A working definition: a sequence of instructions for performing a task

- Some common properties of algorithms:
  - provably correct
  - clear and unambiguous
  - automatable
  - efficient

Why Study Algorithms

- “Wrong” algorithms harmful
  - Online poker
  - Therac-25
- Algorithms have real world value
  - Elevator scheduling
  - Wall Street “quants”
  - LAX airport security
- New problems often related to old ones
- Change the world...

World Changers?

- All applied math (back to ancient times)
- Public key cryptography
- PageRank (Google)

In This Course

We will:

- Study many important/influential algorithms
- Examine methods for analyzing efficiency
- Explore common algorithm design ideas

Example: Sorting

- Input: a list of elements, e.g. integers
- Output: a list of the input elements in sorted order
- A simple solution:
  - Find the minimum element in the list
  - Swap it with the first element in the list
  - Sort the sublist after the first element recursively
- This sorting algorithm is named “selection sort”
Selection Sort

Our simple solution:
- Find the minimum element in the list
- Swap it with the first element in the list
- Recursively sort the sub-list following the first element

Is this clear and unambiguous?
Is it automatable?
Is it correct?
Is it efficient?

Digression: Pseudocode

- Just what it sounds like: it is sort of like code, but it isn’t code (except when it is...)
- Language-independent (except when it isn’t...)
- Informal (or formal)

You get the idea... there is no one way to do pseudocode.

Pseudocode I

```pseudocode
selectionSort(x) // x is a list of N integers
if x has one element, do nothing
otherwise:
    find the minimum element of x
    swap 1st element with minimum element
    call selectionSort on sublist starting at element 2
This is a very informal description. It is easy to understand and explain.
```

Pseudocode II

```pseudocode
selectionSort(x) // x is a list of N integers
if length(x) = 1 then return
k ← 1
N ← length(x)
for i ← 2 to N
    if x[i] < x[k] then k ← i
t ← x[k]
x[k] ← x[1]
x[1] ← t
selectionSort(x[2..N])
```

Yet another approach... generally the form I will use.
Use what works!

Pseudocode III

```pseudocode
selectionSort(x) // x is a list of N integers
if N = 1 then return
i ← arg min x[i] // arg min F returns x value minimizing F
swap x[i] with x[1]
selectionSort(x[2..N])
```

This is a very "program-y" description. It is easy to code in a procedural language.
Correctness of Selection Sort

Is this correct?

\[
\text{selectionSort}(x) \\
\text{if length}(x) = 1, \text{return } x \\
k \leftarrow \text{arg min}_i x[i] \\
\text{swap } x[k] \text{ and } x[1] \\
\text{selectSort}(x[2..N]) \\
\text{Note the last line sets us up for an inductive proof, with the first line providing the base case.}
\]

Digression: Induction

Prove \( A(n) \), for any \( n \).
\( A(n) \) is some statement, such as
“Selection sort correctly sorts lists of length \( n \)”

Method:
show true for a base case such as \( n = 1 \)
show that \( A \) true for \( n \) implies \( A \) true for \( n+1 \)

Convince yourself this works...

Induction Practice

• \( 1 + 3 + \ldots + 2N - 1 = N^2 \), for all \( N \geq 1 \)
• \( 3^N > 2^N \) for all \( N > 0 \)
• Triangle inequality: \( |x_1 + x_2 + \ldots + x_n| \leq |x_1| + |x_2| + \ldots + |x_n| \)

Induction Generalizations

• Sometimes more than one base case is needed: \( n = 1, n = 2, \text{ etc.} \)
• In general, in the inductive step (proving true for \( n+1 \)) you can assume true for \( n, n-1, n-2, \text{ etc.} \) (not just \( n \)).

Efficiency

How fast is selection sort?

\( \text{What do we mean by fast?} \)
**Efficiency**

"Basic computer steps"
- Constant time operations (on the computer)
- Effectively independent of inputs
- Examples
  - Arithmetic (bounded precision/magnitude)
  - Variable assignment
  - Comparisons/branching
  - Array access
- Non-examples (?)
  - Loops
  - Array copy
  - Function calls

**Measure of efficiency:**
# of steps as function of input size
We typically use \( N \) or \( n \) to notate input size
- Sort a list of \( N \) elements
- Decrypt an \( n \)-bit signal
- Multi-dimension input sizes, too: invert \( M \times N \) matrix

We’ll formalize this more later.

---

**Efficiency of Selection Sort**

\[
\text{selectionSort}(x: \text{ a list of } N \text{ integers})
\]

if \( N = 1 \) return 1 step
else
  \( i \leftarrow \text{argmin}_i \) \( x[i] \)<2N steps*
  swap \( x[i] \) and \( x[1] \) 3 steps**
  \( \text{selectionSort}(x[2..N]) \) ???

* \( \text{argmin} \ x \)
  \( k \leftarrow 1 \) 1 step
  for \( i \leftarrow 2 \) to \( N \)
  \( \text{if } x[i] < x[k], k \leftarrow i \) 1-2 steps
  \( t \leftarrow x[i] \)
  \( x[i] = x[k] \)
  \( x[k] = t \)

**Simplifying**

\[
\text{selectionSort}(x: \text{ a list of } N \text{ integers})
\]

if \( N = 1 \) return \(~2N+4\) steps
else
  \( i \leftarrow \text{argmin}_i \) \( x[i] \)
  swap \( x[i] \) and \( x[1] \) ???
  \( \text{selectSort}(x[2..N]) \) ???

Note that \(~2N+4\) is not very precise, and it is based on a sloppy assumption that all "basic computer steps" have the same cost.

As we’ll see, it turns out that constants don’t matter (much). So instead of \(~2N+4\), we’ll write \( O(N) \). [say "order \( N \)""]

---

**Recurrence Relations**

Let’s introduce a new function: let \( T(N) \) be the running time of \( \text{selectionSort} \) on a size \( N \) list. We have

\[
\text{selectionSort}(x: \text{ a list of } N \text{ integers})
\]

... stuff that takes \( O(N) \)
\( \text{selectionSort}(x[2..N]) \) \( T(N-1) \)

Now we can write an expression for the running time:

\( T(N) = O(N) + T(N-1) \).

This kind of recursive expression is called a recurrence relation.

---

**Solving the Recurrence**

A good starting point is to write out \( T(N) \) for small \( N \):

\[
\begin{align*}
T(1) &= O(1) & \text{// if } N = 1 \text{ return} \\
T(2) &= O(2) + T(1) = O(2 + 1) \\
T(3) &= O(3) + T(2) = O(3 + 2 + 1) \\
& \cdots \\
\text{the pattern seems clear now!} \\
T(N) &= 1 + 2 + \cdots + N = O((N^2+N)/2)
\end{align*}
\]
Wrapping Up

We’ve determined that the running time for selection sort is $O((N^2+N)/2)$.

[As previously mentioned, we can ignore the factor of $\frac{1}{2}$; as we’ll also see, we can ignore lower order polynomials, so we can write simply $O(N^2)$]

Coming up next:
Asymptotic analysis: a formal treatment of “big O”