COMP2007/2907 – Algorithms

Course page: Blackboard and Piazza (or Ed)

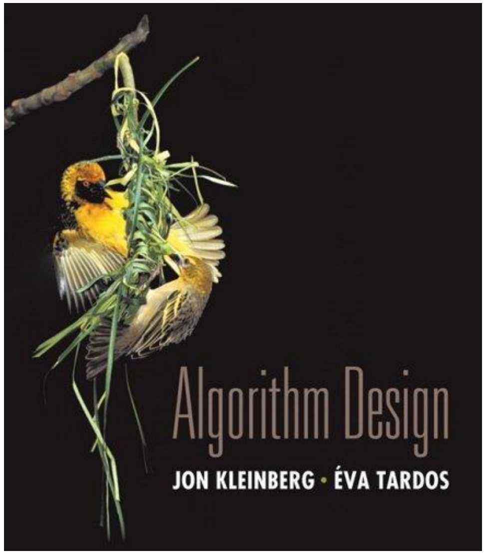
Lecturer: Joachim Gudmundsson
Level 4, Room 416, School of IT
joachim.gudmundsson@sydney.edu.au
Ph. 9351 4494

Tutors:

Xavi Holt (TA)	Yilun He
Natalie Tridgell	Joe Godbehere
Jessica McBroom	Patrick Eades
Anton Jurisevic	Shumin Kong
Gengxing Wang	Hisham Husein
Mingshen Cai	

Course book:

J. Kleinberg and E. Tardos
Algorithm Design
Addison-Wesley



Outline:

12 lectures
5 assignments
10+1 quizzes
Exam

Tutorials:

12 tutorials

Aims of this unit

- This unit provides an introduction to the design and analysis of algorithms. Its main aims are
  - (i) learn how to develop algorithmic solutions to computational problem
  - (ii) develop understanding of algorithm efficiency.

- Assumes basic knowledge of discrete math
  - graphs
  - big O notation
  - proof techniques

Assessment:

- Quizzes 20% (average of best 8 out of 10)
- Each assignment 6% (5 assignments – total 30%)
- Exam 50% (minimum 40% required to pass)

Assignments submitted via Blackboard. Turnitin will be used to check every submission

Collaboration:

- General ideas – Yes!
- Formulation and writing – No!

Read Academic Dishonesty and Plagiarism.

Assignments

- There will be 5 homework assignments
- The objective of these is to teach problem solving skills
- Each assignment represents 6% of your final mark. Late submissions will be penalized by 25% of the full marks per day.

For example, say you get 80% on your assignment:
If submitted on time = 4.8
Late but within 24 hours = 4.8 * 0.75 = 3.6
Between 24 and 48 hours = 4.8 * 0.5 = 2.4
Between 48 and 72 hours = 4.8 * 0.25 = 1.2
More than 72 hours = 4.8 * 0 = 0

Final exam

- The final will be 2.5 hours long. It will consist of 6 problems similar to those seen in the tutorials and assignments
- The final will test your problem solving skills
- There is a 40% exam barrier
- The final exam represents 50% of your final mark
- Our advice is that you work hard on the assignments throughout the semester. It's the best preparation for the final.

Tutorials

- To get the most out of the tutorial, try to solve as many problems as you can *before* the tutorial. Your tutor is there to help you out if you get stuck, not to lecture.

- We will post solutions to tutorials (see Ed).

Preliminary schedule

- **Lecture 1** [Mon 31 July]: Introduction
- **Lecture 2** [Mon 7 Aug]: Graphs
- **Lecture 3** [Mon 14 Aug]: Greedy algorithms
- **Lecture 4** [Mon 21 Aug]: Divide & Conquer algorithms
- **Lecture 5** [Mon 28 Aug]: Sweepline algorithms
- **Lecture 6** [Mon 4 Sep]: Dynamic programming: basic techniques
- **Lecture 7** [Mon 11 Sep]: Dynamic programming: interval scheduling and Bellman-Ford
- **Lecture 8** [Mon 18 Sep]: Network flows I: Theory

Mon 25 Sep: University break

Mon 2 Oct: Labour Day

- **Lecture 9** [Mon 9 Oct]: Network flows II: Applications
- **Lecture 10** [Mon 16 Oct]: NP and intractability
- **Lecture 11** [Mon 23 Oct]: Coping with hardness
- **Lecture 12** [Mon 30 Oct]: Recap

COMP2007/2907: Algorithms

Algorithms then, and now

What's in an algorithm?

- Algorithms can have huge impact
- For example –

  A report to the White House from 2010 includes the following.

  – Professor Martin Grotschel
    - A benchmark production planning model solved using linear programming would have taken 82 years to solve in 1988, using the computers and the linear programming algorithms of the day.
    - Fifteen years later, in 2003, this same model could be solved in roughly 1 minute, an improvement by a factor of roughly 43 million! [Extreme case, but even the average factor is very high.]

What's in an algorithm?

- In 2003 there were examples of problems that we can solve 43 million times faster than in 1988
  - This is because of better hardware and better algorithms

What's in an algorithm?

- In 1988
  - Intel 386 and 386SX
    - About 275,000 transistors
    - clock speeds of 16MHz, 20MHz, 25MHz, and 33MHz
  - MSDOS 4.0 and windows 2.0
  - VGA
- In 2003
  - Pentium M
    - About 140 million transistors
    - Up to 2.2 GHz
  - AMD Athlon 64
  - Windows XP

What's in an algorithm?

- In a report to the White House from 2010 includes the following.
  - Professor Martin Grotschel:
    - A benchmark production planning model solved using linear programming would have taken 82 years to solve in 1988, using the computers and the linear programming algorithms of the day.
    - Fifteen years later, in 2003, this same model could be solved in roughly 1 minute, an improvement by a factor of roughly 43 millions

Observation:

- Hardware: 1,000 times improvement
- Algorithms: 43,000 times improvement

Efficient algorithms

- Efficient algorithms produce results within available resource limits
- In practice
  - Low polynomial time algorithms behave well
  - Exponential running times are infeasible except for very small instances, or carefully designed algorithms
- Performance depends on many obvious factors
  - Hardware
  - Software
  - Algorithm
  - Implementation of the algorithm
- This unit: Algorithms

Efficient algorithms

- Efficient algorithms “do the job” the way you want them to...
  - Do you need the exact solution?
  - Are you dealing with some special case and not with a general problem?
  - Is it ok if you miss the right solution sometimes?

Efficient algorithms

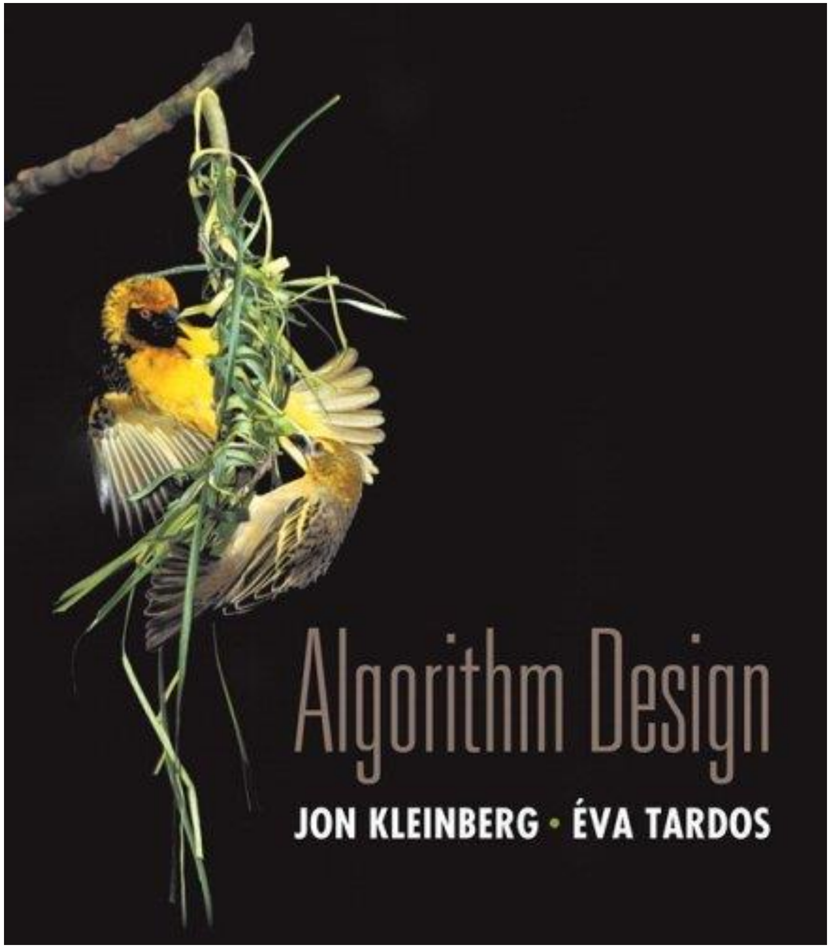
- Complex, highly sophisticated algorithms can greatly improve performance

  But…

- Reasonably good algorithmic solutions that avoid simple, or “lazy” mistakes, can have a much bigger impact!

List of topics

- Greedy algorithms
- Divide and conquer
- Sweepline
- Dynamic programming
- Network flows
- Mincut theorems
- Approximation
- Optimization problems

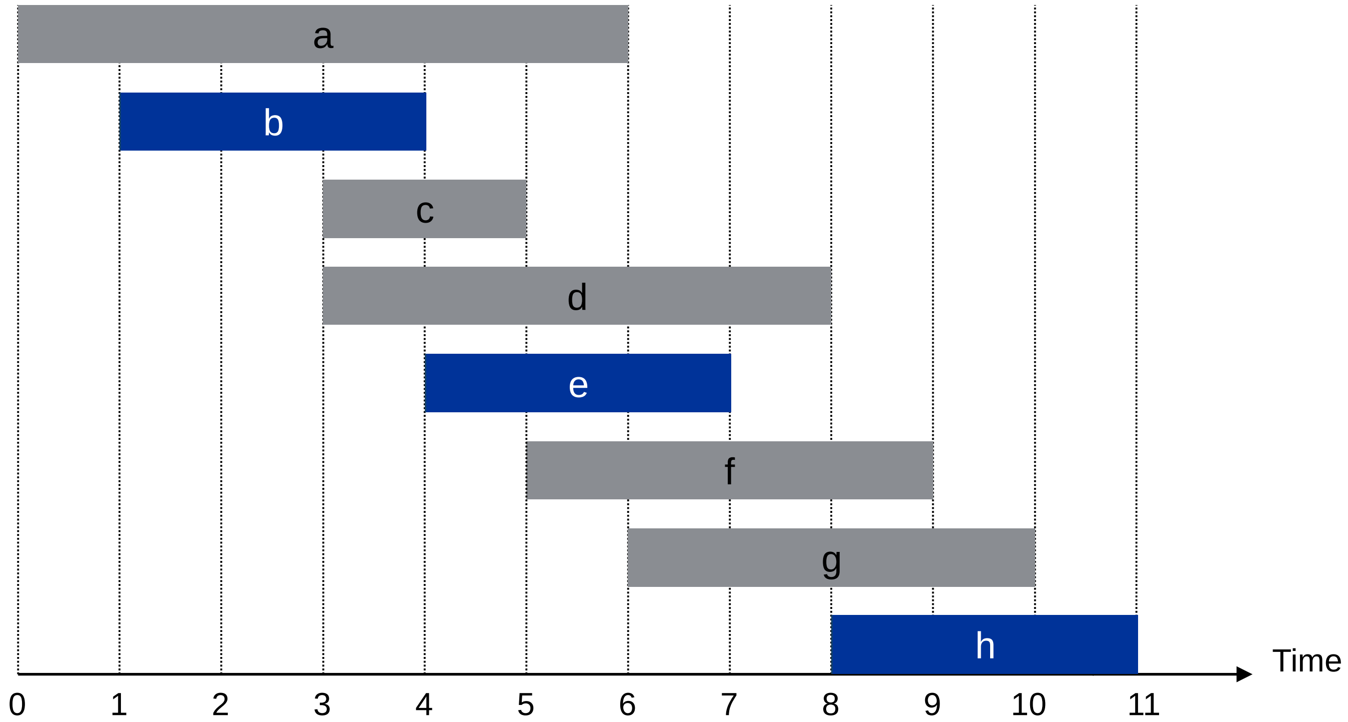


Introduction: Some Representative Problems

Four Representative Problems: Interval Scheduling

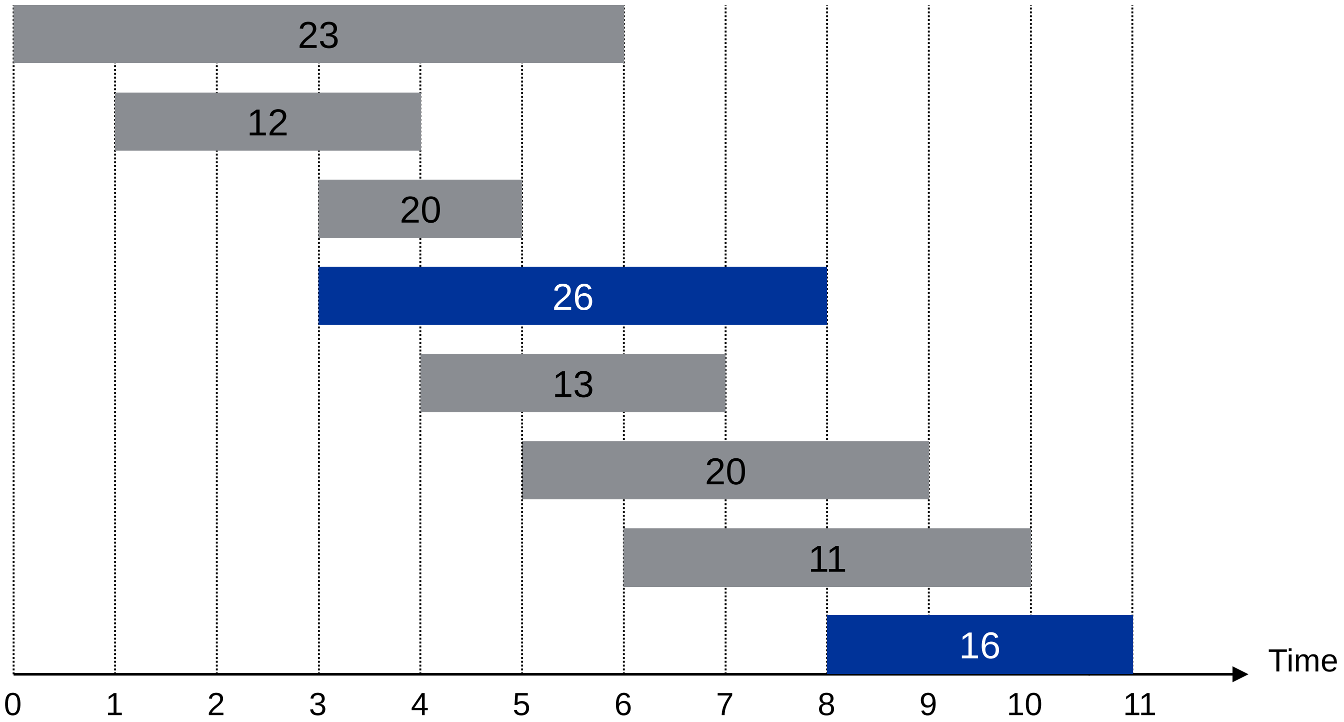
- Input. Set of jobs with start times and finish times.
- Goal. Find maximum cardinality subset of mutually compatible jobs.

jobs don't overlap



Weighted Interval Scheduling

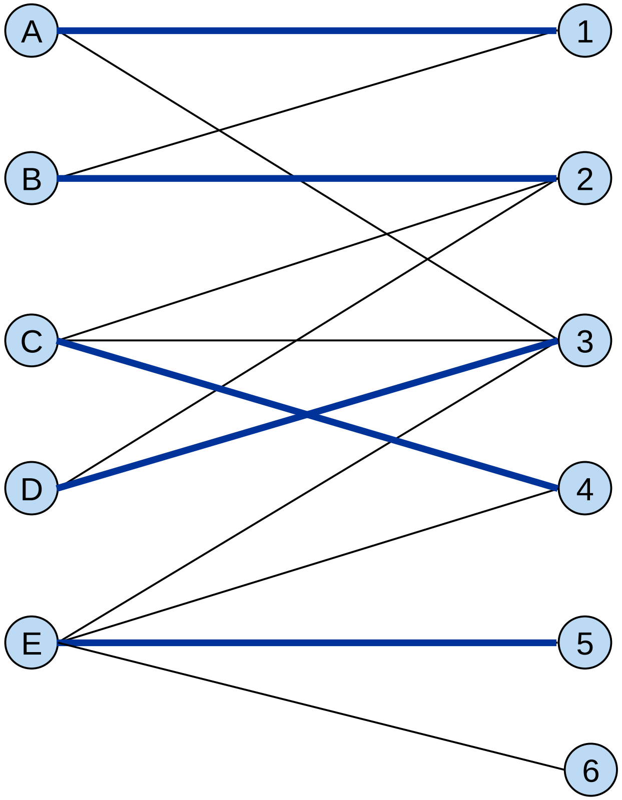
- Input. Set of jobs with start times, finish times, and weights.
- Goal. Find maximum weight subset of mutually compatible jobs.



Bipartite Matching

› Input. Bipartite graph.

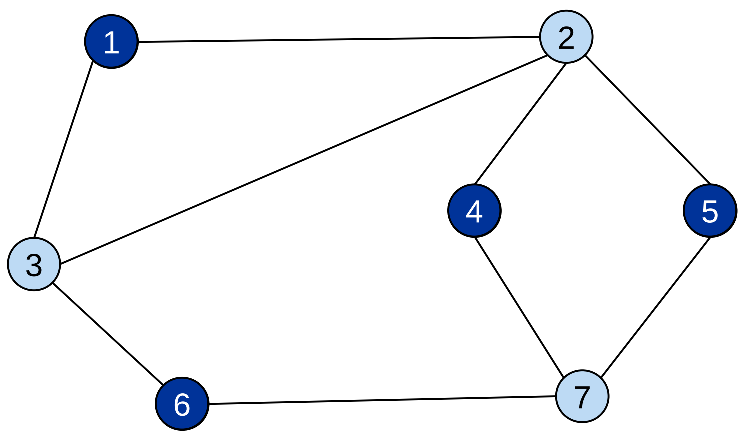
› Goal. Find maximum cardinality matching.



Independent Set

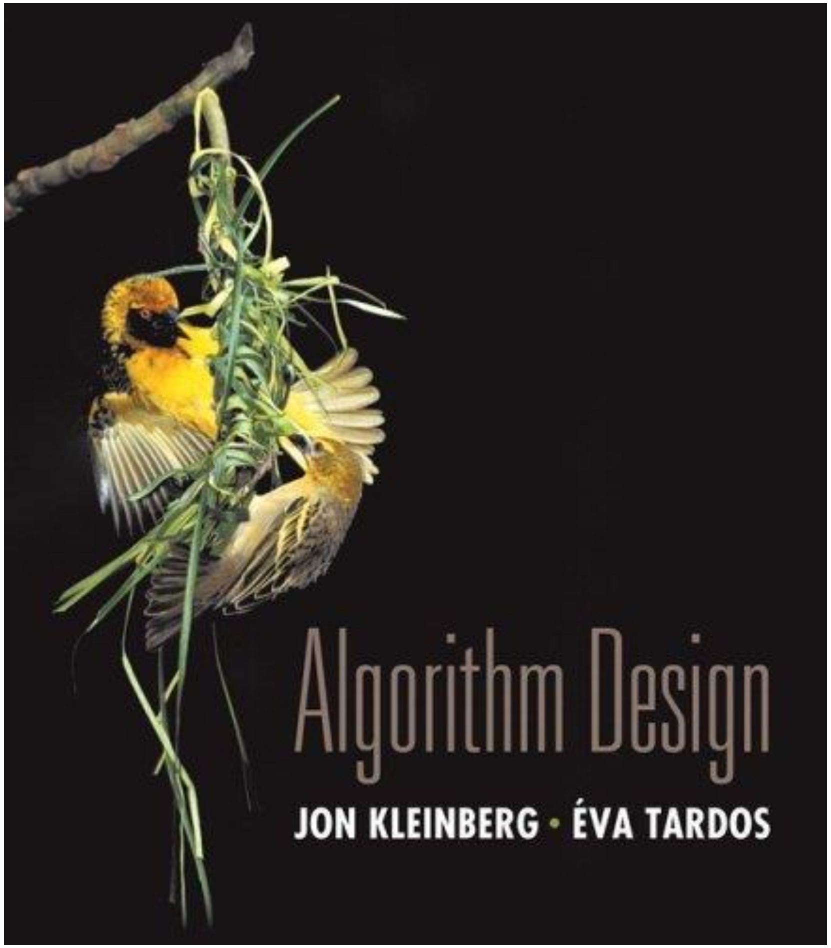
- Input. Graph.
- Goal. Find maximum cardinality independent set.

subset of nodes such that no two
joined by an edge



Four Representative Problems

- Interval scheduling: n log n greedy algorithm.
- Weighted interval scheduling: n log n dynamic programming algorithm.
- Bipartite matching: n^k maxflow based algorithm.
- Independent set: NP-complete.



Algorithm Analysis
&
Data Structures

Polynomial-Time

- Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.
  - Typically takes 2^N time or worse for inputs of size N.
  - Unacceptable in practice.

- Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C.

> There exists constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by $c N^d$ steps.

- Definition: An algorithm is poly-time if the above scaling property holds.

Worst-Case Analysis

- Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.
  - Generally captures efficiency in practice.
  - Draconian view, but hard to find effective alternative.

- Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.
  - Hard (or impossible) to accurately model real instances by random distributions.
  - Algorithm tuned for a certain distribution may perform poorly on other inputs.

Worst-Case Polynomial-Time

› Definition: An algorithm is efficient if its running time is polynomial.

› Justification: It really works in practice!

- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

› Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

simplex method

Unix grep

Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Asymptotic Order of Growth

› Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

› Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

› Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

› Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Notation

- Slight abuse of notation. $T(n) = O(f(n))$.
  - Asymmetric:
    - $f(n) = 5n^3$; $g(n) = 3n^3$
    - $f(n) = O(n^3) = g(n)$
    - but $f(n) \neq g(n)$.
  - Better notation: $T(n) \in O(f(n))$.

- Meaningless statement. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.
  - Statement doesn't "type-check."
  - Use Ω for lower bounds.

Properties

- Transitivity
  - If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
  - If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
  - If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

- Additivity
  - If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
  - If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
  - If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$.

Asymptotic Bounds for Some Common Functions

- **Polynomials.** $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

- **Polynomial time.** Running time is $O(n^d)$ for some constant d independent of the input size n.

- **Logarithms.** $O(\log_a n) = O(\log_b n)$ for any constants a, b > 0.

  can avoid specifying the base

- **Logarithms.** For every x > 0, $\log n = O(n^x)$.

  log grows slower than every polynomial

- **Exponentials.** For every r > 1 and every d > 0, $n^d = O(r^n)$.

  every exponential grows faster than every polynomial

Linear Time: O(n)

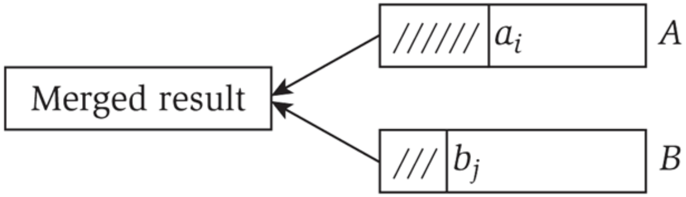
› Linear time. Running time is at most a constant factor times the size of the input.

› Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a₁
for i = 2 to n
{
    if (aᵢ > max)
        max ← aᵢ
}
```

Linear Time: O(n)

› Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into one sorted list.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (a_i ≤ b_j) then append a_i to output list and
increment i
   else append b_j to output list and increment j
}
append remainder of nonempty list to output list
```

O(n log n) Time

- O(n log n) time. Arises in divide-and-conquer algorithms.

- Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

- Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

- O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic Time: $O(n^2)$

› Quadratic time. Enumerate all pairs of elements.

› Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

› $O(n^2)$ solution. Try all pairs of points.

```
min ← (x1 - x2)² + (y1 - y2)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)² + (yi - yj)²
      if (d < min)
         min ← d
   }
}
```

don't need to take square roots

see chapter 5

Cubic Time: $O(n^3)$

› Cubic time. Enumerate all triples of elements.

› Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?

› $O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```
foreach set Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
   }
}
```

Polynomial Time: $O(n^k)$ Time

› Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

k is a constant

› $O(n^k)$ solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

poly-time for k=17,
but not practical

- Check whether S is an independent set = $O(k^2)$.
- Number of k element subsets = $\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!}$
- $O(k^2 n^k / k!) = O(n^k)$.

Exponential Time

› Independent set. Given a graph, what is maximum size of an independent set?

› $O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* ← ϕ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

Summary: Algorithm analysis

› You must learn the asymptotic order of growth. It is fundamental when measuring the performance of an algorithm.

- O-notation
- Ω-notation
- Θ-notation

› Transitivity and additivity

Basic dynamic data structures

Assumed knowledge:

- Linked lists
- Queues
- Stacks
- Balanced binary trees

Why data structures?

- Programs manipulate data
- Data should be organized so manipulations will be efficient
  - Search (e.g. Finding a word/file/web page)
- Good programs are powered by good data structures
- Naïve choices are often much less efficient than clever choices
- Data structures are existing tools that can help you
  - guide your design, and
  - save you time (avoid re-inventing the wheel)

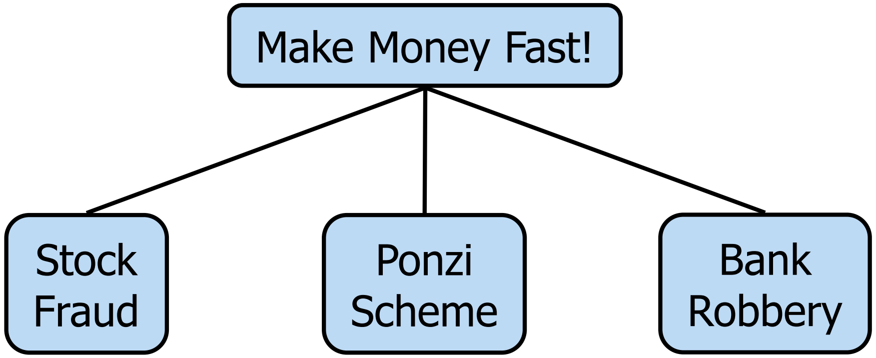
The Queue data structure

- The Queue data structure stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - **enqueue**(object): inserts an element at the end of the queue
  - object **dequeue**(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

The Stack

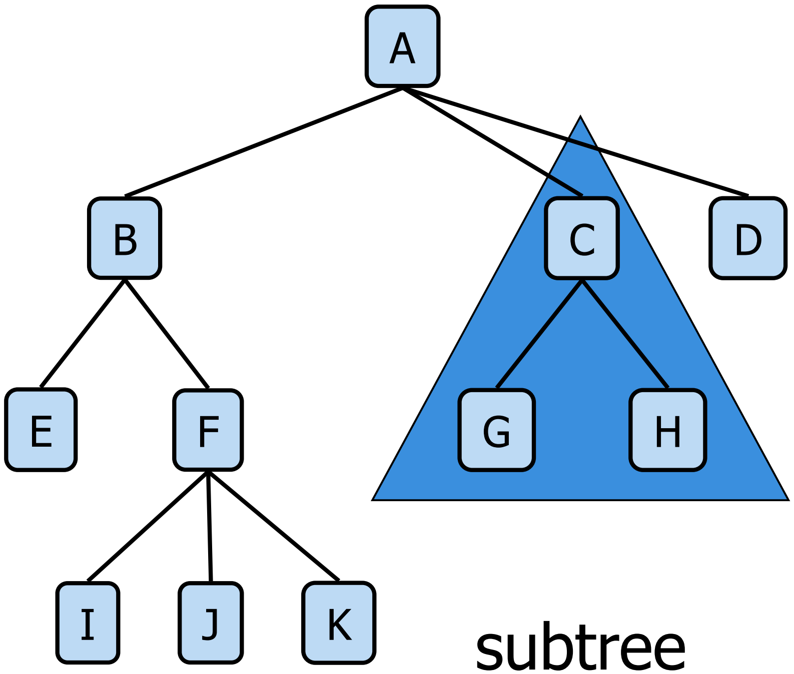
- The Stack data structure stores arbitrary objects
- Insertions and deletions follow the last-in first-out (LIFO) scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Trees



Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Subtree: tree consisting of a node and its descendants



Binary Trees

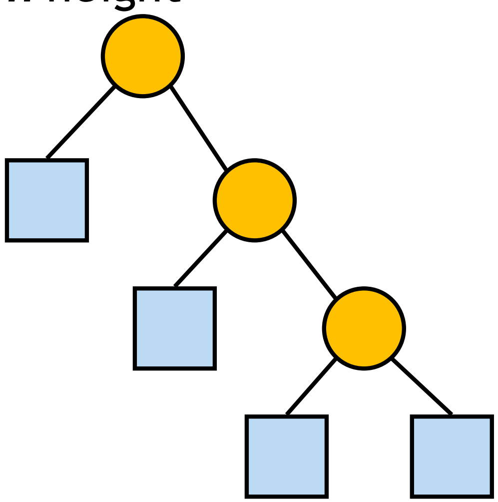
› Notation

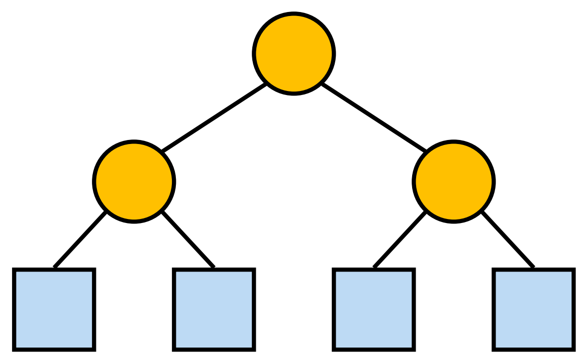
n number of nodes

e number of external nodes

i number of internal nodes

h height





Properties:

- $e = i + 1$
- $n = 2e - 1$
- $h \leq i$
- $h \leq (n - 1)/2$
- $e \leq 2^h$
- $h \geq \log_2 e$
- $h \geq \log_2 (n + 1) - 1$

Running Times for AVL Trees

- find is O(log n)
  - height of tree is O(log n), no restructures needed
- insert is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- remove is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

Summary data structures

› Queues
  - Enqueue, dequeue, first and size operations in O(1) time.

› Stacks
  - Push, pop, top and size operations in O(1) time

› Balanced binary trees (e.g. AVL trees)
  - Insert, delete and find operations in O(log n) time