Computer Science 431
Algorithms
The College of Saint Rose
Spring 2013

Topic Notes: Introduction and Overview

Welcome to Algorithms!

What is an Algorithm?

A possible definition: a step-by-step method for solving a problem.

An algorithm does not need to be something we run on a computer in the modern sense. The notion of an algorithm is much older than that. But it does need to be a formal and unambiguous set of instructions.

The good news: if we can express it as a computer program, it's going to be pretty formal and unambiguous.

Example: Computing the Max of 3 Numbers

Let's start by looking at a couple of examples and use them to determine some of the important properties of algorithms.

Our first example is finding the maximum among three given numbers.

Any of us could write a program in our favorite language to do this:

```
int max(int a, int b, int c) {
  if (a > b) {
    if (a > c) return a;
    else return c;
  }
  else {
    if (b > c) return b;
    else return c;
  }
}
```

The algorithm implemented by this function or method has *inputs* (the three numbers) and one *output* (the largest of those numbers).

The algorithm is defined *precisely* and is *deterministic*.

This notion of determinism is a key feature: if we present the algorithm multiple times with the same inputs, it follows the same steps, and obtains the same outcome.

A *non-deterministic* procedure could produce different outcomes on different executions, even with the same inputs.

Code is naturally deterministic – how can we introduce non-determinism?

It's also important that our algorithm will eventually terminate. In this case, it clearly does. In fact, there are no loops, so we know the code will execute in just a few steps. An algorithm is supposed to solve a problem, and it's not much of a solution if it runs forever. This property is called *finiteness*.

Finally, our algorithm gives the right answer. This very important property, *correctness*, is not always easy to achieve.

It's even harder to *verify* correctness. How can you tell if you algorithm works for all possible valid inputs? An important tool here: formal *proofs*.

A good algorithm is also *general*. It can be applied to all sets of possible input. If we did not care about generality, we could produce an algorithm that is quite a bit simpler. Consider this one:

```
int max(int a, int b) {
   if (a > 10 && b < 10) return a;
}
```

This gives the right answer when it gives any answer. But it does not compute any answer for many perfectly valid inputs.

We will also be concerned with the *efficiency* in both time (number of instructions) and space (amount of memory needed).

Why Study Algorithms?

The study of algorithms has both *theoretical* and *practical* importance.

Computer science is about problem solving and these problems are solved by applying algorithmic solutions.

Theory gives us tools to understand the efficiency and correctness of these solutions.

Practically, a study of algorithms provides an arsenal of techniques and approaches to apply to the problems you will encounter. And you will gain experience designing and analyzing algorithms for cases when known algorithms do not quite apply.

We will consider both the *design* and *analysis* of algorithms, and will implement and execute some of the algorithms we study.

We said earlier that both time and space efficiency of algorithms are important, but it is also important to know if there are other possible algorithms that might be better. We would like to establish theoretical *lower bounds* on the time and space needed by any algorithm to solve a problem, and to be able to prove that a given algorithm is *optimal*.

Some Course Topics

Some of the problems whose algorithmic solutions we will consider include:

- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Primality testing
- Traveling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Sorting
- Program termination

Some of the approaches we'll consider:

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Greedy approach
- Dynamic programming
- Backtracking and Branch and bound
- Space and time tradeoffs

The study of algorithms often extends to the study of advanced data structures. Some should be familiar; others likely will be new to you:

- lists (arrays, linked, strings)
- stacks/queues
- priority queues

- graph structures
- tree structures
- sets and dictionaries

Finally, the course will often require you to write formal analysis and often proofs. You will practice your technical writing. As part of this, you may wish to gain experience with the mathematical typesetting software LaTeX.

Pseudocode

We will spend a lot of time looking at algorithms expressed as *pseudocode*.

Unlike a real programming language, there is no formal definition of "pseudocode". In fact, any given textbook is likely to have its own style for pseudocode.

Our text has a specific pseudocode style. My own style looks more like Java or C++ code. I will not be picky about the pseudocode style you use as long as it's clear what you mean.

A big advantage of using pseudocode is that we do not need to define types of all variables or specify complex structures.

Example: Greatest Common Denominator

We first consider a very simple but surprisingly interesting example: computing a greatest common denominator (or divisor) (GCD).

Recall the definition of the GCD:

The gcd of m and n is the largest integer that divides both m and n evenly.

For example: gcd(60,24) = 12, gcd(17,13) = 1, gcd(60,0) = 60.

One common approach to finding the gcd is *Euclid's Algorithm*, specified in the third century B.C. by Euclid of Alexandria.

Euclid's algorithm is based on repeated application of the equality:

$$\text{gcd}(m,n) = \text{gcd}(n, m \bmod n)$$

until the second number becomes 0, which makes the problem trivial.

Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

More precisely, application of Euclid's Algorithm follows these steps:

Step 1 If n = 0, return m and stop; otherwise go to Step 2

Step 2 Divide m by n and assign the value of the remainder to r

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

And a pseudocode description:

```
// m,n are non-negative, not both zero
Euclid(m, n) {

  while (n != 0) {

    r = m mod n
    m = n
    n = r
  }
  return m
}
```

It may not be obvious at first that this algorithm must terminate.

How can we convince ourselves that it does?

- the second number (n) gets smaller with each iteration and can never become negative
- so the second number in the pair eventually becomes 0, at which point the algorithm stops.

Euclid's Algorithm is just one way to compute a GCD. Let's look at a few others:

Consecutive integer checking algorithm: check all of the integers, in decreasing order, starting with the smaller of the two input numbers, for common divisibilty.

Step 1 Assign the value of min$\{m,n\}$ to t

Step 2 Divide m by t. If the remainder is 0, go to Step 3; otherwise, go to Step 4

Step 3 Divide n by t. If the remainder is 0, return t and stop; otherwise, go to Step 4

Step 4 Decrease t by 1 and go to Step 2

This algorithm will work. It always stops because every time around, Step 4 is performed, which decreases t. It will eventually become t=1, which is always a common divisor.

Let's run through the computation of gcd(60,24):

Step 1 Set t=24

Step 2 Divide m=60 by t=24 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=23, proceed to Step 2

Step 2 Divide m=60 by t=23 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=22, proceed to Step 2

Step 2 Divide m=60 by t=22 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=21, proceed to Step 2

Step 2 Divide m=60 by t=21 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=20, proceed to Step 2

Step 2 Divide m=60 by t=20 and check the remainder. It is 0, so we proceed to Step 3

Step 3 Divide n=24 by t=20 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=19, proceed to Step 2

Step 2 Divide m=60 by t=19 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=18, proceed to Step 2

Step 2 Divide m=60 by t=18 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=17, proceed to Step 2

Step 2 Divide m=60 by t=17 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=16, proceed to Step 2

Step 2 Divide m=60 by t=16 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=15, proceed to Step 2

Step 2 Divide m=60 by t=15 and check the remainder. It is 0, so we proceed to Step 3

Step 3 Divide n=24 by t=15 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=14, proceed to Step 2

Step 2 Divide m=60 by t=14 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=13, proceed to Step 2

Step 2 Divide m=60 by t=13 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=12, proceed to Step 2

Step 2 Divide m=60 by t=12 and check the remainder. It is 0, so we proceed to Step 3

Step 3 Divide n=24 by t=12 and check the remainder. It is 0, so we return t=12 as our gcd

However, it does not work if one of our input numbers is 0 (unlike Euclid's Algorithm). This is a good example of why we need to be careful to specify valid inputs to our algorithms.

Another method is one you probably learned in around 7th grade.

Step 1 Find the prime factorization of m

Step 2 Find the prime factorization of n

Step 3 Find all the common prime factors

Step 4 Compute the product of all the common prime factors and return it as gcd(m,n)

So for our example to compute gcd(60,24):

Step 1 Compute prime factorization of 60: 2, 2, 3, 5

Step 2 Compute prime factorization of 24: 2, 2, 2, 3

Step 3 Common prime factors: 2, 2, 3

Step 4 Multiply to get our answer: 12

While this took only a total of 4 steps, the first two steps are quite complex. Even the third is not completely obvious. The description lacks an important characteristic of a good algorithm: *precision*.

We could not easily write a program for this without doing more work. Once we work through these, it seems that this is going to be a more complicated method.

We can accomplish the prime factorization in a number of ways. We will consider one known as the *sieve of Eratosthenes*:

```
Sieve(n) {
  for p = 2 to n { // set array values to their index
    A[p] = p
  }
  for p = 2 to floor(sqrt(n)) {
    if A[p] != 0 { //p hasn't been previously eliminated from the list
      j = p * p
      while j <= n  {
        A[j] = 0  //mark element as eliminated
        j = j + p
      }
  }
  // nonzero entries of A are the primes
```

Given this procedure to determine the primes up to a given value, we can use those as our candidate prime factors in steps 1 and 2 of the middle school gcd algorithm. Note that each prime may be used multiple times.

So in this case, the seemingly simple middle school procedure ends up being quite complex, since we need to fill in the vague portions.

Fundamental Data Structures

Before we get into algorithms, we will review and/or introduce some of the data structures we will be using all semester.

Linear Structures

The basic *linear structures* are your standard "one-dimensional" *list* structures: *arrays*, *linked lists*, and *strings*.

Some characteristics of arrays:

- allow efficient contiguous storage of a collection of data
- efficient direct access to an arbitrary element by *index*
- cost of add/remove depends on index

Strings are usually built using arrays, and normally consist of bits or characters.

Important operations on strings include finding the length (whose efficiency depends on whether the strings is *counted* or *null-terminated*), comparing, and concatenating.

Some characteristics of linked lists:

- data stored in a *list node* along with a reference to the next list node (and to the previous one for a *doubly linked list*)
- cost of access/add/remove depends on position within the list
- lends itself to an efficient *traversal*

These basic structures are used for many purposes, including as building blocks for more restrictive linear structures: *stacks* and *queues*.

For a stack, additions (*pushes*) and removals (*pops*) are allowed only at one end (the *top*), meaning those operations can be made to be very efficient. A stack is a *last-in first-out (LIFO)* structure.

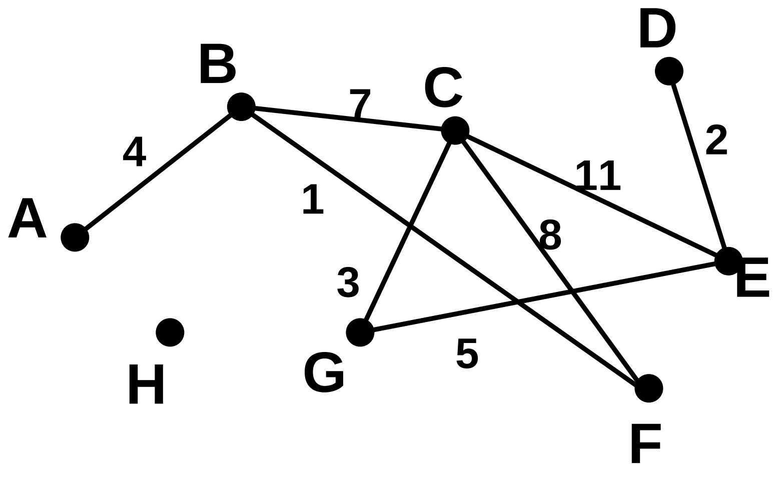
For a queue, additions (*enqueues*) are made to one end (the *rear* of the queue) and removals (*dequeues*) are made to the other end (the *front* of the queue). Again, this allows those operations to be made efficient. A queue is a *first-in first-out (FIFO)* structure.

A variation on a queue is that of a *priority queue*, where each element is given a "ranking" and the highest-ranked item is the only one allowed to be removed, regardless of the order of insertion. A clever implementation using another structure called a *heap* can make both the insert and remove operations on a priority queue efficient.

Graphs

A *graph* G is a collection of *nodes* or *vertices*, in a set V, joined by *edges* in a set E. Vertices have labels. Edges can also have labels (which often represent *weights*). Such a graph would be called a *weighted graph*.

The graph structure represents relationships (the edges) among the objects stored (the vertices).



- Two vertices are *adjacent* if there exists an edge between them.

  e.g., A is adjacent to B, G is adjacent to E, but A is not adjacent to C.

- A *path* is a sequence of adjacent vertices.

  e.g., A-B-C-F-B is a path.

- A *simple path* has no vertices repeated (except that the first and last may be the same).

  e.g., A-B-C-E is a simple path.

- A simple path is a *cycle* if the first and last vertex in the path are same.

  e.g., B-C-F-B is a cycle.

- *Directed graphs* (or *digraphs*) differ from *undirected graphs* in that each edge is given a direction.

- The *degree* of a vertex is the number of edges incident on that vertex.

  e.g., the degree of C is 3, the degree of D is 1, the degree of H is 0.

  For a directed graph, we have more specific *out-degree* and *in-degree*.

- Two vertices u and v are *connected* if a simple path exists between them.
- A subgraph S is a *connected component* iff there exists a path between every pair of vertices in S.

  e.g., {A,B,C,D,E,F,G} and {H} are the connected components of our example.
- A graph is *acyclic* if it contains no cycles.
- A graph is *complete* if every pair of vertices is connected by an edge.

There are two principal ways that a graph is usually represented:

1. an *adjacency matrix*, or
2. *adjacency lists*.

As a running example, we will consider an undirected graph where the vertices represent the states in the northeastern U.S.: NY, VT, NH, ME, MA, CT, and RI. An edge exist between two states if they share a common border, and we assign edge weights to represent the length of their border.

We will represent this graph as both an adjacency matrix and an adjacency list.

In an adjacency matrix, we have a two-dimensional array, indexed by the graph vertices. Entries in this array give information about the existence or non-existence of edges.

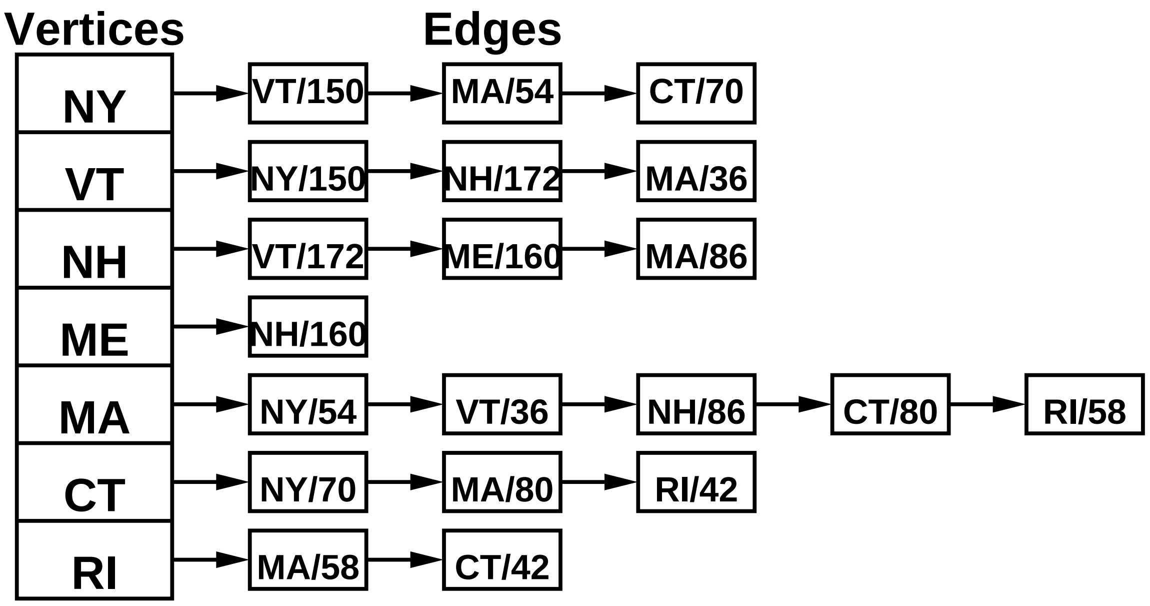
We represent a missing edge with `null` and the existence of an edge with a label (often a positive number) representing the edge label (often representing a weight).

Adjacency matrix representation of NE graph

	NY	VT	NH	ME	MA	CT	RI
NY	`null`	150	`null`	`null`	54	70	`null`
VT	150	`null`	172	`null`	36	`null`	`null`
NH	`null`	172	`null`	160	86	`null`	`null`
ME	`null`	`null`	160	`null`	`null`	`null`	`null`
MA	54	36	86	`null`	`null`	80	58
CT	70	`null`	`null`	`null`	80	`null`	42
RI	`null`	`null`	`null`	`null`	58	42	`null`

If the graph is undirected, then we could store only the lower (or upper) triangular part, since the matrix is symmetric.

An adjacency list is composed of a list of vertices. Associated with each each vertex is a linked list of the edges adjacent to that vertex.



In some cases, a matrix representation is more desirable. In other cases, it is the list representation. It depends on the density of the graph and which graph operations need to be most efficient for the task at hand.

Trees

In a linear structure, every element has unique successor.

In a *tree*, an element may have many successors.

We usually draw trees upside-down in computer science.

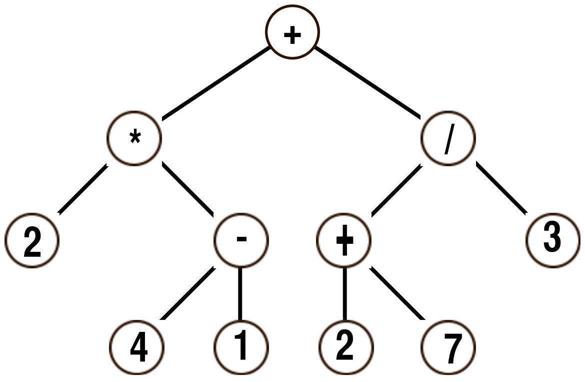
You won't see trees in nature that grow with their roots at the top (but you can see some at Mass MoCA over in North Adams).

One example of a tree is an *expression tree*:

The expression

```
(2*(4-1))+((2+7)/3)
```

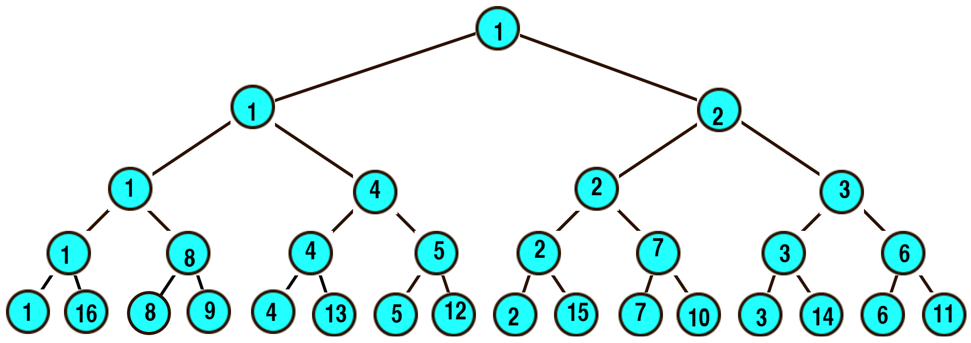
can be represented as



Once we have an expression tree, how can we evaluate it?

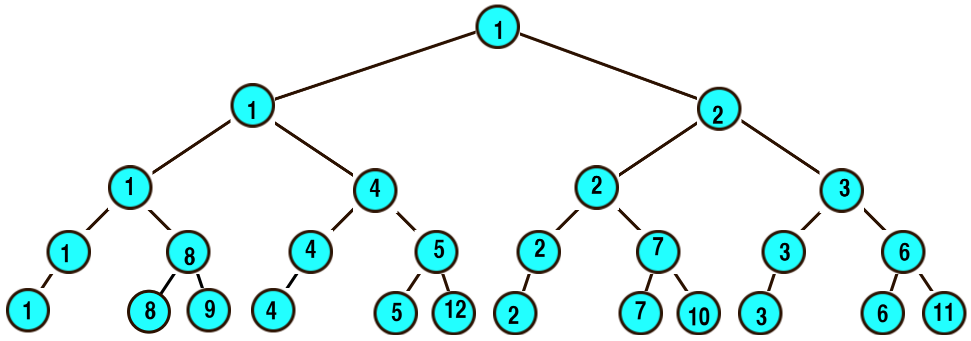
We evaluate left subtree, then evaluate right subtree, then perform the operation at root. The evaluation of subtrees is recursive.

Another example is a tree representing a tournament bracket:



(a *complete* and *full* tree)

or



(neither complete nor full)

There are a lot of terms we will likely encounter when dealing with tree structures:

A *tree* is either empty or consists of a *node*, called the *root node*, together with a collection of (disjoint) trees, called its *subtrees*.

- An *edge* connects a node to its subtrees
- The roots of the subtrees of a node are said to be the *children* of the node.
- There may be many nodes without any successors: These are called *leaves* or *leaf nodes*. The others are called *interior nodes*.
- All nodes except root have unique predecessor, or *parent*.
- A collection of trees is called a *forest*.

Other terms are borrowed from the family tree analogy:

- sibling, ancestor, descendant

Some other terms we'll use:

- A *simple path* is series of distinct nodes such that there is an edge between each pair of successive nodes.
- The *path length* is the number of edges traversed in a path (equal to the number of nodes on the path - 1)
- The *height of a node* is length of the longest path from that node to a leaf.
- The *height of the tree* is the height of its root node.
- The *depth of a node* is the length of the path from the root to that node.
- The *degree of a node* is number of its direct descendents.
- The idea of the *level* of a node defined recursively:
  - The root is at level 0.
  - The level of any other node is one greater than the level of its parent.

  Equivalently, the level of a node is the length of a path from the root to that node.

We often encounter *binary trees* – trees whose nodes are all have degree ≤ 2.

We will also orient the trees: each subtree of a node is defined as being either the *left* or *right*.

Iterating over all values in linear structures is usually fairly easy. Moreover, one or two orderings of the elements are the obvious choices for our iterations. Some structures, like an array, allow us to traverse from the start to the end or from the end back to the start very easily. A singly linked list however, is most efficiently traversed only from the start to the end.

For trees, there is no single obvious ordering. Do we visit the root first, then go down through the subtrees to the leaves? Do we visit one or both subtrees before visiting the root?

There are four standard *tree traversals*, considered here in terms of binary trees (though most can be generalized):

1. *preorder*: visit the root, then visit the left subtree, then visit the right subtree.
2. *in-order* visit the left subtree, then visit the root, then visit the right subtree.
3. *postorder*: visit the left subtree, then visit the right subtree, then visit the root.
4. *level-order*: visit the node at level 0 (the root), then visit all nodes at level 1, then all nodes at level 2, etc.

For example, consider the preorder, in-order, and postorder traversals of the expression tree

```
        /
  *          2
 +   -
4 3 10 5
```

- preorder leads to prefix notation:
  / * + 4 3 - 10 5 2
- in-order leads to infix notation:
  4 + 3 * 10 - 5 / 2
- postorder leads to postfix notation:
  4 3 + 10 5 - * 2 /

Sets and Dictionaries

A *set*, just like in mathematics, is a collection of distinct *elements*.

There are two main ways we might implement a set.

If there is a limited, known group of possible elements (a *universal set*) U, we can represent any subset S by using a *bit vector* with the bit at a position representing whether the element at that position in U is in the subset S.

If there is no universal set, or the universal set is too large (meaning the bit vector would also be large, even for small subsets), a linear structure such as a linked list of the elements of the set can be used.

A *dictionary* is a set (or *multiset*, if we allow multiple copies of the same element) which is designed for efficient addition, deletion, and search operations. The specific underlying implementation (array, list, sorted array, tree structure) depends on the expected frequency of the operations.

We will consider many of these data structures more carefully, and will see several more advanced data structures later in the couese.