"Two ideas lie gleaming in the scientist's chest: the first is the calculus, the second, the algorithm. The calculus made modern science possible; but it has been the algorithm that has made possible the modern world."

David Berlinski, May 2000.

2

Why Study Algorithmics?

- Algorithms constitute a core technology of human activity, one that affects significantly the behavior of *every* software system.
  - Many real world applications
  - Moore's Law
    - What is better, switch from a $\Theta(n^2)$ algorithm to a $\Theta(n \log n)$ algorithm or buy a faster computer?
- Depend on and impact other computer technologies:
  - Hardware with high clock rates, pipelining, cache, parallel computing (GPGPU, OpenMP, etc.), quantum computing
  - Local and wide area networking

3

Real World Applications

- Finding optimal paths
  - Internet routing
  - Shortest path in road network
  - route inspection
  - Delivery vehicle routing: UPS, FedEx

- Data compression
  - Lossless: text
  - Lossy: music, image, video
- Searching and indexing
  - Web search: Google, Yahoo
  - Text editor: emacs, vi, word
  - Human genome: 100,000 genes, 3 billion base pairs
  - Preference management systems: Netflix, Amazon

4

Real World Applications…

- Privacy and security
  - RSA encrypting: generate $n = pq$ where p and q are large prime numbers
  - Factoring: find a non-trivial factor of a large integer
- Programming tools
  - Is `main.c` a valid C program?
  - Debugging
    - Does `main.c` go into an infinite loop on input 101110?
- Optimization: find the cheapest/best way to do things
  - Airline schedules
  - Triangular mesh decimation in CAD
  - Network design: AT&T, Sprint

5

Class Goals

- Develop algorithmic *intuition*
  - How to approach a problem you are seeing for the first time
  - Learn new design techniques and choose one for a problem
  - How to decide if one algorithm is better than another
  - When do you stop searching for a better algorithm
  - Improve your ability to find (counter)examples
  - Obtain a better understanding of the limits of computation (intractability, computability)
- Improve your algorithmic *language*
  - Code is read more often that it is written
  - Not enough to understand a solution, you need to be able to explain your solution (not just your code) to others

6

Prerequisites

- COMP 2300/3200 (discrete structures) and COMP 2370 (introduction to data structures and algorithms)
- What you should already know:
  - Asymptotic notation (O, Ω, Θ)
  - Basic predicate and propositional logic
  - Proof techniques: induction, contradiction, direct & indirect proofs
  - Math: logs, sum of geometric and arithmetic series, unrolling of recurrences, functions, relations, basic calculus, matrix operations
  - Counting techniques, permutations, combinations, inclusion/exclusion
  - Discrete probability theory: expectation, conditional, independence
  - Trees and Graphs: representation, properties (connectivity, chromatic number, isomorphism, etc.), basic algorithms, including traversals (DFS)
  - Data structures and algorithms: arrays, linked lists, stacks, queues, balanced search trees, heaps, basic sorting and searching algorithms
  - Programming in Python, C, C++, or Java

7

Algorithms: Definition

- Abstraction of a computer program
- *Finite* and *effective* procedure that takes one or more values as *input* and, in *finite time*, produces one or more values as *output*
  - Finite sequence of instructions expressed in the language of a processing agent
- Must solve a *general problem*, specified by:
  - Set of infinitely many valid input instances
  - Properties output must satisfy

Example: Sorting, GCD, shortest path in a graph

8

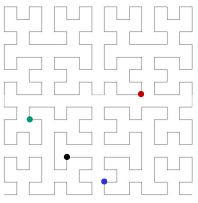
Example: Sorting

- *Input*: A sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ of n values taken from a *totally ordered* set U
- *Output*: a permutation π of X such that $x_{\pi(i)} \leq x_{\pi(i+1)}$, for $1 \leq i < n$
- Some input instances

  ⟨ 20, 5, 4, 13, 9 ⟩

  ⟨ Bill, Tom, Katie, Mary, Bob ⟩

  ⟨ (5, 3), (11, 8), (8, 1), (2,6) ⟩



9

A Sorting Algorithm

Step	Sort(A, n)
1	**for** $i \leftarrow 2$ **to** n
2	**do** key $\leftarrow A[i]$
3	$j \leftarrow i - 1$
4	**while** $j > 0$ and $A[j] >$key
5	**do** $A[j+1] \leftarrow A[j]$
6	$j \leftarrow j - 1$
7	$A[j+1] \leftarrow$ key

- How does this solve a general problem?
- Can you explain why it is correct?
- How good is it?
- Can the approach used be applied to other problems?

10

Exercise

```
f(x, y)
1  z ← 0
2  while x > 0
3        do
4           if x mod 0 = 1
5              then z ← z + y
6           x ← x div 2
7           y ← 2 · y
8  return z
```

- What does $f(x, y)$ compute?
- Why is it correct? Is it general?
- How many times are lines 6 and 7 executed?

11

Issues

- What formal models do we use to describe algorithms?
  - Turing machines, Java, C++, Python, pseudo-code
- What characterizes "good" algorithms?
  - Proofs of correctness
    - Does an algorithm need to be correct to be good?
  - Cost models
- How do we design good algorithms?
  - Algorithm design paradigms
- What sort of problems can be solved by algorithms and at what cost?
  - Computability and complexity

12

Pseudocode Guidelines

1. Aim for clarity and precision. A competent programmer should be able to implement the algorithm in any language without understanding why it works
2. Avoid the urge to describe repeated operations informally
3. Use the constructs of programming languages (loops, conditionals, etc.) to reflect the structure of the algorithm
4. Use indentation carefully and consistently
5. Use mnemonic names (except for idioms such as loop indices). Never use pronouns
6. Write individual steps using English, standard mathematical notation, or a combination
7. Avoid math notation if English is clearer (e.g., Insert x into S)
8. One statement or structuring element per line
9. Font should enhance structure and functionality

13

"Put the right kind of software into a computer, and it will do whatever you want it to. There may be limits to what you can do with the machines themselves, but there are no limits on what you can do with software"

Time Magazine, April 1984

14

A Class of Problems

- We investigate an intriguing question: *which set is larger, the set of computing problems or the set of algorithms?*
- For every real number r define problem Π_r

  Input: a natural number n

  Output: r truncated to the first n digits after the decimal point.

  Denote by A_r an algorithm that solves Π_r

  Example: $A_{4/3}(5)$ outputs 1.33333

  $A_{\pi}(4)$ outputs 3.1415

  $A_{\sqrt{2}}(3)$ outputs 1.414

15

Countable Sets

- A set *countable* if it has the same cardinality as some subset of the natural numbers
- Two sets have the same cardinality if there is a *bijection* from one to the other

Example: the set of prime numbers is countable

- Which of the following sets are countable?
  - The positive rational numbers
  - The number of months in a year
  - The numbers of type float that can be used in a C program.
  - The set of all valid C programs
  - The set of problems Π_r

16

Diagonalization

- Assume a 1-1 correspondence between [0,1] and N

	0	1	2	3	4	…	i	…
1	0.	$\boxed{a_{11}}$	a_{12}	a_{13}	a_{14}	…	a_{1i}	…
2	0.	a_{21}	$\boxed{a_{22}}$	a_{23}	a_{24}	…	a_{2i}	…
3	0.	a_{31}	a_{32}	$\boxed{a_{33}}$	a_{34}	…	a_{3i}	…
4	0.	a_{41}	a_{42}	a_{43}	$\boxed{a_{44}}$	…	a_{4i}	…
⋮	⋮	⋮	⋮	⋮	⋮	⋱		
i	0.	a_{i1}	a_{i2}	a_{i3}	a_{i4}	…	a_{ii}	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

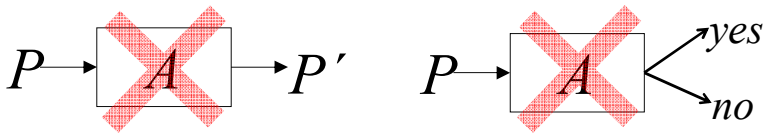
	0	1	2	3	4	5	6	…
1	0.	$\boxed{7}$	3	2	1	1	0	…
2	0.	0	$\boxed{0}$	0	0	0	0	…
3	0.	9	9	$\boxed{8}$	1	0	3	…
4	0.	2	3	4	$\boxed{0}$	7	8	…
5	0.	3	5	0	1	$\boxed{1}$	2	…
6	0.	3	1	4	0	5	$\boxed{7}$	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱
i	0.	7	6	5	0	0	1	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

- Construct $c = 0.c_1c_2c_3 \ldots c_i \ldots$ $\quad c_i = \begin{cases} a_{ii} - 1 & \text{if } a_{ii} \neq 0 \\ 1 & \text{if } a_{ii} = 0 \end{cases}$
  - Number c is not in the list
  - A_c does not exist!

17

Computability

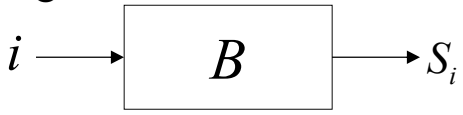
- Since the set $\{\Pi_r, r \in \mathbb{N}\}$ is not countable $\Rightarrow$ not all problems can be solved by an algorithm
- Many practical problems are not computable
  - This includes many useful program optimizations
  - As well as the detection of many (non-trivial) program properties



18

Example: Size Reduction

- Design an algorithm *A* that reduces the size (in bytes) required to store a program that solves an arbitrary problem *P*
- Design an algorithm *B* that finds the shortest program S_i to print an arbitrary input integer *i*



19

Size Reduction Impossibility

For arbitrary constant n define D_n as follows:

D_n : $i \leftarrow 0$
repeat
$i \leftarrow i+1$
$S_i \leftarrow B(i)$
$\ell \leftarrow |S_i|$
until $\ell \geq n$
print i

D_{987564} : $i \leftarrow 0$
repeat
$i \leftarrow i+1$
$C_i \leftarrow B(i)$
$\ell \leftarrow |C_i|$
until $\ell \geq 987564$
print i

$$\lceil \log n \rceil + c \geq n, \forall n \text{? No!}$$

20

Computation Model

- Abstraction of machine that executes algorithms

Example: RAM abstracts standard von Neumann architecture, CUDA abstracts GPUs, etc.

- We will use *Unit Cost RAM*
  - Similar to high-level procedural language (C, Java, etc.)
  - Unbounded random access memory
  - Basic data type is integer, real, character
  - Individual values of input fit in a memory cell
  - Arithmetic and logical operations on memory cells take constant time
  - Arrays allow random access in constant time
  - Pointer dereferencing takes constant time

21

Properties of Good Algorithms

- Correctness
  - An algorithm must implement the correct input to output transformation for *all* input instances
  - Not enough to try your algorithm on a few instances
- Efficiency
  - Time
  - Memory
- Simplicity
  - Always prefer an algorithm that is easier to implement
- Generality

22

Algorithm Design Paradigms

- An *algorithm design technique* is a general approach to solving a problem that is applicable to other problems
- Distills the common structure of "similar" algorithms

Example. The sorting example is an instance of an ***incremental design***, summarized as follows, for problems that take input values $x_1, x_2, \ldots, x_n$

1. Solve the problem for $x_1, \ldots, x_c$, for some constant c
2. **for** $i \leftarrow c+1$ **to** n **do**
   extend the solution of $x_1, \ldots, x_{i-1}$ to a solution of $x_1, \ldots, x_i$

```
Sort(A, n)
for i ← 2 to n
    do key ← A[i]
       j ← i − 1
       while j > 0 and A[j] >key
          do A[j + 1] ← A[j]
             j ← j − 1
       A[j + 1] ← key
```

23

Exercise

- Provide a high-level description of an *incremental algorithm* to compute each of the following:
  - The minimum of a set of values
  - The median of a set of values
  - The longest increasing sequence of a set of values
  - The convex hull of a set of points on the plane
  - The two closest points among a set of points

24

A Brief Tour of Problems and Design Paradigms

- Will illustrate various algorithm design paradigms using the following problems
  1. Matrix multiplication
  2. Greatest common divisor
  3. Optimal facility location
  4. Stable matching
  5. Eight queens
  6. Subset Sum
  7. Optimal robot tours

25

Problem 1: Matrix Multiplication

- *Input*: matrices A, B of order p.
- *Output*: matrix C of order p, where $C = A \times B$.

$$C_{ij} = \sum_{k=1}^{p} A_{ik} \cdot B_{kj}$$

Example:

$$\begin{bmatrix} 2 & -3 & 3 \\ -2 & 6 & 5 \\ 4 & 7 & 8 \end{bmatrix} \times \begin{bmatrix} -1 & 9 & 1 \\ 0 & 6 & 5 \\ 3 & 4 & 7 \end{bmatrix} = \begin{bmatrix} 7 & 12 & 8 \\ 17 & 38 & 63 \\ 20 & 110 & 95 \end{bmatrix}$$

26

Matrix Multiplication Algorithm

```
MATRIXMUL(A, B, p)
1  for i ← 0 to p − 1
2      do for j ← 0 to p − 1
3          do sum ← 0
4             for k ← 0 to p − 1
5                 do sum ← sum + A[i · p + k] · A[k · p + j]
6             C[i · p + j] ← sum
7  return C
```

- How many operations are performed?
  - What is the input size? $n = p^2$
  - Express answer as a function of n

$$\Theta\left(n\sqrt{n}\right)$$

27

Algorithm Design Paradigms

- MatrixMult is is an example of a ***brute force*** algorithm, referring to the fact that it uses a straightforward approach, directly based on the definition of the problem statement
- In an *optimization problem* you are required to maximize or minimize an *objective function*
- Can often be solved by brute force, by performing an ***exhaustive search***: first, generate and evaluate all candidate solutions and select the optimal one
- Can you think of other problems that you have solved by brute force?

28

Problem 2: Greatest Common Divisor

- *Input*: positive integers *a*, *b*.
- *Output*: largest integer *c* that divides both *a* and *b*.
- Applications:
  - Number of roots of an algebraic equation.
  - Unlimited precision integer and rational arithmetic.

Examples:

GCD(60, 24) = ?

GCD(1881234123551234, 4321435914) = ?

- Do you remember the middle school procedure?
- What algorithm do you propose?

29

Middle-School Procedure

GCD(a, b)

1. Find the prime expansions of a and b
2. Identify all the prime factors common to a and b and output each with the lowest multiplicity occurring in a or b.
3. Compute the product of the output factors

Exercise. Find GCD(60, 24)

Does this even qualify as an algorithm?

30

Sieve of Eratosthenes

```
//Input: An integer n ≥ 2
//Output: Array L of all prime numbers less than or equal to n
for p ← 2 to n do A[p] ← p
for p ← 2 to ⌊√n⌋ do    //see note before pseudocode
    if A[p] ≠ 0    //p hasn't been eliminated on previous passes
        j ← p * p
        while j ≤ n do
            A[j] ← 0    //mark element as eliminated
            j ← j + p
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L
```

- How many operations does this perform?

31

Consecutive Integer Check

```
GCD(a, b)
1  if a = 0 then return b
2  if b = 0 then return a
3  t ← min{a, b}
4  while a mod t ≠ 0 or b mod t ≠ 0
5        do t ← t − 1
6  return t
```

- What is the input size ? $n = \lceil \log a \rceil + \lceil \log b \rceil$
- How many iterations are performed in the worst case?

$\Theta(\min\{a,b\})$ or, in terms of n, $\Theta(2^{n/2})$

32

Euclid's Algorithm (c. 300 B.C.E.)

$$\gcd(a,b) = \begin{cases} a & \text{if } b = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

```
gcd(a, b)
   if b = 0 then
        return a
   return gcd(b, a mod b)
```

a	b	$a \bmod b$
3978	1590	798
1590	798	792
798	792	6
792	6	0
6	0	

33

Analysis

gcd(a, b)
  if $b = 0$ **then**
    return a
  return gcd$(b, a \bmod b)$

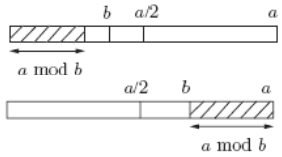
How do you measure efficiency?

- time, e.g., seconds
- some other variable proportional to time

Input size: $n = \log a + \log b$

Theorem. $a > b \Rightarrow r < a/2$

1. $b \le a/2 \Rightarrow r < b \le a/2$
2. $b > a/2 \Rightarrow r = a - b < a/2$



- Euclid's algorithm performs $O(n)$ recursive calls

34

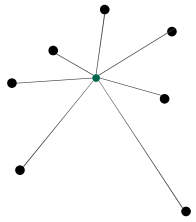
Algorithm Design Paradigms

- Euclid's Algorithm is an example of ***decrease-and-conquer***, meaning that the input of a problem is transformed to the input of the same or another problem.
- In this paradigm, the input of one problem is transformed to a simpler instance of *the same* problem.
- Can you think of other examples that use a similar approach?

35

Problem 3: Facility Location

- Given a set of points in the plane, find a point that minimizes the sum of distances to the input points

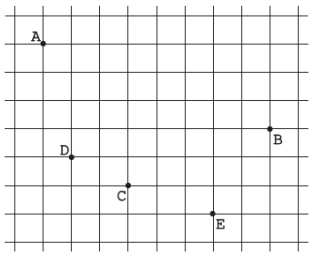


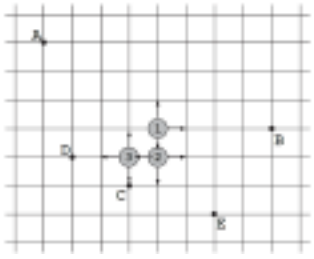
- There is no exact algorithm for solving this problem for general n

36

A Variant

- Given a set of points on a grid (e.g. downtown Manhattan), find a point that minimizes the sum of Manhattan distances to the input points





- Start with the point halfway between min and max coordinates
- Given a candidate solution, consider the 4 neighbors in order and move to first that yields an improvement
- Stop when all neighbors yield a worse solution

37

Algorithm Design Paradigms

- Our algorithm is an example of ***iterative improvement***
- The basic idea is to start with a feasible solution and then improve it by repeated application of a simple heuristic until no further improvements are possible
- Can you think of what obstacles one may encounter in a successful implementation?

38

Problem 4: Stable Matching

- Given a set of preferences among hospitals and (~13K) interns, assign a hospital to each intern so that the assignment is *self-enforcing*, i.e., unlikely to change
- Originated in the 1960's when economists studied recruiting / admission processes
- Intern *x* and hospital *Y* are an *unstable* pair if
  1) *x* prefers *Y* to his/her currently assigned hospital *X*, and
  2) *Y* prefers *x* to one of its currently assigned interns *y*
- A *stable* assignment is one with no unstable pairs
  - Individual interests cannot undo the assignment
  - No deals "under the table"

39

A Simplification: Perfect Matching

- For fun, we consider N boys and N girls, and look for matchings with no one left out
- Each person ranks all members of the opposite gender with a unique number between 1 and N in order of preference
- Avoid matchings where a boy and a girl prefer each other rather than their current partners
- If no such pair exists, all matches are "stable"

40

Modeling the Problem

- Let B be a set of N boys and G a set of N girls
- A *matching* is a subset M of $B \times G$ such that each member of B and each member of G appears in *at most* one pair of M
- A matching is *perfect* if it is a bijection between B and G
  - Each boy is matched to exactly one girl
  - Each girl is matched to exactly one boy

41

The General Matching Problem

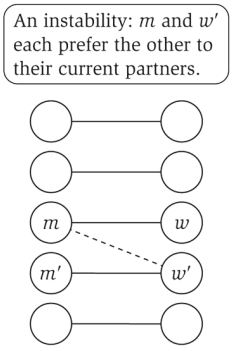
- Given an undirected graph (V, E), a *matching* is a subset $M \subseteq E$ such that no two edges in M share a vertex.
- A *maximum matching* is a matching of maximum cardinality. If every vertex is matched, i.e., incident on an edge in *M*, the matching is called *perfect.*
- In our problem (V, E) is the complete bipartite graph $K_{n,n}$, and we ask for a specific type of perfect matching

42

Stable Matching Problem

- An unmatched pair *m*-*w*′ is *unstable* if man *m* and woman *w*′ prefer each other over current partners

- *m* and *w*′ can improve their situation by eloping



- A *stable matching* is a perfect matching with no unstable pairs.
- Given the preference lists of *N* men and *N* women, find a stable matching, if one exists

43

Example 1

Men's Preferences

	1st	2nd
Victor	Bertha	Amy
Wyatt	Bertha	Amy

Women's Preferences

	1st	2nd
Amy	Victor	Wyatt
Bertha	Victor	Wyatt

Example 2

Men's Preferences

	1st	2nd
Victor	Bertha	Amy
Wyatt	Amy	Bertha

Women's Preferences

	1st	2nd
Amy	Victor	Wyatt
Bertha	Wyatt	Victor

- How many perfect matchings are there?
- How many of these are stable?

44

Example 3

Men's Preference Profile

	1st	2nd	3rd	4th	5th
Victor	Bertha	Amy	Diane	Erika	Clare
Wyatt	Diane	Bertha	Amy	Clare	Erika
Xavier	Bertha	Erika	Clare	Diane	Amy
Yancey	Amy	Diane	Clare	Bertha	Erika
Zeus	Bertha	Diane	Amy	Erika	Clare

Women's Preference Profile

	1st	2nd	3rd	4th	5th
Amy	Zeus	Victor	Wyatt	Yancey	Xavier
Bertha	Xavier	Wyatt	Yancey	Victor	Zeus
Claire	Wyatt	Xavier	Yancey	Zeus	Victor
Diane	Victor	Zeus	Yancey	Xavier	Wyatt
Erika	Yancey	Wyatt	Zeus	Xavier	Victor

- How many perfect matchings are there?
- How many of these are stable?

45

Brute-Force Algorithm

- For each perfect matching S, check if S is stable
- How many perfect matchings are there?
- How long does it take to check if a perfect matching is stable?
- A brute-force algorithm can be implemented in $O(N!\ N^3)$ time

Example: $n = 20$

- $20! = 2{,}432{,}902{,}008{,}176{,}640{,}000 = 2 \times 10^{18}$
- If you can evaluate 1 matching in 1 μsec, then you need 63,419 years to get the answer

46

The Boston Pool Algorithm

- Used by the National Resident Matching Program to assign interns to hospitals. Also used for faculty recruiting in France, college admission in Germany, public school admission in Boston, and more
- Proceeds in rounds:
  1. An arbitrary unmatched man (hospital) m proposes to his favorite woman (intern) who has not rejected him
  2. Each woman (intern) w plans to accept her best possible suitor (hospital) according to her preference list. If w is unmatched, she tentatively accepts m. If w is matched but prefers m, she rejects her current match and tentatively accepts m. Otherwise, w rejects m

47

The Boston Pool Algorithm…

- Keeps a set S of tentative matches until all matched

```
1  Initialize all persons to free and S to empty
2  while (some man m is free and hasn't proposed to every woman)
3       choose m
4       let w be the highest-ranked woman in m's list to whom
          m has not proposed
5       if w is free
6            add (m, w) to S
7       else let m' be w's fiancee
8            if w prefers m over m'
9                 add (m, w) to S
10                set m' free
11           else w rejects m
12 return S
```

Example 3

Men's Preference Profile

	0th	1st	2nd	3rd	4th
Victor	Bertha	Amy	Diane	Erika	Clare
Wyatt	Diane	Bertha	Amy	Clare	Erika
Xavier	Bertha	Erika	Clare	Diane	Amy
Yancey	Amy	Diane	Clare	Bertha	Erika
Zeus	Bertha	Diane	Amy	Erika	Clare

Women's Preference Profile

	0th	1st	2nd	3rd	4th
Amy	Zeus	Victor	Wyatt	Yancey	Xavier
Bertha	Xavier	Wyatt	Yancey	Victor	Zeus
Claire	Wyatt	Xavier	Yancey	Zeus	Victor
Diane	Victor	Zeus	Yancey	Xavier	Wyatt
Erika	Yancey	Wyatt	Zeus	Xavier	Victor

Algorithm Properties

Observation 1. Men propose to women in decreasing order of preference.

Observation 2. An engaged man can become unengaged if his fiancée dumps him in favor of a better candidate.

Observation 3. Once a woman becomes engaged, she remains so. After this point, she can only trade up.

Observation 4. Upon termination, each man is either engaged or has proposed to everyone.

Observation 5. At all times S contains a matching.

50

Proof of Correctness

Termination. Each iteration of the while loop (line 2) results in a new proposal. There can be at most N^2 proposals $\Rightarrow$ there are at most N^2 iterations

Perfection. Upon termination everyone is matched.

- Suppose, for contradiction, that man m' ends unmatched.
- Then, some woman w' is also unmatched (Observation 5).
- m' proposed to everyone (Observation 4).
- w' was not proposed to by anyone (Observation 3).
- But m' proposed to everyone, a contradiction

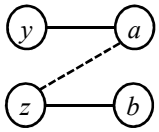
51

Proof of Correctness: Stability

Claim. Upon termination there are no unstable pairs**.**

***Proof* (by contradiction).** Assume *a*-*z* is unstable and that, upon termination, (*y*, *a*) and (*z*, *b*) ∈ *S*

Case 1: *z* never proposed to *a*



⇒ *z* prefers *b* over *a*

⇒ *a*-*z* is stable

Case 2: *z* proposed to *a* and was rejected

⇒ *a* prefers current partner over *z* (since *a* can only trade-up)

⇒ *a*-*z* is stable

- Either way, *a*-*z* is stable, a contradiction

52

Conclusions

- Our algorithm *always* finds a stable matching
- What is the running time?
  - Depends on implementation
  - What is the input size?
- If there are multiple solutions, does the stable matching found depend on the proposal order?
- The Boston Pool algorithm is another example of the *iterative improvement* paradigm

53

Efficient Implementation

- Both men/women are represented by index 1..N
- Engagements.
  - Maintain free men in a queue or stack
  - Keep 2 arrays, wife[] and husband[] so that if (x, a) in S then wife[x]=a and husband[a] = x
  - Use 0 for unengaged
- Men proposing
  - Each man keeps array of women sorted by preference
  - Each man keeps *count* of number of proposals made

54

Efficient Implementation…

- Men proposing
  - Each man keeps array of women sorted by preference
  - Each man keeps *count* of number of proposals made
- Women accepting/rejecting
  - Need to know if woman w prefers m over m'
  - A preference list is a bijection, hence invertible
  - Each woman keeps the inverse of preference list

Amy	1st	2nd	3rd	4th	5th	6th	7th	8th
Pref	8	3	7	1	4	5	6	2

Amy	1	2	3	4	5	6	7	8
Inverse	4th	8th	2nd	5th	6th	7th	3rd	1st

55

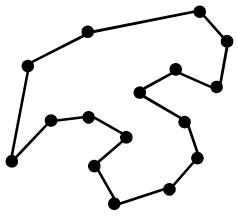
Variants

- Not everyone ranks everyone
  - Partial preference lists, missing table entries
- Most are not will to partner with some
  - Some men/women label others "unacceptable"
  - A doctor refuses to move to a rural area
- The two partition sets have unequal number of elements
- How about limited "polygamy"?
  - E.g., hospital wants 5 residents

56

Problem 5: Short Robot Tours

- Robot arm equipped with a tool, say a soldering iron or drill, to assemble a circuit board
- Robot needs to visit (and solder) the contact points in some order
- We seek an order which minimizes the time it takes to assemble the circuit board.



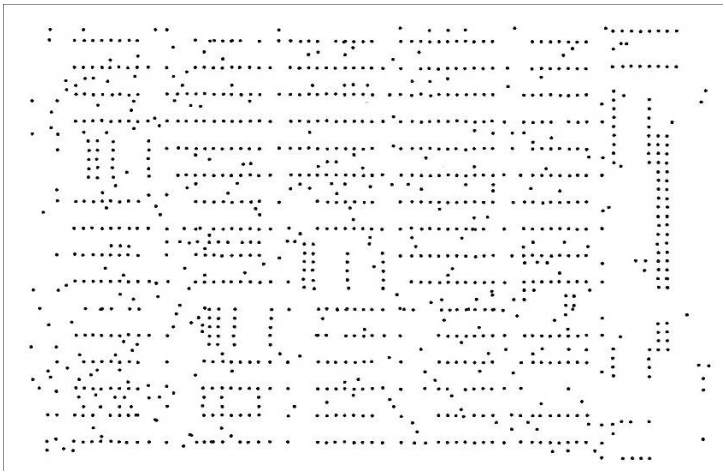
Input: a set of points in the plane

Output: a shortest tour that visits each point once

57

Example: PCB Drilling

- 1,173 locations to drill in a printed circuit board



58

Brute Force: Exhaustive Search

```
1  d = ∞
2  foreach permutation π of P
3       if cost(π) < d
4            d = cost(π)
5            π_best = π
6  return π_best
```

- How to generate all permutations?
- How long does it take?

59

Analysis

- Need to evaluate ½ $(n-1)!$ tours. Why?

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

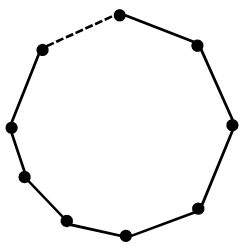
Example: $n = 21$

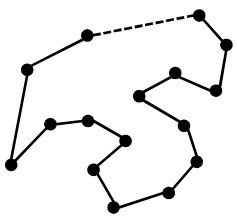
- $20! = 2{,}432{,}902{,}008{,}176{,}640{,}000 = 2 \times 10^{18}$
- Assume you can evaluate 1 tour in 1 μsec
- Need 38,573 years to get the answer
- In real boards $n \approx 1000$

60

Nearest Neighbor Tour

```
1  Select an initial point p_0 ∈ P, say highest point
2  i = 0
3  while there are unvisited points of P
4        i = i + 1
5        p_i = the unvisited point closest to p_{i-1}
6        visit p_i
7  return ⟨p_0, p_1, ..., p_{n-1}, p_0⟩
```

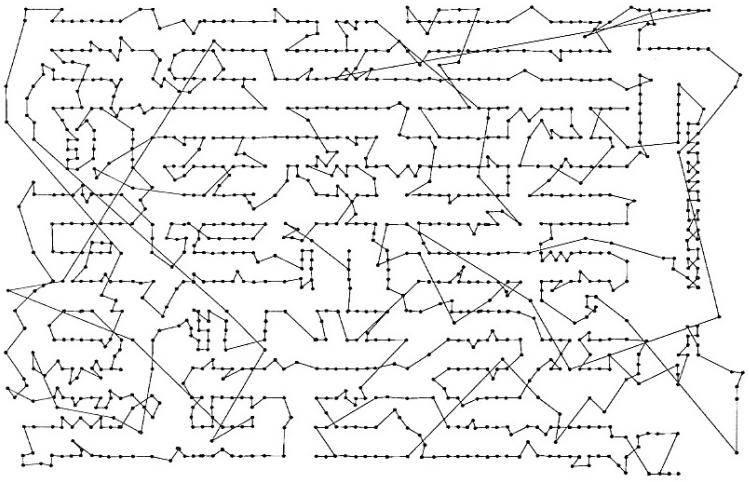




61

PCB Drilling NN Tour

- $n = 1173$



62

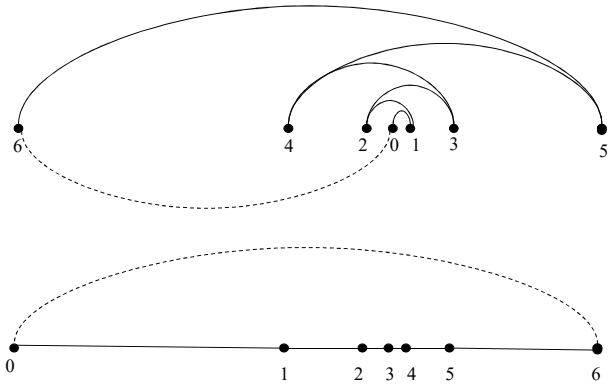
Algorithm Design Paradigms

- Nearest-neighbor is an example of a ***greedy algorithm.***
- The idea is to construct the solution as a sequence of steps, choosing at each step what "looks best" (the greedy heuristic) at the moment
- Each step is *feasible*, *locally optimal* and (we hope) consistent with a *globally optimal* solution
- Can you think of other examples that follow this approach?
  - How about if you are asked to write an algorithm to produce exact change with minimum # of coins for a certain currency?

63

Correctness

- NN tour is incorrect!



- Starting at leftmost vertex does not always help.

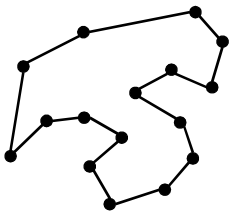
64

Closest-Pair Tour

- Add the next cheapest edge that doesn't produce a cycle or 3-way branch

```
1  for i ← 1 to n − 1
2        do d ← ∞
3           foreach pair of endpoints p, q of distinct chains
4                do if dist(p, q) ≤ d
5                      then s = p, t = q, d ← dist(p, q)
6           Connect (s, t) by and edge
7  Connect the two remaining endpoints by an edge
```

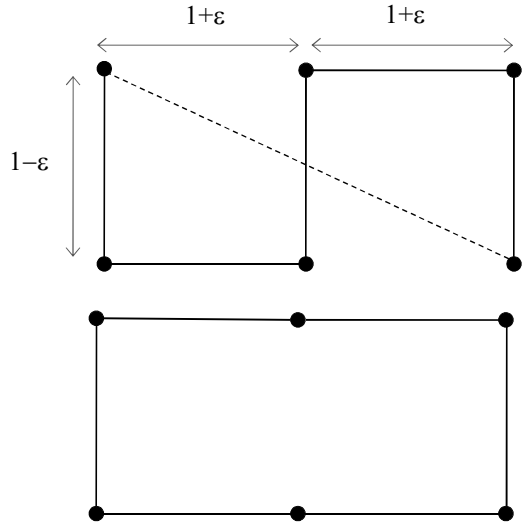
- How long does this take?



65

Correctness

- CP tour is incorrect



Optimal tour

66

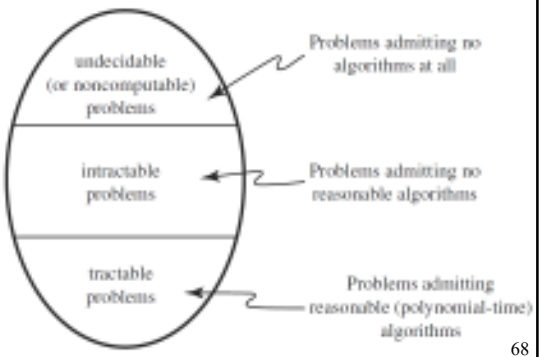
Exercise

- Describe a backtracking algorithm to determine if a given graph is Hamiltonian
  - What does the state-space tree look like?
  - Why is the solution better than exhaustive search?
- Optimization problems use a variant of backtracking known as ***branch-and-bound***
  - Based on computing a bound on the objective function
  - Prune subtrees that fall outside the desired bound
- Can you adapt the backtracking solution to solve the shortest robot tour optimally?

67

Tractability

- Unfortunately, nobody knows an efficient algorithm for the shortest robot tour problem
- Not all problems that can be solved *in principle* can be solved *in practice*
- A problem is said to be *intractable* if we cannot solve it efficiently (in polynomial time); otherwise the problem is *tractable*



68

Two Related Problems

- Given a set S of $n = 2k$ points in the plane
  1. Find a circuit of minimal length that visits each point exactly once (our robot tour problem).
  2. Find a *perfect* matching of minimum weight, i.e., a set of segments incident on S such that each point is the endpoint of *exactly one* segment and the sum of the lengths of the segments is minimized.
- Both problems require that we select an optimal set of segments from an exponential set of choices and both can be modeled using the language of graph theory.
- Problem 1 is not known to be tractable while problem 2 can be easily solved in $O(n^3)$ time.

69

Approximation Algorithms

- An *approximation algorithm* is one that returns an approximate answer to an optimization problem
- Trades loss of accuracy for better running time
- Usually comes with quality guarantee

  Example: 1.5-approximation means answer is at most 50% worse than the optimal (in practice, may be much better)
- Why settle for less?
  - Sometimes exact answer doesn't exist, e.g. square roots
  - Exact solution may take too long
  - Approximate answer may be the first step in finding optimal answer

70

Approximate Shortest Robot Tours

- Nearest neighbor tour yields approximation:

$$\frac{NN}{OPT} \leq \frac{1}{2}\left(\lceil \log n + 1 \rceil\right)$$

- Can do much better using MST-based approximation $\frac{MST}{OPT} \leq 2$
- Can improve by augmenting MST with *minimum-weight perfect matching* of odd degree vertices

71