Lecture 2 – Algorithms and their cost
Pages 23-28 in your textbook

Algorithms

The concept of an algorithm is key in computer programming because ultimately every program that you write will be implementing some sort of algorithm. Some algorithms will be simple and some will be very complex.

An algorithm is a defined set of steps, or a process, that is followed to solve a problem. You can think of an algorithm as being similar to a recipe. There is an objective to accomplish (problem to solve) and the order of the steps that you take to solve that problem matters. Algorithms generally have a set of values that is the input, and then transforms the input in some way to produce an output.

For example, assume we have input as a sequence of numbers:
A = <54, 34, 23, 45, 56, 90>

An algorithm that puts those numbers in order (aka. Sort them) would produce an output of

A = <23, 34, 45, 54, 56, 90>

Algorithms often require a way to store the data that allows for implementation of the algorithm: a data structure. In the sorting example above, the data structure could be an array, for example. In C++, we could write

int A[6] = {54, 34, 23, 45, 56, 90};

How do we evaluate an algorithm?
Given a problem to solve, such as sorting a sequence of numbers, how do we evaluate whether one algorithm is better than another algorithm? Of course, we want our algorithm to produce a correct solution, so we will assume that this is the case in our evaluation, that we are only evaluating correct algorithms. For example, if our sorting algorithm produces:

A = <90, 54, 23, 34, 45, 56>

that's not correct, and any other method of evaluation is irrelevant. The algorithm did not produce a correct answer.

Cost
Algorithms are evaluated by their cost, which is calculated from the number of instructions executed for different sizes of input. From the cost, we can calculate an algorithms complexity.

Example 1: Write an algorithm to determine the index of a specified value in an array, e.g. we want to know if A contains a 50 and the index of the 50.

First, the pseudocode:

```
findItem(A, v) //A is the array, and v is the value to find
  index = -1
  for i=1 to A.length //pseudocode convention, start at 1
    if A[i] == v
      index = i
  return index
```

This is pseudocode, so we're not concerned with types, and indexing is starting at 1. We use A.length as a variable here. In code we may not have this available, we may need to pass an additional variable to the function since the C++ array doesn't have a length method. We assume that there is other code to call this function that handles the -1 to signify that the value is not found. This algorithm is implemented in C++ in the Lecture2AlgsAndCost.cpp file.

Pseudocode	**Worst-case cost***
`findItem(A, v)`	
`  index = -1`	1
`  for i=1 to A.length`	A.length
`    if A[i] == v`	A.length
`      index = i`	A.length
`  return index`	1

*Assume each line has a cost = 1.

The cost is calculated from the lines within the algorithm. The function header `findItem` is not included. The line `index = i` is only evaluated when `if A[i] == v` is true.

Calculate cost of the following calls to findItem(A,v)

A = <45, 34, 32, 34>
findItem(A, 34) = 1+4+4+2+1=12
findItem(A, 25) = 1+4+4+0+1=10
A = <45, 34, 32, 34, 56, 23, 12>
findItem(A, 34) = 1+7+7+2+1=18
findItem(A, 25) = 1+7+7+0+1=16
A = <45, 45, 45, 45, 45, 45, 45>
findItem(A, 45) = 1+7+7+7+1=23

The for loop is the biggest contributor to the cost of the algorithm, which changes as the size of the for loop changes based on the size of A. The cost of index=-1 is constant, it is 1 regardless of how big A is. Worst case happens when every value in the array is the value you're looking for.

Change the code slightly to return when the value is found. The cost changes, as does the result of the algorithm. The previous algorithm would actually return the index of the last instance of the value found. This algorithm will return the index of the first instance of the value.

Pseudocode	**Worst-case cost***
`findItem(A, v)`	1
`  index = -1`	1
`  for i=1 to A.length`	A.length
`    if A[i] == v`	A.length
`      return index`	1
`  return index`	1

*Assume each line has a cost = 1.

Calculate cost of the following calls to findItem(A,v)

A = <45, 34, 32, 34>
findItem(A, 34) = 1+2+2+1=6 //item found in the array
findItem(A, 25) = 1+4+4+0+1=10 //item not found in the array
A = <45, 34, 32, 34, 56, 23, 12>
findItem(A, 34) = 1+2+2+1=6 //item found in the array, array has more items
findItem(A, 25) = 1+7+7+0+1=16
A = <45, 45, 45, 45, 45, 45, 45>
findItem(A, 45) = 1+1+1+1=4

The situation that produced the worse case is different for this algorithm, it happens when the value is not found in the array. However, it's still the size of the for loop that influences worse case more than the other operations. The `index=-1` line, for example, still contributes a cost of only 1.

Note: there's a coding practice is this algorithm that isn't really advised. It's good practice to have only one return point in your functions, but we have two. A better approach here that would produce the same result is to use a while loop that exits when the value is found.

Array shifting example

Example 2: Write an algorithm that finds a specified value in an array, removes it, and then shifts the array to fill the empty spot. The algorithm should remove all instances of the value, not just the first one found. For example, given the array

<54, 34, 45, 23, 90, 91>,

if you want to remove the 45, you would generate a new array that looks like <54, 34, 23, 90, 91, 0>. We're assuming a fixed array size here, rather than changing the size of the array as we remove items.

Pseudo-pseudo code
Check each item in array. If item is value we're searching for, then shift all items from that position to end of the array forward by one position.

Pseudocode	**Worst-case cost***
`removeItem(A, v)`	
`  for i=1 to A.length`	A.length
`    if A[i] == v`	A.length
`      for j=i to A.length-1`	$\sum_{j}^{n-1} j$
`        A[j] = A[j+1]`	$\sum_{j}^{n-1} j$
`      A[A.length] = 0`	A.length
`  return A`	1

*Assume each line has a cost = 1, and n = A.length

Calculate cost of the following calls to removeItem(A,v)
A = <45, 34, 32, 36>
removeItem(A, 34) = 4+4+2+2+1+1=14 //item found in the array

There's a limitation to this algorithm. It doesn't work correctly when there are instances of the number to remove right next to each other. For example, for an input of <34, 34, 32, 36> the algorithm would produce <34, 32, 36, 0>, which is not correct. How could we design an algorithm that wouldn't have this problem and what would its cost be?