ArsDigita University
Month 5: Algorithms - Professor Shai Simonson

Lectures

The study of algorithms concentrates on the high level design of data structures and methods for using them to solve problems. The subject is highly mathematical, but the mathematics can be compartmentalized, allowing a student to concentrate on *what* rather than *why*. The assumed prerequisite is that a student can take a description of an algorithm and relevant data structures, and use a programming tool to implement the algorithm. For most computer scientists, this is exactly how they might interact with algorithms in their future careers. Keeping this in mind, whenever we *write* the code for an algorithm we will use a pseudo procedural C-like language, and the student is expected to be able to implement the details using OOP ideas in Java or C++, or functional style like Scheme.

A complete understanding of algorithms is more than just learning a few particular methods for a few particular problems. The course focuses not just on details of particular algorithms but on styles and patterns that can be used in new situations. The second focus of the course is teaching the tools that help you distinguish between problems that are efficiently solvable and ones that are not.

Let's get to some actual examples of this latter point by listing pairs of problems that although superficially similar, have one problem that is efficiently solvable and one that is NP-complete. For now, NP-Complete means that the problem as far as any reasonable person is concerned has no efficient solution. We will later give a real definition of the term and make the intuition more clear.

1. Euler Circuit vs. Hamiltonian Circuit
2. Shortest Path vs. Longest Path
3. Minimum Spanning Tree vs. Dominating Set
4. Edge Cover vs. Vertex Cover
5. Min Cut vs. Max Cut
6. 2-Dim Matching vs. 3-Dim Matching
7. 2-Colorability vs. Colorability
8. 2-Satisfiability vs. Satisfiability

Formal definitions of these problems can be found in Garey and Johnson's comprehensive list of NP-Complete problems found in:

Computers and Intractibility:
A guide to the theory of NP-completeness
Michael R. Garey and David S. Johnson
W. H. Freeman, 1979.

Here are some informal descriptions:

1. Euler Circuit vs. Hamiltonian Circuit

Euler Circuit asks, given an undirected graph, whether you can trace the edges starting and ending at the same place, and tracing each edge exactly once. (A given vertex can be traced through multiple times).

Hamiltonian circuit asks, given an undirected graph, whether you can trace through the vertices of the graph, starting and ending at the same place and tracing through each vertex exactly once.

2. Shortest Path vs. Longest Path

Shortest Path asks, given a weighted undirected graph and two vertices in the graph, what is the shortest path between the two vertices. (The assumption is that there are no negative weight cycles in the graph).

Longest Path is the natural analogue of shortest path. (The assumption is that there may be positive weight cycles).

3. Minimum Spanning Tree vs. Dominating Set

Minimum Spanning Tree, given a weighted undirected graph, asks for the minimum weight tree that contains all the vertices in the graph. The tree must be a subgraph of the given graph. Say your town gets flooded, this problem asks for the minimum miles of roads that need to be repaved to reconnect all the houses in town.

Dominating Set, given an undirected graph, asks for the minimum size set of vertices, such that every vertex is either in this set or else is connected to it directly by an edge. This is like your town gets flooded and you want to put rescue stations at intersections so that every road has at least one rescue station at each end.

4. Edge Cover vs. Vertex Cover

Edge Cover, given an undirected graph, asks for the smallest set of edges such that every vertex in the graph is incident to at least one of the edges.

Vertex Cover, given an undirected graph, asks for the smallest set of vertices such that every edge in the graph is incident to at least one of the vertices.

5. Min Cut vs. Max Cut

Min Cut asks, given a weighted undirected graph, what is the minimum weight set of edges whose removal separates the graph into two or more disconnected components.

Max Cut is the natural analogue of Min Cut.

6. 2-Dim Matching vs. 3-Dim Matching

2-Dim Matching, is also called the marriage problem. Given a certain number of men and an equal number of woman, and a list of pairs of men/woman who are willing to be married, is there a way to arrange marriages so that everyone gets paired up, and all pairs are in the preferred list.

3-Dim Matching is the natural *3-gender* analogue to 2-Dim Matching, where each marriage must have one of each of the three genders.

7. 2-Colorability vs. Colorability

Colorability is the famous problem that asks for the minimum number colors needed to color a graph, so that no two connected vertices have the same color. Note for a planar graph, the 4-color theorem implies that the number is no larger than four.

2-Colorability asks simply whether a given graph can be colored with at most two colors. This equivalent to determining whether or not a graph is bi-partite.

8. 2-Satisfiability vs. Satisfiability

Satisfiability asks, given a wff in conjunctive normal form, is there a T/F assignment to the variables, such that the value of the formula ends up being true.

2-Sat is the restricted version of Satisfiability where every conjunct has exactly two variables.

Try to guess which one of each pair is the hard one and which one is the easy one. I should point out that the *easy* one does not necessarily have an easily designed algorithm.

Algorithms can be categorized by style and by application.

Commonly used styles are divide and conquer (recursion), dynamic programming (bottom-up or memoization), and greedy strategy (do the best thing locally and hope for the best). Common application categories include mathematics, geometry, graphs, string matching, sorting and searching. Combinatorial algorithms are a larger category including any algorithm that must consider the best result among a large sample space of possibilities. Many combinatorial problems are NP-Complete. Do not confuse dynamic programming with *linear programming.* The latter refers to a particular problem in linear algebra and numerical analysis with important application in industry and operations research. It is not a style or technique, nor does it have anything to do with programs that run in linear time. It is a problem that can be used to solve many combinatorial problems.

Correctness and Analysis of Algorithms

A large concern of this course is not just the design of algorithms and the techniques used in designing then, but the proofs that the algorithms work and the analysis of the time and space requirements of the algorithms.

Behind every algorithm there is a proof of correctness often based on many theorems and lemmas. The proofs are often by mathematical induction. When you are designing your own algorithms, you must be able to convince yourself that they work. When you publish or share them, you can't really on your own confidence and instincts, but must prove that they work. These proofs can often be quite tedious and technical, but understanding *why* an algorithm work gives you better tools for creating new algorithms than merely knowing how an algorithm works.

There are many ways to analyze the time and space requirements of an algorithm. We describe the time requirements of an algorithm as a function of the input size, rather than a list of measured times for particular inputs on a particular computer. The latter method is useful for engineering issues but not useful for general comparisons. In the early days of CS before theory was well developed, the method of measuring actual times of program runs gave no fair way to compare algorithms due to the random engineering differences between computers, implementations and languages. However, it should be emphasized that theory is not always a fair assessment either, and ideally one calculates the time complexity theoretically while engineers fine tune the constant factors, for practical concerns. There are many examples of this. Fibonacci heaps are the fastest data structures for certain algorithms but in practice require too much overhead to make them worthwhile. The Simplex algorithm of Dantzig is worst case exponential time but in practice runs well on real life problems.

There is no perfect theory for modeling all aspects of input distributions and time measurement.

Usually, the time requirements are the main concern, so the space requirements becomes a secondary issue. One way to analyze the time complexity of an algorithm is worst case analysis. Here we imagine that we never get lucky, and calculate how long the program will take in the worst case. The average case analysis may be more useful, when the worst case does not show up very often, and this method averages the time complexity of the algorithm over all possible inputs, weighted by the input distribution. Average case analysis is not as common as worst case analysis, and average case complexity is often difficult to compute due to the need for careful probabilistic analysis.

A third method of measuring time complexity is called *amortized* analysis. Amortized analysis is motivated by the following scenario. Let's say that there is an algorithm whose average case complexity is linear time, however in practice this algorithm is run in conjunction with some other algorithms, whose worst case complexities are constant time. It turns out that when you use these algorithms, the slow one is used much less frequently than the fast ones. So much less that we can distribute the linear cost of the slow one over the fast costs so that the total *amortized* time over the use of all the algorithms is constant time per run. This kind of analysis comes up with with the fancier data structures like Fibonacci heaps, which support a collection of algorithms. The amortized analysis is the only way in which the fancier data structure can be proved better than the standard binary heap data structure.

Lower Bounds and NP-Completeness

Most of the time we will do worst case analysis in this course. This gives us an upper bound on the time requirements of an algorithm. It is also useful, however, to get lower bounds on our time requirements for solving a problem. These lower bound arguments are much harder, because instead of simply analyzing a particular method, they require that we analyze all possible methods to solve a problem. Only when we can quantify over all possible solutions can we claim that a particular problem. A famous lower bound result is that sorting in general requires at least O(n log n) time.

Most of the time, it is too hard to prove any lower bound time complexity on a problem. In that case, a weaker way to show a lower bound on solving a problem, is to prove that the problem is NP-complete. Intuitively, this means that although we cannot prove that the problem requires more than polynomial time, we can prove that finding a polynomial time for our problem would imply a polynomial time algorithm for hundreds of other problems (those in NP) for which no polynomial time algorithms are currently known. That is an NP-Complete is the hardest of a collection of problems called NP. More details on this later. It suffices to say that many hard problems that defy a polynomial approach are in this collection called NP.

Determining the frontier between which versions of a problem are polynomial time solvable and which are NP-complete, is a skill honed by experience. You will get plenty of practice with this sort of thing in this course.

Coping with NP-Completeness

For the most part, proving that a problem is NP-complete is a negative result. All it really gives you is the knowledge that you should not waste your time trying to come up with a fast algorithm. However, there are many ways of coping with an NP-Complete problem.

1. Look for special cases that may be polynomial time solvable. Determine the frontier for the problem.
2. Try to design an *approximation* algorithm. This is an algorithm that will not give the right answer, but will get an answer within a certain percentage of the correct answer.
3. Try a *probabilistic* algorithm. This is an algorithm that will get the right answer some percent of the time.
4. Experiment by engineering the exponential time algorithms to see how far you can get with current speeds and technologies.
5. Use a different computational model (DNA for example).

Mathematical Preliminaries

Discrete mathematics is used a lot in studying algorithms. Proofs by induction abound in proving the correctness of an algorithm. Recurrence equations and sums for the analysis of algorithms are crucial. Counting and probability come up everywhere. Graphs and trees are standard data structures. Logic and Boolean algebra come up in many examples. Finally, Big-O notation for describing asymptotic complexity is a standard tool.

Max and Min Algorithms – A Warm Up

There are simple iterative and recursive algorithms for calculating max (or min) of a list of n numbers. One method recursively finds the max of n-1 of the numbers, and then compares this to the last number. The recurrence equation is $T(n) = T(n-1) + 1$ and $T(1) = 0$. The solution to this is $T(n) = n-1$, which agrees with the simple iterative method of initializing a LargestSoFar to the first number, and then comparing the rest of the numbers one by one to LargestSoFar, and swapping when necessary.

Another recursive method to find the max of a list of numbers is to split the list into two equal parts, recursively find the max of each, and compare the two values. The recurrence equation for this is $T(n) = 2T(n/2)+1$, $T(1) = 0$, which also has a solution of n-1.

Now what if we wanted to calculate both the max and min. We could just do any of the above algorithms twice, giving 2n-2 steps. But can we do better? We will not do better asymptotically. That is we will still need O(n), but can we make the constant factors better? Normally, we do not care so much about this, but sometimes constant factors are an important practical matter, and here it motivates the idea of doing better than what you get with brute force.

We can calculate the max and min simultaneously by using the following recursive idea. Note it is possible to describe this iteratively, but the recursive version emphasizes the technique more elegantly. The idea is that we can process two numbers at a time. We recursively compute the max and min of n-2 of the numbers in the list. Then we compare the larger of the two remaining numbers to the max and the lower of the two remaining numbers to the min. This method gives a recurrence equation of $T(n) = T(n-2) + 3$, $T(2) = 1$. The solution to this is $(3n-1)/2$. We can also split the list into two, and recursively compute the min and max of each list. Then we compare the max of one to the max of the other, and the min of one to the min of the other. This gives $T(n) = 2T(n/2) + 2$, $T(2) = 1$. How does this method compare? Knowing the solutions to recurrence equations helps you choose which algorithm variation to try!

The Max and Second Biggest

We could try to do the same trick for finding the max and second biggest number simultaneously. How exactly? But there is an additional thing we can leverage in this case. We can run a *tournament* to find the max, that takes n-1 steps. This tournament is exactly the same as the n/2 recursion of the previous example. We keep track of all the elements that lost a match with the eventual winner. There will be lg n of these more or less. It is the largest of these which is the second biggest. This method takes n-1 + lg n –1 comparisons. How does this compare with (3n-1)/2? The drawback here is that we must do some extra work in keeping track of who played the eventual winner. No matter how you do this, it does affect the constant factors of the overall algorithm. Therefore, comparing the methods at this level requires experiments and measurements more than just analysis.

Sorting

The bread and butter of algorithms are sorting algorithms. They deal with the simplest and most common kind of problem, and still exhibit a wide variety of techniques, data structures, and methods of analysis that are useful in other algorithms. There are many ways to compare sorting algorithms, but the main way is simply by time complexity. Other aspects of sorting relate to the practical issues of constant factors, recursive overhead, space requirements, performance on small lists, suitability for parallel computation and whether the sort preserves the order of equal sized elements (stable). There are hundreds of sorting algorithms, of which we study a representative sampling.

O(n^2) Time Sorting Algorithms

Bubble Sort and Insertion Sort are two O(n^2) sorting algorithms. Despite the slow time complexity, they have their place in practice. The former is excellent for sorting things that are almost already sorted. The latter is excellent for small lists. Both these special traits can be explained intuitively and can be demonstrated by experiments. This illustrates that the theory does not always address the reality so exactly. A good computer scientist will combine both theory and engineering to explore the truth.

Bubble Sort

```
SwitchMade = true;
for (i = 1;  (i < n) and SwitchMade;  i++)
        SwitchMade = false;
        for (j = 1;  j <= n- i;  j++)
                if (a[j] > a[j+1]) then {swap (a[j], a[j+1]);
SwitchMade = true;}
```

Bubble Sort works by doing n-1 Bubble procedures. Each Bubble procedure (the inner loop) compares adjacent elements and decides whether to swap them, proceeding downward through the array up to the slot looked at last in the last Bubble procedure.

The SwitchMade flag is used to remember whether any swaps were made in the last bubble loop. If not, then the list is sorted and we are done. An example is shown below where the list of numbers is shown after each iteration of the outer loop.

10	8	5	1	16	13
8	5	1	10	13	16
5	1	8	10	13	16
1	5	8	10	13	16

Note that with the ith iteration of the outer loop, the inner loop bubble procedure pushes the ith largest value to the bottom of the array while the other lighter values *bubble* up in a less predictable way. The worst case is that the algorithm does n-1 + n-2 + n-3 + … + 1 comparisons. These are the triangle numbers from discrete math which are O(n^2). You can see that if the array is sorted to begin with, then the algorithm takes O(n) time. If the array gets sorted earlier than expected, then the algorithm stops immediately.

Insertion Sort

```
for j = 2 to n          {
            next = a[j];
            for  (k = j-1; ((k>0) && (a[k]> next));  --k)   a[k+1] = a[k];
            a[k+1] = next;
}
```

This algorithm sorts by keeping a sorted area from 1 to j, and expanding it by one with each complete execution of the inner loop. The inner loop inserts the next element into the appropriate slot in the sorted area from 1 to j-1. It does this by looking at the area in reverse order shifting elements to the right until it reaches an element which it is not less than. It inserts the element in that spot. Note that looking in the opposite order would take extra time. This should remind you of the collapsing loops in your SameGame program from last month. An example is shown below:

10	8	5	1	16	13
8	10	5	1	16	13
5	8	10	1	16	13
1	5	8	10	16	13
1	5	8	10	13	16

The worst case time complexity of this algorithm is also O(n^2) when the list was originally in reverse order. The same triangle number sum as Bubble Sort shows up, however unlike Bubble Sort, it does not stop early if the list was almost sorted to begin with. It does however still use only O(n) total operations on a sorted list. It also has particularly few operations in the inner loop, making the constant factor overhead very low. It actually runs faster than many O(n log n) algorithms for small values of n, since the constant factors and/or recursive overhead for the O(n log n) algorithms are relatively high. Of course for large n, the O(n^2) algorithms are not practical.

The book does a careful analysis of the average case for Insertion Sort by summing up all the time requirements of the equally likely possibilities, dividing by the number of possibilities, and showing that it too is O(n^2). (pages 8-9). It is often the case that worst case is the same as average case. That is why we rarely calculate average case time complexity.

A notable exception is Quicksort which is worst case O(n^2) and average case O(n log n). We calculate average case for Quicksort because it explains why the thing works so well despite its bad worst-case complexity. Sometimes practice motivates theory and sometimes theory motivates practice.

Heapsort, Mergesort and Quicksort – O(n log n) Time Algorithms

Mergesort

Mergesort is an O(n log n) sort that works recursively by splitting the list into two halves, recursively sorting each half, and then *merging* the results. The merge algorithm takes two sorted lists and creates a merged sorted list by examining each list and comparing pairs of values, putting the smaller of each pair of values in a new array. There are two pointers, one for each array of numbers, and they start at the beginning of the array. The pointer to the list from which the smaller value was taken is incremented. At some point one of the pointers hits the end of the array, and then the remainder of the other array is simply copied to the new array.

The time complexity for merging is the sum of the lengths of the two arrays being merged, because after each comparison a pointer to one array is moved down and the algorithm terminates when the pointers are both at the ends of their arrays. This gives a recurrence equation of $T(n) = 2T(n/2) + O(n)$, $T(2) = 1$, whose solution is $O(n \log n)$.

One important feature of Mergesort is that is not *in-place*. That is, it uses extra space proportional to the size of the list being sorted. Most sorts are *in-place,* including insertion sort, bubble sort, heapsort and quicksort. Mergesort has a positive feature as well in that the whole array does not need to be in RAM at the same time. It is easy to merge files off disks or tapes in chunks, so for this kind of application, mergesort is appropriate. You can find a C version of mergesort in assignment 1 of How Computers Work (month 3).

Heapsort

Heapsort is the first sort we discuss whose efficiency depends strongly on an abstract data type called a *heap.* A heap is a binary tree that is as complete as possible. That is, we fill it in one level at a time from right to left on each level. It has the property that the data value at each node is less than or equal to the data value at its parent. (Note that there is another abstract data type called a binary search tree that is not the same as a heap. Also, there is another *heap* used in the context of dynamic allocation of storage and garbage collection for programming languages such as Java or Scheme. This other heap has nothing to do with our heap. The heap from dynamic memory allocation has more of a usual English meaning as in a heap of free of memory, and is actually more like a linked list.)

A heap supports a number of useful operations on a collection of data values including GetMax(), Insert(x), and DeleteMax(). The easiest way to implement a heap is with a simple array, where A[1] is the root, and the successive elements fill each level from left to right. This makes the children of A[i] turn up at locations A[2i] and A[2i+1]. Hence moving from a parent to a child or vice versa, is a simple multiplication or integer division. Heaps also allow changing any data value while maintaining the heap property, Modify(i, x), where i is the index of the array and x is the new value. Heaps are a useful way to implement priority queues that is a commonly used abstract data type (ADT) likes stacks and queues.

To GetMax(), we need only pull the data value from the root of the tree. The other operations Insert(x), DeleteMax() and Modify(i,x) require more careful work, because the tree itself needs to be modified to maintain the heap property. The modification and maintenance of the tree is done by two algorithms called *Heapify* (page 143) and *Heap-Insert* (page 150). These correspond to the need to push a value up through the heap (Heap -Insert) or down through the heap (Heapify). If a value is smaller or equal to its parent but smaller than at least one of its children, we push the value downwards. If a value is larger or equal than both its children, but larger than its parent, then we push it upwards. Some texts call these two methods simply *PushUp* and *PushDown.* The details of these two methods using examples will be shown in class. The time complexity for these operations is $O(h)$ where h is the height of the tree, and in a heap h is $O(\lg n)$ because it is so close to perfectly balanced. An alternate way to calculate the complexity is the recurrence $T(n) = T(2n/3) + O(1)$, $T(1) = 0$, whose solution is $O(\log n)$. The recurrence comes from the fact that the worst case splitting of a heap is 2/3 and 1/3 (page 144) on the two children.

Heapsort works in two phases. The first phase is to build a heap out of an unstructured array. The next step is:

```
for index = last to 1 {
        swap(A[0], A[index]);
        Heapify (0);
}
```

We will discuss the buildheap phase in class, and there is a problem on it in your Pset. It is also discussed at length in the text. The next phase works assuming the array is a heap. It computes the largest value in the array, and the next etc., by repeatedly removing the top of the heap and swapping it with the next available slot working backwards from the end of the array. Every iteration needs to restoe theheap property since a potentially small value has been placed at the top of the heap. After n-1 iterations, the heap is sorted. Since each PushUp and PushDown takes O(lg n) and we do O(n) of these, that gives O(n log n) total time.

Heapsort has some nice generalizations and applications as you will see in your Pset, and it shows the use of heaps, but it is not the fastest practical sorting algorithms.

Quicksort

Quicksort and its variations are the most commonly used sorting algorithms. Quicksort is a recursive algorithm that first *partitions* the array in place into two parts where all the elements of one part are less than or equal to all the elements of the second part. After this step, the two parts are recursively sorted.

The only part of Quicksort that requires any discussion at all is how to do the partition. One way is to take the first element a[0] and split the list into parts based on which elements are smaller or larger than A[0]. There are a number of ways to do this, but it is important to try to do it without introducing O(n) extra space, and instead accomplish the partition in place. The issue is that depending on A[0], the size of the two parts may be similar or extremely unbalanced, in the worst case being 1 and n-1. The worst case of Quicksort therefore gives a recurrence equation of T(n) = T(n-1) + O(n), T(1) = 0, whose solution is O(n^2).

The partition method we will review in class keeps pointers to two ends of the array moving them closer to each other swapping elements that are in the wrong places. It is described on pages 154-155. An alternative partition algorithm is described in problem 8-2 on page 168.

The question is why is Quicksort called an O(n log n) algorithm even though it is clearly worst case O(n^2)? It happens to run as fast or faster than O(n log n) algorithms so we better figure out where the theory is messing up. It turns out that if we calculate the average case time complexity of Quicksort, we get an O(n log n) result. This is very interesting in that agrees with what we see in practice. Moreover it requires the solution of a complicated recurrence equation, T(n) = (2/n) (T(1) + T(2) + … + T(n-1)) + O(n), T(1) = 0, whose solution is obtained by guessing O(n log n) and verifying by mathematical induction, a technique with which you may be familiar. The solution also requires the closed form summation of k log k for k = 1 to n, another technique from discrete mathematics that you have seen before.

Bucket Sort and Radix Sort – Linear Time Sorts for Special Cases

Counting Sort

The idea behind bucket sort is based on a simpler idea our text calls counting sort. This method is equivalent to the following way to sort quizzes whose grades can be anything between 0 and 10. Set up 11 places on your desk and mark them 0 through 10. Then go through each quiz one at a time and place it in the pile according to its grade. When you are finished, gather the quizzes together collecting the piles in order from 0 to 10. This method generalizes to sorting an array of integers where the number of data values is limited to a range between 0 and m. The time complexity of counting sort is O(n+m) where n is the number of data values, and m is the largest possible data value. Note it is O(n), because we can place the quiz grades in their appropriate slots with a loop of instructions like: for i=0 to n-1 {B[A[i]] ++}. B[j] holds the number of quizzes with grade j. The algorithm is an additional O(m) because we must initialize and collect the piles at the end. This is done by: for j = 0 to m-1 {for k = 1 to B[j] {print (j);}}. This loop takes time equal to the maximum of n and m. Note it does not take O(nm).

Bucket Sort

Counting sort is O(n) whenever m is O(n), but it can be very slow if m is O(2^n). Bucket Sort is a way to extend Counting Sort when the values are not limited in size. Instead, we artificially divide the n numbers into n different groups. So for example if we have 100 five-digit positive numbers where the maximum is 99999, then we divide the range into 100 different intervals. We can do this by simply using the first two digits of the number as its interval value, so that 45678 would be placed in interval 45. (In general, to find the correct interval for a data entry, we would have to divide the data by m/n, where m is the maximum value and n is the number of intervals.) The sort works by looking at each value and placing it in its appropriate interval. Each interval has a linked list that holds all the values in that interval since there may of course be more than one in any interval. After all the values are placed in some interval, each interval is sorted and then they are collected together in order of interval size.

The implicit assumption that makes this sort O(n) time, is that the distribution of values into intervals is uniform, hence Bucket Sort lets us trade the assumption of uniform distribution for Counting Sort's assumption of a limited number of values. Note if there was one interval that contained all the numbers, the sort would time at best O(n log n). The analysis assuming uniform distribution requires a little probability and discrete math (see page 182).

Radix Sort

Radix Sort is another generalization of Counting Sort, where we assume nothing about the list of numbers. Radix Sort is an idea originally seen in the punch card machines invented by Hermann Hollerith to do the 1890 USA census. The same trick was used in IBM punch cards of the 1960's and 1970's. A neat feature of Radix Sort is that it must use a *stable* sort as a subroutine. Recall that a stable sort is one that preserves the order of equal valued elements.

Radix Sort works by repeatedly sorting the numbers by looking at the digits, from right to left. A good example would be sorting strings of length 4 in alphabetical order. Say we want to sort:

SHAI
FRAN
SHAM
FANA
FRAM

We make 26 boxes labeled A-Z, and we sort the strings using counting sort on the rightmost character. To implement this, we use an array of linked lists whose indices are the A through Z. After the first step, all strings in the original input array that end in A, are in the linked list headed by A etc. Then we copy all the strings from the linked lists in order back into the original input array, overwriting the original array. Note that this is a *stable* process. This gives the list below:

FANA
SHAI
SHAM
FRAM
FRAN

We repeat this step sorting on the column that is the second to the right, making sure (very important) to preserve the order of equal characters. (The only time this preservation does not matter is on the very first iteration.) This gives:

SHAI
SHAM
FRAM
FRAN
FANA

You should note that after this step the strings are sorted correctly if we look just at the last two characters. After each subsequent step, this sorting will be correct for one more column to the left. You can prove naturally by induction, that it works in general.

Here are the last two stages:

FANA
SHAI
SHAM
FRAM
FRAN

FANA
FRAM
FRAN
SHAI
SHAM

The same algorithm works on integers by splitting each integer up into its digits. It is fine to use the binary digits. The time complexity is O(n+k) for each step, where n is the number of elements and k is the number of different digits (2 for binary). There are d steps where d is the number of digits in each number giving a total of O(d(n+k)). Since k is constant and d is worst case O(log n) then radix sort works in O(n log n) worst case. It can be linear time when d happens to be O(n).

One thing to note with radix sort is that if we sort from the most significant bit to the least significant bit, then we do not actually sort the array. For example:

356 189 338 185 266 325 turns into:

189 185 266 356 338 325 which turns into:

325 338 356 266 185 189

and we are getting nowhere fast.

If we try to fix this by sorting subsets of the list in each subsequent iteration, we end taking too much space and too much time. How much extra I leave to you to think about.

This processing from least significant to most significant seems unintuitive at first, but is actually the key to the whole algorithm. It allows us to reuse the same original Counting Sort array for each iteration, and it keeps the time complexity down.

Lower Bounds on Sorting

There is a well known argument that any algorithm using comparisons requires at least O(n log n) comparisons to sort a list of n numbers. The argument turns any algorithm into a decision tree, which must have *at least* n! leaves. Each leaf represents a particular permutation of the input list, and since the input list is arbitrary, there must be at least n! leaves. From discrete math, we recall that a binary tree with m leaves has depth at least lg m, and here that gives lg n! which is O(n log n). Hence we do not expect a general sorting algorithm using standard methods of computation to ever do better than we already know how to do. This idea of decision trees can be used to get primitive lower bounds on certain other problems, but lower bounds in general are elusive for most problems.

Median and the Kth Largest in Linear Time

We can certainly find the median of a list by sorting in O(n log n) time. The question is whether we can do better. Heapsort can be generalized in an obvious way to get the kth largest value in O(k log n). A problem in your Pset discusses a way to get this down to O(k log k). Unfortunately, when k=n/2 (the median), both these are still O(n log n). Is there any way to find the median of a list of n number in O(n) time?

There is a recursive scheme that solves the kth largest problem in linear time, although the constant factor and overhead can be quite high. Lets see how it works.

We arrange the list into a 2-dimensional array of 5 by n/5. We then find the median of each column and partition it. We now have n/5 columns, each one of which has the higher two values on top and the lower two on the bottom. We then look at the middle row, and recursively calculate its median, m, and then partition (ala Quicksort) the row so that the left side has numbers smallers than the median and the right side has numbers larger or equal. The partitioning moves columns along with their middle elements, so that at this point we have the upper left quadrant smaller than m, and the lower right quadrant larger or equal to m. The remaining two quadrants must be checked element by element to completely partition the array into two parts one smaller than m, and one larger. Call these two parts S_1 and S_2 respectively. If S_1 has more than k elements then we recursively call our algorithm on S_1. Otherwise we recursively call it on S_2. We will do a detailed example in class.

The recurrence equation for this process is a mess. To find the median of a column of 5 elements can be done in 6 comparisons, so this step takes 6n/5. To recursively find median of middle row takes T(n/5). To partition the array and move the columns around takes time about n/5 + n. To construct S_1 and S_2, including checking each element in the upper right and lower left quadrants, takes n time. To recursively call the algorithm on S_1 or S_2 takes worst case T(3n/4), because at least ¼ of the array is in each set. This gives T(n) = T(n/5) + T(3n/4) + 17n/5, T(5) = 6.

Solving this explicitly is difficult, but we can guess that the solution is linear, and prove it is by induction. The constant that works for the proof is constructed, and may be fairly large. The question of whether this algorithm is practical needs to consider the actual data and size of the lists that we are processing.

The key point about this recurrence equation is that it resembles $T(n) = T(n/2) + O(n)$. If it resembled $T(n) = 2T(n/2) + O(n)$ then the complexity would be $O(n \log n)$ rather than $O(n)$. The reason for this is that $3n/4 + n/5 < n$, and this explains why we use five rows in this method. Five is the smallest number of rows that allows a linear time recurrence. With three rows we would have $T(n/3) + T(3n/4)$ and that would give an $O(n \log n)$ recurrence.

Data Structures

There are a number of basic data structures that computer scientists use in designing algorithms. These include stacks, queues, linked lists, priority queues, heaps, trees of various types, graphs, and hashing. For the most part, this small collection and its variations are enough to handle almost any problem you approach. You rarely need to design a new data structure from scratch, but you may well need to design a variation of one of these, or at least know which one is appropriate for which task. Except for heaps, Red-Black binary search trees and graphs, we will leave the discussions of basic data structures and implementations for recitations and reviews.

Binary Search Trees

The flipside of sorting is searching. Searching is perhaps even more fundamental than sorting, in that one of the largest specialties in computer science, data bases, is concerned primarily with ways of organizing and managing data that maintain integrity, consistency and allow for general searching. The theory of databases requires a separate course, and the data structure usually used for the underlying physical layer of a data base is a B-tree, or variations thereof. We will leave the study of B-trees for the database course, and discuss searching at it related to simpler smaller scale applications.

One of the first algorithms that children discover is what we call binary search. It is when a child tries to guess a secret number, and is given high or low answers, in order to determine his next guess. If the range of number is 1-16, the child would guess 8; if that is too high, then the next guess would be 4, etc. until the correct value is found. This process is naturally recursive and cuts the list in half with each subsequent guess. The recurrence equation is $T(n) = T(n/2) + 1 \quad T(1) = 0$, and the solution is $T(n) = O(\log n)$.

Binary search can be done on elements stored in an array, but although this allows searches that are $O(\log n)$ time, the insertions need $O(n)$. If we use linked lists then insertions are $O(1)$ but the search is $O(n)$. The data can also be stored in a *binary search tree*. A binary search tree is a data structure that supports searching, insertions and deletions. Each node stores a data value on which we will search. All numbers in the left subtree of a node are smaller than the number in the node, and all numbers in the right subtree are larger than or equal. We keep pointers from each node to its children, and we can also include a pointer to its parent. To find a value we compare it to the root and move left or right depending on the answer. When we get an equal result, we stop and return the information associated with that value.

Insertions are done by adding a new leaf to the tree in the appropriate place after we hit a null pointer. Deletions are a bit trickier. Examples will be shown in class.

Tree Traversals

Before we discuss deletions from a binary search tree, it is worthwhile to review tree traversals. There are three natural recursive ways to traverse a tree, and one other non-recursive way. The recursive ways are called inorder, preorder and postorder. Inorder traversal means that we first recursively traverse the left subtree, then the root, then recursively traverse the right subtree. Preorder traversal (which is also definable on d-ary trees) means that we visit the root and then recursively traverse the subtrees. This is like depth first search. Postorder means the opposite, that is, first recursively traverse the subtrees then visit the root. There are applications for each of these, and good examples come from compiler design and parsing. However, the inorder traversal is very useful for binary search trees. *An inorder traversal of a binary search tree, prints out the values in sorted order.*

The natural non-recursive traversal of a tree is called *level order.* It is associated with breadth first search, and just as depth first search and the recursive traversals use stacks as a fundamental data structure, so does breadth first search and level order traversals use a queue.

Back to deletion of nodes in a binary search tree… To delete a node in a binary search tree, we need to use the inorder traversal. The idea is that we do not want to lose the property of our ordered structure. When searching or adding this is no problem, but deletion will mangle the tree. The trick is to try to delete a leaf if possible because this does not affect the tree ordered structure. When the node we must delete is not a leaf, but it has only one child, then we can just delete the node by *splicing* it away. When the node has two children then we use the following trick. We find its successor in the inorder traversal, we splice out the successor (which we can prove must have at most one child), and replace the value of our node with that of the deleted successor. Intuitively this works, because we are basically replacing the node to be deleted with the next highest value in the tree. The fact that this next highest value must have at most one child and therefore can be spliced out is very helpful. There are many ways to find the inorder successor of a node but a simple one is just to do a linear traversal time inorder traversal and store the values in an array. The text gives faster and more efficient ways that do not have to traverse the whole tree (page 249). Using a successor guarantees that the binary tree retains its ordered structure.

What's the Problem with Binary Search Trees?

The problem with binary search trees is that they can get thin and scrawny, and to support fast insertions, searches and deletions they must be fat and bushy. All the operations we perform on a binary search tree take time proportional to the height of the tree. But the tree in the worst case can turn into a long thin straight line, so the time complexity becomes O(n) instead of O(log n).

We can just keep the stuff in an array, but then insertions take O(n) – because we have to shift over elements to make room for a new leaf, like insertion sort. The solution is to come up with some sort of data structure that has the dynamic structure of pointers like a tree, but which is guaranteed never to grow too thin and scrawny. Historically, there have been a number of candidates including height balanced trees like AVL trees and 2-3 trees, and weight balanced trees. The height balanced trees keep the left and right heights from each node balanced, while the weight balanced trees keep the number of nodes in each right and left subtree balanced. They are similar in theory but height balanced won favor over weight-balanced trees. 2-3 trees evolved into B-trees used for disk storage, and AVL trees got replaced by Red-Black trees, because the Red-Black trees were slightly more efficient. An advanced data structure called a Splay tree accomplishes the same thing as Red-Black trees, but uses amortized analysis to distribute the cost over the set of all operations on the data structure. Binomial heaps and Fibonacci heaps are also advanced data structures that do for simple binary heaps, what splay trees do for Red-Black trees.

Red-Black Trees

We discuss only Red-Black trees, leaving the simpler and more advanced data structures for your own personal study or recitations and review. The result that makes operations on a Red-Black tree efficient, is that the height of a Red-Black tree with n nodes is at most 2 lg(n + 1), hence they are relatively bushy trees.

The proof of this result depends on the exact definition of a Red-Black tree and a proof by induction you can find on page 264 of your text. A Red-Black is a binary search tree where each node is colored Red or Black, every Red node has only Black children, every leaf node (nil) is Black, and all the paths from a fixed node to any leaf contain the same number of Black nodes.

We show how to do searches and insertions on a Black-Red tree. The text should be consulted for details on deletions that is mildly more complex (as it is in general binary search trees). The details of implementation and pointer details is left to you, with the text providing plenty of help on pages 266, 268, 273 and 274.

To search a Red-Black tree you just do the normal binary tree search. Since the height is at most O(log n), we are okay. The hard part is to do insertions and deletions in O(log n) while maintaining the Red-Black property.

Insertions into a Red-Black Tree

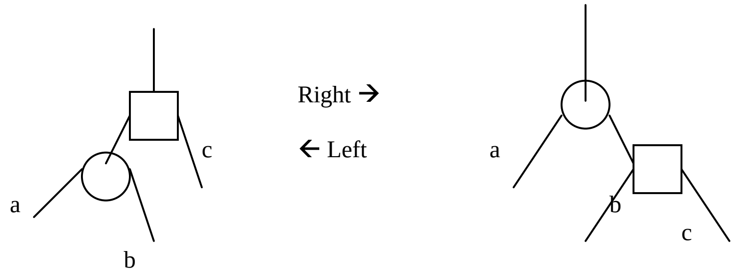
Inserting a value into a binary search tree takes place at a leaf. What properties of the Red-Black tree might this violate. If we color the new node Red, and make its nil children Black, then the number of Black nodes on any path has not changed, all nodes are still either Red or Black, all leaves are Black, and the children of the new Red node are Black. The only property to worry about is whether the parent of the new leaf is also colored Red, which would violate the rule about Red nodes having to have only Black children.

Fixing this property is not simple. We will need to recolor nodes, pushing up the Red-Red clash until finally get rid of it. In the worst case we need to recolor values all the way up the tree. In order to do this recoloring, we require 1 or 2 *rotations* at the end which involve a mutation of the search tree in addition to a recoloring.

Rotations

A rotation is a way to reshape a binary search tree that maintains the structure of the inorder traversal. It is an important tool for managing any height-balanced trees.

Left and right rotations are inverses of one another and the basic movements are shown as pictures below. The details of the code that actually moves the pointers to accomplish this, can be found on page 266 of your text.



You can check that the inorder traversal of these two trees is the same. The improvement is that the heights of the subtrees have become more balanced. For example if the subtree at ‘a’ was a bit long, then a right rotation will balance it up more. It is kind of like pulling up droopy socks, and moving the slack over the top.

I will do a detailed description of inserting an element into a Red-Black tree, showing the cases where the Red-Red clash is pushed upwards. When the Uncle of the bottom Red node is also Red, we can just push it up and recolor. The problem occurs when the Uncle of the bottom Red node is Black. In this case 1 or possibly two rotations are necessary to restore the Black-Red properties. The good news is that if any rotation is necessary, then that ends all future Red-Red clashes, and we can stop pushing the clash up the tree. See class examples in real time for details, or see your text on page 271.

Graph Algorithms

Graph algorithms represent the most diverse collection of applications and problems of any algorithm category. Graphs are used to represent games, networks, process dependencies, scheduling, physical and virtual connectivity. Many graph problems are NP-complete. There a few basic graph algorithms that are solvable in polynomial time including minimum spanning tree, shortest path, maximum flow, and maximum matching. There are a number of general algorithms on graphs that are flexible and can be used to solve a host of other problems. These general techniques are depth first search and breadth first search. The former in particular has great flexibility and its variations are myriad. It can be used to find cycles, connected components, strongly connected components, bi-connected components, triangles, and toplogical orderings. DFS can also be used as the basis for brute force combinatorial algorithms that are NP-complete and come up in AI related applications.

There are many kinds of graphs, and plenty of theorems about graphs that give us a good foundation on which to build our algorithms. Most of these can be looked up in your discrete math text, and reviewed if necessary in recitation. We will focus first on how we can represent a graph inside a computer; that is, what does a graph data structure look like?

The thing that makes graph algorithms a little hard for beginners is that we are used to solving graph algorithms by using our eyes and all the built in visual processing that goes with that. For example, if I show you a picture with a square and a triangle, it is kind of obvious what the connected components are, and whether or not there is a cycle. The beginner does not often know just what is necessary to describe in a graph algorithm. The best way to get a good sense for graph algorithms is to look at the graph the way a computer sees it, and then try to describe your method or algorithm.

The Graph Data Structure

Graphs are stored as an array of linked lists. Each node x has a slot in the array and each has a linked list that holds the numbers of all the nodes that adjacent to x. For example:

0: A → 3 → 4 → nil
1: B → 2 → 5 → 7 → nil
2: C → 1 → 2 → 7 → nil
3: D → 0 → 4 → nil
4: E → 0 → 3 → nil
5: F → 1 → 2 → 7 → nil
6: G → 2 → 7 → nil
7: H → 1 → 2 → 5 → nil

Quick! What does this graph look like? What are its connected components? Does it have a cycle? If you draw the picture, you will be able to answer these questions immediately. But you should design your algorithms without being able to see the picture, because that is the data structure that your program will use. Maybe one day we will have data structures whose methods are the analogue of our visual processing? Don't hold your breath.

There are many ways to augment this data structure. Header information like the indegree or outdegree of a node can be added. Weights on each edge can be indicated with another field in each linked list node. Note that in an undirected graph each edge appears in two distinct linked list nodes. In general, any algorithm that can run by a constant number of traversals of a graph data structure runs in time O(n + e) where n is the number of nodes in the graph and e is the number of edges.

Of course a graph can also be stored in a two dimensional array, which is useful for certain things. Matrix multiplication and variations of inner product, help calculate the number of walks and paths and related stats about a graph, as you saw in discrete math. However, for most algorithms the two dimensional methods just slows down time complexity from O(n + e) to O(n^2). When the graph has a lot of edges this slowdown doesn't matter much.

A Warm Up for Graph Algorithms – Topological Sorting

A directed graph can be used to represent dependencies between nodes. For example, the nodes may represent sections of a large software project and an edge from node x to node y means that x must be complete before y can be completed. Or the nodes represent courses and an edge from x to y means that x is a prerequisite of y. A *topological ordering* or *topological sort* of a directed graph is a list of the nodes in order where the edges all point in a left to right direction. In the case of courses, a topological sort of the courses is an ordering of the courses guaranteeing correct prerequisites. We will discuss an algorithm to find a topological sort of a digraph. Note that later on when we discuss depth first search, there will be another more elegant method based on DFS.

One strategy to topologically sort a digraph is to repeatedly delete a node of in-degree 0 from the graph. How can we do this?

While the graph is not empty do

a. Find a node of indegree zero.
b. Delete it from the graph.

How can we accomplish the steps above and how much time does each take? To check if a graph is empty, we would have to look through the whole array and check for nil's which takes O(n) time. To accomplish this in constant time, we can just keep a static variable that holds the number of nodes in the graph and check if this is zero. This might require O(n) time once at the creation time of the data structure in the appropriate constructor. This variable would be modified by delete.

Finding a node of in-degree zero can be done by traversing the graph data structure and marking which nodes get visited. This takes O(n+e) time. Deleting a node from the graph can be done by setting the pointer to nil on that node, and traversing the other linked lists, deleting the node anytime it is found. This is also O(n+e) time. The loop gets executed at most n times so we get a total time complexity of O(n(n+e)).

This is way too slow. We can do better by preprocessing the graph, calculating the indegree of each node in advance and storing them in header nodes. Now we can solve the whole problem in O(e). First we find a node of indegree 0, which takes O(n) time. Then we traverse its linked list and for each node in the list, we subtract 1 from the appropriate header node. After we finish traversing the list, if the indegree of any node turns to 0 we output it, and then we traverse its linked list.

The natural data structure is a queue that holds the nodes whose linked lists must be traversed. We initialize the queue to all the nodes that initially have indegree zero. While the queue is not empty, we delete a node from the queue, traverse its linked list, subtract one from the appropriate header node, and if the header node turned to zero, then add it to the queue. The queue makes sure that we delete a node completely before deleting any nodes to which it pointed. In this implementation we don't actually delete any nodes from the graph, instead we just modify the indegree values. A detailed example will be done in class.

Note that if the digraph has a cycle, then this algorithm will never terminate.

Minimum Spanning Tree

Let's consider a famous problem on graphs whose solution will help us elaborate more on data structures and their relationship to algorithms. Prim's algorithm will use a priority queue that can be implemented with Red-Black trees or heaps, and Kruskal's algorithm can be implemented using a new data structure called the Union-Find data structure which is composed of trees and linked lists.

Both algorithms use a *greedy* strategy. This means that the overall problem is solved by repeatedly making the choice that is best locally, and hoping that the combination is best globally. It is not reasonable to expect greedy strategies to work, yet they sometimes do. For example, if you wanted to find the best move in a chess game, it would be naïve to think that taking the opponents queen is always better than a slow defensive pawn move. The queen capture might be followed by you being checkmated the next move, while the defensive move may result in a win for you thirty moves down the line. Nevertheless, there are circumstances when the greedy strategy works and a general mathematical discussion of what these circumstances are brings us to some serious mathematics about *Matroids* and the Matroid Intersection Problem. The interested reader is referred to Lawler's book, Combinatorial Optimization – Networks and Matroids.

Prim's algorithm

The basic idea is to start at some arbitrary node and grow the tree one edge at a time, always adding the smallest edge that does not create a cycle. What makes Prim's algorithm distinct from Kruskal's is that the spanning tree grows connected from the start node. We need to do this n-1 times to make sure that every node in the graph is spanned. The algorithm is implemented with a priority queue. The output will be a tree represented by a parent array whose indices are nodes.

We keep a priority queue filled with all the nodes that have not yet been spanned. The *value* of each of these nodes is equal to the smallest weight of the edges that connect it to the partial spanning tree.

1. Initialize the Pqueue with all the nodes and set their values to a number larger than any edge, set the value of the root to 0, and the parent of the root to nil.
2. While Pqueue is not empty do{

```
        Let x be the minimum value node in the Pqueue; Delete x;
        For every node y in x's adjacency list do {
                If y is in Pqueue and the weight on the edge (x,y)
        is less than value(y) {
        Set value(y) to weight of (x,y);  Set parent of y to x.
                }
        }
}
```

An example will be done in class, and you can find one in your text on page 508. There are some details in the implementation that are not explicit here but which are crucial. For example, when we set value(y) to weight of (x,y), the heap must be modified and this can only be done if you have an inverted index from the values to the heap locations where each is located. This can be stored in an a array whose values must be continually updated during any heap modifications.

The analysis of Prim's algorithm will show an O(e log n) time algorithm, if we use a heap to implement the priority queue. Step 1 runs in O(n) time, because it is just building a heap. Step 2 has a loop that runs O(n) times because we add a new node to the spanning tree with each iteration. Each iteration requires us to find the minimum value node in the Pqueue. If we use a simple binary heap, this takes O(log n). The total number of times we execute the last two lines in step 2 is O(e) because we never look at an edge more than twice, once from each end. Each time we execute these two lines, there may be a need to change a value in the heap, which takes O(lg n). Hence the total time complexity is O(n) + O(n log n) + O(e log n), and the last term dominates.

A final note is that if we use an adjacency matrix instead of adjacency lists, Prim's idea can be implemented without a heap, and the minimum calculation O(n) gets done while we decide whether to updatet he values of the nodes in each iteration. This gives n iterations of O(n) and an O(n^2) total algorithm with much simpler data structures. The question of which method of implementation is best depends obviously on the relationship between e and n. The number of edges is of course bounded between O(n) and O(n^2). For sparse graphs (few edges) the heap and adjacency list method is better, and for dense graphs the two-dim adjacency matrix is best without a heap is better.

Kruskal's Algorithm

Kruskal's algorithm also works by growing the tree one edge at a time, adding the smallest edge that does not create a cycle. However, his algorithm does not insist on the edges being connected until the final spanning tree is complete. We start with n distinct single node trees, and the spanning tree is empty. At each step we add the smallest edge that connects two nodes in different trees.

In order to do this, we sort the edges and add edges in ascending order unless an edge is already in a tree. The code for this is shown below:

For each edge (u,v) in the sorted list in ascending order do {
If u and v are in different trees then add (u,v) to the spanning tree, and union the trees that contain u and v.

Hence we need some data structure to store sets of edges, where each set represents a tree and the collections of sets represents the current spanning *forest.* The data structure must support the following operations: Union (s, t) – which merges two trees into a new tree, and Find-Set(x) which returns the tree containing node x.

The time complexity of Kruskal's algorithm will depend completely on the implementation of Union and Find-Set. The Union-Find data structure can be implemented in a number of ways, the best one showing its efficiency only through *amortized analysis.*

Union-Find Data Structure

The Union-Find data structure is useful for managing *equivalence classes*, and is indispensable for Kruskal's algorithm. It is a data structure that helps us store and manipulate equivalence classes. An equivalence class is simply a set of things that are considered equivalent (satisfies reflexive, symmetric and transitive properties). Each equivalence class has a representative element. We can union two equivalence classes together, create a new equivalence class, or find the representative of the class which contains a particular element. The data structure therefore supports the operations, Makeset, Union and Find. The Union-Find can also be thought of as a way to maipulate disjoint sets, which is just a more general view of equivalence classes.

Makeset(x) initializes a set with element x. Union(x, y) will union two sets together. Find(x) returns the set containing x. One nice and simple implementation of this data structure used a tree defined by a parent array. A set is stored as a tree where the root represents the set, and all the elements in the set are descendents of the root. Find(x) works by following the parent pointers back until we reach nil (the root's parent). Makeset(x) just initializes an array with parent equal to nil, and data value x. Union(x,y) is done by pointing the parent of x to y. Makeset and Union are O(1) operations but Find is an O(n) operation, because the tree can get long and thin, depending on the order of the parameters in the calls to the Union. In particular it is bad to point the taller tree to the root of the shorter tree.

We can fix this by changing Union. Union(x,y) will not just set the parent of x to y. Instead it will first calculate which tree, x or y, has the greater number of nodes. Then it points the parent of the tree with the fewer nodes to the root of the tree with the greater nodes. This simple idea guarantees (a proof by induction is on page 453), that the height of a tree is at most lg n. This means that the Find operation has become O(log n).

Analysis of Kruskal's Algorithm

With the Union-Find data structure implemented this way, Kruskal's algorithm can be analyzed. The sorting of the edges can be done in O(e log e) which is O(e log n) for any graph (why?). For each edge (u,v) we check whether u and v are in the same tree, this is done with two calls to Find which is O(log n), and we union the two if necessary which is O(1). Therefore the loop is O(e log n). Hence the total time complexity is O(e log n).

It turns that if we throw in another clever heuristic to our implementation of the Union-Find data structure, we can improve the performance of the algorithm, but the analysis requires an *amortized analysis.* The heuristic is easy to describe and the final result too, but the analysis is a little involved, and the reading (22.4) is optional. We will have an optional recitation for anyone interested in studying the details.

The trick heuristic is called *path compression.* It is another way to make the trees even shorter. Every time we traverse pointers back to do a Find, we point all nodes up to the root of the tree. This is done in two phases by first finding the root as normal, and then going back to reassign the parents of all visited nodes to the root. Although the worst case remains O(log n) for a Find, the extra constant time work for path compression balances the occasional log n searches. That is, every time we have a long search in a Find, it makes many other Find searches short. The details are hairy, but the upshot is that p operations of Union and Find using weighted union and path compression takes time O(p lg*n). That is, each operation on the average is taking O(lg*n). Hence Kruskal's algorithm runs in time O(e lg*n).

The Function lg*n

Note that lg*n is a very slow growing function, much slower than lg n. In fact is slower than lg lg n, or any finite composition of lg n. It is the inverse of the function f(n) = 2 ^2^2^…^2, n times. For n >= 5, f(n) is greater than the number of atoms in the universe. Hence for all intents and purposes, the inverse of f(n) for any real life value of n, is constant. From an engineer's point of view, Kruskal's algorithm runs in O(e). Note of course that from a theoretician's point of view, a true result of O(e) would still be a significant breakthrough. The whole picture is not complete because the actual best result shows that lg*n can be replaced by the inverse of A(p,n) where A is Ackermann's function, a function that grows explosively. The inverse of Ackermann's function is related to lg*n, and is a nicer result, but the proof is even harder.

Amortized Analysis

Amortized analysis is kind of like average case analysis but not quite. In average case analysis, we notice that the worst case for an algorithm is not a good measure of what turns up in practice, so we measure the complexity of all possible cases of inputs and divide by the number of different cases. In Quicksort, for example, this method of average case analysis resulted in an O(n log n) time analysis rather than O(n^2) which is its worst case performance.

In amortized analysis, we are not working with one algorithm, rather we are working with a collection of algorithms that are used together. Typically this occurs when considering operations for manipulating data structures. It is possible that one of the algorithms takes a long time worst or average case, but that it happens relatively infrequently. So that even if the worst or average case for one operation is O(n), it is possible that this is balanced with enough O(1) operations to make a mixture of p operations have a time complexity of O(p), or O(1) per operation.

We will not get into the details of the proof that p operations of Union-Find needing at most O(p lg*n). However, you should get a flavor of what amortized analysis is used for, and this problem is a perfect motivation for that. Let's discuss a simpler data structure with a collection of operations whose efficiency can be more easily analyzed with amortized analysis.

Stacks and a Simple Amortized Analysis

One way to implement a stack is with a linked list, where each push and pop takes O(1) time. Another way is with an array. The problem with an array is that we need a way to make it larger dynamically. One way to do this is called *array doubling.* In this scheme, a pop is the same as you would expect and uses O(1), but a push can be O(1) or O(n) depending on whether the array is full and needs to be dynamically extended. The idea is that we will double the size of the array anytime a push will overflow the current space. If a push demands a doubling of the array it takes O(n) to do it.

Hence in this scheme, pops are O(1) but pushes are O(n) worst case. The thing is that the O(n) pushes don't happen that often. We can calculate this explicitly with an example. Let's say we are pushing nine elements into the array. The array needs to get doubled when we add the 2nd, 3rd, 5th, and 9th elements. The time for these doublings is 1, 2, 4, and 8 steps respectively. The time for the actual pushes is 9. This gives a total time of 2(8) – 1 + (8 + 1) = 3(8). In general, the time to push n elements into the array is 3n, (recall the that sum 1+2+4+… +n equals 2n-1). This means that over n pushes we use an average of 3 steps per push, even though the worst case push is O(n).

There are many ways to think about amortized analysis, but I think the above idea will give you the flavor in the clearest way. Another way to think of it, is that we add two steps into a savings account, every time we do a fast push, making the expenditure for each fast push three instead of one. Then we cash in on this savings on a slow push, by withdrawing n-2 steps for the doubling. This way each long push (n+1 steps) is accomplished with three steps plus the n-2 we saved up from the short pushes.

It is this accounting scheme that must be determined and specified in every amortized analysis. Each problem requires its own ingenuity and cleverness.

Depth and Breadth First Search

With any data structure the first basic algorithm that we write is one that traverses it. In the case of graphs, the two basic methods of traversal are breadth first and depth first. It turns out that each one of these, but depth first search in particular, can be modified to solve a variety of problems on graphs while it traverses the graph. Both these algorithms run in time proportional to the edges of the graph.

Breadth First Search

Breadth first search traverses a graph in what is sometime called *level order*. Intuitively it starts at the source node and visits all the nodes directly connected to the source. We call these level 1 nodes. Then it visits all the unvisited nodes connected to level 1 nodes, and calls these level 2 nodes etc.

The simple way to implement breadth first search is using a queue. In fact when hear you *breadth* you should think *queue*, and when you hear *depth* you should think *stack*. We have the algorithm output a tree representing the breadth first search, and store the level of each node. The code can be found on page 470 of your text. Here is a perhaps more readable version. We have a parent array to store the search tree, a Level array to store the level, and a visited array to remember who has already been placed on the queue. The book uses a three valued color system white, grey and black instead of this Boolean array. I don't know why this is necessary and I am not sure it is..

```
Initialize:  Queue Q = source;  level[source] = 0, p[source] = nil;
             For all nodes x do visited[x] = false;
             visited[source]= true;

Loop:  While Q is not empty do {
                x = deleteq(Q);
                for all y adjacent to x do
                        if visited[y] = false {
```

```
                visited[y]=true; level[y]=level[x]+1;
                p[y] = x; addq(Q, y)}
        }
```

The total time for initializing is O(n) and the total time for the queuing operations is O(n) because each node is put on the queue exactly once. The total time in the main loop is O(e) because we look at each edge at most twice, once from each direction. This gives a time complexity of O(n+e).

Depth First Search

Depth first search (DFS) traverses a graph by going as deeply as possible before backtracking. It is surprisingly rich with potential for other algorithms. It also returns a search tree. It does not return the level of each node, but can return a numbering of the nodes in the order that they were visited. We first show a depth first search skeleton and define the different kinds classes of edges. Then we show how to augment the skeleton to solve two very basic algorithms: topological sorting, connected components. Each of leverages the power of DFS at a different location in the skeleton. We conclude with a sophisticated use of DFS that finds strongly connected components of a directed graph. You may recall that in month 0 we discussed a method in linear algebra using matrix multiplication that solved this algorithm in O(n^3). Our method will work in O(n+e). There are other sophisticated uses of depth first search including an algorithm to find bi-connected components in undirected graphs, and an algorithm to determine whether a graph is planar. Neither one of these problems has an obvious brute force solution and certainly not an efficient one.

A similar DFS skeleton can be found in your text on page 478.

Depth First Search Skeleton

```
DFS(G, s)

Mark s visited; Dfsnum[s] = count; count++;
//count is a global counter initialized to 1.
/* Process s – previsit stage */
Recursive Loop:  For every y adjacent to s do
        if y is unvisited then {DFS(G, y); parent[y] = x;} else…
                /* process edges {s,y}*/;
/* Process s – postvisit stage */
Mark s finished;
```

The points at which this algorithm can be augmented are threefold:

1. After the node is marked, before looking forward on its edges, (previsit stage).
2. While we process each edge (process edge stage).
3. After all children of a node have been searched (postvisit stage).

Stage two processes edges. Edges can be classified into four categories (only the first two exist for undirected graphs): tree edges, back edges, cross edges and descendant edges. Definitions can be found on page 482 of your text but the best way to understand the classification is to go through a DFS example, exhibit the search tree and identify the different edges. We will do this in class. An example appears in the book on page 479.

DFS in Comparison with BFS

It is stage three that gives depth first search all its potential power. At this postorder time, many nodes have already been examined and a lot of information may be available to the algorithm. There is no analogue to this in breadth first search because there is no recursion. This is why there are so many more applications of DFS than there are for BFS.

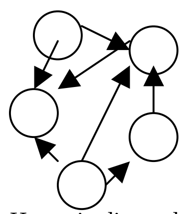
Connected Components – A Simple Application

Our eyes can pick out the connected components of an undirected graph by just looking at a picture of the graph, but it is much harder to do it with a glance at the adjacency lists. Both BFS and DFS can be used to solve this problem, because there is no postorder time processing. The basic trick is to wrap the search in a loop that looks like this:

For each node x in the graph do
    If x is unvisited {mark x; Search(x);}

Doing this with DFS, we can keep a global counter and assign a number to each connected component. The counter k is initialized to 1. During the *process edge* stage, we throw any edge visited on a stack. When we finish Search(x); then before we go back up to the top of the for loop, we pop all edges off the stack until the stack is empty. Then we output these edges with a header that says connected component k, and we increment k.

For directed graphs, it is possible for a search in a single underlying connected component to finish without traversing all of the nodes. This is because the direction of the arrows might effectively disconnect one part of the graph one another even though they are in the same underlying undirected connected component. See the picture below.



Hence in directed graphs a call DFS is always wrapped in a loop like the one above. In both algorithms that follow, we make use of the postorder processing available in DFS, and we assume that the DFS call is wrapped in a loop that checks unvisited nodes.

Topological Sorting – Revisited with DFS

When we introduced the graph data structure, we designed an algorithm for topological sorting. Here we design an alternative algorithm using DFS. This algorithm depends very strongly on postorder time processing. We use a global counter in a postorder processing step that assigns finishing times to each node. The nodes are pushed onto a stack in the order they finish. When the search is over, popping the stack lists the nodes in topological order. This makes sense because we only finish a node after all its descendants are finished. Hence the finishing order is the reverse of a topological sorting.

Strongly Connected Components – A Cool Application of DFS

The description of this algorithm like most variations of depth search is deceptively simple to describe but tedious and complex to prove that it works.

1. Call DFS and calculate finishing times for each node.

2. Call DFS on the transpose (reverse all edges), but consider the nodes in reverse finishing time order.
3. Each connected tree in the DFS forest of trees is a separate strongly connected component.

This algorithm seems to work like magic, and it is indeed a bit amazing. Your text spends four pages and three theorems convincing you of this, and we do not reproduce that information here.

Shortest Path Algorithms

These algorithms are perhaps the most well known algorithm outside of sorting. Most people have heard of Dijkstra's shortest path algorithm. The presentation here follows the spirit of Tarjan in *Data Structure and Network Algorithms.* Our text's presentation is very similar. By the way, the previously mentioned book is an excellent resource for advanced data structures and basic graph algorithms.

The shortest path algorithm gives a graph and a start node, and asks for the shortest paths from that node to every other node in the graph. Note that it does not save any time in general to consider a particular goal node, hence we might as well calculate the shortest paths to all the other nodes. There are version where you wish to calculate all the shortest paths between any two nodes can be solved by repeating the singe source algorithm n times, or with a completely different technique using dynamic programming. We speak about the all pairs shortest path algorithm next week when discussing the topic of dynamic programming.

The shortest path problem is NP-complete when the graph has negative weight cycles, hence the longest path problem is NP-complete in the presence of positive weight cycles, which is the common situation with a graph.

Single Source Shortest Path

The output of this algorithm is a shortest path tree and a distance array. The array dist[x] stores the shortest path distance from the source to x, where dist[s] = 0. The tree is stored via a parent array, (like the minimum spanning tree), where parent[source] is nil. The main tool used in the algorithm is called *scanning* a node. Scanning a node looks at all the nodes adjacent to it and decides whether to change their parent and dist values. Your textbook calls an affirmative decision *relaxing* a node. Ideally, we would like to scan each node exactly once, but that is not always possible.

```
Scan(x)

For every node y adjacent to x do {
        If dist[x] + length(x,y) < dist[y] {
                dist[y] = dist[x] + length(x,y);  parent[y] = x;
        }
}
```

The code looks at each node y adjacent to x and sees whether or not the path from the source to y through x is shorter than the shortest path currently known from the source to y. If it is better, then we change the current dist and parent values for y.

We start the algorithm by initializing the dist and parent arrays, and scanning the source node.

Shortest Path Skeleton

Initialize: for all nodes x do {dist[x] = MAX; parent[x] = nil;}
dist[s] = 0;

Main: Scan(s);
Scanning Loop: Scan the other nodes in some suitable order;

Dijkstra's Algorithm

The point now is to consider the order in which to scan the nodes, and this depends on the kind of graph we are given. For a graph with only positive edges, we use a greedy approach of scanning nodes in ascending order of current distance values. This is called Dijkstra's algorithm. Once a node is scanned, it is never scanned again because we can prove that scanning in this order means that the dist and parent values for that node will subsequently never change. If we never *relax* a node after it is scanned then there is no need to scan it ever again.

An efficient way to implement this algorithm is to keep a heap of the currently unscanned nodes by their dist values, and maintain the heap through possible changes in dist values. Getting the next node to scan is O(log n), and we scan each node exactly once due to the theorem we mentioned. This gives O(n log n). We also must consider the maintenance of the heap, which takes O(log n) but can happen as many as O(e) times. Hence the total time complexity is O(e log n). Without using a heap, the time complexity can be analyzed to be O(n^2). Note that if you use a d-heap (a heap with d children, where d = 2 + e/n), or if the edge weights are restricted to small integers, then we can improve these time complexities, but we will not talk about these advanced variations.

Examples of Dijkstra's algorithm can be found in your text (page 528) and we will do one in class. Note that we did not discuss just what happens with Dijkstra's algorithm in the presence of negative weight edges. I leave this for you to think about.

Acyclic Directed Graphs – Topological Order Scanning

Another way to guarantee that once a node is scanned that it never needs to be scanned again, is to scan the nodes in topological sorted order. In this case, no scanned node can ever be *relaxed* later, because there are no edges coming back to this node in a topological ordering. No fancy theorem here, just simple common sense. Of course constructing a topological ordering is only possible in a directed acyclic graph. Note this works whether or not the graph has negative weight edges.

The Bellman Ford Shortest Path Algorithm for Graphs with Negative Weight Edges but No Negative Weight Cycles – Breadth First Scanning

The shortest path problem is NP-complete in the presence of negative weight cycles, but it can be solved in polynomial time for a graph with negative weight edges and cycles, as long as there are no negative weight cycles.

The algorithm uses a breadth first scanning order. The main difference between this algorithm and the previous algorithms, is that in this case we cannot guarantee that every node will be scanned exactly once. We may have to rescan a node multiple times. The key is that we must scan each node at most n times. This results in an O(ne) time algorithm. The details behind these clams are not at all obvious.

To understand why each node is scanned at most n times and why the complexity is O(ne), it helps to talk about the details of implementation. Breadth first scanning calls for the use of a queue. We initialize a queue to the source node, and while the queue is not empty, we remove a node from the queue, scan it, and set the parent and dist values of its adjacent node appropriately. We add a node to the queue whenever it is relaxed (that is, when its parent and dist values are changed). Note if a node, is already on the queue when it is relaxed, then we do not put it on again, we simply leave it on. This implies that at any given time no node is on the queue more than once.

Let's analyze the algorithm by looking at *phases.* The 0th phase of the queue is when it consists of just the source node. The ith phase consists of the nodes on the queue after the i-1 st phase nodes have been removed. There is a crucial theorem proved by induction that states that if there is a shortest path from the source to a node x containing k edges, then just before the kth phase of the algorithm, dist[x] will equal the length of this path. Since any path without cycles from the source to a node can contain at most n-1 edges, this means the algorithm needs at most O(n) phases. Moreover, since each individual phase has no duplicate nodes on the queue, at most n nodes can be on the queue in a given phase. Processing a phase means deleting and scanning these n nodes. This processing takes O(e) time, because the worst case is that we look at every edge adjacent to thee nodes. Since we process at most O(n) phases with O(e) time perphase, this gives the O(ne) time complexity.

Geometric Algorithms

Geometric algorithms are wonderful examples for programming, because they are deceptively easy to do with your eyes, yet much harder to implement for a machine.

We will concentrate on a particular problem called convex hull, which takes a set of points in the plane as its input and outputs their convex hull. We will stay away from formal definitions and proofs here, since the intuitive approach will be clearer and will not lead you astray. To understand what a *convex hull* is, imagine that a nail is hammered in at each point in the given set, the convex hull contains exactly those points that would be touched by a rubber band which was pulled around all the nails and let go. The algorithm is used as a way to get the natural border of a set of points, and is useful in all sorts of other problems.

Convex Hull is the sorting of geometric algorithms. It is fundamental, and as there are many methods for sorting, each of which illustrates a new technique, so it is for convex hull.

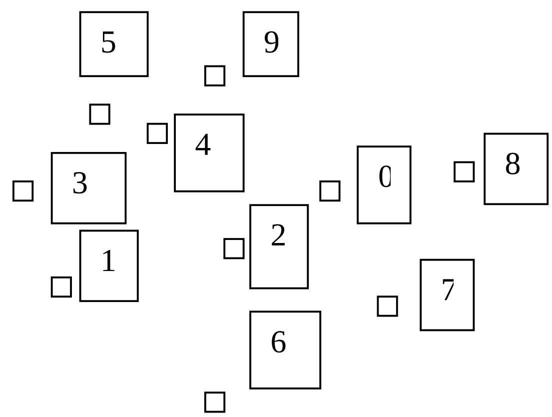
Graham Scan

The particular algorithm we will implement for Convex Hull is due to Ron Graham and was discovered in 1972. Graham Scan, as it is called, works by picking the lowest point p, i.e. the one with the minimum p.y value (note this must be on the convex hull), and then scanning the rest of the points in counterclockwise order with respect to p. As this scanning is done, the points that should remain on the convex hull, are kept, and the rest are discarded leaving only the points in the convex hull at the end.

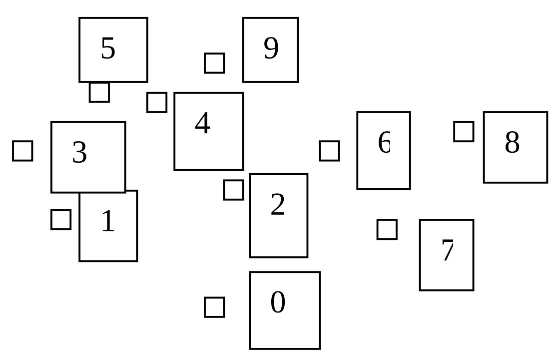
To see how this is done, imagine first that, by luck, all the points scanned are actually on the convex hull. In that case, every time we move to a new point we make a left turn with respect to the line determined by the last two points on the hull. Therefore, what Graham Scan does, is to check if the next point is really a left turn. If it is NOT a left turn, then it backtracks to the pair of points from which the turn would be a left turn, and discards all the points that it backs up over. Because of the backtracking, we implement the algorithm with a stack of points. An example is worth a thousand words. The input list of points is:

(A, 0, 0) (B, -5, -2) (C, -2, -1) (D, -6, 0) (E, -3.5, 1) (F, -4.5, 1.5)

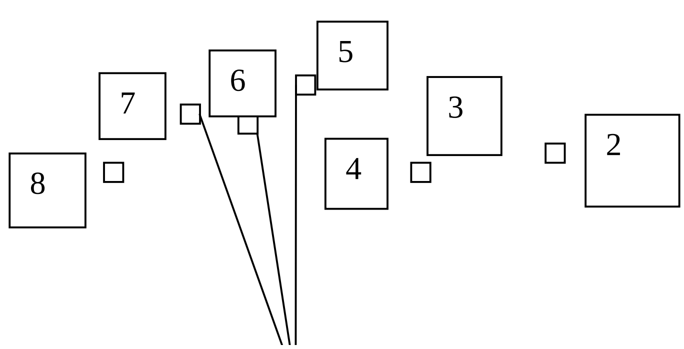
(G, -2.5, -5) (H, 1, -2.5) (I, 2.5, .5) (J, -2.2, 2.2).

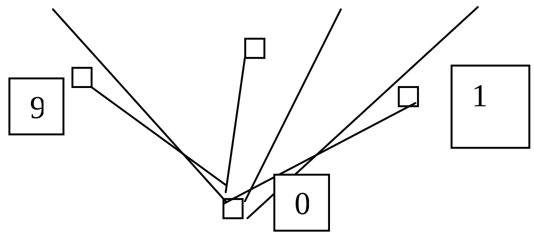


The array of input points is shown above labeled by index in the array (rather than their char label). The point labeled A is in index 0, B is in index 1, etc. The lowest point is computed and swapped with the point in index 0 of the array, as shown below.

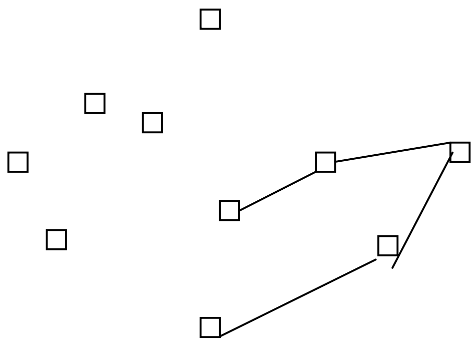


The points are then sorted by their polar angles with respect to the lowest point.

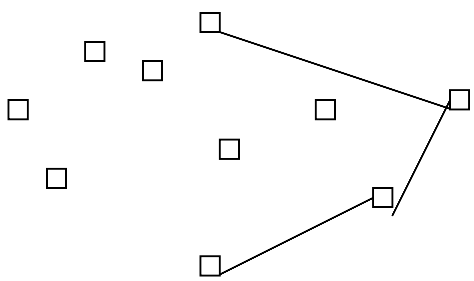




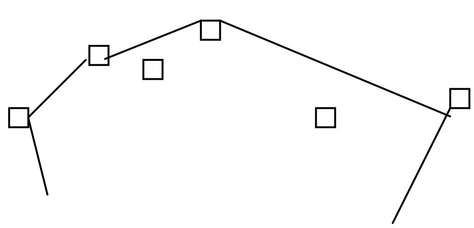
The points are sorted and rearranged in the array as shown above. The turn from line 0-1 to point 2 is left, from 1-2 to 3 is left, from 2-3 to 4 is left. Now the stack contains the points 01234. This represents the partial hull in the figure below.

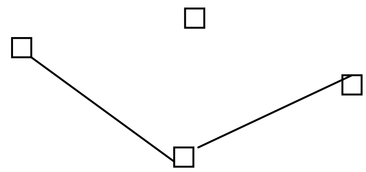


The turn from line 3-4 to point 5 is right, so we pop the stack. The turn from 2-3 to 5 is right, so we pop again. The turn from 1-2 to 5 is left, so we push 5 on the stack. The stack now has 0125, and the picture looks like this:



The turn from line 2-5 to 6 is left so 6 is pushed on the stack. Then the turn from 5-6 to 7 is right, so 6 is popped and 7 is pushed because the turn from line 2-5 to 7 is left. The rest of the turns are left, so 8 and 9 are pushed on the stack. The final stack is 0125789, and the convex hull is shown below:





Graham Scan Pseudo-code: The algorithm takes an array of points and returns an array of points representing the convex hull.

1. Find the lowest point p, (the point with the minimum y coordinate). If there is more than one point with the minimum y coordinate, then use the leftmost one.
2. Sort the remaining points in counterclockwise order around p. If any points have the same angle with respect to p, then sort them by increasing distance from p.
3. Push the first 3 points on the stack.
4. For each remaining point c in sorted order, do the following:
   b = the point on top of the stack.
   a = the point below that on the stack.
   While a left turn is NOT made while moving from a to b to c do
      pop the stack.
      b = the point on top of the stack.
      a = the point below that on the stack.
   Push c on the stack.
5. Return the contents of the stack.

Implementation Details for the Point Class:

Private Data:

We start by defining a simple geometric class *point* and deciding on the appropriate private data and member functions. A *point* should have three fields: two are float for the *x* and *y* coordinates, and one is a char for the name of the point.

Constructors:

A three parameter constructor should be created to set up points.

Methods:

An output method to print out a point by printing its name (char) along with its coordinates.

Accessor methods to extract the *x* or *y* coordinates of a point.

A static distance method to determine the distance from one point to another.

A *turn-orientation* method that takes two points *b* and *c* and returns whether the *sweeping movement* from the line a-b to the line a-c goes clockwise (1), counterclockwise (-1) or neither (0). (The result is neither (0) when a, b and c are all on the same line.) This function is necessary for deciding whether a left or right turn is made when moving from *a* to *b* to *c* in step 4 of the pseudo-code above. It is also useful for sorting points by their polar angles.

It may not be obvious how to implement this function. One method is based on the idea of the cross product of two vectors. Let *a*, *b* and *c* be points, where *x* and *y* are accessor methods to extract the *x* and *y* coordinates respectively.

if $(c.x - a.x)(b.y - a.y) > (c.y - a.y)(b.x - a.x)$ then the movement from line *a-b* to line *a-c* is clockwise.
if $(c.x - a.x)(b.y - a.y) < (c.y - a.y)(b.x - a.x)$ then the movement from line *a-b* to line *a-c* is counterclockwise.
Otherwise the three points are co-linear.

To understand this intuitively, concentrate on the case where the lines a-b and a-c both have positive slope. A clockwise motion corresponds to the line a-b having a steeper (greater) slope than line a-c. This means that $(b.y - a.y)/(b.x - a.x) > (c.y - a.y)/(c.x - a.x)$. Multiply this inequality by $(c.x - a.x)(b.x - a.x)$ and we get the inequalities above.

The reason for doing the multiplication and thereby using this *cross product* is:
1. To avoid having to check for division by zero, and
2. So that the inequality works consistently for the cases where both slopes are not necessarily positive. (You can check for yourself that this is true).

Graham Scan should be coded using an abstract STACK class of points. The sorting in step two can be done by comparing pairs of points via the turn-orientation method with respect to the lowest point (object *p*). An *interface* (if you use Java) may be convenient to allow the sorting of points.

A Note on Complexity:

The complexity of Graham Scan is $O(n \log n)$. We will discuss informally what this means and why it is true. It means that the number of steps in the algorithm is bounded asymptotically by a constant times n log n where n is the number of points in the input set. It is true because the most costly step is the sorting in step 2. This is $O(n \log n)$. Step 1 takes time $O(n)$. Step 3 takes $O(1)$. Step 4 is trickier to analyze. It is important to notice that although each of the $O(n)$ points are processed, and each might in the worst case have to pop the stack $O(n)$ times, overall this does NOT result in $O(n^2)$. This is because overall, every point is added to the stack exactly once and is removed at most once. So the sum of all the stack operations is $O(n)$.

There are many $O(n \log n)$ and $O(n^2)$ algorithms for the convex hull problem, just as there are both for sorting. For the convex hull there is also an algorithm that runs in $O(nh)$, where *n* is the number of points in the set, and h is the number of points in the convex hull. For small convex hulls (smaller than *log n*) this algorithm is faster than *n log n*, and for large convex hulls it is slower.

Jarvis' Algorithm for Convex Hull

Jarvis' algorithm uses some of the same ideas as we saw in Graham Scan but it is a lot simpler. It does no backtracking and therefore does NOT need to use a STACK class, although it still makes use of the ARRAY class template with your point class.

As before, we start by adding the lowest point to the convex hull. Then we repeatedly add the point whose polar angle from the previous point is the minimum. This minimum angle computation can be done using the clockwise/counterclockwise member function, similar to how the sorting step (step 2) of Graham Scan uses the function.

The complexity of this method is $O(nh)$ where *h* is the number of points in the convex hull, because in the worst case we must examine $O(n)$ points to determine the minimum polar angle for each point in the hull.