



CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

# LEC00
# SYLLABUS & INTRO TO ALGORITHMS

Lecture slide courtesy of Prof. Steven Skiena

# ANALYSIS OF ALGORITHMS – FALL 2016

- Instructor: Sael Lee
- Office: Academic Building B422
- E-mail: sael@sunykorea.ac.kr
- Webpage: http://www3.cs.stonybrook.edu/~sael/teaching/cse373/
- Course Time: Mon/Wed 17:00~18:20
- Office Hours: Tu 15:30~17:30 Academic Bldg. B422

# TEXT BOOK & RESOURCES

- Textbook:
  - Required: Steven Skiena, The Algorithm Design Manual, 2nd edition, Springer-Verlag
  - Reference:
    - MIT Open Courseware Introduction to Algorithms (Fall 2005).
    - Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. Introduction to Algorithms.
    - Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. Algorithms.
    - Jon Kleinberg and Eva Tardos. Algorithm Design
- Lectures will follow Steven Skiena's class
  - His lectures can be found at
    - http://www3.cs.stonybrook.edu/~algorith/video-lectures/

# GRADING & ASSIGNMENTS

- Grading: Grades will be assigned based on the following formula, with cut-offs determined by my opinion of students on the boundary.
  - **Participation** will be worth 5% of the grade.
  - **Homework** will be worth approximately 20% of the grade.
  - **Midterm** I will be worth 20% of your grade.
  - **Midterm** II will be worth 20% of your grade.
  - **Final Exam (cumulative)** will be worth 35% of your grade.
- Participation
  - You will turn in “daily problems” at the beginning of the class.
  - “daily problems” will be posted in the web;
- Assignments
  - There will be 4-5 assignments.
  - I will drop the lowest grade from among your homework scores.
  - No late assignments will be accepted.

# EXAMS

- All homework, daily and *midterms/exam* problems will be drawn from the textbook.
- Thus the correct way to study for this course is to review these problems and figure out how to solve them.
- The *exams* will be closed book, but there is no need to memorize solutions. Once you have solved them once you should be able to reconstruct them on demand.

# RULES OF THE GAME

- You should get the required textbook.
- I will lecture from slides, which are also available on the course page. They are available on-line to be read online. *Any student caught printing the slides on the CS department machines will get in trouble*.
- The best way to learn the material is by solving problems. You are encouraged to work in pairs, for the best way to understand the subtleties of the homework problems is to argue about the answers. Each of you should look at all the problems independently, and not just divide the list in two parts each time. Don't be a leech and let your partner do all the work. Unless you learn how to solve problems, I *promise* that you will get burned on the exams and thus for your final grade.

# ASSIGNMENTS

- Assignment will be handed out in class and are due at the start of class of the due date. Legible handwritten copies of the assignments should be turned in. Programing components should be turned in through Blackboard before 23:55 of the due date.
- The partner system relies upon a certain maturity among the students. If you don't have a partner, tell me and I will hook you up with one. If you are having trouble with your partner and want a divorce, tell me and I will set you up with a new one. I will act as a broker *but not* as a counselor. I do not want to hear what a louse your old partner is, and you will get a dirty look from me when you demand a divorce regardless of who was at fault.
- At the start of each class, I will work out one previously identified homework problem, emphasizing the thought process leading to the solution. To get the most benefit from this, you should try to work out the problem before lecture,
- The daily problems should be worked on individually. I will collect your solutions for these daily problems at the beginning of each class.

- Only one solution to the assignment per pair should be turned in, with the partners alternating who writes up the final solution. The scribe for each assignment will have to label themselves as such. Unless announced otherwise in class, any solution to a part of a homework problem which takes more than one side of a sheet of paper will not be graded. This is to save you the ordeal of trying to impress with volume instead of quality.
- Because a primary goal of the course is to teach professionalism, any academic dishonesty will be viewed as evidence that this goal has not been achieved, and will be grounded for receiving a grade of F. (See CEAS Procedures and Guideline Governing Academic Dishonesty, 1/81.)
- Homework assignments will be due at the *beginning of class.*

# LECTURE SCHEDULE (TENTATIVE)

| # | DATE('14) | CONTENT (reading) |
|---|---|---|
| 1 | 29-Aug | Introduction to Algorithms (1-27) |
| | 31-Aug | Asymptotic Notations (30-40) |
| 2 | 5-Sept | Program Analysis |
| | 7-Sept | Elementary Data Structure |
| 3 | 12-Sept | Dictionaries |
| | 14-Sept | NO CLASS Harvest Moon |
| 4 | 19-Sept | Dictionary Data Structures / Trees |
| | 21-Sept | Heapsort / Priority Queues |
| 5 | 26-Sept | Sorting |
| | 28-Sept | MIDTERM I |

| | | |
|---|---|---|
| 6 | 3-Oct | NO CLASS National Foundation Day |
| | 5-Oct | Sorting (cont.) |
| 7 | 10-Oct | Sorting (cont.) |
| | 12-Oct | Graph Data Structures |
| 8 | 17-Oct | NO CLASS - Attending Conference (make up date TBD) |
| | 19-Oct | Breadth-First Search |
| 9 | 24-Oct | Depth-First Search |
| | 26-Oct | Min. Spanning Tree |
| 10 | 31-Oct | Shortest Path |
| | 2-Nov | Backtracking |
| 11 | 7-Nov | Exhaustive Search & Backtracking |
| | 9-Nov | Dynamic Programming |
| 12 | 14-Nov | MIDTERM II |

| | | |
|---|---|---|
| | 16-Nov | Edit Distance |
| 13 | 21-Nov | DP Examples |
| | 23-Nov | NP-Completeness |
| 14 | 28-Nov | NP-Completeness Proofs |
| | 30-Nov | Reductions |
| 15 | 5-Dec | Harder reductions |
| | 7-Dec | |
| F | 21-Dec | FINIALS Content: cumulative; Duration: 12:30 - 15:00 |

Textbook Ch 1.

# INTRODUCTION TO ALGORITHMS

# WHAT IS AN ALGORITHM?

- Algorithms are the ideas behind computer programs.
- An algorithm is the thing which stays the same
  - whether the program is in Pascal running on a Cray in New York or is in BASIC running on a Macintosh in Kathmandu!
- To be interesting, an algorithm has to solve a general specified problem.
- An algorithmic problem is specified by describing the
  - **set of instances it must work** on and
  - what **desired properties the output must have**.

# EXAMPLE: SORTING

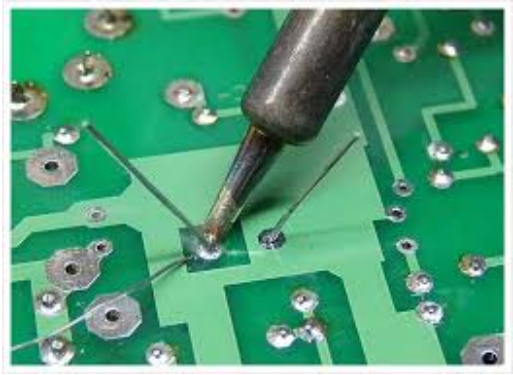
- *Input*: A sequence of N numbers a1:::an
- *Output*: the permutation (reordering) of the inp ut sequence such as $a_1 \leq a_2 \ldots \leq a_n$ .
- We seek algorithms which are
  - ***correct*** and
  - ***efficient***.

# CORRECTNESS

- For any algorithm, we must prove that it ***always*** returns the desired output for **all** legal instances of the problem.
- For sorting, this means even if
  - (1) the input is already sorted,
  - (2) it contains repeated elements.
- *Algorithm correctness is not obvious in many optimization problems!*

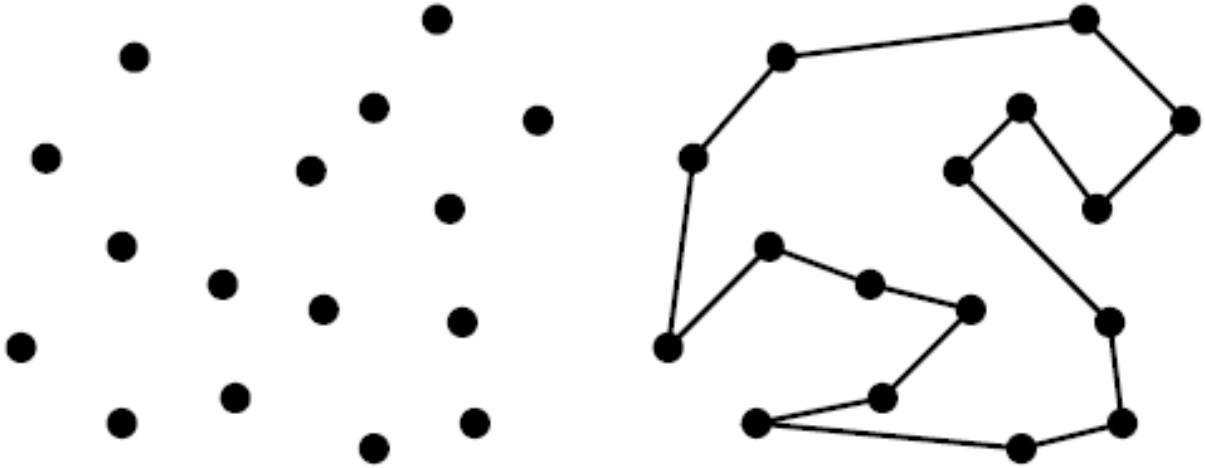
# ROBOT TOUR OPTIMIZATION

- Suppose you have a robot arm equipped with a tool, say a soldering iron.



- To enable the robot arm to do a soldering job, we must **construct an ordering of the contact points, so the robot visits (and solders) the points in order.**
- We seek the order which **minimizes the testing time** (i.e. travel distance) it takes to assemble the circuit board.

# FIND THE SHORTEST ROBOT TOUR



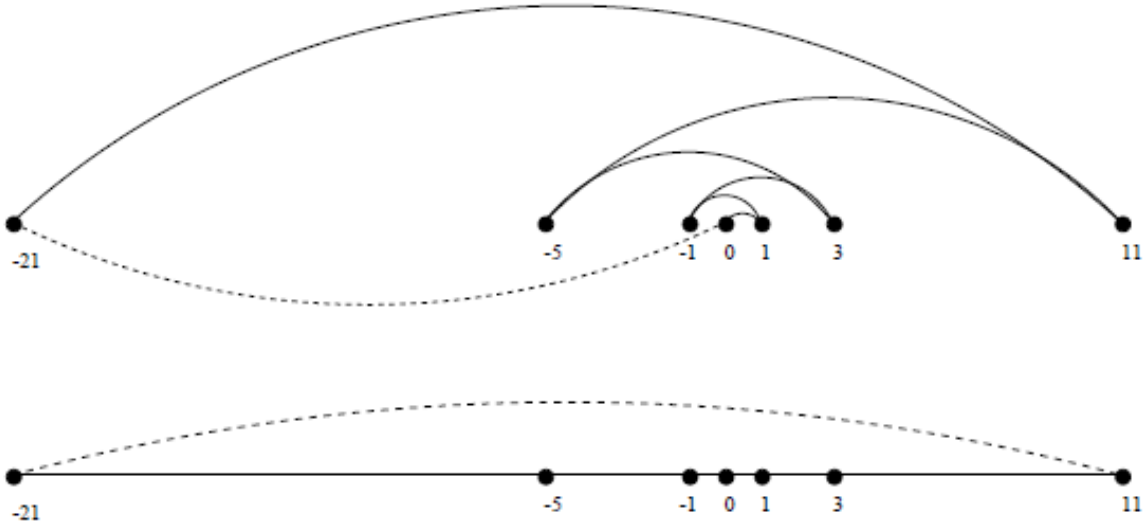
You are given the job to program the robot arm.
Give me an algorithm to find the best tour!

# NEAREST NEIGHBOR TOUR

- A popular solution starts at some point $p_0$ and then walks to its nearest neighbor $p_1$ first, then repeats from $p_1$, etc. until done.

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

# NEAREST NEIGHBOR TOUR IS WRONG!




Starting from the leftmost point will not fix the problem.

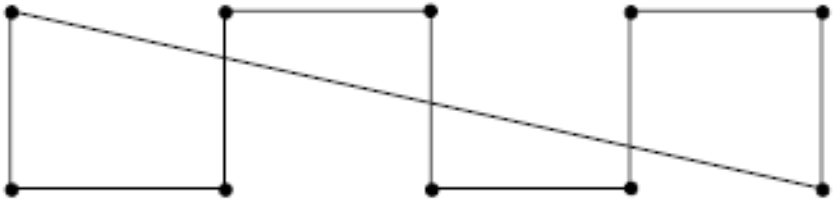
# CLOSEST PAIR TOUR

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.

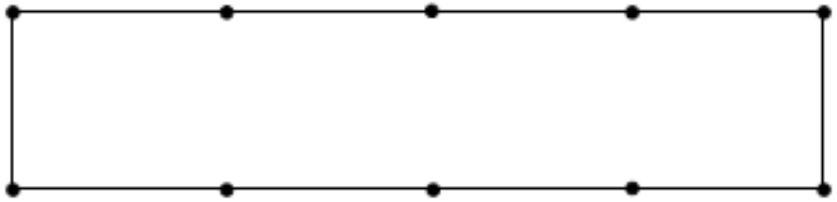
Let $n$ be the number of points in the set
$d = \infty$
For $i = 1$ to $n - 1$ do
For each pair of endpoints $(x, y)$ of partial paths
    If $dist(x; y) \leq d$ then
        $x_m = x$, $y_m = y$, $d = dist(x; y)$
    Connect $(x_m, y_m)$ by an edge
Connect the two endpoints by an edge.

# CLOSEST PAIR TOUR IS WRONG!

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



What the Algorithm does outputs



What the output should output

# A CORRECT ALGORITHM: EXHAUSTIVE SEARCH

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

Let $n$ be the number of points in the set
$d = \infty$
For each of the $n!$ permutations, $\Pi_i$, of the $n$ points
    If ($cost(\Pi_i) \leq d$) then
        $d = cost(\Pi_i)$ and $P_{min} = i$
Return $P_{min}$

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

# EXHAUSTIVE SEARCH IS SLOW!

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

No efficient, correct algorithm exists for the ***traveling salesman problem***, as we will see later.

# EFFICIENCY: WHY NOT USE A SUPERCOMPUTER?

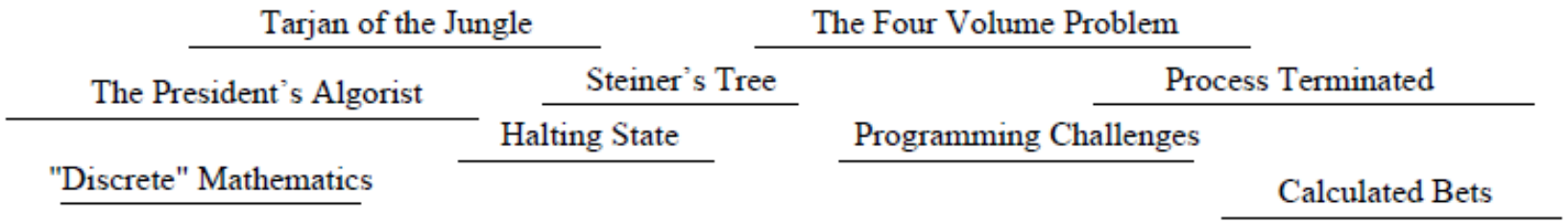
- A faster algorithm running on a slower computer will ***always*** win for sufficiently large instances, as we shall see.
- Usually, problems don't have to get that large before the faster algorithm wins.

# EXPRESSING ALGORITHMS

- We need some way to express the sequence of steps comprising an algorithm. In order of increasing precision, we have
  - **English,**
  - **pseudocode,** and
  - real **programming languages.**
- Unfortunately, ease of expression moves in the reverse order.
- I prefer to describe the *ideas* of an algorithm in English, moving to pseudocode to clarify sufficiently tricky details of the algorithm.
- Algorithms *problems* must be carefully specified to allow a provably correct algorithm to exist. We can find the "shortest tour" but not the "best tour".

# SELECTING THE RIGHT JOBS

A movie star wants to the select the maximum number of staring roles such that no two jobs require his presence at the same time.



# THE MOVIE STAR SCHEDULING PROBLEM

- *Input:* A set I of n intervals on the line.
- *Output:* What is the largest subset of mutually non-overlapping intervals which can be selected from *I*?
- **Give an algorithm to solve the problem!**

# EARLIEST JOB FIRST

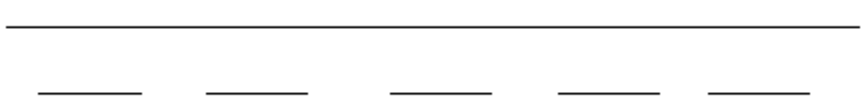
Start working as soon as there is work available:

EarliestJobFirst(I)

Accept the earliest starting job *j* from *I* which does not overlap any previously accepted job, and repeat until no more such jobs remain.

# EARLIEST JOB FIRST IS WRONG!

The first job might be so long (War and Peace) that it prevents us from taking any other job.



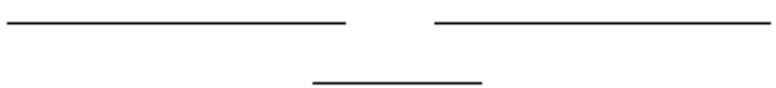
# SHORTEST JOB FIRST

Always take the shortest possible job, so you spend the least time working (and thus unavailable).

```
ShortestJobFirst(I)
While (I != NULL) do
    Accept the shortest possible job j from I.
    Delete j, and intervals which intersect j from I.
```

# SHORTEST JOB FIRST IS WRONG!

Taking the shortest job can prevent us from taking two longer jobs which barely overlap it.



# DEMONSTRATING INCORRECTNESS

- Searching for counterexamples is the best way to disprove the correctness of a heuristic.
  - *Think exhaustively:* There are only a small number of possibi lities for the smallest nontrivial value of *n*.
  - *Think small:* Think about all small examples.
  - *Go for a tie:* Think about examples with ties on your decision criteria (e.g. pick the nearest point)
  - *Seek extremes:* Think about examples with extremes of big and small.
  - *Hunt for the weakness: If a proposed algorithm is of the form "always take the biggest" (better known as the greedy algorithm), think about why that might prove to be the wrong thing to do.*

# FIRST JOB TO COMPLETE

Take the job with the earliest completion date:

```
OptimalScheduling(I)
While (I != NULL) do
    Accept job j with the earliest completion date.
    Delete j, and intervals which intersect j from I.
```

# FIRST JOB TO COMPLETE IS OPTIMAL!

Other jobs may well have started before the first to complete (x), but all must at least partially overlap each other.

Thus we can select at most one from the group.

The first these jobs to complete is x, so the rest can only block out more opportunities to the right of x.

# INDUCTION AND RECURSION

- Failure to find a counterexample to a given algorithm does not mean *"it is obvious"* that the algorithm is correct.
- Mathematical **induction** is a very useful method for proving the correctness of **recursive algorithms.**
- Recursion and induction are the same basic idea:
  - (1) basis case,
  - (2) general assumption,
  - (3) general case.

Ex> proving

$$\sum_{i=1}^{n} i = n(n+1)/2$$