BLG 336E

Analysis of Algorithms II

Lecture 1:

Logistics, and introduction

Slides partially based on CS 161-Stanford Course

The big questions

- Who are we?
  - Professor, TAs, students?
- Why are we here?
  - Why learn about algorithms?
- What is going on?
  - What is this course about?
  - Logistics?

Who am I??

Assistant Professor, **January 2020-present**
Computer Engineering Department
Istanbul Technical University, Istanbul, Turkey

Research Associate, **August 2017-present**
Biomedical Engineering Department
King's College London, London, UK

Visiting Post-graduate Student, **January 2017-July 2017**
Institute for Digital Communications
The University of Edinburgh, Edinburgh, UK

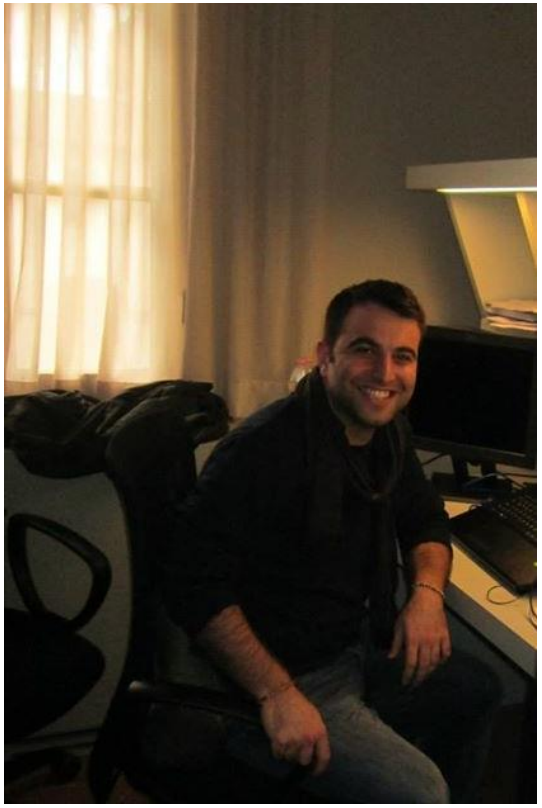
Doctor of Philosophy, **November 2013-December 2017**
Computer, Decision and Systems Science
IMT Institute for Advanced Studies, Lucca, Italy
Thesis: Joint registration and segmentation of CP-BOLD MRI

Visiting Post-graduate Student, **November 2015-September 2016**
Diagnostic Radiology Department
Yale University, New Haven, Connecticut, USA

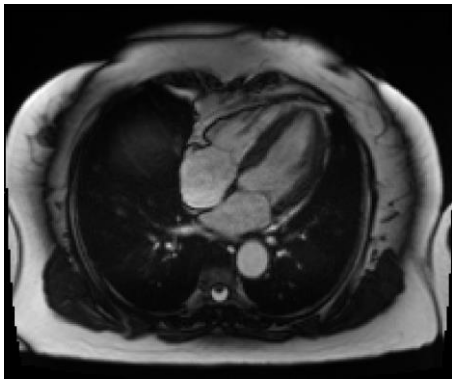
Master of Science, **October 2013**
Electrical and Electronics Engineering
Bahcesehir University, Istanbul, Turkey
Thesis: 3D Vessel Segmentation and Analysis in Coronary CT Angiography Images

Bachelor of Science, **September 2010**
Electronics Engineering
Istanbul Technical University, Istanbul, Turkey

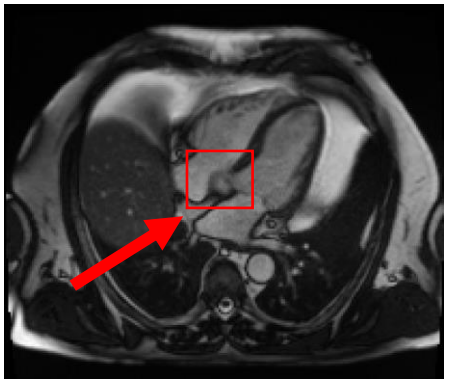
Erasmus Exchange Student, **September 2009**
Electronics Engineering
Darmstadt Technical University, Darmstadt, Germany



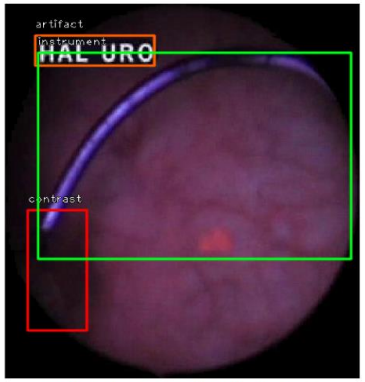
- TÜBİTAK International Outstanding Researchers Grant
- Diagnostic assessment and interpretation of the patient's cardiovascular health/disease
- Smart Magnetic Resonance (MR) scanner aims
  - ***Extracting clinically useful information***
  - Eliminating **dead time** between separate specialized acquisitions
  - Allowing extraction of multiple dynamic as well as **tissue contrast parameters** simultaneously



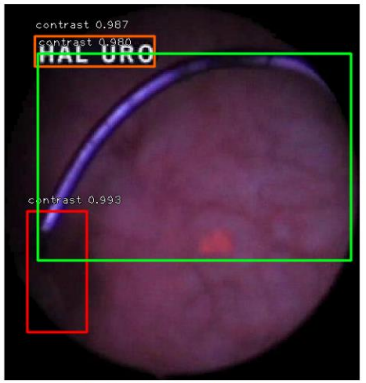
Good Planning



Bad Planning



(a) Example ground truth bounding boxes



(b) Predicted bounding boxes

Instructors

- İlkay Öksüz
- Mehmet Baysan

Teaching Assistants

- Erdi Sarıtaş
- Enes Erdoğan
- Mustafa Selahaddin Şentop
- Doğukan Arslan
- Yusuf Kızılkaya




Who are you?

- Sophomores
- Juniors
- Seniors
- MS Students
- PhD Students

Why are we here?

- I’m here because I’m super excited about algorithms!

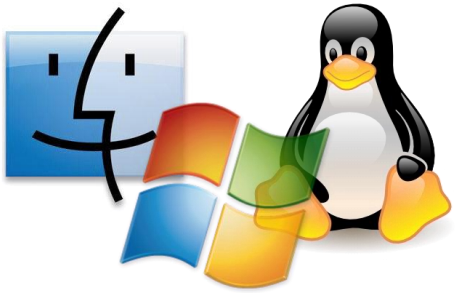
Why are you here?

- Algorithms are fundamental.
- Algorithms are useful.
- Algorithms are fun!
- BLG 336E is a required course.

Why is BLG 336E required?

- Algorithms are fundamental.
- Algorithms are useful.
- Algorithms are fun!

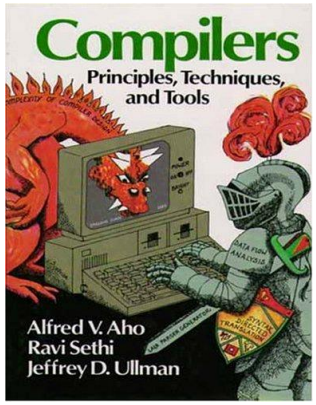
Algorithms are fundamental



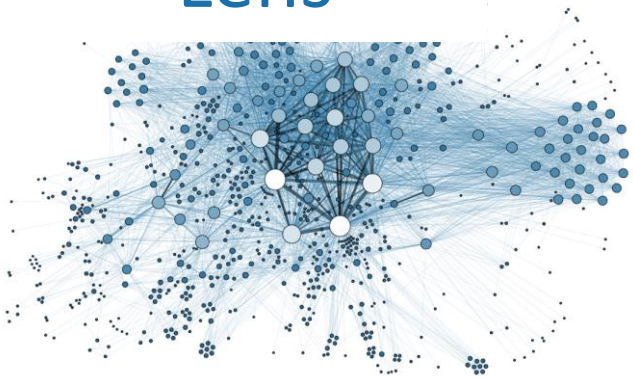
Operating Systems

The Algorithmic Lens

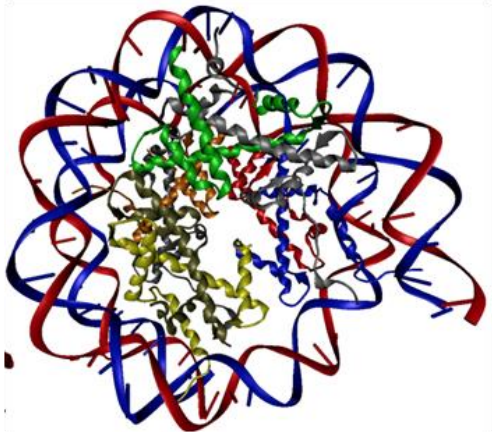
Security



Compilers



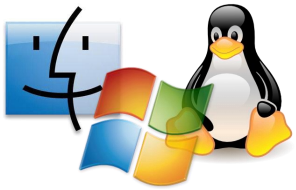
Networking

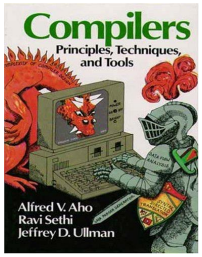


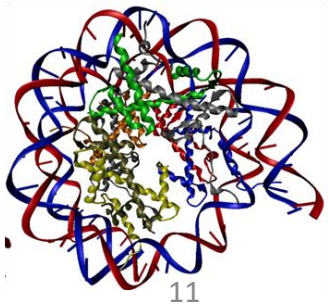
Computational Biology

Algorithms are useful

- As inputs get bigger and bigger, having good algorithms becomes more and more important!
- Most companies base their interview questions on this course







Algorithms are fun!

- Algorithm design is both an art and a science.
- Many surprises!
- Many exciting research questions!

What’s going on?

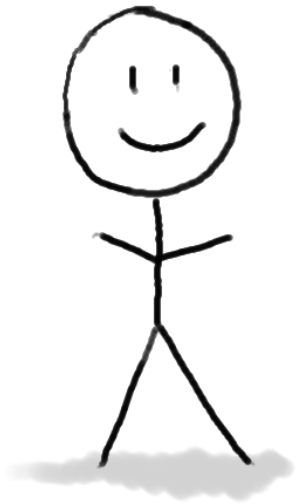
- Course goals/overview
- Logistics

Course goals

- The design and analysis of algorithms
  - These go hand-in-hand

- In this course you will:
  - Learn to think analytically about algorithms
  - Flesh out an "algorithmic toolkit"
  - Learn to communicate clearly about algorithms

Our guiding questions:



Does it work?

Is it fast?

Can I do better?

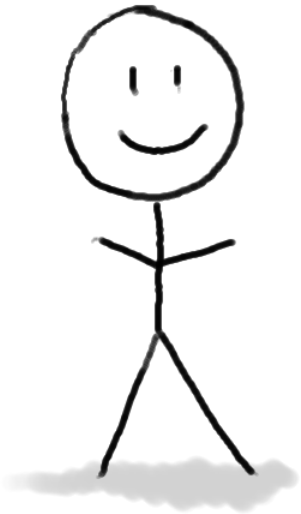
Our internal monologue…

Precision

What exactly do we mean by better? And what about that corner case? Shouldn't we be zero-indexing?

Detail-oriented
Precise
Rigorous

Does it work?
Is it fast?
Can I do better?



Intuition

Dude, this is just like that other time. If you do the thing and the stuff like you did then, it'll totally work real fast!

Big-picture
Intuitive
Hand-wavey

Both sides are necessary!

Aside: the bigger picture

- Does it work?
- Is it fast?
- Can I do better?

- Should it work?
- Should it be fast?

- We want to reduce crime.
- It would be more “efficient” to put cameras in everyone’s homes/cars/etc.

- We want advertisements to reach to the people to whom they are most relevant.
- It would be more “efficient” to make everyone’s private data public.

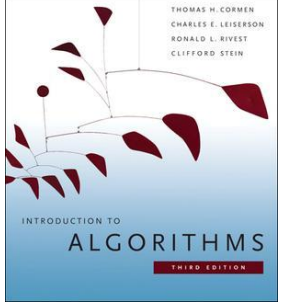
- We want to design algorithms, that work well, on average, in the population.
- It would be more “efficient” to focus on the majority population.

How to get the most out of lectures

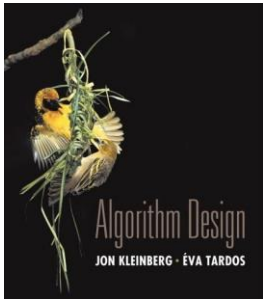
- **During lecture:**
  - Show up or tune in, ask questions.
  - Engage with in-class questions.
- **Before lecture:**
  - Get prepared to the topic
- **After lecture:**
  - Go through the exercises and assignments.

- ***Do the reading***
  - either before or after lecture, whatever works best for you.
  - **do not wait to "catch up" the week before the exam.**

Textbook

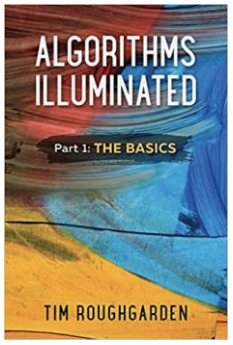


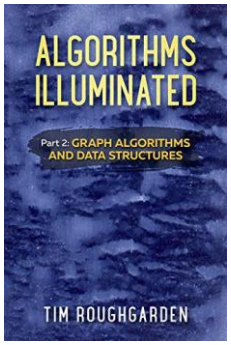
"CLRS": Introduction to Algorithms by Cormen, Leiserson, Rivest, and Stein.

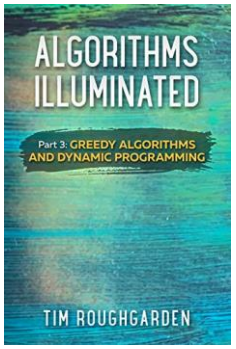


"Algorithm Design" by Kleinberg and Tardos

- CLRS and Algorithm Design are the reference books
- Stanford CS-161 Course
- We may also refer to to the following (optional) books:







Syllabus and Grading

Week	Date	Topic
1	13-Feb	Introduction. Some representative problems
2	20-Feb	Stable Matching
3	27-Feb	Basics of algorithm analysis.
4	5-Mar	Graphs (Project 1 announced)
5	12-Mar	Greedy algorithms-I
6	19-Mar	Greedy algorithms-II
7	26-Mar	Divide and conquer (Project 2 announced)
8	2-Apr	Dynamic Programming I
	9-Apr	HOLIDAY
9	16-Apr	Dynamic Programming II
10	23-Apr	**National Sovereignty and Children's Day** (Project 3 announced)
11	29/30-Apr	**Midterm**
12	7-May	Network Flow I
13	14-May	Network Flow II
14	21-May	NP and computational intractability I&II

Grading

- 3 Programming Projects (30%)
- 1 Midterm (30%)
- Final (40%)

Please note:

- VF Condition **15/50 (First 2 Hws+Midterm)**

Applications

Wide range of applications.

- Caching.
- Compilers.
- Databases.
- Scheduling.
- Networking.
- Data analysis.
- Signal processing.
- Computer graphics.
- Scientific computing.
- Operations research.
- Artificial intelligence.
- Computational biology.
- . . .

We focus on algorithms and techniques that are useful in practice.

Homework!

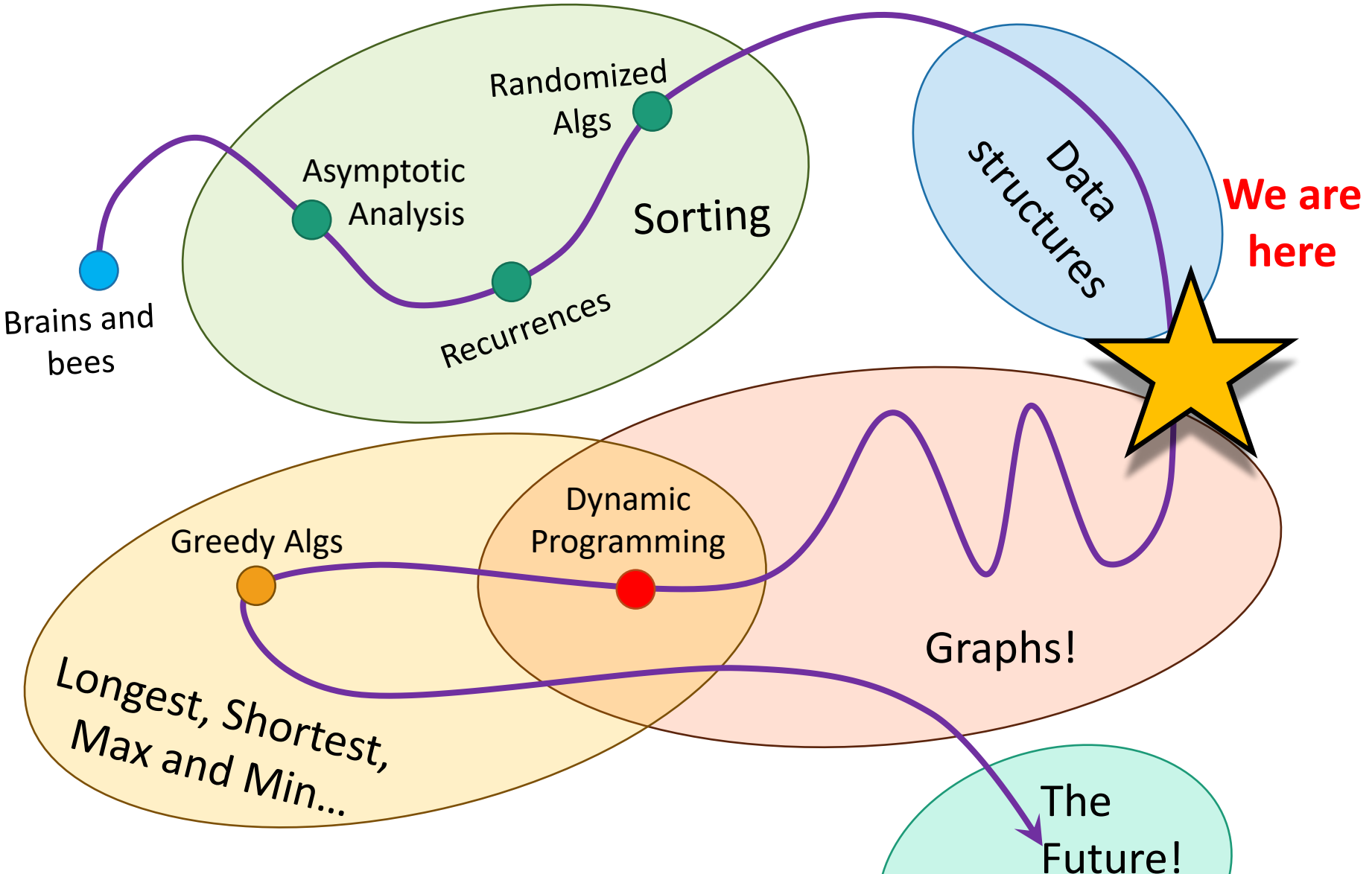
- Use C++ and object oriented approach in your assignments.
- The goal is to practice your implementation skills, so copy-pasting is not encouraged.
- There will be explicit instructions on which files to submit, how it should be compiled, example cases etc.
- Some examples will be provided so that you can test your program yourselves.
- However, your program will be graded based on how good it is in solving the given problem.

Talk to each other!

- Recitation sections:
  - See Ninova for schedule (to be posted soon)
  - Extra practice with the material, example problems, etc.
  - Technically optional, but ***highly recommended***!

Talk to us!

Roadmap


Randomized Algs
Asymptotic Analysis
Sorting
Brains and bees
Recurrences
Data structures
We are here
Dynamic Programming
Greedy Algs
Graphs!
Longest, Shortest, Max and Min…
The Future!

Contents

Introduction, Some representative problems

Stable matching problem

- How to prove the number of iterations of the algorithm
- How to prove that the solution returned is correct (stable)

Five representative problems:

- Interval Scheduling (Greedy Algorithm)
- Weighted Interval Scheduling (Dynamic Programming)
- Bipartite Matching (network flow problems, augmentation)
- Independent Set (NP Complete problem, can check a given solution in linear time, finding soln takes a lot of time)
- Competitive Facility Location (PSPACE-complete poblem)

Basics of Algorithm Analysis

- Computational Tractability = running efficiently=in polynomial time
- Asymptotic Order of Growth ($O()$, $\Omega()$, $\Theta()$)
- Implementation of Stable Matching Problem Using Arrays and Lists
- Survey of Common Running Times (Linear, nlogn, Quadratic, Cubic, $O(n^k)$, Beyond Polynomial, Sublinear)
- Implementation of Stable Matching Problem Using Priority Queues (Heap)

Graphs

- Definition and Applications (Transportation networks, communication networks, information networks, social networks, dependency networks)
- Paths and connectivity, trees
- Graph Connectivity and Graph Traversal (***Breadth-First (BFS), Depth First Search (DFS))***
- Implementation using Queues and Stacks
- Testing bipartideness: An Application of BFS
- Directed Acyclic Graphs, Topological Ordering

Greedy Algorithms

- An algorithm that builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion.
- Interval Scheduling: Design, Analysis (How to prove that a greedy algorithm produces an optimal solution)
- Scheduling all intervals
- Scheduling to minimize lateness
- Optimal Caching
- **Shortest Paths in a Graph (Dijksta)**
- Minimum(-cost) spanning trees (Kruskal, Prim)
- Implementation of Kruskal's Algo: Union-Find Data Structure

Divide and Conquer

- An algorithm that breaks up the problem into smaller problems, solves each part separately and then combines them. Recurrence relation.
- Mergesort
- Recurrences
- Divide and Conquer Applications
    - Counting inversions (collaborative filtering)
    - Finding the closest pair of points
    - Integer Multiplication
    - Convolutions and the FFT (maybe)

Dynamic Programming

- Drawn from the intuition behind divide and conquer and is opposite of greedy strategy. Explore space of all possible solutions by carefully decomposing things into a series of subproblems. Then build up solutions to larger and larger subproblems. Dynamic programming is better than brute force search. Does not explore all possibilites but only the ones it needs to explore.
- Weighted Interval Scheduling
- Principles of Dynamic Programming
- Segmented Least Squares
- Subset sums and knapsacks
- Shortest Paths in a Graph

Network Flow

- Bipartite matching is a special case, but there are many diverse applications.
- Max flow problem and Ford-Fulkerson algorithm
- Maximum flows and minimum cuts
- Choosing good augmenting paths
- A first application: bipartite matching
    - Joint paths in directed and undirected graphs
    - Extensions to max flow problem

NP and Computational Intractability

- NP complete problems: A large set of problems for which we do not know an efficient (polynomial time) solution. Once a solution is given though, we can check if it is correct in polynomial time.
- Polynomial time reductions (of one problem to another)
- Satisfiability problem reductions
- Definition of NP
- NP Complete problems

How was the word “algorithm” created?

Muhammad ibn Mūsā al-Khwārizmī 780-850

The words **'algorithm**' and **'algorism**' come from the name **al-Khwārizmī**. Al-Khwārizmī (Persian: خوارزمی, 8th century) was a Persian *mathematician, astronomer, geographer, and scholar* in the House of Wisdom in Baghdad.

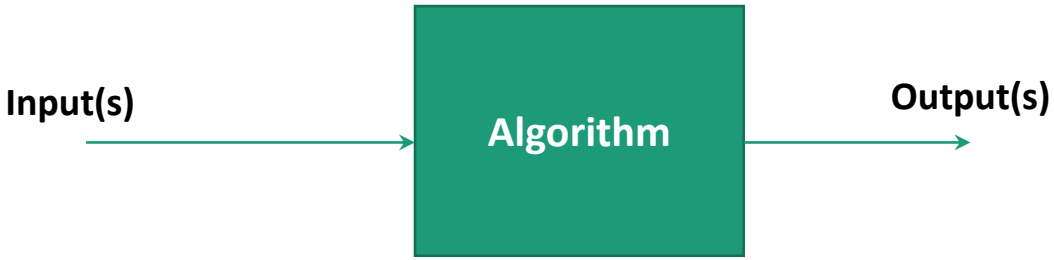
About 825, he wrote a treatise in the Arabic language, which was translated into Latin in the 12th century under the title Algoritmi de numero Indorum. This title means "Algoritmi on the numbers of the Indians", where **"Algoritmi" was the translator's Latinization of Al-Khwarizmi's name**. Al-Khwarizmi was the most widely read mathematician in Europe in the late Middle Ages, primarily through his book, the Algebra.

His name has taken on a special significance in computer science, where the word “algorithm” has come to refer to a method that can be used by a computer for the solution of a problem.

In Class discussion: what is an algorithm?

- Program?
- Function?

A Picture of an Algorithm



First Algorithm

- It is in human nature to think algorithmically!
- Long before computers were invented, people could create precise algorithms that could compute. One of the first algorithms (computing the greatest common divisor) is 300 years older than the Bible! It was written by Euclid (~300 BC).

- Here is an idea:

Gcd(m,n) = n, if m **mod** n = 0 or

Gcd(n, (m **mod** n)) otherwise.

Gcd (54,24)

Use Euclid's algorithm to determine the greatest common divisor of 54 and 24?

Gcd(m,n) = n, if m **mod** n = 0

or Gcd(n, (m **mod** n)) otherwise

Gcd (54,24)

Use Euclid's algorithm to determine the greatest common divisor of 54 and 24?

Gcd(m,n) = n, if m **mod** n = 0

or Gcd(n, (m **mod** n)) otherwise

Solution:

Gcd(54,24) = ?
54 mod 24 = 6 (=54 – 2 x 24)
Gcd(24,6) = 6 (since 24 **mod** 6 = 0)

First Algorithm

What is the greatest common divisor of 54 and 24?

The number 54 can be expressed as a product of two integers in several different ways:

$$54 \times 1 = 27 \times 2 = 18 \times 3 = 9 \times 6.$$

Thus the **divisors of 54** are: $1, 2, 3, 6, 9, 18, 27, 54.$

Similarly, **the divisors of 24** are: $1, 2, 3, 4, 6, 8, 12, 24.$

The numbers that these two lists share in common are the **common divisors** of 54 and 24:

$$1, 2, 3, 6.$$

The greatest of these is 6. That is, the **greatest common divisor** of 54 and 24. One writes:

$$\gcd(54, 24) = 6.$$

"Homework": implement the Euclidean algorithm in the language of your choice.

First Algorithm (Euclid's) = **function**

$$f(x) = y$$

input(s) output(s)

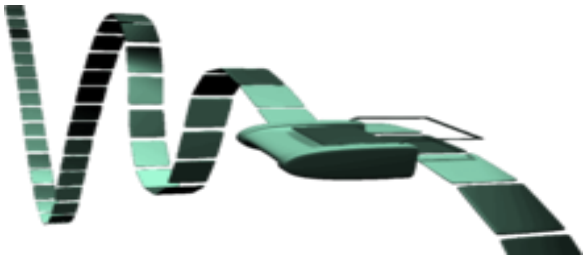
$$GCD(x_1, x_2) = y$$

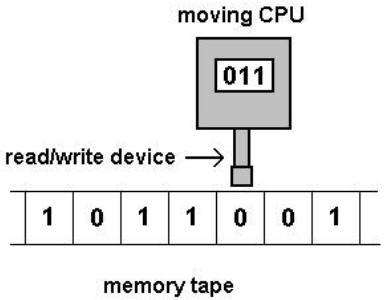
How to think about an algorithm differently?

Alan Turing. 1912 - 1954

Alan Turing was an English **mathematician, logician, cryptanalyst and computer scientist**. He was influential in the development of computer science and providing a formalization of the concept of the algorithm and computation with the Turing machine, playing a significant role in the creation of the modern computer

Turing Machine





A Turing machine is a kind of a *state machine*. At any time the machine is in any one of a finite number of states.

A Turing machine has an infinite one-dimensional *tape* divided into cells. Traditionally we think of the tape as being horizontal with the cells arranged in a left-right orientation. The tape has one end, at the left say, and stretches infinitely far to the right. Each cell is able to contain one symbol, '0' or '1'.

The machine has a *read-write head*, which scans a single cell on the tape. This read-write head can move left and right along the tape to scan successive cells.

The **action of a Turing machine** is determined completely by (1) the current state of the machine (2) the symbol in the cell currently being scanned by the head and (3) a table of transition rules, which serve as the "program" for the machine.

Turing Machine

Each transition rule is a 4-tuple:

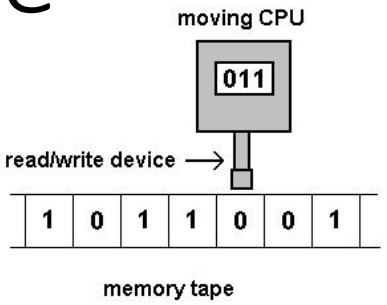
$\langle\ State_0, Symbol, State_{next}, Action\ \rangle$

which can be read as saying "if the machine is in state $State_0$ and the current cell contains *Symbol* then move into state $State_{next}$ taking *Action*".

The actions available to a Turing machine are either to **write a symbol on the tape in the current cell** (which we will denote with the symbol in question), or to **move the head one cell to the left or right**.

If the machine reaches a situation in which there is not exactly one transition rule specified, i.e., none or more than one, then the machine halts.

In modern terms, the tape serves as the memory of the machine, while the read-write head is the memory bus through which data is accessed (and updated) by the machine.



A **function** will be **Turing-computable** if there exists a set of instructions that will result in the machine computing the function, regardless of the amount of time it takes.

Although the device looks **primitive**,
even very complex modern computers can perform *only* Turing-computable tasks!

ARTIFICIAL INTELLIGENCE

Deep neural reasoning

The human brain can solve highly abstract reasoning problems using a neural network that is entirely physical. The underlying mechanisms are only partially understood, but an artificial network provides valuable insight. SEE ARTICLE P.471

HERBERT JAEGER

A classic example of logical reasoning is the syllogism, "All men are mortal. Socrates is a man. Therefore, Socrates is mortal." According to both ancient and modern views[1], reasoning amounts to a rule-based mental manipulation of symbols — in this example, the words 'All', 'men', and so on. But human brains are made of neurons that operate by exchanging jittery electrical pulses, rather than word-like symbols. This difference encapsulates a notorious scientific and philosophical enigma, sometimes referred to as the neural–symbolic integration problem[2], which remains unsolved. On page 471, Graves *et al.*[3] use the machine-learning methods of 'deep learning' to impart some crucial symbolic-reasoning mechanisms to an artificial neural system. Their system can solve complex tasks by learning symbolic-reasoning rules from examples, an achievement that has potential implications for the neural–symbolic integration problem.

A key requirement for reasoning is a working memory. In digital computers, this role is served by the random-access memory (RAM). When a computer reasons — when it executes a program — information is bundled together in working memory in ever-changing combinations. Comparing human reasoning to the running of computer programs is not a far-fetched metaphor. In fact, a venerable historical alley leads from Aristotle's definition of syllogisms to the modern model of a programmable computer (the Turing machine). Alan Turing himself used 'mind' language in his groundbreaking work[4]: "The behaviour of the computer at any moment is determined by the symbols which he is observing and his 'state of mind' at that moment."

The authors' neural system cannot and need not be programmed — instead, it is trained.

Although there are clear parallels between human reasoning and the running of computer programs, we lack an understanding of how either of them could be implemented in biological or artificial neural networks. Graves

Church-Turing thesis

Everything computable is computable by a Turing Machine.

Alonzo Church
(1903–1995)

Turing's Algorithm = **program**

"a series of instructions that can be put into a computer in order to make it perform an operation"

Is an algorithm a function or a program?

- It can be both:

1. **Declarative/functional** (Euclid's way of thinking)
2. **Imperative/ via machine instructions** (Turing's way of thinking)

What is an Algorithm?

All algorithms must satisfy the following criteria:

1. Input. Zero or more quantities are externally supplied.
2. Output. At least one quantity is produced.
3. Definiteness. Each instruction is clear and unambiguous. E.g., "add 6 or 7 to x" is not permitted.
4. Finiteness. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps. (Termination!)

We can also add "effectiveness". Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper; it must be feasible.

4 questions to ask about algorithms

- **How to devise algorithms?**

Creating an algorithm is an art which will never be fully automated.

During your studies, you can learn useful techniques.

- **How to validate/verify algorithms?**

Once an algorithm is devised, it is necessary to show that it computes a correct answer for all possible inputs —this is called algorithm validation.

Once the validity of the method is shown, a program can be written. Then a program verification is needed.

- **How to analyze algorithms?**

Performance analysis determines how much computing time and storage an algorithm requires [in best case, in worst case, and in average].

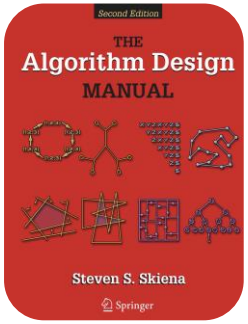
- **How to test a program?** Testing a program has two phases – debugging and profiling.

"Debugging can only point to the presence of errors, and not to their absence!" A proof of correctness is much more certain.

Profiling is the process of executing a correct program on data sets and measuring the time and space required.

How do we know that an algorithm is correct?

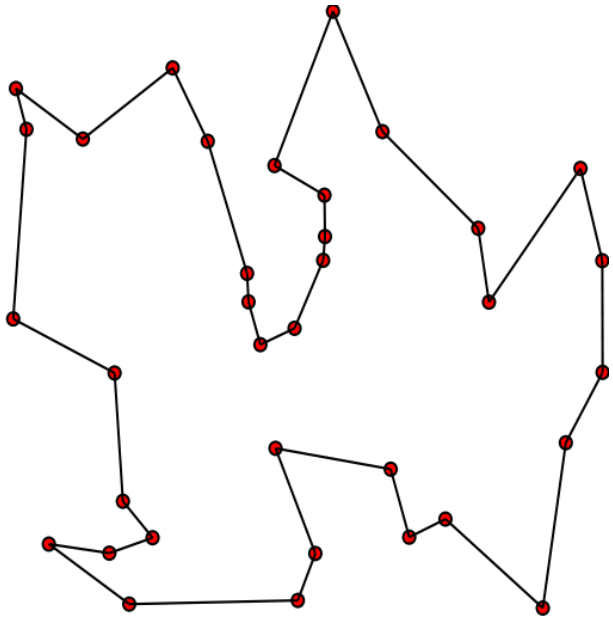
There are three desirable properties for a good algorithm. We seek algorithms that are *correct* and *efficient*, while being *easy to implement*. These goals may not be simultaneously achievable. In industrial settings, any program that seems to give good enough answers without slowing the application down is often acceptable, regardless of whether a better algorithm exists. The issue of finding the best possible answer or achieving maximum efficiency usually arises in industry only after serious performance or legal troubles.



algorithm correctly solves a given problem. Correct algorithms usually come with a proof of correctness, which is an explanation of *why* we know that the algorithm must take every instance of the problem to the desired result. However, before we go further we demonstrate why *"it's obvious"* never suffices as a proof of correctness, and is usually flat-out wrong.

Shortest travel distance in a graph

The **travelling salesman problem** (**TSP**) asks the following question: *"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"*



Stop right now and think up and algorithm to solve this problem (make the travelling salesman happy!).

Nearest Neighbour Tour

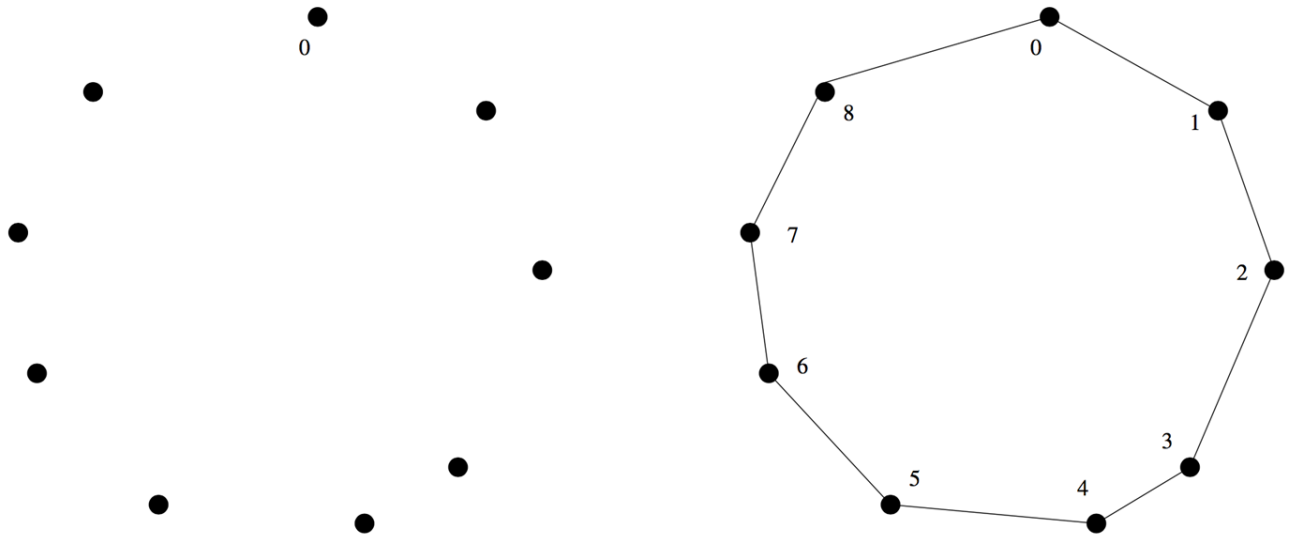
- A popular solution starts at some point p0 and then walks to its nearest unvisited neighbor p1, then repeats from p1, etc. Until done.

NearestNeighbor(P)
    Pick and visit an initial point p_0 from P
    $p = p_0$
    $i = 0$
    While there are still unvisited points
        $i = i + 1$
        Select p_i to be the closest unvisited point to p_{i-1}
        Visit p_i
    Return to p_0 from p_{n-1}

Is this solution correct?

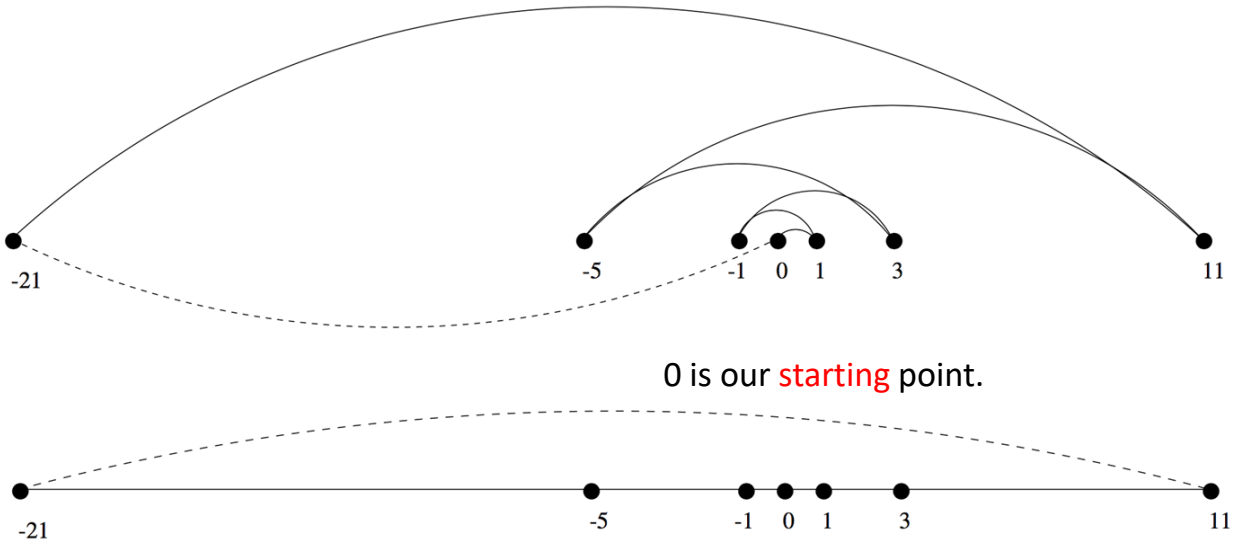
Let us try another example …

Nearest Neighbor Algorithm works perfectly for this case too! ☺



To make sure that NN algorithm is a correct solution to the TSP problem, we need to keep testing it on different examples.

What about this example?



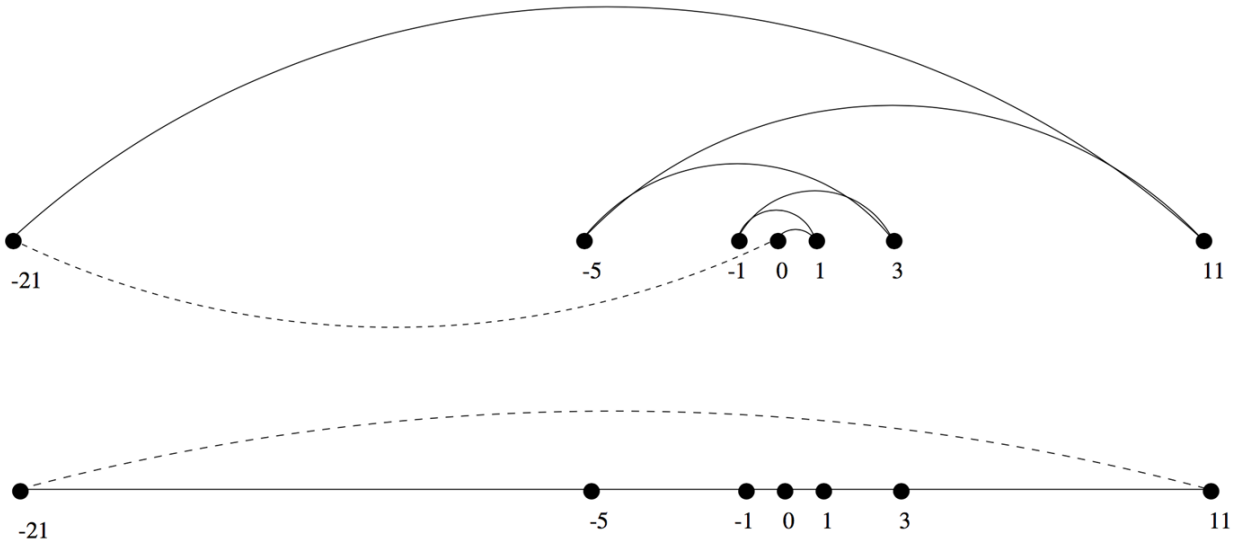
Closest Pair Tour

- Another idea is to repeatedly connect the closest pair of points whose connection will not cause a cycle or a three-way branch, until all points are in one tour.

ClosestPair(P)

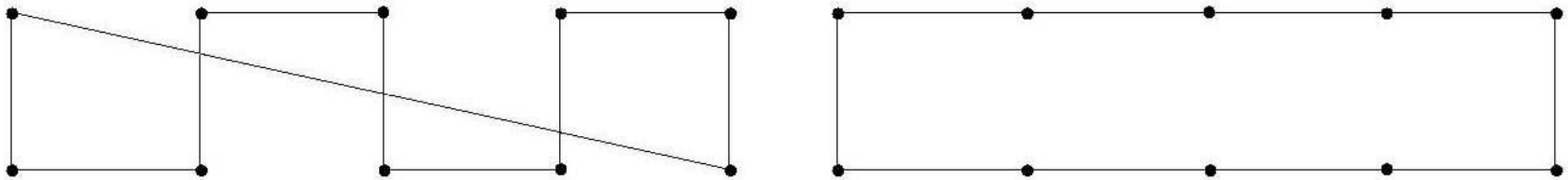
Let n be the number of points in set P.
For $i = 1$ to $n - 1$ do
$d = \infty$
For each pair of endpoints (s, t) from distinct vertex chains
if $dist(s, t) \leq d$ then $s_m = s$, $t_m = t$, and $d = dist(s, t)$
Connect (s_m, t_m) by an edge
Connect the two endpoints by an edge

What about this example?



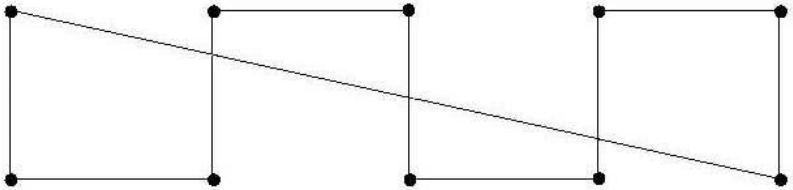
A good solution

What about this example? Does the Closest Pair Tour work on this?

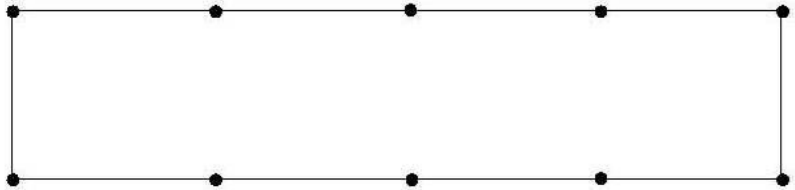


Although it works correctly on the previous example, other data causes trouble.

What about this example? Does the Closest Pair Tour work on this?



A good solution



Although it works correctly on the previous example, other data causes trouble.

Does a GOOD algorithm for the problem exist?

A Correct Algorithm: Exhaustive Search

- We could try all possible orderings of the points, then select the one which minimizes the total length:

OptimalTSP(P)
$d = \infty$
For each of the $n!$ permutations P_i of point set P
If $(cost(P_i) \leq d)$ then $d = cost(P_i)$ and $P_{min} = P_i$
Return P_{min}

- Since all possible orderings are considered, **we are guaranteed to end up with the shortest possible tour**.

Factorial function:
n! = 1*2*3*...*n (e.g. 3! = 1*2*3 = 6)

Exhaustive Search is Slow!

Because it tries all **n!** permutations, it is extremely slow to use when there are more than 10-20 points.

The fastest computer in the world couldn't hope to enumerate all the

20! =2,432,902,008,176,640,000 orderings of 20 points within a day.

[For n=1000, will not be achieved in your lifetime!!!]

No efficient and correct algorithm exists for the *travelling salesman problem.*

William Rowan Hamilton
(1805–1865)
Irish mathematician

*The **problem of joining dots** is also known as "**Hamiltonian cycle**"*

Take-Home Lesson: There is a fundamental difference between *algorithms*, which always produce a correct result, and *heuristics*, which may usually do a good job but without providing any guarantee.

Take-Home Lesson: An important and honorable technique in algorithm design is to narrow the set of allowable instances until there *is* a correct and efficient algorithm. For example, we can restrict a graph problem from general graphs down to trees, or a geometric problem from two dimensions down to one.

Take-Home Lesson: Searching for counterexamples is the best way to disprove the correctness of a heuristic.

Searching for **counterexamples** is the best way to disprove the correctness of an algorithm.

If the algorithm works some cases and fails in others, it is generally called a *heuristic*

"In which case my algorithm might fail?"

Think algorithmically ☺
Top interview questions on algorithms

- Sorting (plus searching binary search)
- Divide-and-conquer
- Dynamic programming / memoization
- Greediness
- Recursion
- Algorithms associated with a specific data structure

A sample of interview questions

- How to find middle element in a linked list through a single pass?
- Write a Java program to sort an array using Bubble Sort algorithm?
- How to reverse a linked list using recursion and iteration?
- Design an algorithm to find all the common elements in two sorted lists of numbers.

Sorting Problem

Input: A sequence of *n* numbers

$\langle a_1, a_2, \ldots, a_n \rangle$

Output: A permutation (reordering)

$\langle a_1', a_2', \ldots, a_n' \rangle$

such that

$a_1' <= a_2' <= \ldots <= a_n'$

Insertion Sort

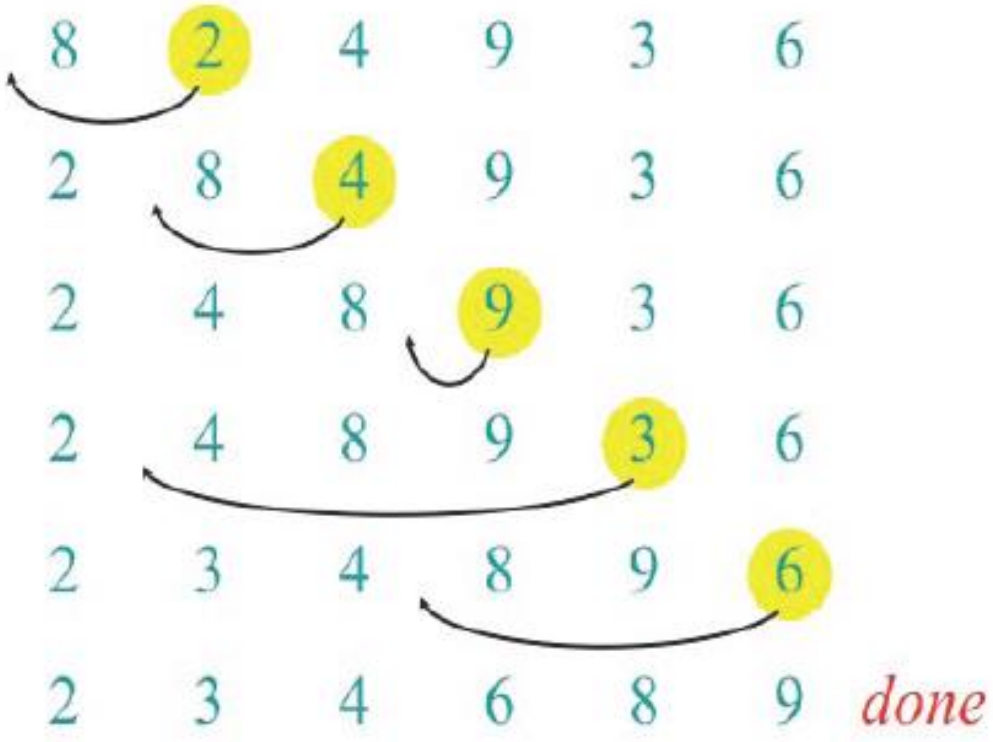
- Simple algorithm
- Basic idea:
  - Assume initial *j*-1 elements are sorted
  - Until you find place to insert j^{th} element, move array elements to right
  - Copy j^{th} element into its place
- Insertion Sort is an “in place” sorting algorithm. No extra storage is required.

Insertion Sort Example

8 2 4 9 3 6

Insertion Sort Example

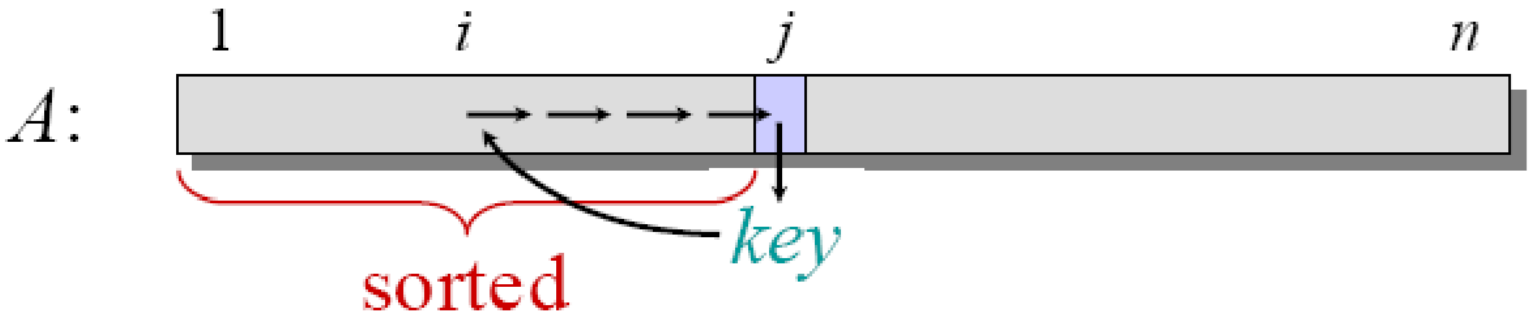
Insertion Sort Example



Insertion Sort

"pseudocode"

```
INSERTION-SORT (A, n)      ▷ A[1 . . n]
    for j ← 2 to n
        do key ← A[j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```



Pseudocode Conventions

- Indentation
  - indicates block structure
  - saves space and writing time
- Looping constructs (**while, for, repeat**) and conditional constructs (**if, then, else**)
  - like in C, C++, and Java
  - we assume that loop variable in a **for** loop is still defined when loop exits
- Multiple assignment $i \leftarrow j \leftarrow e$ assigns to both variables *i* and *j* value of *e (== j ← e, i ← j)*
- Variables are local, unless otherwise specified

Pseudocode Conventions

- Array elements are accessed by specifying array name followed by index in square brackets
  - A[i] indicates ith element of array A
  - Notation ".." is used to indicate a range of values within an array (A[i..j] = A[1], A[2],…, A[j])
- We often use **objects**, which have **attributes** (equivalently, **fields**)
  - For an attribute *attr* of object x, we write *attr*[*x*]
  - Equivalent of *x.attr* in Java or *x*-> *attr* in C++
- Objects are treated as references, like in Java
  - If *x* and *y* denote objects, then assignment $y \leftarrow x$ makes *x* and *y* reference same object
  - It does not cause attributes of one object to be copied to another

Pseudocode Conventions

- Parameters are passed **by value**, as in Java and C (and the default mechanism in C++).
  - When an object is passed by value, it is actually a reference (or pointer) that is passed
  - Changes to the reference itself are not seen by caller, but changes to the object's attributes are
- Boolean operators "and" and "or" are **short-circuiting**
  - If after evaluating left-hand operand, we know result of expression, then we do not evaluate right-hand operand
  - If x is FALSE in "x and y", then we do not evaluate y
  - If x is TRUE in "x or y", then we do not evaluate y

Efficiency

- Correctness alone is not sufficient
- **Brute-force** algorithms exist for most problems
- To sort *n* numbers, we can enumerate all permutations of these numbers and test which permutation has the correct order
  - Why cannot we do this?
  - Too slow!
  - By what standard?

How to measure complexity?

- Accurate running time is not a good measure
- It depends on **input**
- It depends on the **machine** you used and who implemented the algorithm

- We would like to have an analysis that **does not depend** on those factors

Machine-independent

- A generic uniprocessor random-access machine (RAM) model
  - No concurrent operations
  - Each simple operation (e.g. +, -, =, *, if, for) takes 1 step.
    - Loops and subroutine calls are *not* simple operations.
  - All memory equally expensive to access
    - Constant word size
    - Unless we are explicitly manipulating bits
    - No memory hierarch (caches, virtual mem) is modeled

Running Time

- **Running Time:T(n):** Number of primitive operations or steps executed for an input of size n.
- Running time depends on input
  - already sorted sequence is easier to sort
- Parameterize running time by size of input
  - short sequences are easier to sort than long ones
- Generally, we seek upper bounds on running time
  - everybody likes a guarantee

Kinds of Analysis

- **Worst-case:** (usually)
  - T(*n*) = maximum time of algorithm on any input of size *n*
- **Average-case:** (sometimes)
  - T(*n*) = expected time of algorithm over all inputs of size *n*
  - Need assumption about statistical distribution of inputs
- **Best-case:** (bogus)
  - Cheat with a slow algorithm that works fast on some input

Analyzing Insertion Sort

- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 S + c_5(S - (n-1)) + c_6(S - (n-1)) + c_7(n-1)$
  $= c_8 S + c_9 n + c_{10}$
- What can S be?
  - Best case -- inner loop body never executed
    - $t_j = 1$ ⇒ $S = n - 1$
    - $T(n) = an + b$ is a linear function
  - Worst case -- inner loop body executed for all previous elements
    - $t_j = j$ ⇒ $S = 2 + 3 + \ldots + n = n(n+1)/2 - 1$
    - $T(n) = an^2 + bn + c$ is a quadratic function
  - Average case
    - Can assume that in average, we have to insert A[j] into the middle of A[1..j-1], so $t_j = j/2$
    - $S \approx n(n+1)/4$
    - T(n) is still a quadratic function

Insertion Sort Running Time

Theta Notation, see next week.

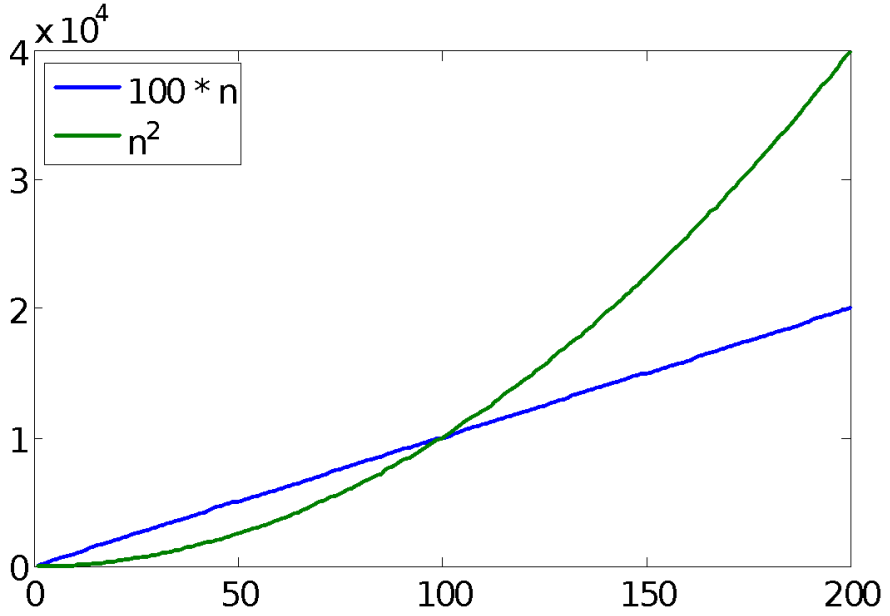
- **Best-case:**
  - $\Theta(n)$, inner loop not executed at all
- **Worst-case:** Input reverse sorted
  - $T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$ [Arithmetic series]
- **Average-case:** All permutations equally likely
  - $T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$

Is Insertion Sort a fast sorting algorithm?

- Moderately so, for small *n*
- Not at all, for large *n*

Asymptotic Analysis

- Ignore actual and abstract statement costs
- *Order of growth* is the interesting measure:
  - Highest-order term is what counts
    - As the input size grows larger it is the high order term that dominates



Next Week

- Stable Matching

Week	Date	Topic
1	13-Feb	Introduction. Some representative problems
2	20-Feb	Stable Matching
3	27-Feb	Basics of algorithm analysis.
4	5-Mar	Graphs (Project 1 announced)
5	12-Mar	Greedy algorithms-I
6	19-Mar	Greedy algorithms-II
7	26-Mar	Divide and conquer (Project 2 announced)
8	2-Apr	Dynamic Programming I
	9-Apr	HOLIDAY
9	16-Apr	Dynamic Programming II
10	23-Apr	**National Sovereignty and Children's Day** (Project 3 announced)
11	29/30-Apr	**Midterm**
12	7-May	Network Flow I
13	14-May	Network Flow II
14	21-May	NP and computational intractability I&II