Lecture 1: Introduction to Algorithms

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

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 $a_1 ... a_n$

Output: the permutation (reordering) of the input 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.

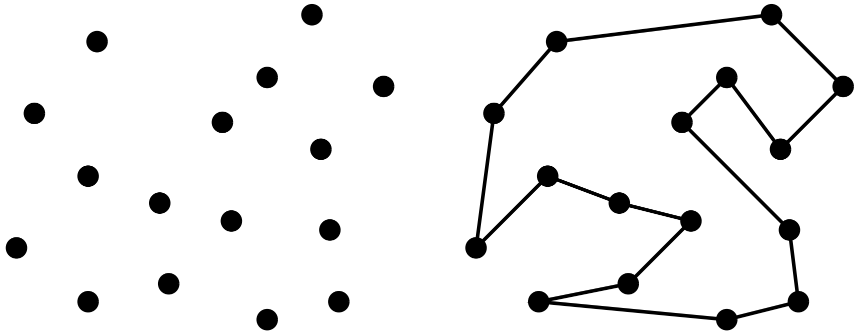
For sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements.

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



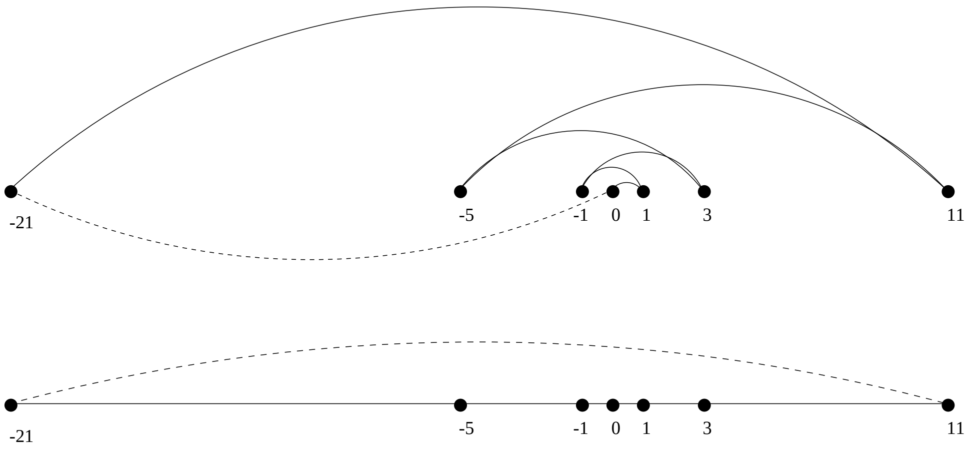
You are given the job to program the robot arm. Give me an algorithm to find the best tour!

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.

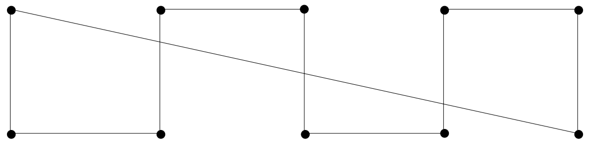
Closest Pair Tour

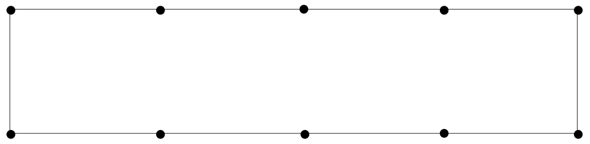
Another idea is to repeatedly connect the closest pair of points whose connection will not cause a cycle or a three-way branch, until all points are in one tour.

```
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:





A Correct Algorithm: Exhaustive Search

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

```
$d = \infty$
For each of the $n!$ permutations $\Pi_i$ of the $n$ points
        If $(cost(\Pi_i) \leq d)$ then
                $d = cost(\Pi_i)$ and $P_{min} = \Pi_i$
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.

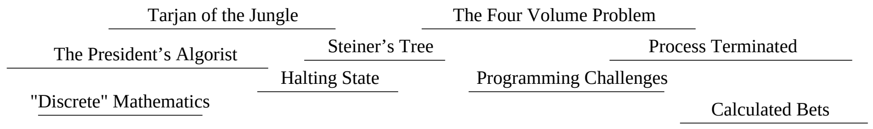
I prefer to describe the *ideas* of an algorithm in English, moving to pseudocode to clarify sufficiently tricky details of the algorithm.

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".

%swallow

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!

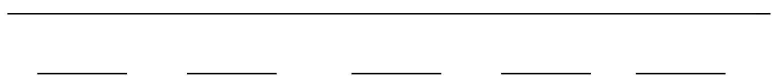
Earliest Job First

Start working as soon as there is work available:

```
EarliestJobFirst(I)
    Accept the earlest 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.



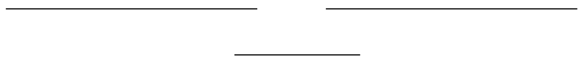
Shortest Job First

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

```
ShortestJobFirst(I)
    While $(I \neq \emptyset)$ 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.



First Job to Complete

Take the job with the earliest completion date:

```
OptimalScheduling(I)
    While $(I \neq \emptyset)$ do
        Accept job $j$ with the earliest completion date.
        Delete $j$, and whatever intersects $j$ 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 opportunties to the right of x.

Demonstrating Incorrectness

Searching for counterexamples is the best way to disprove the correctness of a heuristic.

- Think about all small examples.
- Think about examples with ties on your decision criteria (e.g. pick the nearest point)
- Think about examples with extremes of big and small. . .

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.

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