Lecture 1:
Introduction to Algorithms (1997)

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.

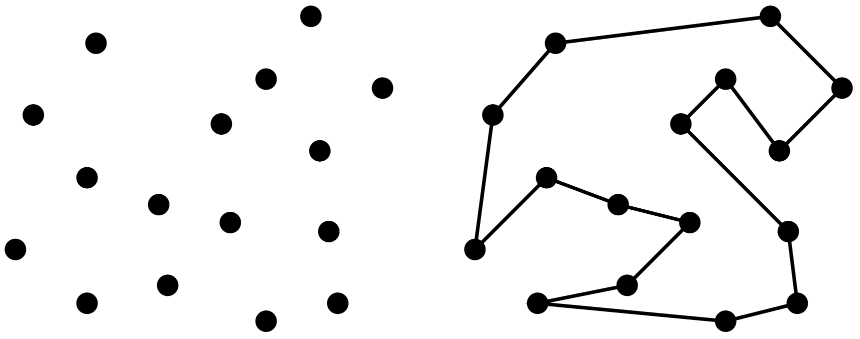
For sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements.

Correctness is Not Obvious!

The following problem arises often in manufacturing and transportation testing applications.

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 first contact point, then visits the second point, third, and so forth until the job is done.

Since robots are expensive, we need to find the order which minimizes the time (ie. travel distance) it takes to assemble the circuit board.



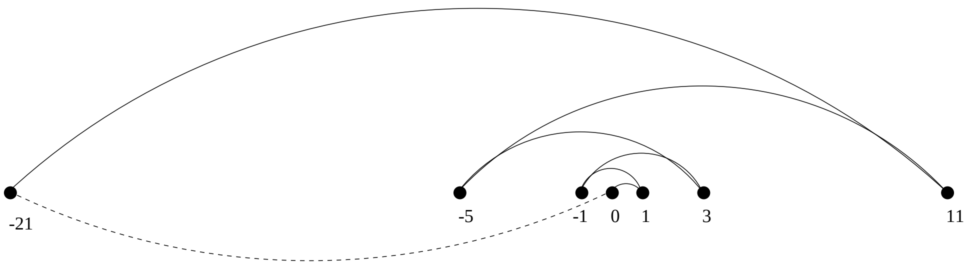
You are given the job to program the robot arm. Give me an algorithm to find the best tour!

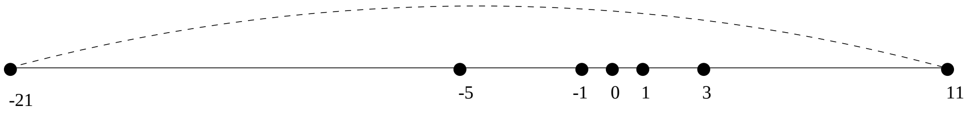
Nearest Neighbor Tour

A very 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 $p_i$
```

This algorithm is simple to understand and implement and very efficient. However, it is **not correct!**





Always starting from the leftmost point or any other point will not fix the problem.

Closest Pair Tour

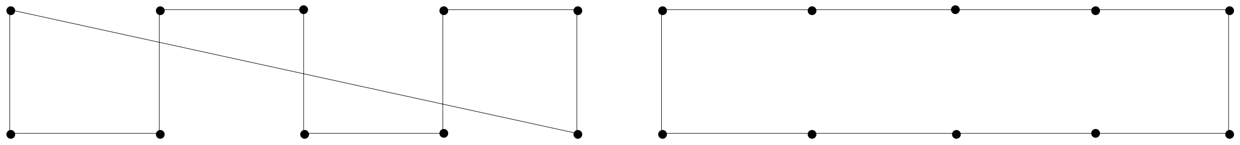
Always walking to the closest point is too restrictive, since that point might trap us into making moves we don't want. Another idea would be to repeatedly connect the closest pair of points whose connection will not cause a cycle or a three-way branch to be formed, until we have a single chain with all the points in it.

```
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.

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



This algorithm is **not correct**!

A Correct Algorithm

We could try all possible orderings of the points, then select the ordering 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.
Because it trys all $n!$ permutations, it is extremely slow, 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 just use a supercomputer?”

Supercomputers are for people too rich and too stupid to design efficient algorithms!

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.

The RAM Model

Algorithms are the *only* important, durable, and original part of computer science *because* they can be studied in a machine and language independent way.
The reason is that we will do all our design and analysis for the RAM model of computation:

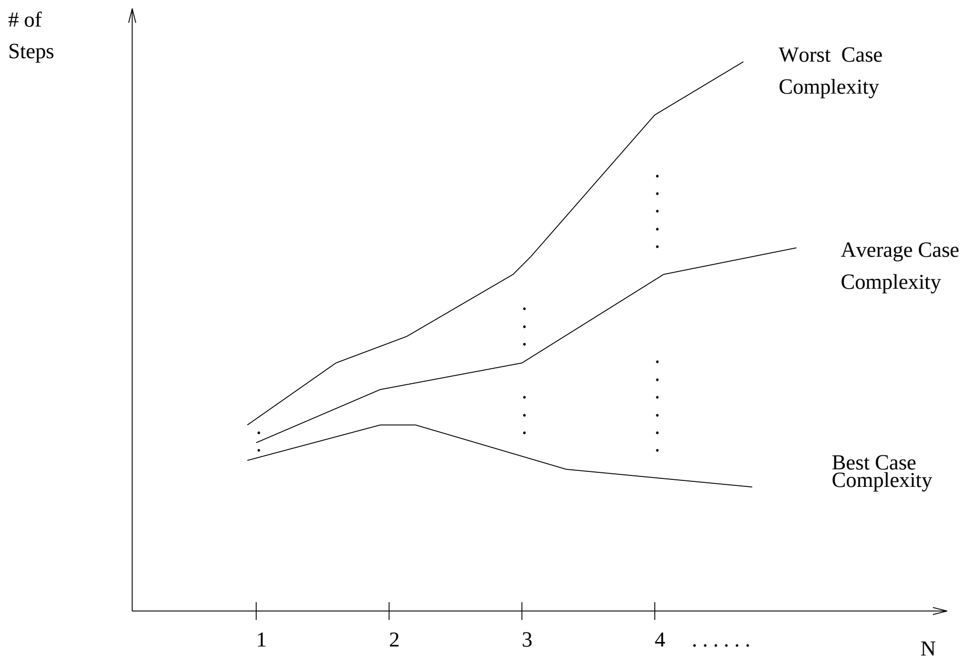
- Each ”simple” operation (+, -, =, if, call) takes exactly 1 step.
- Loops and subroutine calls are *not* simple operations, but depend upon the size of the data and the contents of a subroutine. We do not want “sort” to be a single step operation.

- Each memory access takes exactly 1 step.

We measure the run time of an algorithm by counting the number of steps.
This model is useful and accurate in the same sense as the flat-earth model (which *is* useful)!

Best, Worst, and Average-Case

The *worst case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size n.



The *best case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.

The *average-case complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size n.
Each of these complexities defines a numerical function – time vs. size!

Insertion Sort

One way to sort an array of n elements is to start with a_n empty list, then successively insert new elements in the proper position:

$$a_1 \leq a_2 \leq \ldots \leq a_k \mid a_{k+1} \ldots a_n$$

At each stage, the inserted element leaves a sorted list, and after *n* insertions contains exactly the right elements. Thus the algorithm must be correct.

But how *efficient* is it?

Note that the run time changes with the permutation instance! (even for a fixed size problem)

How does insertion sort do on sorted permutations?

How about unsorted permutations?

Exact Analysis of Insertion Sort

Count the number of times each line of pseudocode will be executed.

Line	InsertionSort(A)	#Inst.	#Exec.
1	for j:=2 to len. of A do	c1	n
2	key:=A[j]	c2	n-1
3	/* put A[j] into A[1..j-1] */	c3=0	/
4	i:=j-1	c4	n-1
5	while $i > 0 \& A[1] > key$ do	c5	tj
6	A[i+1]:= A[i]	c6	
7	i := i-1	c7	
8	A[i+1]:=key	c8	n-1

The **for** statement is executed $(n-1)+1$ times (why?)
Within the **for** statement, ”key:=A[j]” is executed n-1 times.
Steps 5, 6, 7 are harder to count.
Let $t_j = 1+$ the number of elements that have to be slide right to insert the *j*th item.
Step 5 is executed $t_2 + t_3 + ... + t_n$ times.
Step 6 is $t_{2-1} + t_{3-1} + ... + t_{n-1}$.

Add up the executed instructions for all pseudocode lines to get the run-time of the algorithm:

$c_1 * n + c_2(n-1) + c_4(n-1) + c_5 \Sigma_{j=2}^{n} t_j + c_6 \Sigma_{j=2}^{n}(t_j - 1)$
$+ c_7 \Sigma_{j=2}^{n}(t_j - 1) + c_8$

What are the $t_j's$? They depend on the particular input.

Best Case

If it’s already sorted, all t_j’s are 1.
Hence, the best case time is

$$c_1 n + (c_2 + c_4 + c_5 + c_8)(n - 1) = Cn + D$$

where C and D are constants.

Worst Case

If the input is sorted in *descending* order, we will have to slide *all* of the already-sorted elements, so $t_j = j$, and step 5 is executed

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