Scarborough College
University of Toronto
Fall 1997

CSC A06F: Introduction to Computer Programming
Lecture notes 10:
Algorithms and searching

Reading:

- Lewis and Loftus, section 13.2.

Problem solving

Computer science is about using ***computation*** to solve complex problems:

- Look for a record in a database.
- Compute the cheapest route for a travelling salesman to take between cities.
- Find the optimum ordering of tasks in a large construction project.
- Translate text from one language to another.
- Realistically animate a burning fire.
- Have a robot move without running into things.

So, programming is just a tool: a means to an end.

Algorithms

Use an *algorithm* to describe how to solve a problem.

An **algorithm** is a precisely-defined finite sequence of steps that solves a problem.

Usually you write an algorithm in natural language (like English), or in *pseudocode* .

Then you translate your algorithm into a programming language: Java, C, C++, Fortran, Lisp, ...

Simple algorithms can be implemented in a single method. But complex algorithms might require a huge system of classes and objects.

Specifications

Before you write an algorithm you need a precise **specification** of what it should do.

Preconditions: What needs to be true *before* the algorithm runs. (I.e., what the input looks like.)

Postconditions: What should be true *after* it runs. (I.e., what the output looks like.)

With proper specifications, you'll know what you can assume in writing your program, and others will know what to expect when it runs.

Specifications for simple searching

Specification:

Preconditions: given an *array* of integers, the *length* of the array, and a *key* (an integer) to look for,

Postconditions: return the index of the key in the array, or return -1 if the key is not in the array.

Is this precise enough?

What if the key occurs more than once in the array?

(This is a simple type of a search; normally you would be searching for a record of some sort whose key matches the given key.)

Algorithm for linear search

$found \leftarrow -1$
loop over all elements in the $array$
  if the element at index i equals key then
    $found \leftarrow i$
return i

Method for linear search

```
int linearSearch (int[] array, int length, int key) {




}
```

Efficiency of linear search

How fast is the linear search? How much work does it do?

How many iterations does the loop do?

Let's say there are n integers in the array. How many of these will it have to look at:

- in the worst case?
- in the best case?
- in the average case?

Can we make it better?

Binary Search

Binary search is an order of magnitude faster than linear search.

> Algorithm:
>
> Start at the middle of array. If the key is on the left side, forget about the right side, and repeat this process on the left side. If the key is on the right side, repeat this process on the right side. You eventually close in on the key.

What assumption do we have to add to the preconditions for this to work?

Algorithm for binary search

$left \leftarrow 0$
$right \leftarrow length - 1$
loop until $left = right$
    $middle \leftarrow (left + right)/2$
    if element $middle$ of $array$ is $\leq key$
        $right \leftarrow middle$
    else
        $left \leftarrow middle + 1$
if element $left$ of $array$ equals key
    return $left$
else return -1

How fast is binary search?

Consider an array of n elements.

If you doubled the length of the array to $2n$ elements, how many more loop iterations would it do?

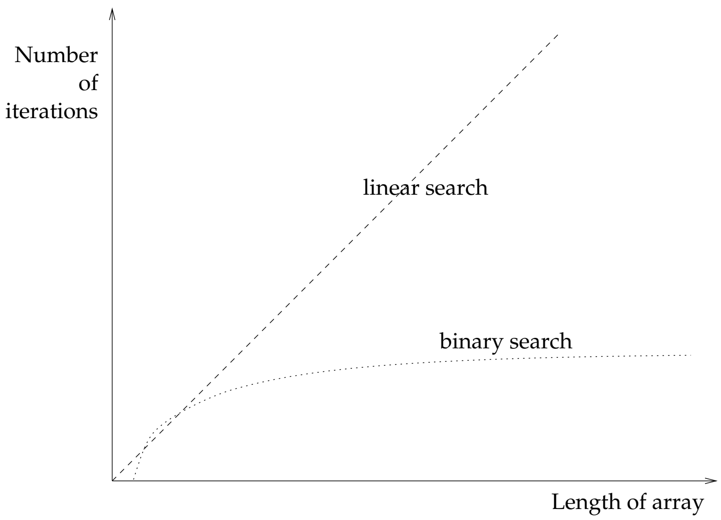
This is a logarithmic relationship:

Worst case: It will do $\log_2 n$ iterations.

Best case:

Average case:

Efficiency of searching



Binary search is more efficient than linear search.