 = a_{22} * (b_{21} - b_{11})$
    - $q_5 = (a_{11} + a_{12}) * b_{22}$
    - $q_6 = (a_{21} - a_{11}) * (b_{11} + b_{12})$
    - $q_7 = (a_{12} - a_{22}) * (b_{21} + b_{22})$
  - It turns out that
    - $c_{11} = q_1 + q_4 - q_5 + q_7$
    - $c_{12} = q_3 + q_5$
    - $c_{21} = q_2 + q_4$
    - $c_{22} = q_1 + q_3 - q_2 + q_6$

Strassen's Algorithm

- Let's verify one of these:

  Given: $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ $B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$ $C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$

  if $C = AB$ **, we know:** $c_{21} = a_{21}b_{11} + a_{22}b_{21}$

- Strassen's Algorithm states:
  - **$c_{21} = q_2 + q_4$,**

    where **$q_4 = a_{22} * (b_{21} - b_{11})$** and **$q_2 = (a_{21} + a_{22}) * b_{11}$**

Strassen's Algorithm

	Mult	Add	Recurrence Relation	Runtime
Regular	8	4	$T(n) = 8T(n/2) + O(n^2)$	$O(n^3)$
Strassen	7	18	$T(n) = 7T(n/2) + O(n^2)$	$O(n^{\log_2 7}) = O(n^{2.81})$

Strassen's Algorithm

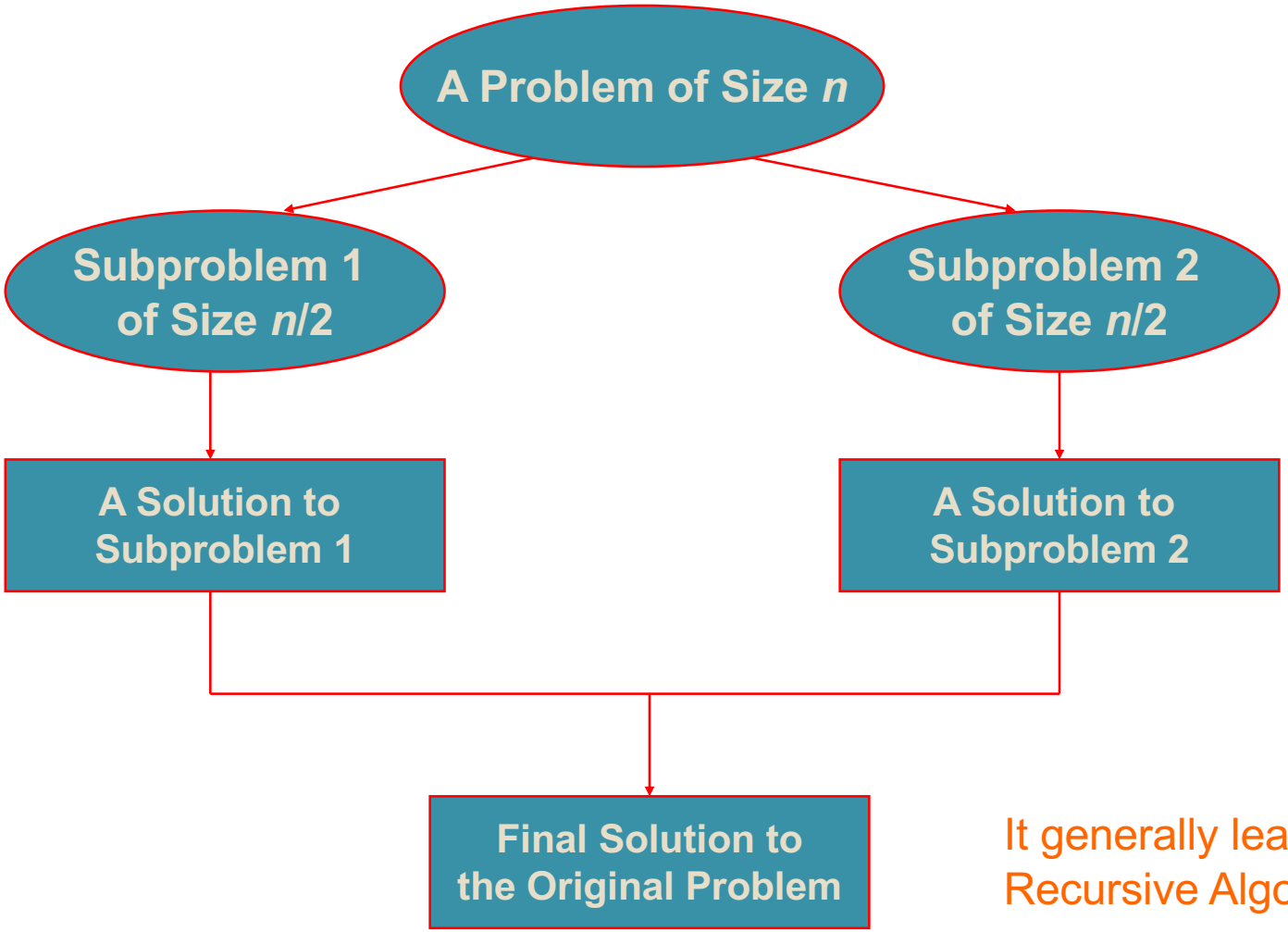
- No idea how Strassen came up with the combinations.
  - He probably realized that he wanted to determine each element in the product using less than 8 multiplications.
    - From there, he probably just played around with it.
- If T(n) be the running time of Strassen's algorithm, then it satisfies the following recurrence relation:
  - $T(n) = 7T(n/2) + O(n^2)$
    - It's important to note that the hidden constant in the $O(n^2)$ term is larger than the corresponding constant for the standard divide and conquer algorithm for this problem.
  - However, for large matrices this algorithm yields an improvement over the standard one with respect to time.

Divide-and-Conquer Summary

The most-well known algo. design strategy:

1. Divide instance of the problem into two or more smaller instances
2. Solve the smaller instances recursively to find the solutions for these smaller instances
3. Obtain the final solution to the original (larger) instance by combining these solutions of the smaller instances

Divide-and-Conquer Technique



It generally leads to a Recursive Algorithm!

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- The Algorithms we've studied:
  - Integer Multiplication
  - Tromino Tiling
  - Closest Pair of Points Problem
  - Skyline Problem
  - Subset Sum Recursive Problem
  - Strassen's Algorithm for Matrix Multiplication

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ $\Theta(n^2)$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ $\Theta(n^2 \log n)$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ $\Theta(n^3)$

$T(n) = 4T(n/4) + n \Rightarrow T(n) \in ?$ $\Theta(n \log n)$

(*Tromino Tiling*)

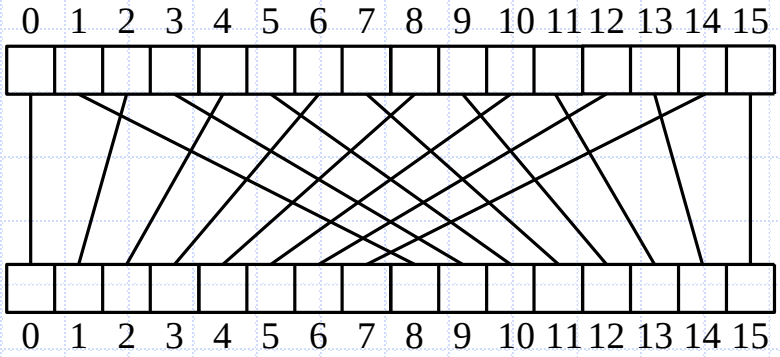
$T(n) = 7T(n/2) + n^2 \Rightarrow T(n) \in ?$ $\Theta(n^{\log_2 7}) = \Theta(n^{2.1})$

(*Strassen's Algorithm for Matrix Multiplication*)

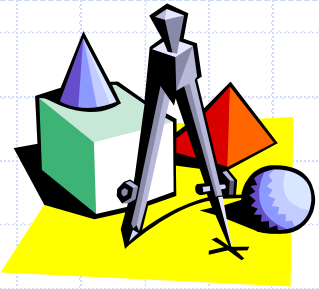
The Fast Fourier Transform

by

Jorge M. Trabal



Outline and Reading



- Polynomial Multiplication Problem
- Primitive Roots of Unity (§10.4.1)
- The Discrete Fourier Transform (§10.4.2)
- The FFT Algorithm (§10.4.3)
- Integer Multiplication (§10.4.4)
- Java FFT Integer Multiplication (§10.5)

Polynomials

- Polynomial:

$$p(x) = 5 + 2x + 8x^2 + 3x^3 + 4x^4$$

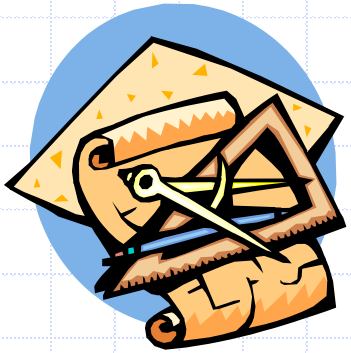
- In general,

$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

or

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

Polynomial Evaluation



- **Horner's Rule:**
  - Given coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$, defining polynomial

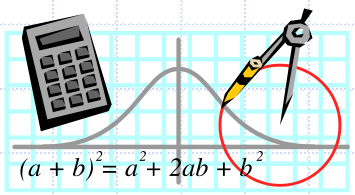
$$p(x)=\sum_{i=0}^{n-1}a_i x^i$$

  - Given x, we can evaluate p(x) in O(n) time using the equation

$$p(x)=a_0+x(a_1+x(a_2+\cdots+x(a_{n-2}+xa_{n-1})\cdots))$$

- **Eval**(A,x): [Where $A=(a_0,a_1,a_2,\ldots,a_{n-1})$]
  - If n=1, then return a_0
  - Else,
    - Let $A'=(a_1,a_2,\ldots,a_{n-1})$ [assume this can be done in constant time]
    - return a_0+x***Eval**(A',x)

Polynomial Multiplication Problem



- Given coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$ and $(b_0,b_1,b_2,\ldots,b_{n-1})$ defining two polynomials, p() and q(), and number x, compute p(x)q(x).

- Horner's rule doesn't help, since

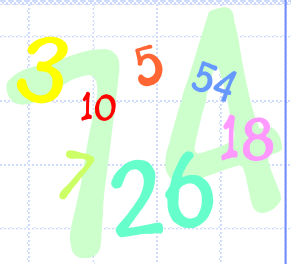
$$p(x)q(x) = \sum_{i=0}^{n-1} c_i x^i$$

where

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

- A straightforward evaluation would take $O(n^2)$ time. The "magical" FFT will do it in O(n log n) time.

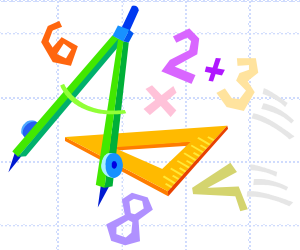
Polynomial Interpolation & Polynomial Multiplication



- Given a set of n points in the plane with distinct x-coordinates, there is **exactly one** (n-1)-degree polynomial going through all these points.
- Alternate approach to computing p(x)q(x):
  - Calculate p() on 2n x-values, $x_0, x_1, \ldots, x_{2n-1}$.
  - Calculate q() on the same 2n x values.
  - Find the (2n-1)-degree polynomial that goes through the points $\{(x_0, p(x_0)q(x_0)), (x_1, p(x_1)q(x_1)), \ldots, (x_{2n-1}, p(x_{2n-1})q(x_{2n-1}))\}$.

- Unfortunately, a straightforward evaluation would still take $O(n^2)$ time, as we would need to apply an O(n)-time Horner's Rule evaluation to 2n different points.
- The "magical" FFT will do it in O(n log n) time, by picking 2n points that are easy to evaluate…

Primitive Roots of Unity



- A number ω is a ***primitive n-th root of unity***, for n>1, if
  - $\omega^n = 1$
  - The numbers 1, ω, ω^2, …, ω^{n-1} are all distinct
- Example 1: The powers of the elements of Z^*_{11}
  - Z^*_{11}:

Set of integers between 1 and n that are relative prime to n.

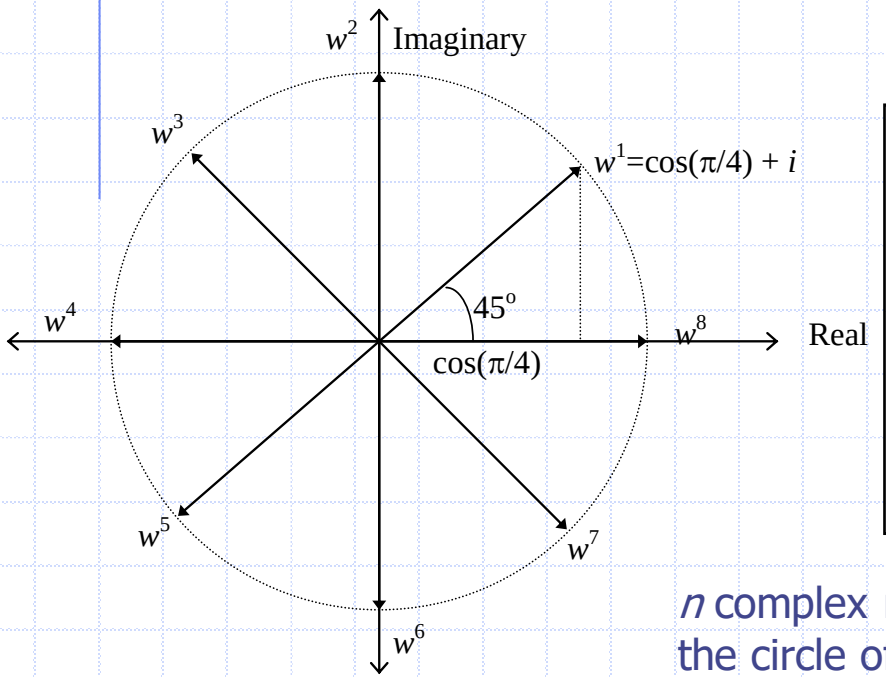
$$\gcd(x,n)=1$$

x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^10
1	1	1	1	1	1	1	1	1	1
2	4	8	5	10	9	7	3	6	1
3	9	5	4	1	3	9	5	4	1
4	5	9	3	1	4	5	9	3	1
5	3	4	9	1	5	3	4	9	1
6	3	7	9	10	5	8	4	2	1
7	5	2	3	10	4	6	9	8	1
8	9	6	4	10	3	2	5	7	1
9	4	3	5	1	9	4	3	5	1
10	1	10	1	10	1	10	1	10	1

- 2, 6, 7, 8 are 10-th roots of unity in Z^*_{11}
- $2^2=4$, $6^2=3$, $7^2=5$, $8^2=9$ are 5-th roots of unity in Z^*_{11}
- $2^{-1}=6$, $3^{-1}=4$, $4^{-1}=3$, $5^{-1}=9$, $6^{-1}=2$, $7^{-1}=8$, $8^{-1}=7$, $9^{-1}=5$

Primitive Roots of Unity

- Example 2: The complex number $e^{2\pi i/n}$ is a primitive n-th root of unity, where $i = \sqrt{-1}$



The properties are satisfied

1. $w^1 = e^{\frac{2\pi i}{n}} \neq 1$
2. $w^n = \left(e^{\frac{2\pi i}{n}}\right)^n = e^{2\pi i} = \cos 2\pi + i \sin 2\pi = 1$
3. $S = \sum_{p=0}^{n-1} w^{jp} = w^0 + w^j + w^{2j} + w^{3j} + \ldots\ldots + w^{j(n-1)} = 0$

n complex roots of unity equally spaced around the circle of unit radius centered at the origin of the complex plane.

Properties of Primitive Roots of Unity

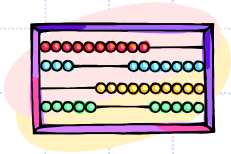
- **Inverse Property:** If ω is a primitive root of unity, then $\omega^{-1}=\omega^{n-1}$
  - Proof: $\omega\omega^{n-1}=\omega^{n}=1$
- **Cancellation Property:** For non-zero -n<k<n, $\sum_{j=0}^{n-1}\omega^{kj}=0$
  - Proof:

$$\sum_{j=0}^{n-1}\omega^{kj}=\frac{(\omega^{k})^{n}-1}{\omega^{k}-1}=\frac{(\omega^{n})^{k}-1}{\omega^{k}-1}=\frac{(1)^{k}-1}{\omega^{k}-1}=\frac{1-1}{\omega^{k}-1}=0$$

- **Reduction Property:** If ω is a primitive (2n)-th root of unity, then ω^2 is a primitive n-th root of unity.
  - Proof: If $1,\omega,\omega^2,\ldots,\omega^{2n-1}$ are all distinct, so are $1,\omega^2,(\omega^2)^2,\ldots,(\omega^2)^{n-1}$
- **Reflective Property:** If n is even, then $\omega^{n/2}=-1$.
  - Proof: By the cancellation property, for k=n/2:

$$0=\sum_{j=0}^{n-1}\omega^{(n/2)j}=\omega^{0}+\omega^{n/2}+\omega^{0}+\omega^{n/2}+\cdots+\omega^{0}+\omega^{n/2}=(n/2)(1+\omega^{n/2})$$

  - Corollary: $\omega^{k+n/2}=-\omega^{k}$.



The Discrete Fourier Transform

- Given coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$ for an (n-1)-degree polynomial p(x)
- The **Discrete Fourier Transform** is to evaluate p at the values
  - $1,\omega,\omega^2,\ldots,\omega^{n-1}$
  - We produce $(y_0,y_1,y_2,\ldots,y_{n-1})$, where $y_j=p(\omega^j)$
  - That is, $$y_j = \sum_{i=0}^{n-1} a_i \omega^{ij}$$
  - Matrix form: **y=Fa,** where **F**$[i,j]=\omega^{ij}$.

- The **Inverse Discrete Fourier Transform** recovers the coefficients of an (n-1)-degree polynomial given its values at $1,\omega,\omega^2,\ldots,\omega^{n-1}$
  - Matrix form: **a=F** $^{-1}$**y**, where **F** $^{-1}[i,j]=\omega^{-ij}/n$.

Correctness of the inverse DFT

- The DFT and inverse DFT really are inverse operations
- Proof: Let $\mathbf{A=F^{-1}F}$. We want to show that $\mathbf{A=I}$, where

$$A[i,j] = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{-ki}\omega^{kj}$$

- If i=j, then

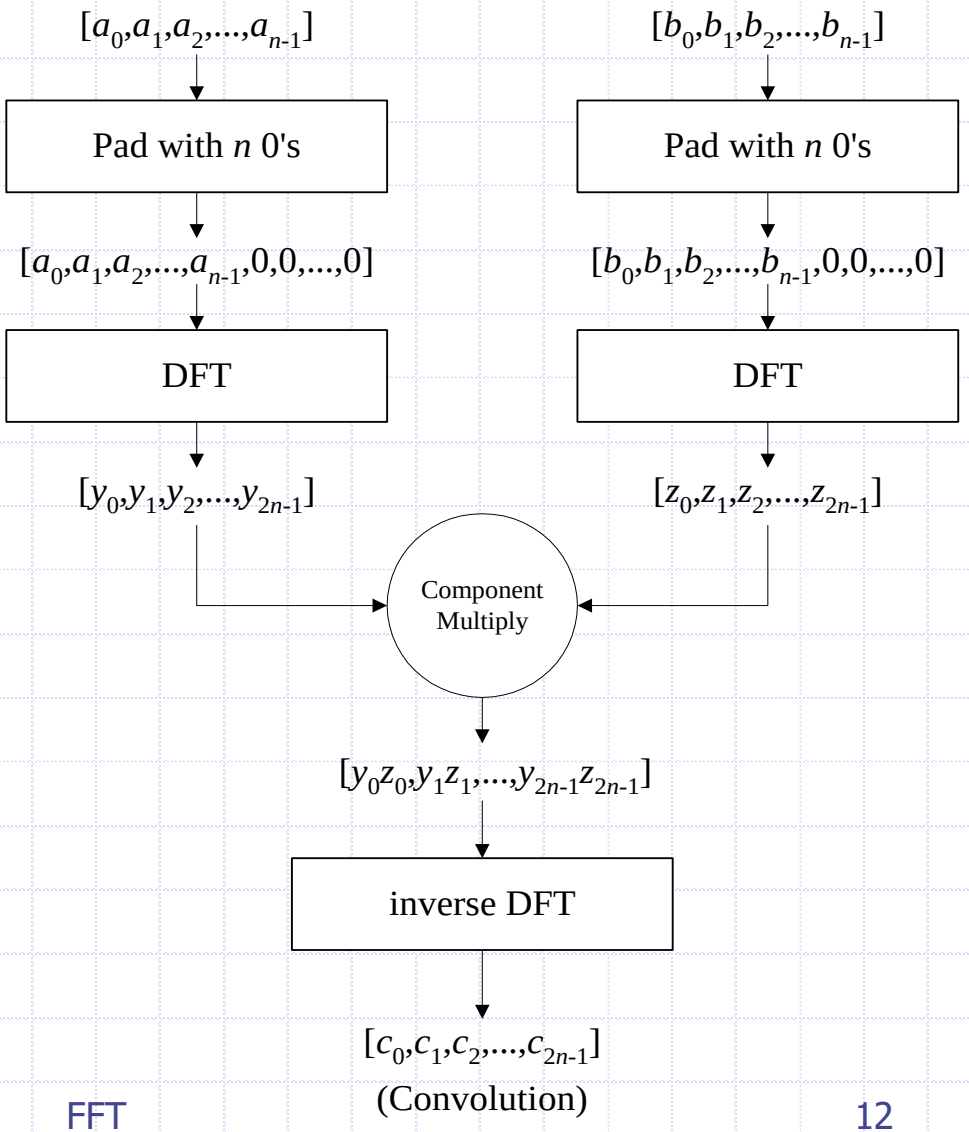
$$A[i,i] = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{-ki}\omega^{ki} = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{0} = \frac{1}{n}n = 1$$

- If i and j are different, then

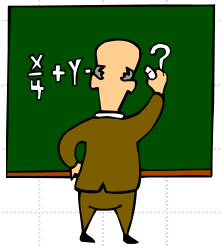
$$A[i,j] = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{(j-i)k} = 0 \qquad \text{(by Cancellation Property)}$$

Convolution

- The DFT and the inverse DFT can be used to multiply two polynomials
- So we can get the coefficients of the product polynomial quickly if we can compute the DFT (and its inverse) quickly…



The Fast Fourier Transform



- The FFT is an efficient algorithm for computing the DFT
- The FFT is based on the divide-and-conquer paradigm:
  - If n is even, we can divide a polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

into two polynomials

$$\begin{aligned} p^{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} \\ p^{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} \end{aligned}$$

and we can write

$$p(x) = p^{\text{even}}(x^2) + x p^{\text{odd}}(x^2).$$

The FFT Algorithm

Algorithm FFT$(\mathbf{a}, \omega)$:

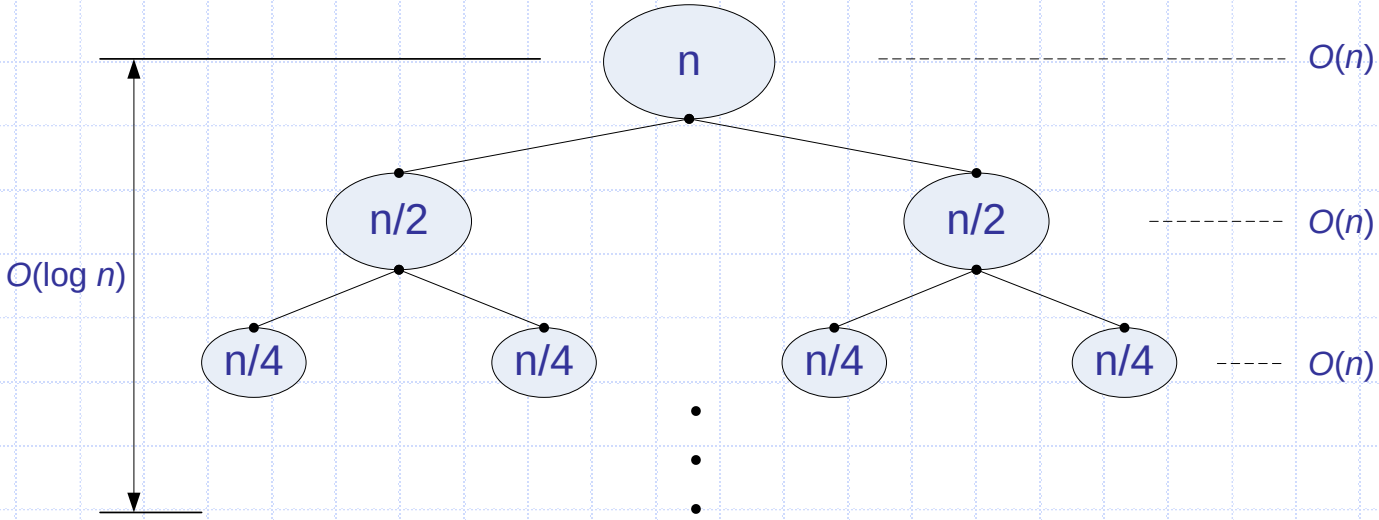
Input: An n-length coefficient vector $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ and a primitive nth root of unity ω, where n is a power of 2

Output: A vector $\mathbf{y}$ of values of the polynomial for $\mathbf{a}$ at the nth roots of unity

if $n = 1$ **then**
  return $\mathbf{y} = \mathbf{a}$.
$x \leftarrow \omega^0$ {x will store powers of ω, so initially $x = 1$.}
{Divide Step, which separates even and odd indices}
$\mathbf{a}^{\text{even}} \leftarrow [a_0, a_2, a_4, \ldots, a_{n-2}]$
$\mathbf{a}^{\text{odd}} \leftarrow [a_1, a_3, a_5, \ldots, a_{n-1}]$
{Recursive Calls, with ω^2 as $(n/2)$th root of unity, by the reduction property}
$\mathbf{y}^{\text{even}} \leftarrow \text{FFT}(\mathbf{a}^{\text{even}}, \omega^2)$
$\mathbf{y}^{\text{odd}} \leftarrow \text{FFT}(\mathbf{a}^{\text{odd}}, \omega^2)$
{Combine Step, using $x = \omega^i$}
for $i \leftarrow 0$ to $n/2 - 1$ **do**
  $y_i \leftarrow y_i^{\text{even}} + x \cdot y_i^{\text{odd}}$
  $y_{i+n/2} \leftarrow y_i^{\text{even}} - x \cdot y_i^{\text{odd}}$ {Uses reflective property}
  $x \leftarrow x \cdot \omega$
return $\mathbf{y}$

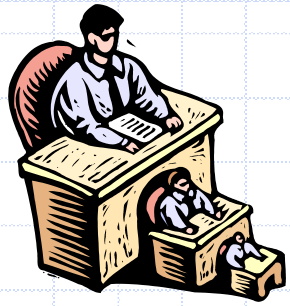
The FFT Algorithm

- The FFT algorithm is based on divide-and-conquer



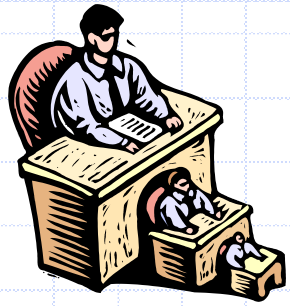
The running time is O(n log n). [inverse FFT is similar]

Non-recursive FFT



- There is also a non-recursive version of the FFT
  - Performs the FFT in place
  - Precomputes all roots of unity
  - Performs a cumulative collection of shuffles on A and on B prior to the FFT, which amounts to assigning the value at index i to the index bit-reverse(i).
- The code is a bit more complex, but the running time is faster by a constant, due to improved overhead

Non-recursive FFT



Algorithm ITERATIVE-FFT(a)

Input: An n-length coefficient vector $\boldsymbol{a} = [a_0,a_1,\ldots,a_{n-1}]$, where n is a power of 2

Output: A vector **y** of values of the polynomial for $\boldsymbol{a}$ at the nth roots of unity

BIT-REVERSE-COPY (a,A)
$n \leftarrow$ length[a] {n is a power of 2}
for $s \leftarrow 1$ **to** log n
do $m \leftarrow 2^s$
$\omega_m \leftarrow e^{2\pi i/m}$
for $k \leftarrow 0$ **to** $n-1$ **by** m
do $\omega \leftarrow 1$
for $j \leftarrow 0$ **to** $m/2-1$
do $t \leftarrow \omega\, A[k+j+m/2]$
$u \leftarrow A[k+j]$
$A[k+j] \leftarrow u+t$
$A[k+j+m/2] \leftarrow u-t$
$\omega \leftarrow \omega\, \omega_m$

Algorithm BIT-REVERSE-COPY(a,A)

Input: An n-length coefficient vector $\boldsymbol{a} = [a_0,a_1,\ldots,a_{n-1}]$ and the polynomial $\boldsymbol{A}$, where n is a power of 2

Output: Bit-reversal permutation of $\boldsymbol{a}$, $\boldsymbol{A}$

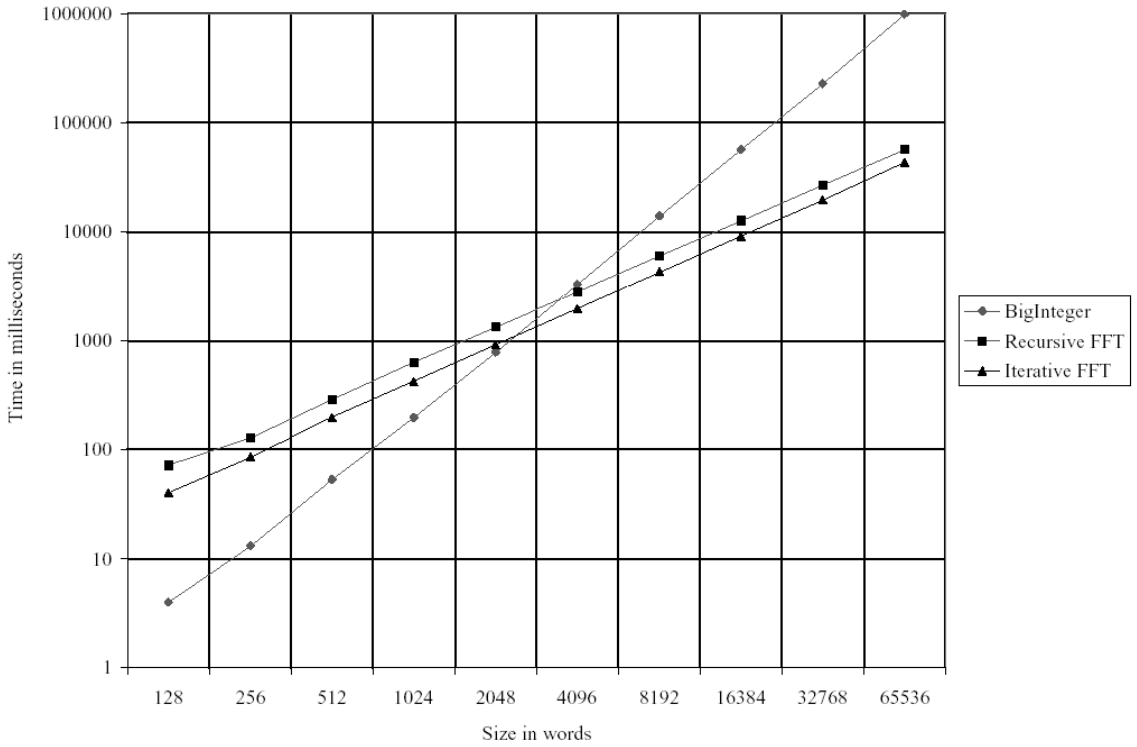
$n \leftarrow$ length[a]
for $k \leftarrow 0$ **to** $n-1$
do $A[\text{rev}(k)] \leftarrow a_k$

Multiplying Big Integers

- Given N-bit integers ($N \geq 64$) I and J, compute IJ.
- Assume: we can multiply words of O(log N) bits in constant time.
- Setup: Find a prime $p=cn+1$ that can be represented in one word, and set $m=(\log p)/3$, so that we can view I and J as n-length vectors of m-bit words.
- Finding a primitive root of unity.
  - Find a generator x of Z^*_p.
  - Then $\omega=x^c$ is a primitive n-th root of unity in Z^*_p (arithmetic is mod p)
- Apply convolution and FFT algorithm to compute the convolution C of the vector representations of I and J.
- Then compute $$K = \sum_{i=0}^{n-1} c_i 2^{mi}$$
- K is a vector representing IJ, and takes O(n log n) time to compute.

Experimental Results

- Log-log scale shows traditional multiply runs in $O(n^2)$ time, while FFT versions are almost linear



Conclusions

- Using the reduction property of the primitive roots of unity and the divide-and-conquer approach the DFT can be computed in O(nlogn) time by the FFT algorithm.
- With the FFT algorithm we can compute the big integers multiplication faster than the traditional multiplication when big size words are required.

Bibliography

1) M. T. Goodrich and R. Tamassia, *Algorithm Design*, New York: John Wiley & Sons, 2002, Chapter 10.
2) T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Massachusetts: The MIT Press, 2001, Chapter 30.

Graph Algorithms

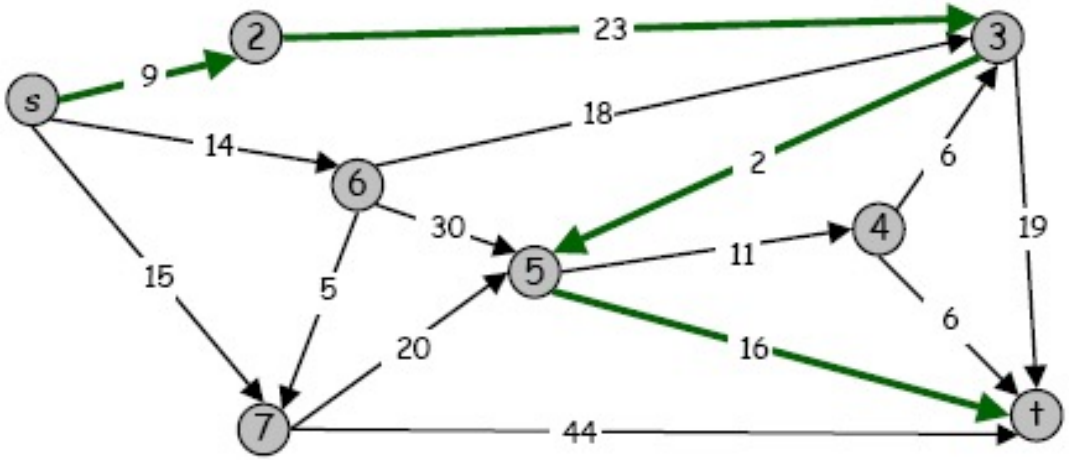
Shortest path problems

Shortest path network.

- Directed graph.
- Source s, destination t.
- cost(v-w) = cost of using edge from v to w.

Shortest path problem: find shortest directed path from s to t.

- Cost of path = sum of arc costs in path.



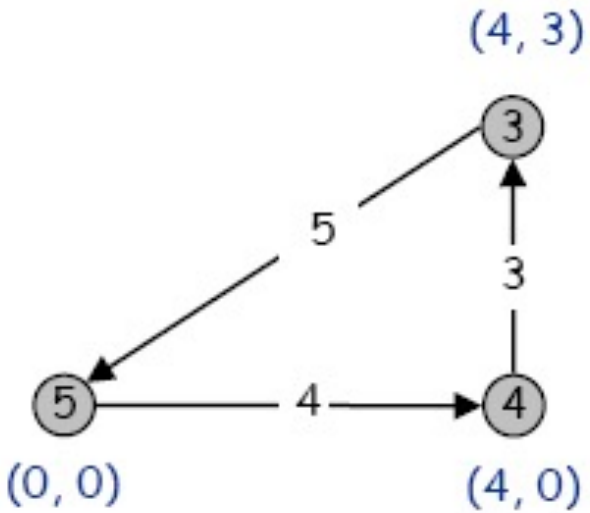
Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.

[Adapted from K.Wayne]

Graph Algorithms

Some versions of the problem that we consider.

- Undirected.
- Directed.
- Single source.
- All-pairs.
- Arc costs are ≥ 0.
- Points in plane with Euclidean distances.



Euclidean

[Adapted from K.Wayne]

Graph Algorithms

Shortest path problems

Applications

More applications.

- Robot navigation.
- Typesetting in TeX.
- Urban traffic planning.
- Tramp steamer problem.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Subroutine in higher level algorithms.
- Routing of telecommunications messages.
- Approximating piecewise linear functions.
- Exploiting arbitrage opportunities in currency exchange.
- Open Shortest Path First (OSPF) routing protocol for IP.
- Optimal truck routing through given traffic congestion pattern.

Reference: *Network Flows: Theory, Algorithms, and Applications*, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

[Adapted from K.Wayne]

Graph Algorithms

Single-Source Shortest Paths

Given graph (directed or undirected) G = (V,E) with weight function w: E $\rightarrow$ **R** and a vertex s$\in$V,
find for all vertices v$\in$V the minimum possible weight for path from s to v.

We will discuss two general case algorithms:

- **Dijkstra's** (positive edge weights only)
- **Bellman-Ford** (positive end negative edge weights)

If all edge weights are equal (let's say 1), the problem is solved by BFS in Θ(V+E) time.

Graph Algorithms

Dijkstra's Algorithm - Relax

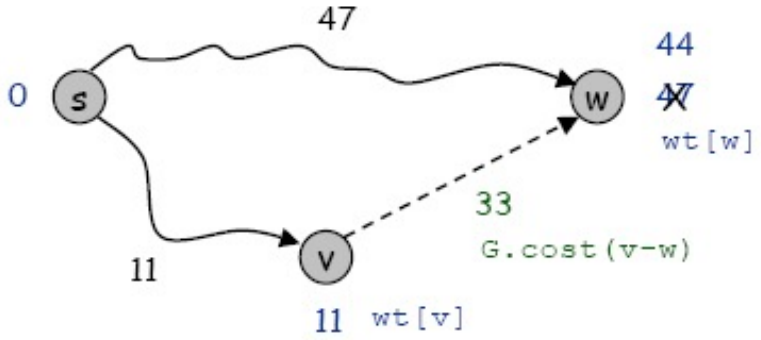
Relax(**vertex** u, **vertex** v, **weight** w)
  if d[v] > d[u] + w(u,v) **then**
    d[v] ← d[u] + w(u,v)
    p[v] ← u

Edsger W. Dijkstra

Edge relaxation.

- Consider edge `v-w` with `G.cost(v, w)`.
- If path from `s` to `v` plus edge `v-w` is better than current path to `w`, then update.

```
if (wt[w] > wt[v] + G.cost(v, w)) {
    wt[w] = wt[v] + G.cost(v, w);
    pred[w] = v;
}
```
edge relaxation



[Adapted from K.Wayne]

Graph Algorithms

Dijkstra's Algorithm - Idea

Dijkstra's algorithm.

- Initialize `wt[v]` = ∞ and `wt[s]` = 0.
- Insert all vertices `v` onto PQ with priorities `wt[v]`.
- Repeatedly delete node `v` from PQ that has min `wt[v]`.
  - add `v` to S
  - for each `v-w`, relax `v-w`

[Adapted from K.Wayne]

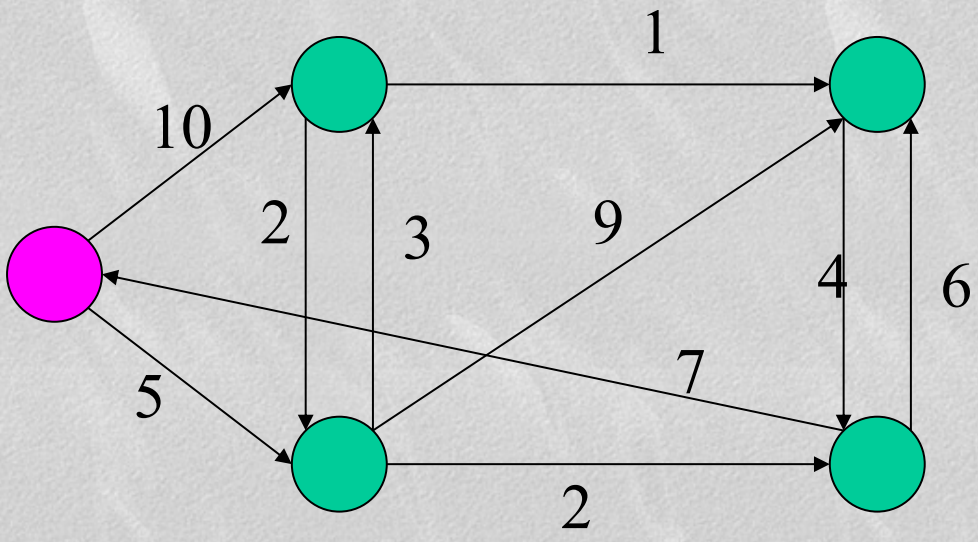
Graph Algorithms

Dijkstra's Algorithm - SSSP-Dijkstra

```
SSSP-Dijkstra(graph (G,w), vertex s)
   InitializeSingleSource(G, s)
   S ← ∅
   Q ← V[G]
   while Q ≠ 0 do
      u ← ExtractMin(Q)
      S ← S ∪ {u}
      for v ∈ Adj[u] do
         Relax(u,v,w)

InitializeSingleSource(graph G, vertex  s)
    for v ∈ V[G] do
        d[v] ← ∞
        p[v] ← 0
    d[s] ← 0
```

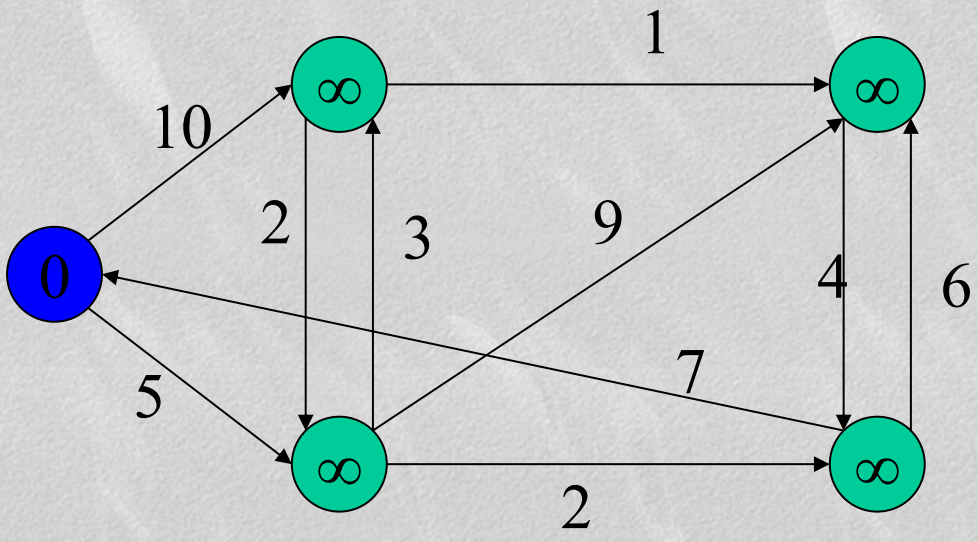
Graph Algorithms

Dijkstra's Algorithm - Example



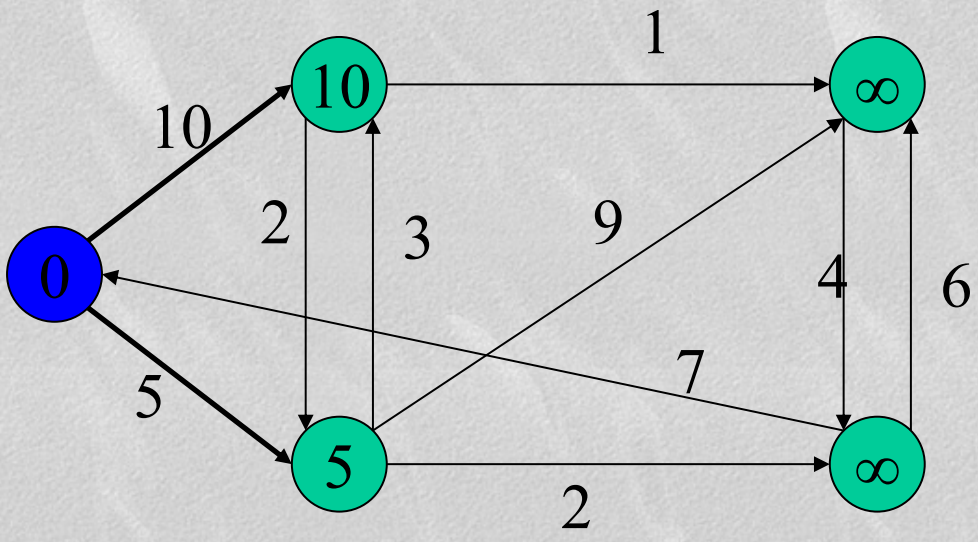
Graph Algorithms

Dijkstra's Algorithm - Example



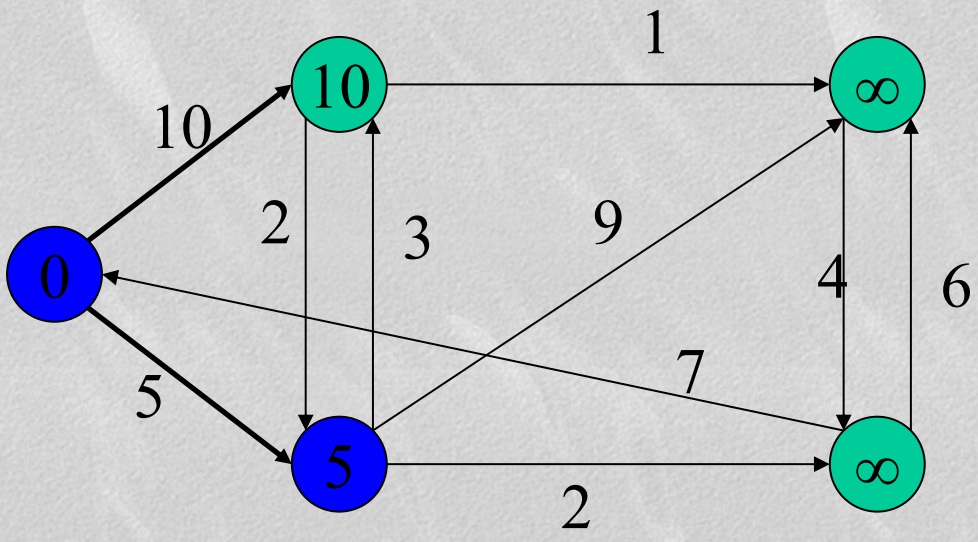
Graph Algorithms

Dijkstra's Algorithm - Example



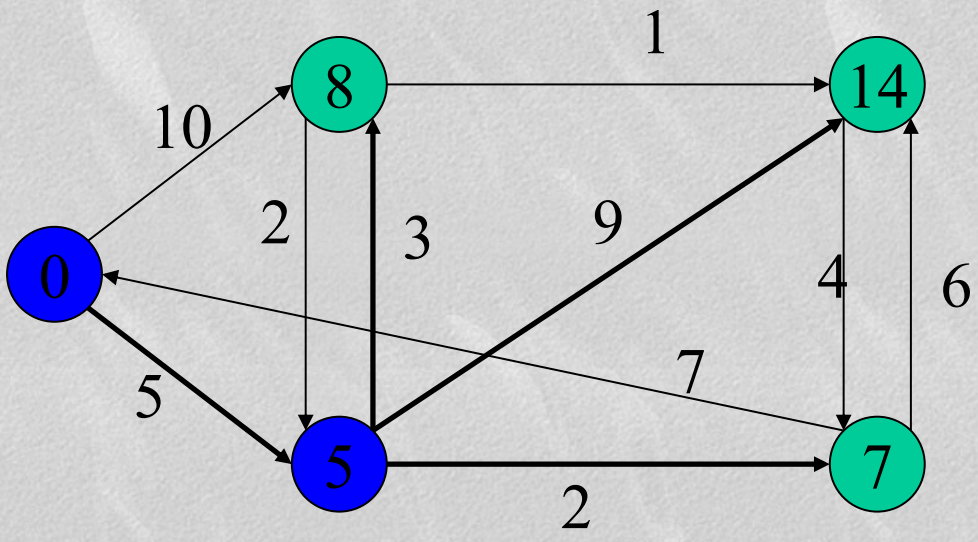
Graph Algorithms

Dijkstra's Algorithm - Example



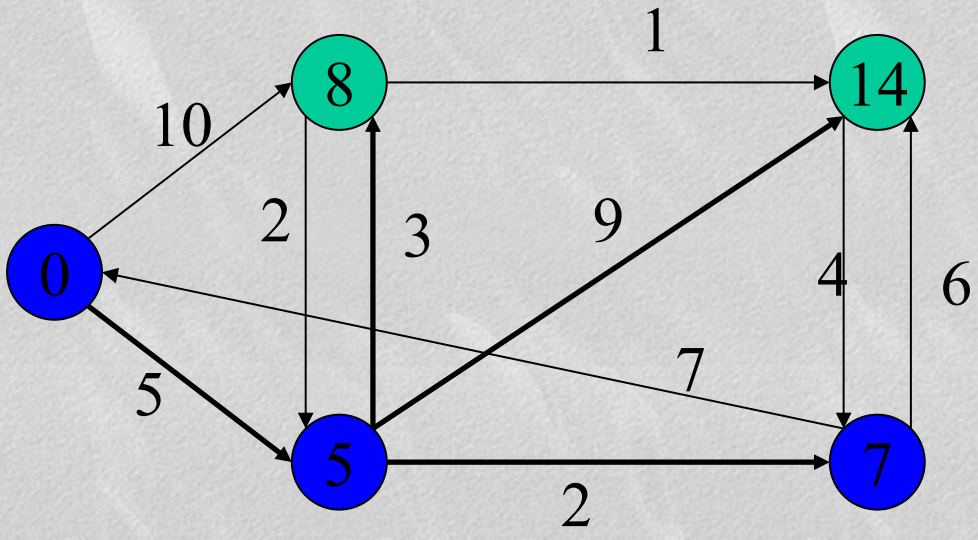
Graph Algorithms

Dijkstra's Algorithm - Example



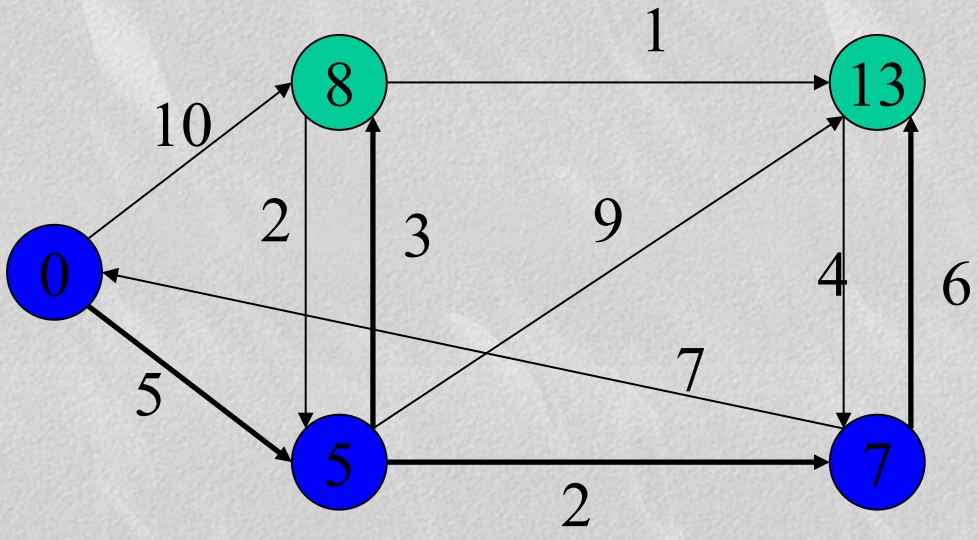
Graph Algorithms

Dijkstra's Algorithm - Example



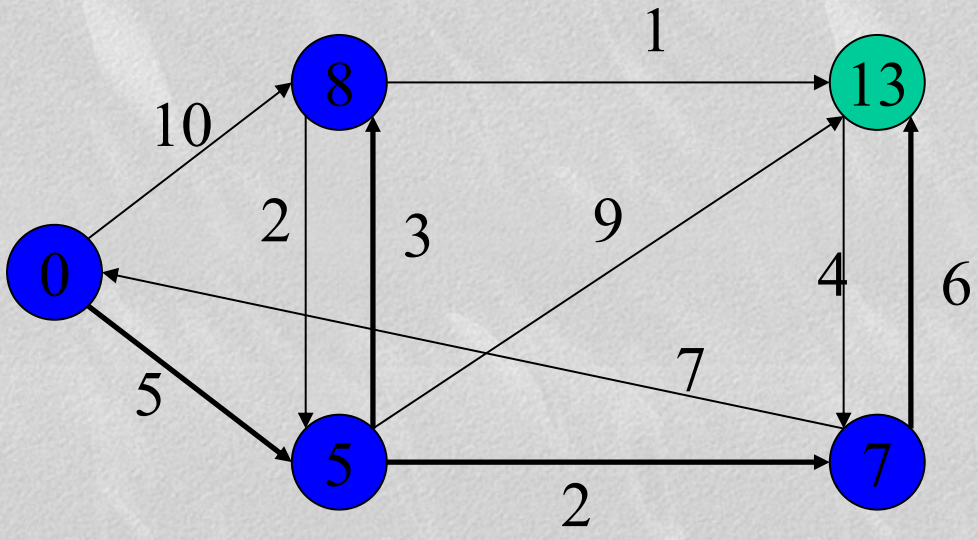
Graph Algorithms

Dijkstra's Algorithm - Example



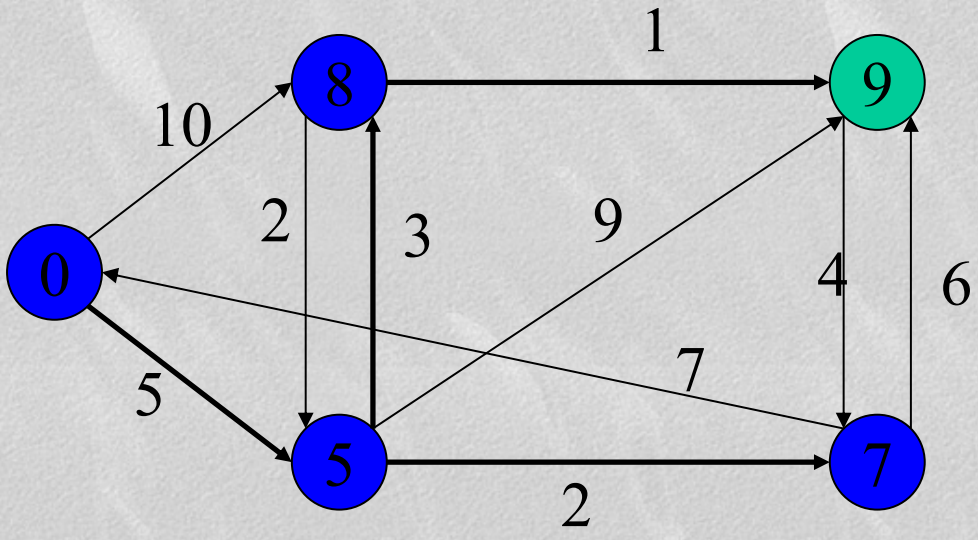
Graph Algorithms

Dijkstra's Algorithm - Example



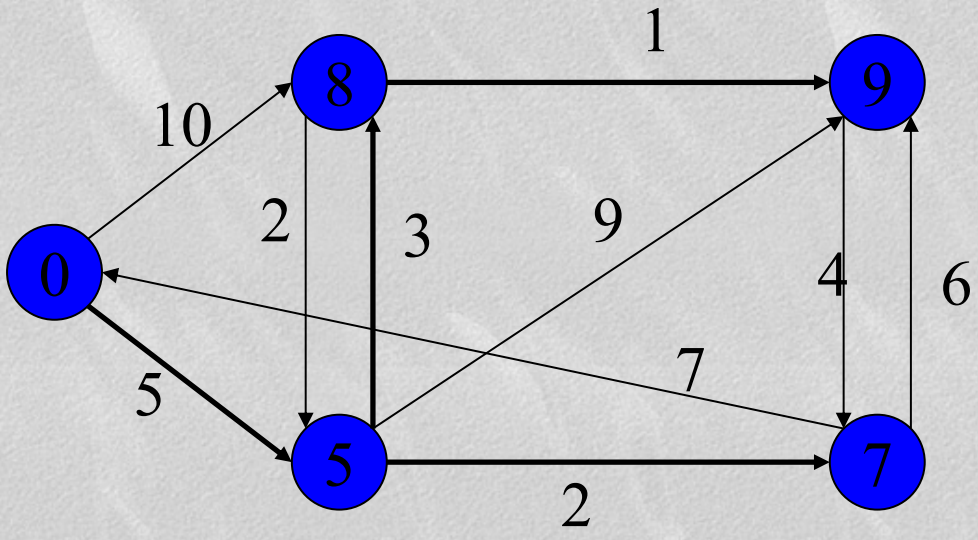
Graph Algorithms

Dijkstra's Algorithm - Example



Graph Algorithms

Dijkstra's Algorithm - Example



Graph Algorithms
Dijkstra's Algorithm - Complexity

executed $\Theta(V)$ times

$\Theta(E)$ times in total

$\Theta(V)$

$\Theta(1)$?

```
SSSP-Dijkstra(graph (G,w), vertex s)
    InitializeSingleSource(G, s)
    S ← ∅
    Q ← V[G]
    while Q ≠ 0 do
        u ← ExtractMin(Q)
        S ← S ∪ {u}
        for u ∈ Adj[u] do
            Relax(u,v,w)
```

```
InitializeSingleSource(graph G, vertex  s)
    for v ∈ V[G] do
        d[v] ← ∞
        p[v] ← 0
    d[s] ← 0
```

```
Relax(vertex  u, vertex v, weight w)
    if d[v] > d[u] + w(u,v) then
        d[v] ← d[u] + w(u,v)
        p[v] ← u
```

Graph Algorithms

Dijkstra's Algorithm - Complexity

InitializeSingleSource $T_I(V,E) = \Theta(V)$

Relax $T_R(V,E) = \Theta(1)$?

SSSP-Dijkstra

$$T(V,E) = T_I(V,E) + \Theta(V) + V\,\Theta(\log V) + E\,T_R(V,E) =$$

$$= \Theta(V) + \Theta(V) + V\,\Theta(\log V) + E\,\Theta(1) = \Theta(E + V \log V)$$

Graph Algorithms

Dijkstra's Algorithm: Implementation Cost Summary

		Priority Queue			
Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
insert	V	V	log V	$d \log_d V$	1
delete-min	V	V	log V	$d \log_d V$	log V
decrease-key	E	1	log V	$\log_d V$	1
is-empty	V	1	1	1	1
total		V^2	E log V	$E \log_{E/V} V$	E + V log V

† Individual ops are amortized bounds

Observation: algorithm is almost identical to Prim's MST algorithm!

Priority first search: variations on a theme.

[Adapted from K.Wayne]

Graph Algorithms

Dijkstra's Algorithm: Proof of Correctness

Invariant. For each vertex v, wt[v] is length of shortest s-v path whose internal vertices are in S; for each vertex v in S, wt[v] = wt*[v] .

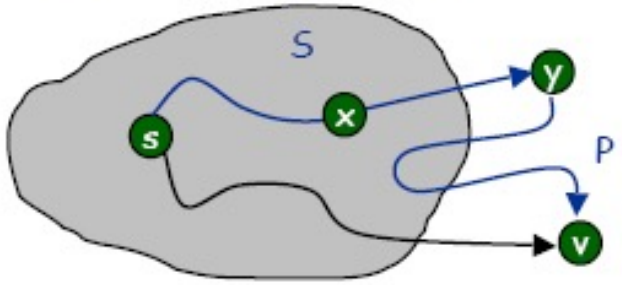
Proof: by induction on |S|.

Base case: |S| = 0 is trivial.

length of shortest s-v path

Induction step:

- Let v be next vertex added to S by Dijkstra's algorithm.
- Let P be a shortest s-v path, and let x-y be first edge leaving S.



- We show wt[v] = wt*[v].

$$wt[v] \geq wt^*[v] \geq wt^*[y] = wt[y] \geq wt[v]$$

wt[v] length of some path

nonnegative weights

induction

Dijkstra chose v before y

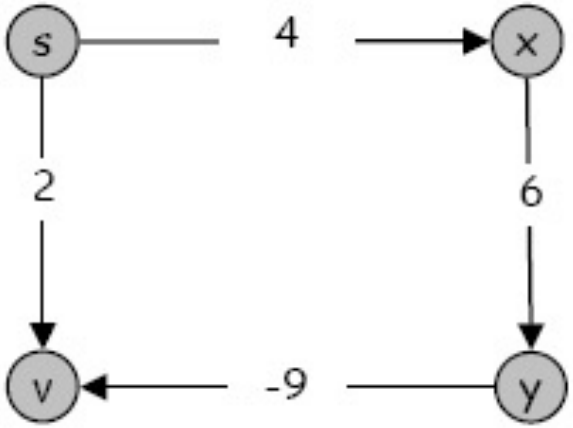
[Adapted from K.Wayne]

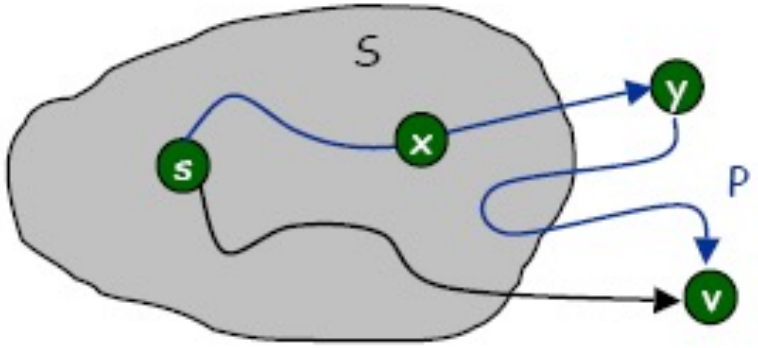
Graph Algorithms

Dijkstra's Algorithm With Negative Costs

Dijkstra's algorithm fails if there are negative weights.

- Ex: Selects vertex `v` immediately after `s`.
  But shortest path from `s` to `v` is `s-x-y-v`.





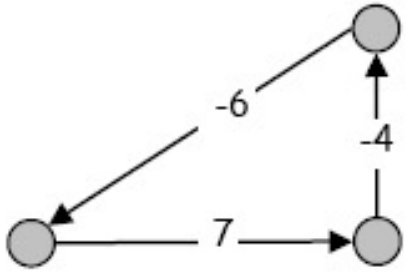
Dijkstra proof of correctness breaks down since it assumes cost of P is nonnegative.

[Adapted from K.Wayne]

Graph Algorithms

Negative Cycles

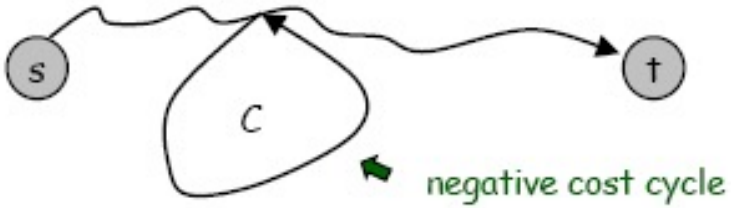
Negative cycle. Directed cycle whose sum of edge costs is negative.



-6 + 7 – 4 = -5

Caveat. Bellman-Ford terminates and finds shortest (simple) path after at most V phases if and only if no negative cycles.

Observation. If negative cycle on path from s to t, then shortest path can be made arbitrarily negative by spinning around cycle.



[Adapted from K.Wayne]

Graph Algorithms

Idea in Bellman-Ford Algorithm

- Repeat the following |V|-1 times:

  relax each edge in E

- Test if there is any negative weight cycle by checking if d[v] > d[u] + w(u, v) for each edge (u, v).

[Adapted from X.Wang]

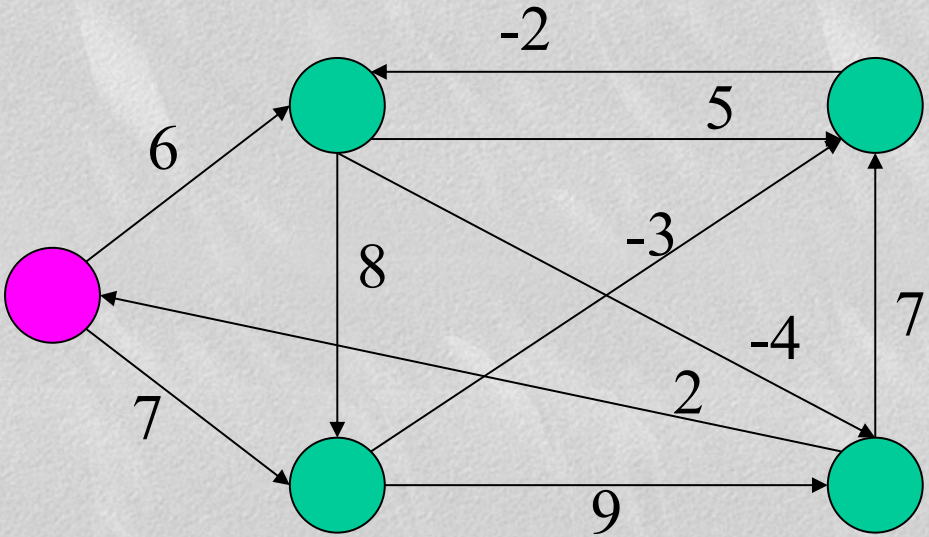
Graph Algorithms

Bellman-Ford Algorithm - SSSP-BellmanFord

```
SSSP-BellmanFord(graph (G,w), vertex s)
    InitializeSingleSource(G, s)
    for i ← 1 to |V[G] – 1| do
        for (u,v) ∈ E[G] do
            Relax(u,v,w)
    for (u,v) ∈ E[G] do
        if d[v] > d[u] + w(u,v) then
            return false
    return true
```

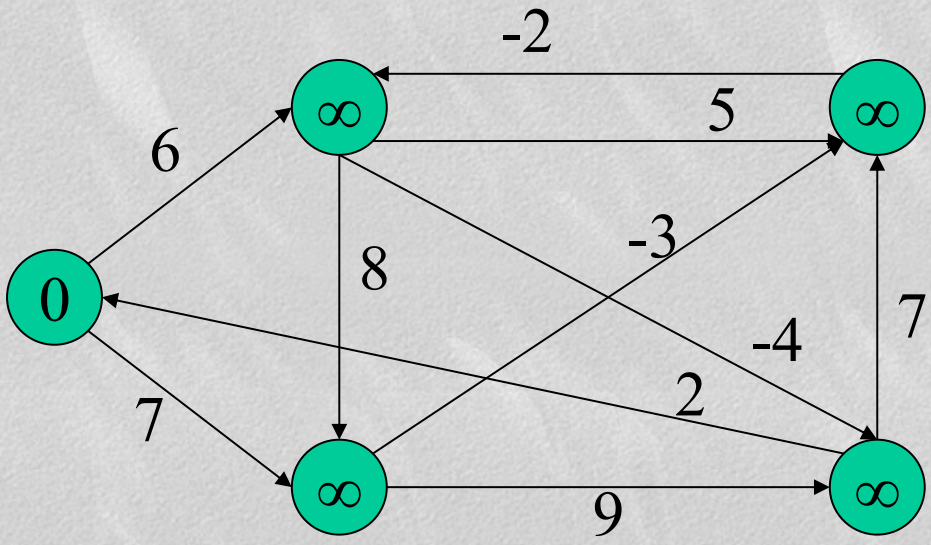
Graph Algorithms

Bellman-Ford Algorithm - Example



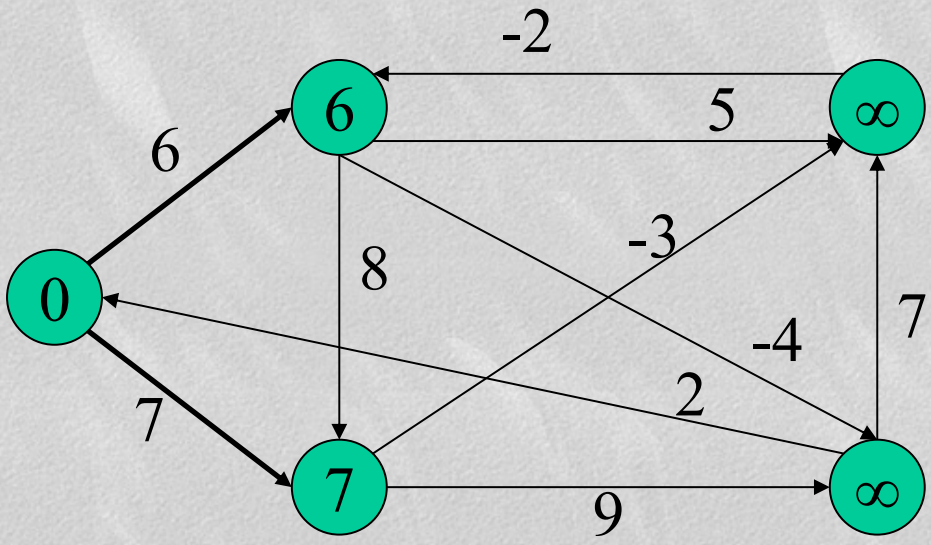
Graph Algorithms

Bellman-Ford Algorithm - Example



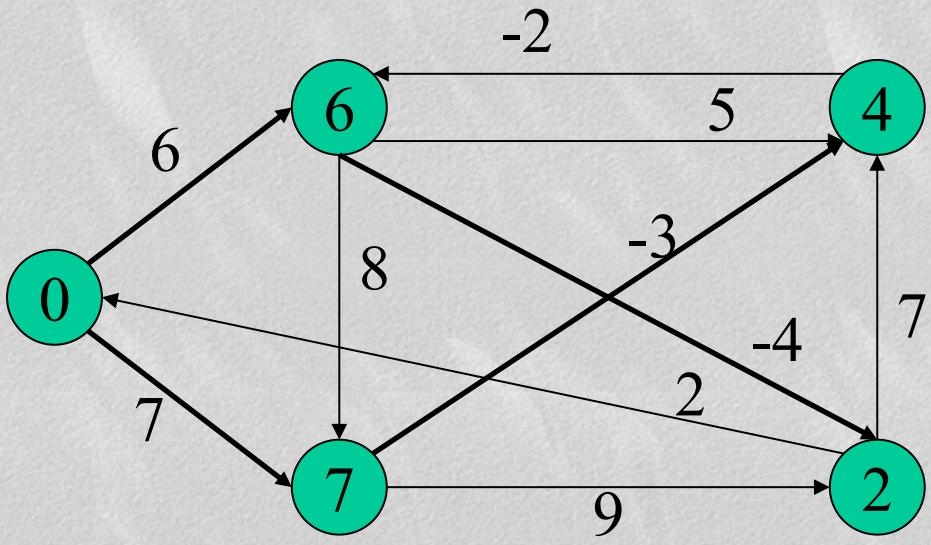
Graph Algorithms

Bellman-Ford Algorithm - Example



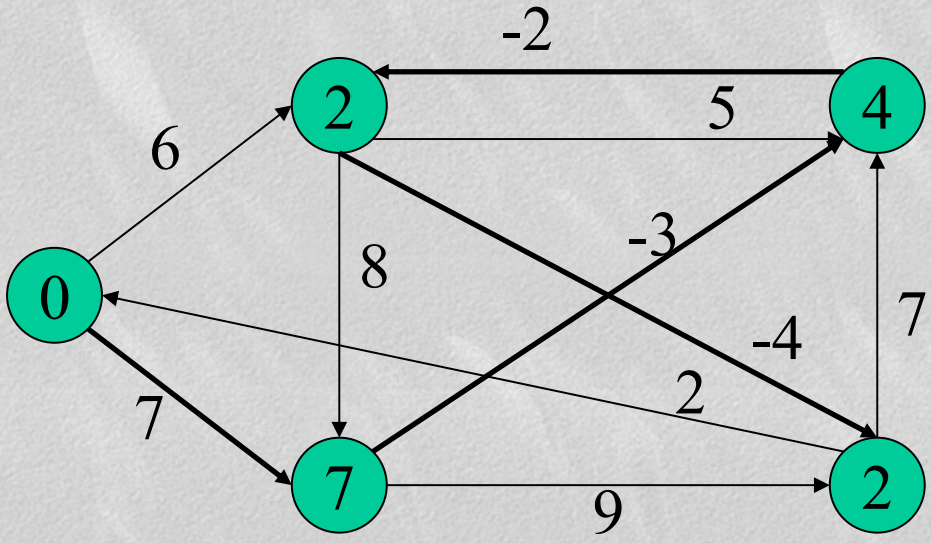
Graph Algorithms

Bellman-Ford Algorithm - Example



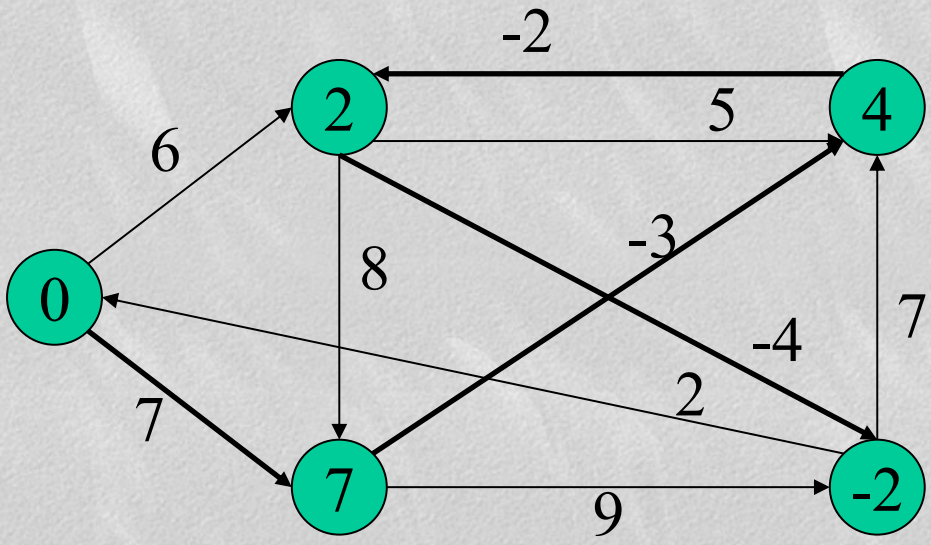
Graph Algorithms

Bellman-Ford Algorithm - Example



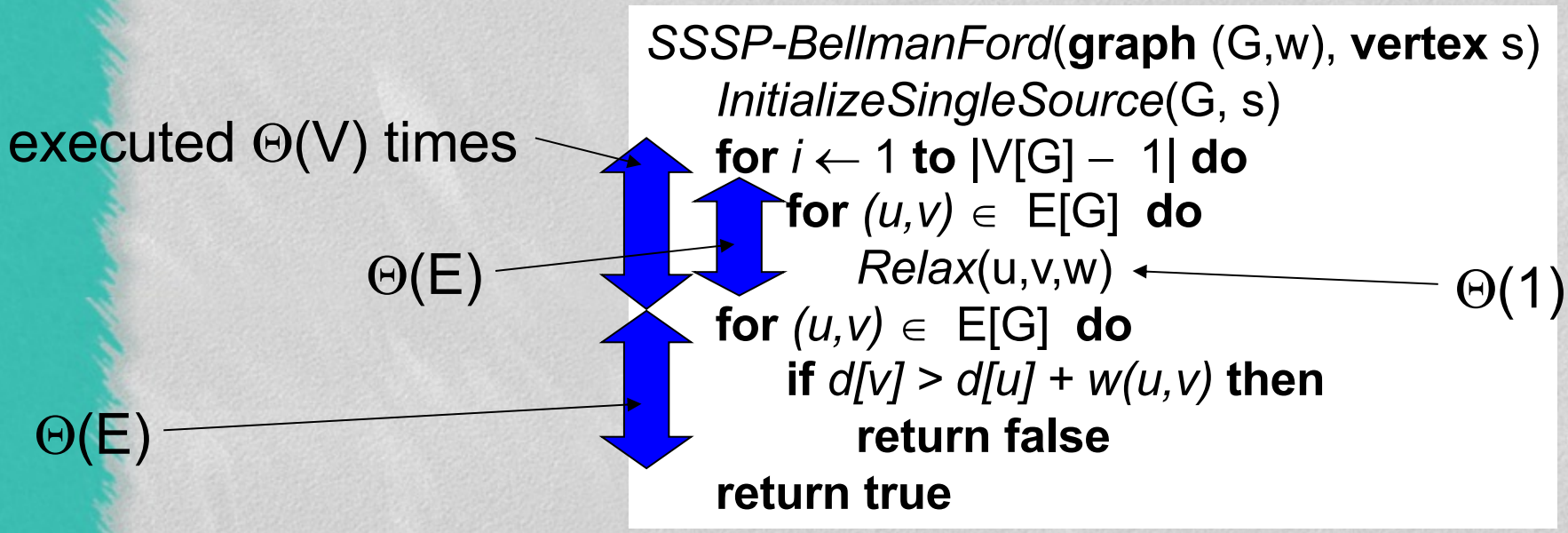
Graph Algorithms

Bellman-Ford Algorithm - Example



Graph Algorithms

Bellman-Ford Algorithm - Complexity



Graph Algorithms

Bellman-Ford Algorithm - Complexity

InitializeSingleSource $T_I(V,E) = \Theta(V)$

Relax $T_R(V,E) = \Theta(1)$?

SSSP-BellmanFord

$T(V,E) = T_I(V,E) + V\ E\ T_R(V,E) + E =$
$= \Theta(V) + V\ E\ \Theta(1) + E =$
$= \Theta(V\ E)$

Graph Algorithms

Bellman-Ford Algorithm - Correctness

Lemma 24.2

Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \rightarrow \mathbf{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, after the $|V| - 1$ iterations of the **for** loop of lines 2-4 of BELLMAN-FORD, we have $d[v] = \delta(s, v)$ for all vertices v that are reachable from s.

Proof We prove the lemma by appealing to the path-relaxation property. Consider any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any acyclic shortest path from s to v. Path p has at most $|V| - 1$ edges, and so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations of the **for** loop of lines 2-4 relaxes all E edges. Among the edges relaxed in the ith iteration, for $i = 1, 2, \ldots, k$, is (v_{i-1}, v_i). By the path-relaxation property, therefore, $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$.

[Adapted from T.Cormen, C.Leiserson, R. Rivest]

Graph Algorithms

Bellman-Ford Algorithm - Correctness

Theorem 24.4: (Correctness of the Bellman-Ford algorithm)

Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow \mathbf{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, we have $d[v] = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof Suppose that graph G contains no negative-weight cycles that are reachable from the source s. We first prove the claim that at termination, $d[v] = \delta(s, v)$ for all vertices $v \in V$. If vertex v is reachable from s, then Lemma 24.2 proves this claim. If v is not reachable from s, then the claim follows from the no-path property. Thus, the claim is proven. The predecessor-subgraph property, along with the claim, implies that G_π is a shortest-paths tree. Now we use the claim to show that BELLMAN-FORD returns TRUE. At termination, we have for all edges $(u, v) \in E$, and so none of the tests in line 6 causes BELLMAN-FORD to return FALSE. It therefore returns TRUE.

[Adapted from T.Cormen, C.Leiserson, R. Rivest]

Graph Algorithms

Bellman-Ford Algorithm - Correctness

$$
\begin{aligned}
d[v] &= \delta(s, v) \\
&\le \delta(s, u) + w(u, v) \text{ (by the triangle inequality)} \\
&= d[u] + w(u, v),
\end{aligned}
$$

Conversely, suppose that graph G contains a negative-weight cycle that is reachable from the source s; let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. Then,

$$
(24.1) \quad \sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 .
$$

Assume for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE. Thus, $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. Summing the inequalities around cycle c gives us

$$
\begin{aligned}
\sum_{i=1}^{k} d[v_i] &\le \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) \\
&= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) .
\end{aligned}
$$

[Adapted from T.Cormen, C.Leiserson, R. Rivest]

Graph Algorithms

Bellman-Ford Algorithm - Correctness

Since $v_0 = v_k$, each vertex in c appears exactly once in each of the summations $\sum_{i=1}^{k} d[v_i]$ and $\sum_{i=1}^{k} d[v_{i-1}]$, and so

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}] .$$

Moreover, by Corollary 24.3, $d[v_i]$ is finite for $i = 1, 2, \ldots, k$. Thus,

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i) ,$$

which contradicts inequality (24.1). We conclude that the Bellman-Ford algorithm returns TRUE if graph G contains no negative-weight cycles reachable from the source, and FALSE otherwise.

[Adapted from T.Cormen, C.Leiserson, R. Rivest]

Graph Algorithms

Shortest Paths in DAGs - SSSP-DAG

SSSP-DAG(**graph** (G,w), **vertex** s)

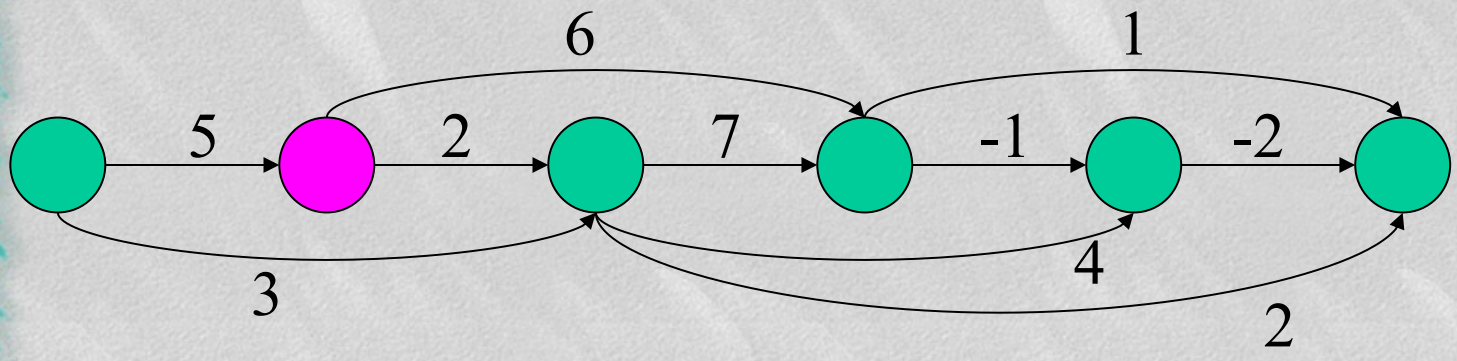
topologically sort vertices of G

InitializeSingleSource(G, s)

for each vertex u taken in topologically sorted order **do**
    for each vertex $v \in$ *Adj*[u] **do**
        Relax(u,v,w)

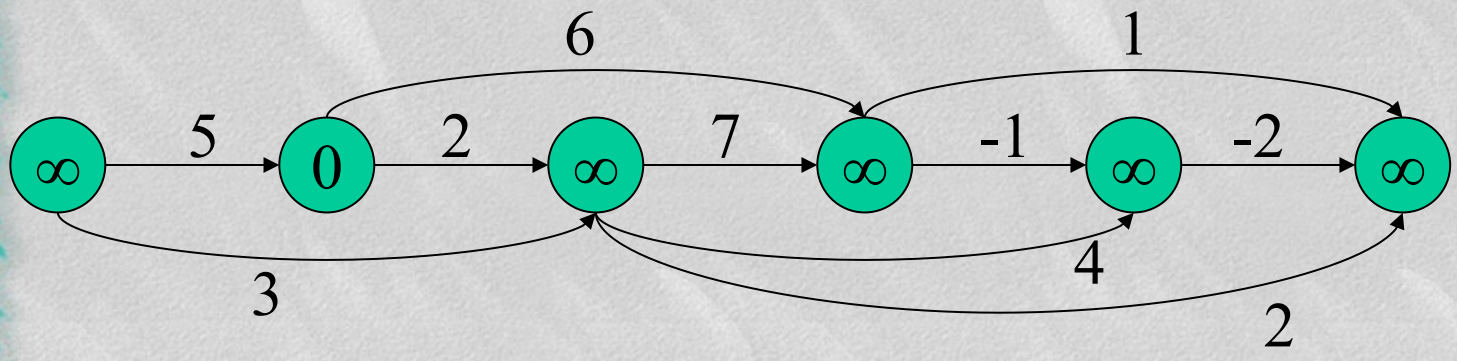
Graph Algorithms

Shortest Paths in DAGs – Example



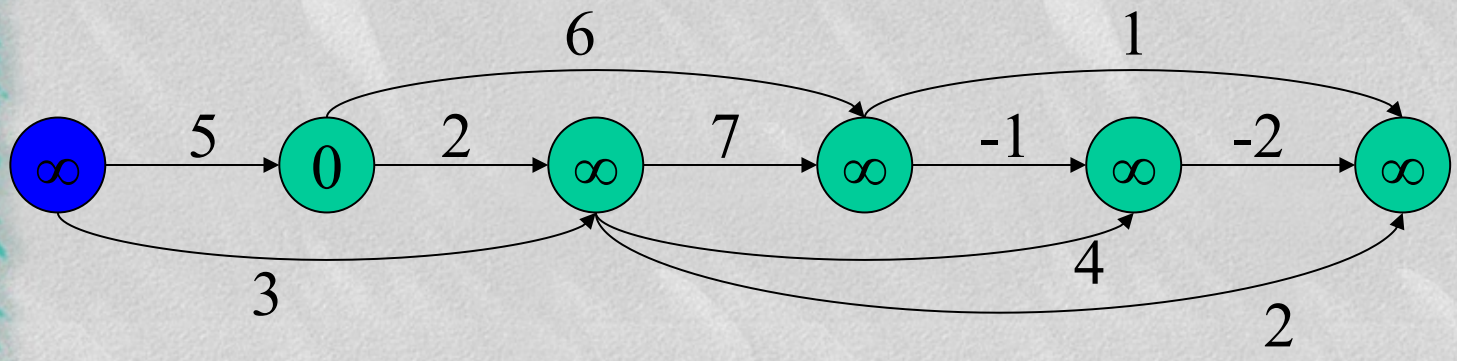
Graph Algorithms

Shortest Paths in DAGs – Example



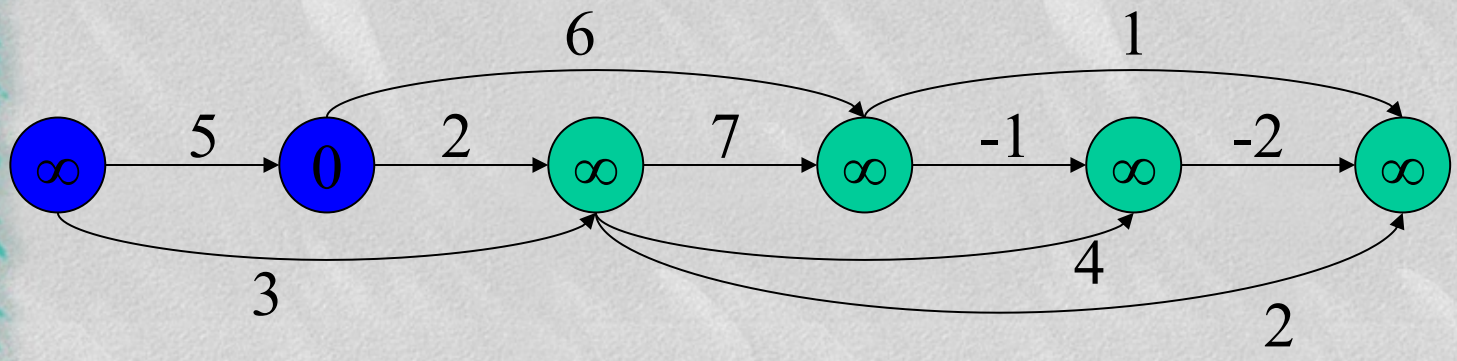
Graph Algorithms

Shortest Paths in DAGs – Example



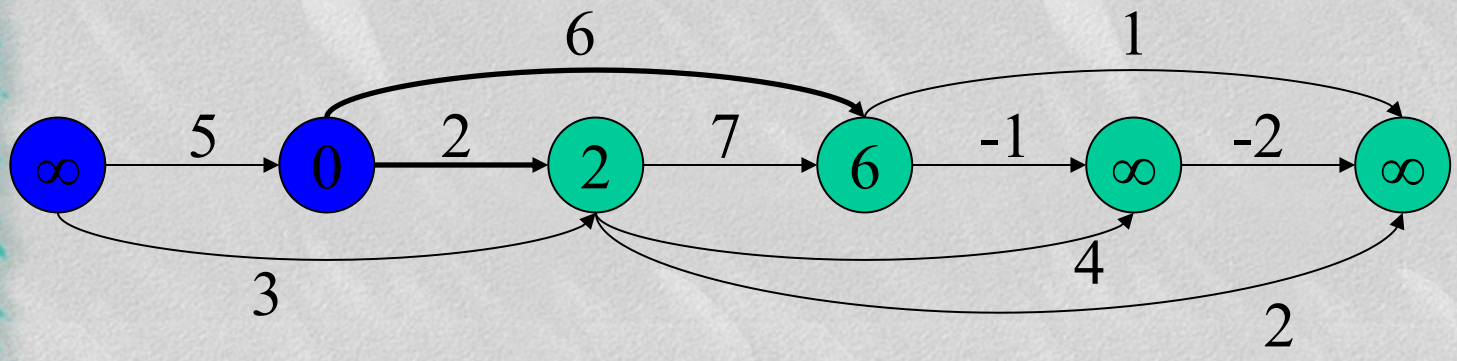
Graph Algorithms

Shortest Paths in DAGs – Example



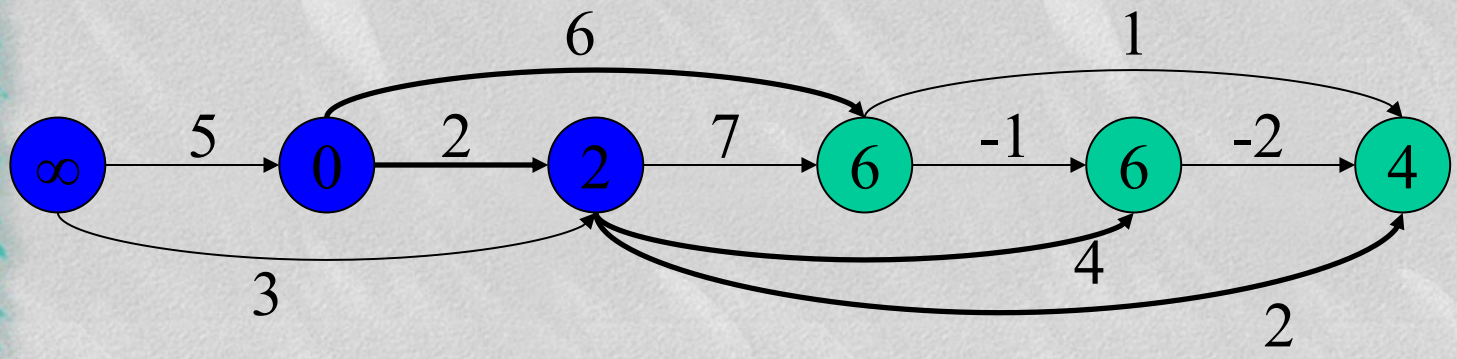
Graph Algorithms

Shortest Paths in DAGs – Example



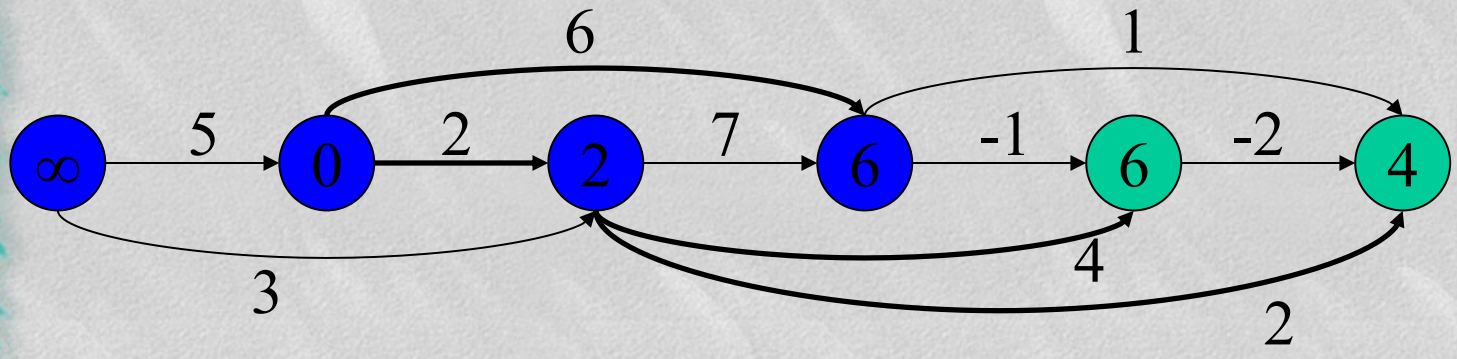
Graph Algorithms

Shortest Paths in DAGs – Example



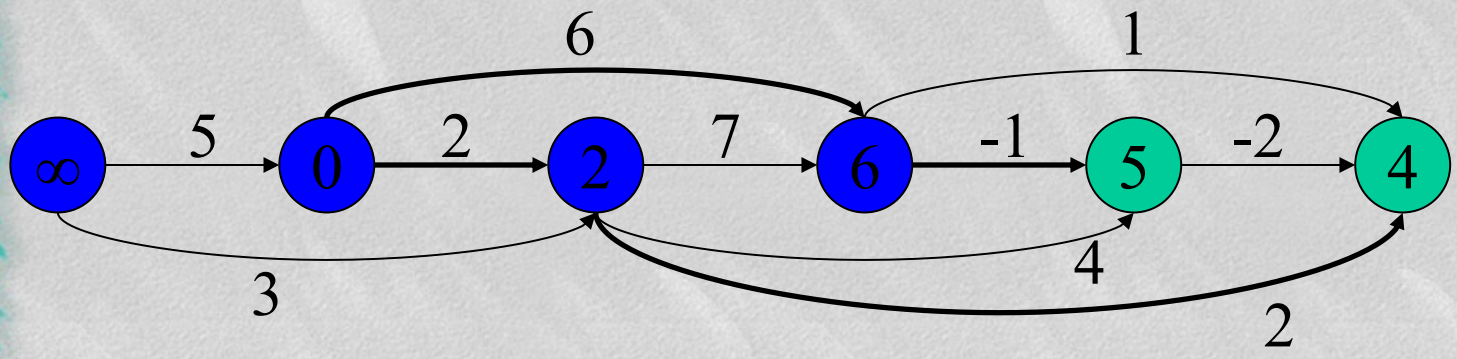
Graph Algorithms

Shortest Paths in DAGs – Example



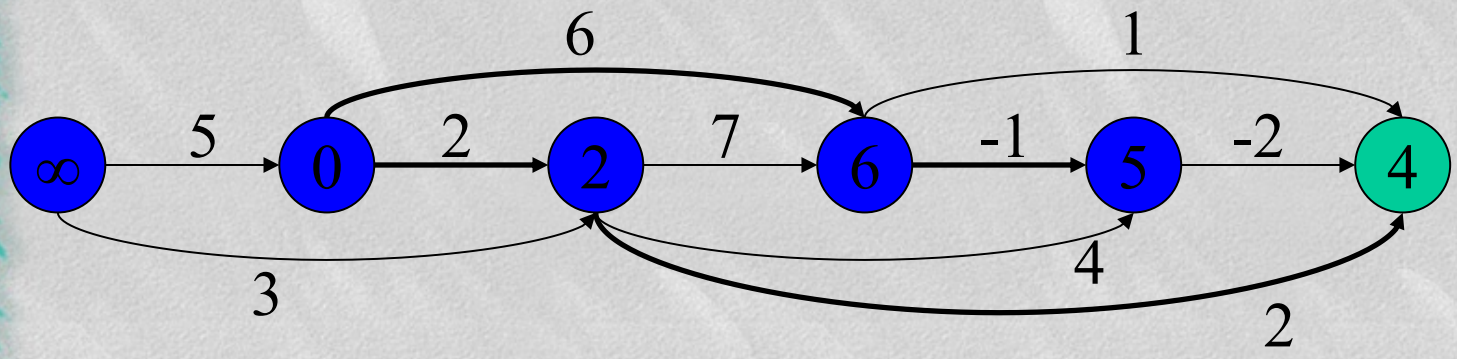
Graph Algorithms

Shortest Paths in DAGs - Example



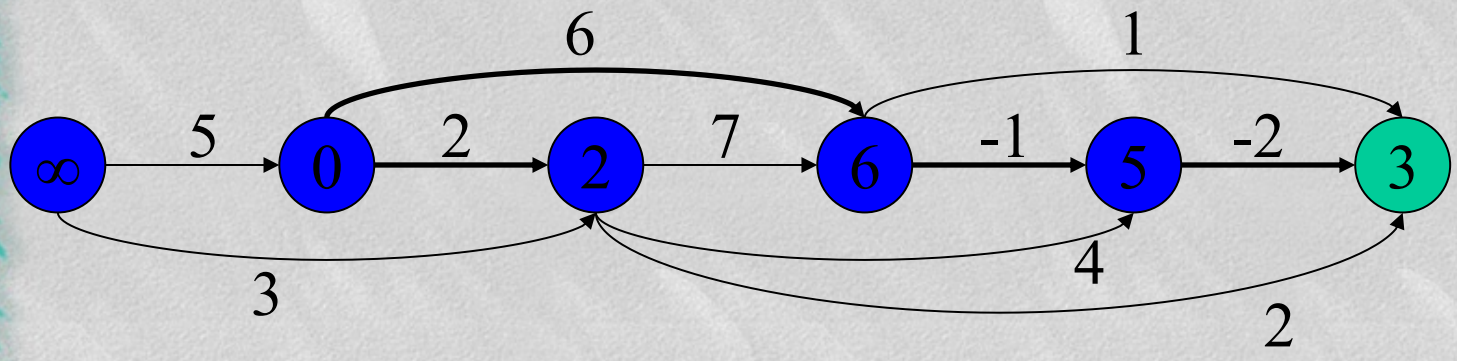
Graph Algorithms

Shortest Paths in DAGs – Example



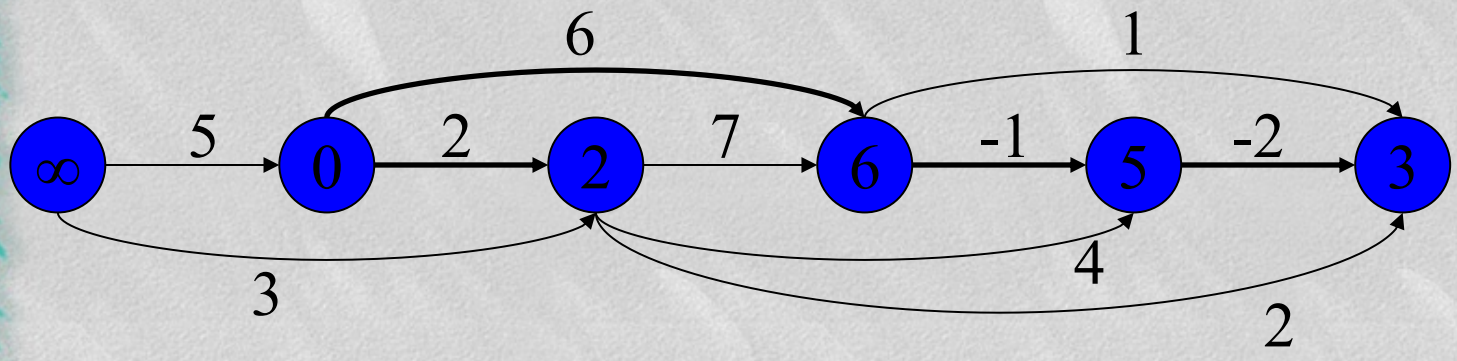
Graph Algorithms

Shortest Paths in DAGs – Example



Graph Algorithms

Shortest Paths in DAGs - Example



Graph Algorithms

Shortest Paths in DAGs - Complexity

```
SSSP-DAG(graph (G,w), vertex s)

    topologically sort vertices of G

    InitializeSingleSource(G, s)

    for each vertex u taken in topologically sorted order do
        for each vertex v ∈ Adj[u] do
            Relax(u,v,w)
```

$$T(V,E) = \Theta(V + E) + \Theta(V) + \Theta(V) + E\ \Theta(1) = \Theta(V + E)$$

Graph Algorithms

Shortest Path Application: Currency Conversion

Given currencies and exchange rates, what is best way to convert one ounce of gold to US dollars?

- 1 oz. gold ⇒ $327.25.
- 1 oz. gold ⇒ £208.10 ⇒ 208.10 (1.5714) ⇒ $327.00.
- 1 oz. gold ⇒ 455.2 Francs ⇒ 304.39 Euros ⇒ $327.28.

Currency	£	Euro	¥	Franc	$	Gold
UK Pound	1.0000	0.6853	0.005290	0.4569	0.6368	208.100
Euro	1.4599	1.0000	0.007721	0.6677	0.9303	304.028
Japanese Yen	189.050	129.520	1.0000	85.4694	120.400	39346.7
Swiss Franc	2.1904	1.4978	0.011574	1.0000	1.3929	455.200
US Dollar	1.5714	1.0752	0.008309	0.7182	1.0000	327.250
Gold (oz.)	0.004816	0.003295	0.0000255	0.002201	0.003065	1.0000

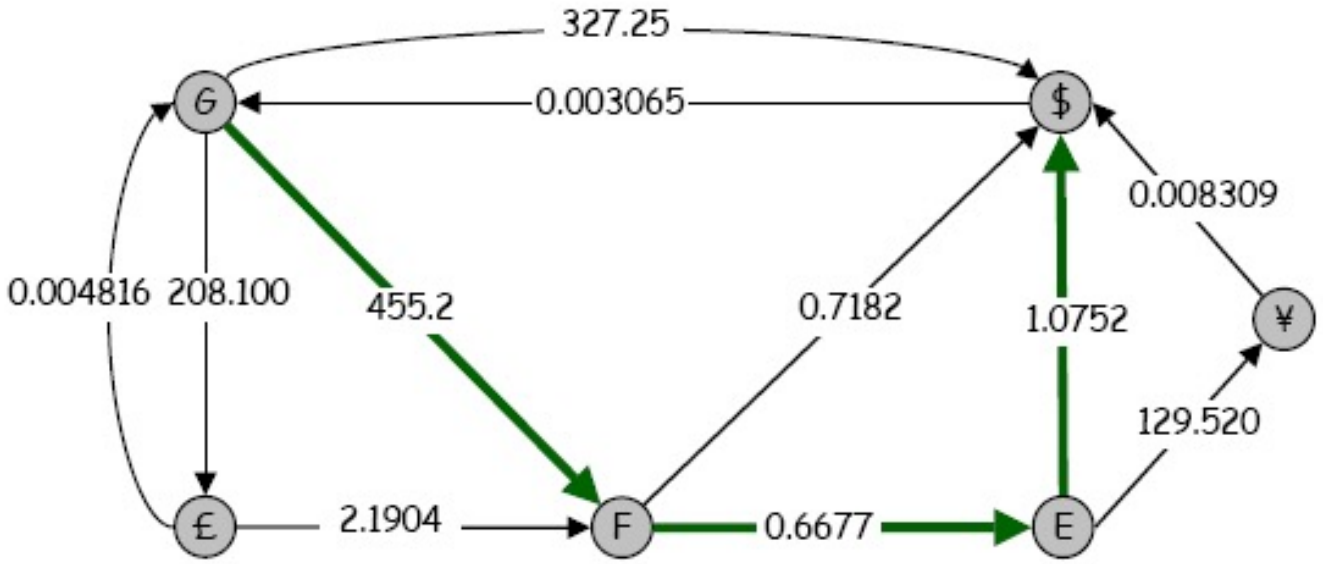
[Adapted from K.Wayne]

Graph Algorithms

Shortest Path Application: Currency Conversion

Graph formulation.

- Create a vertex for each currency.
- Create a directed edge for each possible transaction, with weight equal to the exchange rate.
- Find path that maximizes product of weights.



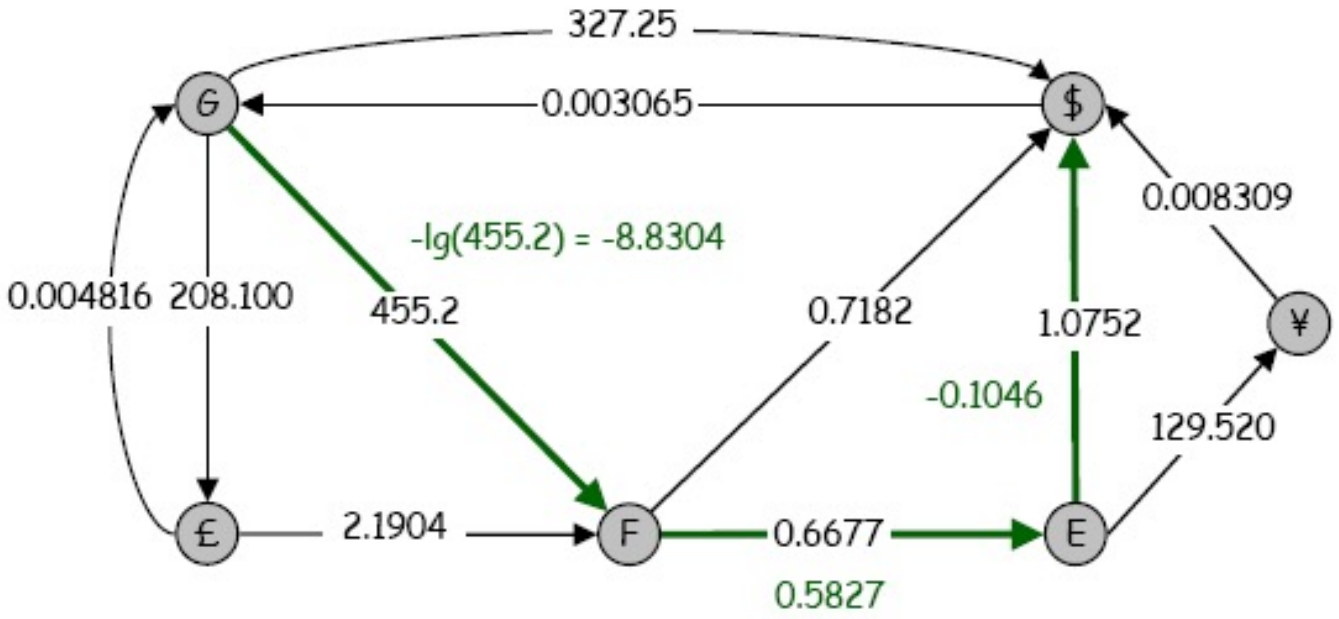
[Adapted from K.Wayne]

Graph Algorithms

Shortest Path Application: Currency Conversion

Reduction to shortest path problem.

- Let γ_{vw} be exchange rate from currency v to w.
- Let $c_{vw} = - \lg \gamma_{vw}$.
- Shortest path with costs c corresponds to best exchange sequence.



[Adapted from K.Wayne]

Graph Algorithms

Application of SSSP - constraint satisfaction

Linear programming

In the general ***linear-programming problem***, we are given an $m \times n$ matrix A, an m-vector b, and an n-vector c. We wish to find a vector x of n elements that maximizes the ***objective function*** $\sum_{i=1}^{n} c_i x_i$ subject to the m constraints given by $Ax \leq b$.

[Adapted from T.Cormen, C.Leiserson, R. Rivest]

Graph Algorithms

Application of SSSP - constraint satisfaction

For example, consider problem of finding the 5-vector $x = (x_i)$ that satisfies

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix} .$$

[Adapted from T.Cormen, C.Leiserson, R. Rivest]

Graph Algorithms

Application of SSSP - constraint satisfaction

This problem is equivalent to finding the unknowns x_i, for $i = 1, 2, \ldots, 5$, such that the following 8 difference constraints are satisfied:

$$
\begin{aligned}
x_1 - x_2 &\le 0, \\
x_1 - x_5 &\le -1, \\
x_2 - x_5 &\le 1, \\
x_3 - x_1 &\le 5, \\
x_4 - x_1 &\le 4, \\
x_4 - x_3 &\le -1, \\
x_5 - x_3 &\le -3, \\
x_5 - x_4 &\le -3.
\end{aligned}
\qquad (25.4)
$$

[Adapted from T.Cormen, C.Leiserson, R. Rivest]

Graph Algorithms

Application of SSSP - constraint satisfaction

More formally, given a system $Ax \leq b$ of difference constraints, the corresponding ***constraint graph*** is a weighted, directed graph $G = (V, E)$, where

$$V = \{v_0, v_1, \ldots, v_n\}$$

and

$$E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ is a constraint}\}$$
$$\cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_n)\} .$$

[Adapted from T.Cormen, C.Leiserson, R. Rivest]

Graph Algorithms

Application of SSSP - constraint satisfaction

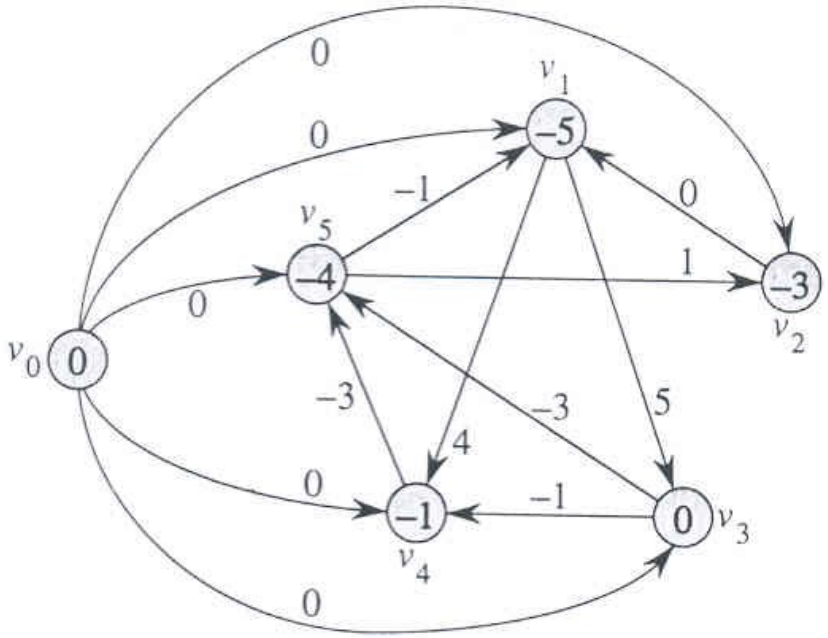


Figure 25.9 The constraint graph corresponding to the system (25.4) of difference constraints. The value of $\delta(v_0, v_i)$ is shown in each vertex v_i. A feasible solution to the system is $x = (-5, -3, 0, -1, -4)$.

[Adapted from T.Cormen, C.Leiserson, R. Rivest]

Graph Algorithms

Application of SSSP - constraint satisfaction

Theorem 25.17

Given a system $Ax \leq b$ of difference constraints, let $G = (V, E)$ be the corresponding constraint graph. If G contains no negative-weight cycles, then

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n)) \tag{25.5}$$

is a feasible solution for the system. If G contains a negative-weight cycle, then there is no feasible solution for the system.

[Adapted from T.Cormen, C.Leiserson, R. Rivest]

Graph Algorithms

All-Pairs Shortest Paths

Given graph (directed or undirected) G = (V,E) with weight function w: E $\rightarrow$ **R** find for all pairs of vertices u,v $\in$ V the minimum possible weight for path from u to v.

- Finding the shortest path between all pairs of vertices in a graph
- For the general case (negative weights allowed), the problem can be solved by running the Bellman-Ford algorithm $|V|$ times, once for each vertex as the source

  Time Complexity: $O(V^2E)$. If the graph is dense: $O(V^4)$

- The **Floyd-Warshall** dynamic programming algorithm solves the same problem in $O(V^3)$
  - Competitive for dense graphs
  - Uses adjacency matrices, as opposed to adjacency lists

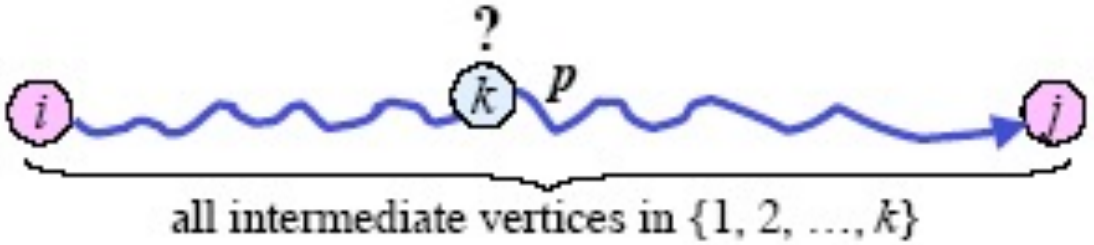
[Adapted from M.Jacome]

Graph Algorithms

Floyd-Warshall Algorithm - Idea

- Let the vertices of G be $V = \{1, 2, \ldots, n\}$ and consider $\{1, 2, \ldots, k\}$ be a subset of vertices for some k.
- For any pair of vertices $i, j \in V$ consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$ and let p be a minimum-weight path among them - **a simple path** (no negative weight cycles).

The Floyd-Warshall algorithm exploits a relationship between path p and a **shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\text{-}1\}$**. The relationship depends on whether or not k is an intermediate vertex of p.

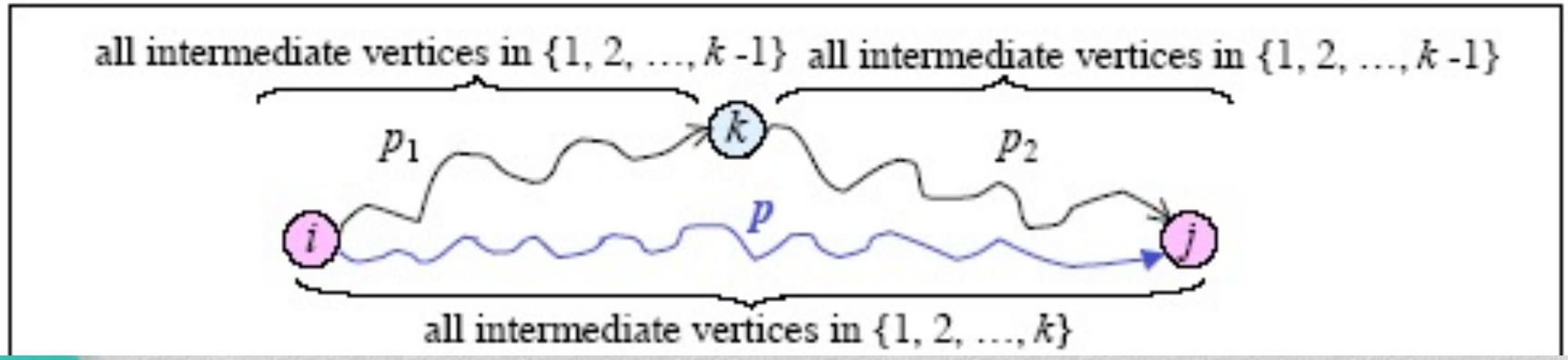


[Adapted from M.Jacome]

Graph Algorithms

Floyd-Warshall Algorithm - Idea

- **If k is not in p**, then a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\text{-}1\}$ is **also a shortest path in the set $\{1, 2, \ldots, k\}$.**
- **If k is in p**, then we break down p into p_1 and p_2, where
  - p_1 is the shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k\text{-}1\}$
  - p_2 is the shortest path from k to j with all intermediate vertices in the set $\{1, 2, \ldots, k\text{-}1\}$



[Adapted from M.Jacome]

Graph Algorithms

Floyd-Warshall Algorithm - Idea

$d_{s,t}^{(i)}$ – the shortest path from s to t containing only vertices $v_1, \ldots, v_i$

$d_{s,t}^{(0)} = w(s,t)$

$$d_{s,t}^{(k)} = \begin{cases} w(s,t) & \text{if } k = 0 \\ \min\{d_{s,t}^{(k-1)}, d_{s,k}^{(k-1)} + d_{k,t}^{(k-1)}\} & \text{if } k > 0 \end{cases}$$

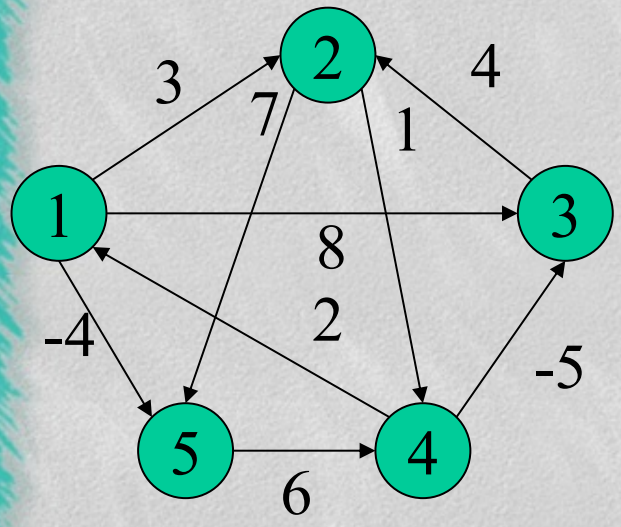
Graph Algorithms

Floyd-Warshall Algorithm - Algorithm

FloydWarshall(**matrix** W, **integer** *n*)
    for *k* $\leftarrow$ 1 **to** *n* **do**
        for *i* $\leftarrow$ 1 **to** *n* **do**
            for *j* $\leftarrow$ 1 **to** *n* **do**
                $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
    return $D^{(n)}$

Graph Algorithms

Floyd-Warshall Algorithm - Example



W

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

Graph Algorithms

Floyd-Warshall Algorithm - Example

$D^{(0)}$

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

$\Pi^{(0)}$

0	0	0		0
	0		0	0
	0	0		
0		0	0	
			0	0

Graph Algorithms

Floyd-Warshall Algorithm - Example

$D^{(1)}$

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	5	-5	0	-2
∞	∞	∞	6	0

$\Pi^{(1)}$

0	0	0		0
	0		0	0
	0	0		
0	1	0	0	1
			0	0

Graph Algorithms

Floyd-Warshall Algorithm - Example

$D^{(2)}$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	5	-5	0	-2
∞	∞	∞	6	0

$\Pi^{(2)}$

0	0	0	2	0
	0		0	0
	0	0	2	2
0	1	0	0	1
			0	0

Graph Algorithms

Floyd-Warshall Algorithm - Example

$D^{(3)}$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	-1	-5	0	-2
∞	∞	∞	6	0

$\Pi^{(3)}$

0	0	0	2	0
	0		0	0
	0	0	2	2
0	3	0	0	1
			0	0

Graph Algorithms

Floyd-Warshall Algorithm – Example

$D^{(4)}$

0	3	-1	4	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

$\Pi^{(4)}$

0	0	4	2	0
4	0	4	0	1
4	0	0	2	1
0	3	0	0	1
4	3	4	0	0

Graph Algorithms

Floyd-Warshall Algorithm - Example

$D^{(5)}$

0	3	-1	2	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

$\Pi^{(5)}$

0	0	4	5	0
4	0	4	0	1
4	0	0	2	1
0	3	0	0	1
4	3	4	0	0

Graph Algorithms

Floyd-Warshall Algorithm - Extracting the shortest paths

The predecessor pointers `pred[i, j]` can be used to extract the final path. The idea is as follows.

Whenever we discover that the shortest path from i to j passes through an intermediate vertex k, we set $pred[i, j] = k$.

If the shortest path does not pass through any intermediate vertex, then $pred[i, j] = nil$.

To find the shortest path from i to j, we consult $pred[i, j]$. If it is nil, then the shortest path is just the edge (i, j). Otherwise, we recursively compute the shortest path from i to $pred[i, j]$ and the shortest path from $pred[i, j]$ to j.

[Adapted from S.Cheng]

Graph Algorithms

Floyd-Warshall Algorithm - Complexity

```
FloydWarshall(matrix W, integer n)
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
    return D^(n)
```

3 **for** cycles, each executed exactly n times

$T(V,E) = \Theta(n^3) = \Theta(V^3)$

Graph Algorithms

All-Pairs Shortest Paths -Johnson's algorithm

Johnson's Algorithm

- Finds shortest paths between all pairs in $O(V^2 \lg V + VE \lg V)$ time
  - For sparse graphs ($|E| << |V|$) it is asymptotically better than Floyd-Warshall algorithm ($O(V^3)$)

$$O(V^2 \lg V + VE \lg V) = O(V^2 \lg V) \text{ for sparse graphs}$$

- Johnson's algorithm:
  - uses the technique of **reweighting, so that all edges are positive**
  - then applies Dijkstra's algorithm, once for each vertex
    - Using binary heaps, Dijkstra's algorithm takes $O(V \lg V + E \lg V)$

[Adapted from M.Jacome]

Graph Algorithms

All-Pairs Shortest Paths - Reweighting

Reweighting

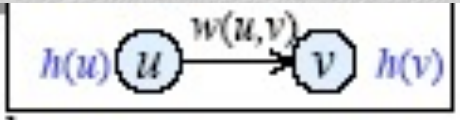
- The **new set of edge weights w'** must satisfy two important properties:
  - (1) For all pairs of vertices $u, v \in V$, a shortest path from u to v using the weight function w is **also** a shortest path using the weight function w'
  - (2) For all edges (u, v), the new weight $w'(u, v)$ is nonnegative

[Adapted from M.Jacome]

Graph Algorithms

All-Pairs Shortest Paths - Reweighting

(1) Reweighting does not change the shortest paths



- Let $h: V \rightarrow \mathbf{R}$ be any function mapping vertices to real numbers
- For each edge $(u,v) \in E$ define $w'(u,v) = w(u,v) + h(u) - h(v)$

Let p be a **path from vertex v_0 to vertex v_k**. Then $w(p) = \delta(v_0, v_k)$ if and only if $w'(p) = \delta'(v_0, v_k)$.

Proof

$$w'(p) = \sum_{i=1}^{k} w'(v_{i-1}, v_i) = \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) = w(p) + h(v_0) - h(v_k)$$

Showing by contradiction that $w(p) = \delta(v_0, v_k) \Rightarrow w'(p) = \delta'(v_0, v_k)$

Suppose that there is **a shorter path p'** from v_0 to vertex v_k using w'.

Then $w'(p') < w'(p)$

$$w(p') + h(v_0) - h(v_k) < w(p) + h(v_0) - h(v_k)$$

the same "cst" is added to all paths between two nodes

$w(p') < w(p)$, but this contradicts the assumption that p is a shortest path for weight function w. √

[Adapted from M.Jacome]

Graph Algorithms

All-Pairs Shortest Paths - Reweighting

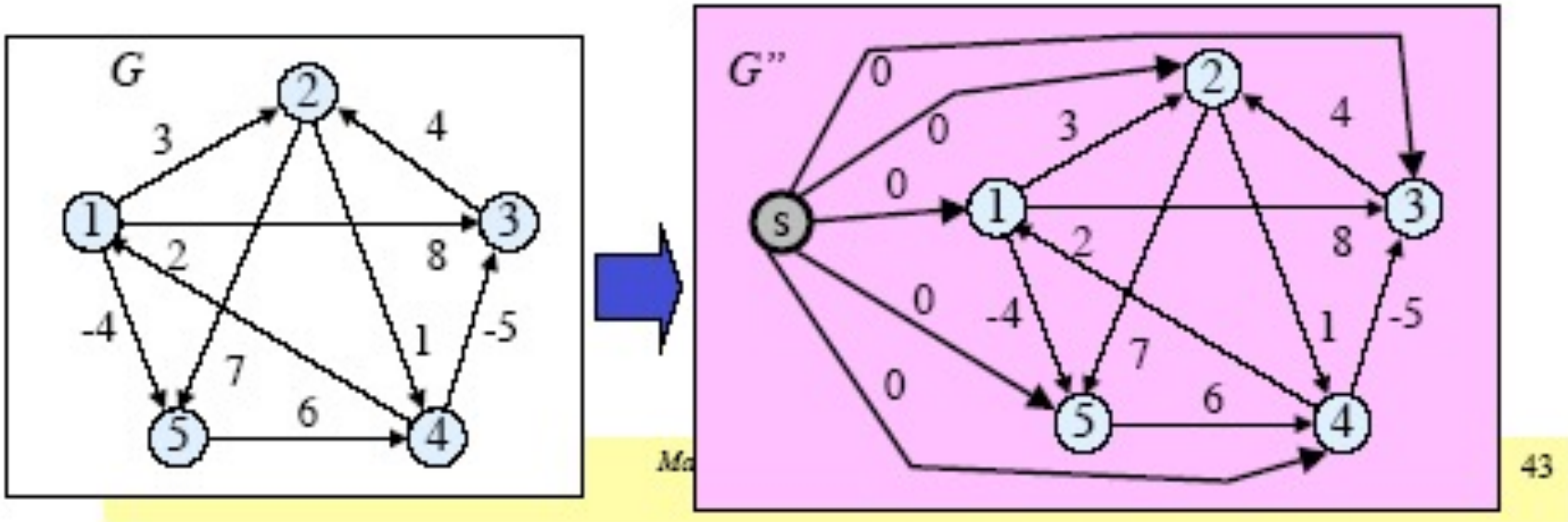
Bellman-Ford algorithm is used to determine $\delta(s, v)$ for all $v \in V$"

- Create a new graph $G" = (V", E")$, where:
  - $V" = V \cup \{s\}$
  - $E = E \cup \{(s,v) : v \in V\}$
  - $w(s,v) = 0$ for all $v \in V\}$

Define $h(v) = \delta(s, v)$ for all $v \in V$"

$\Rightarrow$ $h(v) \leq h(u) + w(u,v)$ for all $(u,v) \in E$"

$\therefore$ $w'(u,v) = w(u,v) + h(u) - h(v) \geq 0$ $\surd$



[Adapted from M.Jacome]