COMPSCE 130
Design and Analysis of Algorithms
1 - Introduction

What Is an Algorithm?
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
  – Example: online poker
• Choice of algorithm impacts utility
  – Example: elevator scheduling
• New problems often related to old ones
• Change the world...

World Changers?
• All applied math (back to ancient times)
• Public key cryptography
• PageRank (Google)
• Kalman filter

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
  – Recursively sort the sublist after the first element

• 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

selectionSort(x)       // x is a list of N integers
    find the minimum element of x
    let k = index of minimum element
    swap 1st element with kth element
    call selectionSort on sublist starting at element 2

This is a very informal description. It is easy to understand and explain.

Pseudocode II

selectionSort(x)
k ← 1
N ← length(x)
for i ← 2 to N
    if x[i] < x[k], k ← i
t ← x[k]
x[k] ← x[i]
x[i] ← t
    selectionSort(x[2..N])

This is a very “program-y” description. It is easy to code in a procedural language.

Pseudocode III

selectionSort(x)
i ← argmin, x[i]
swap x[i] with x[1]
selectionSort(x[2..N])

Yet another approach...
Use what works!

Correctness of Selection Sort

Is this correct?

selectionSort(x)
i ← argmin, x[i]
swap x[i] and x[1]
selectionSort(x[2..N])

Note the last line sets us up for an inductive proof. If we can prove the base case (e.g., for N=1), we are done.
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: n = 1
  show that A true for n implies A true for n+1

Convince yourself this works...

Correctness of Selection Sort

Is this correct?

\[
\text{selectionSort}(x) \\
\{ \\
\quad i = \text{argmin}\, x[i] \\
\quad \text{swap}\, x[i] \text{ and } x[i+1] \\
\quad \text{selectionSort}(x[2..N]) \\
\}
\]

What happens when N = 1?
Hm. It seems we didn’t consider that case.

Correctness of Selection Sort

Let’s be a bit more precise in our pseudocode:

```
selectionSort(x, a list of N integers)
if N = 1 return 
x = [x[1]] is sorted
else 
i = \text{argmin}\, x[i] 
\text{swap}\, x[i] \text{ and } x[i+1] 
\text{selectionSort}(x[2..N])
```

First line \(\Rightarrow\) we sort lists of length 1 correctly
Last line \(\Rightarrow\) we sort lists of length n correctly if we sort lists of length n+1 correctly.
By induction, selectionSort is correct. \(\square\)

Efficiency

How fast is selection sort?

What do we mean by fast?

“Basic computer steps”
- Constant time operations (on the computer)
- Effectively independent of inputs
- Examples
  - Arithmetic (not arbitrary precision/magnitude)
  - Array access
  - sin, cos, log, exp
- Non-examples (?)
  - Array copy
  - Set queries (e.g., hasKey())
  - Vector/matrix math

Efficiency

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 x 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

\[\begin{align*}
\text{else} & \quad 1 \text{ step} \\
\text{i} & \leftarrow \text{argmin}_x \{x[i]\} \\
\text{swap } x[i] \text{ and } x[1] & \quad <2N \text{ steps}\* \\
\text{selectionSort}(x[2..N]) & \quad 3 \text{ steps}\* \\
\text{swap } x[i] \text{ and } x[1] & \quad ???
\end{align*}\]

\*argmin \( x \)
for \( i \leftarrow 2 \) to \( N \)
if \( x[i] < x[k] \), \( k \leftarrow 1 \)
1 step
N-1 loops
1-2 steps

Recurrence Relations

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

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

... stuff that takes \( O(N) \)
... 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.

Simplifying

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

if \( N = 1 \) return

\[\begin{align*}
\text{else} & \quad \sim 2N + 4 \text{ steps} \\
\text{i} & \leftarrow \text{argmin}_x \{x[i]\} \\
\text{swap } x[i] \text{ and } x[1] & \quad <2N \text{ steps}\*
\end{align*}\]

Note that \( \sim 2N+4 \) 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 \( \sim 2N+4 \), we’ll write \( O(N) \). [say “order \( N \)”]

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”