Contents

Algorithms

Why should I learn about algorithms?

1. Your knowledge will be tested in CSci 202, and all dependent CSci classes.
2. Solving problems is a bankable skill and algorithms help you solve problems.
3. Job interviews test your knowledge of algorithms.
4. You can do things better if you know a better algorithm.
5. Inventing and analyzing algorithms is a pleasant hobby.
6. Publishing a novel algorithm can make you famous. Several Computer Scientists started their career with a new algorithm.

What is an algorithm?

An algorithm is a clearly described procedure that tells you how to solve a well defined problem.

A clearly described procedure is made of simple clearly difined steps. It often includes making decisions and repeating earlier steps. The procedure can be complex, but the steps must be simple and unambiguous. An algorithm describes how to solve a problem in steps that a child could follow without understanding the problem.

Algorithms solve *problems*. Without a well defined problem an algorithm is not much use. For example, if you are given a sock and the problem is to find a matching sock in a pile of similar socks, then an algorithm is to take every sock in turn and see if it matches the given sock, and if so stop. This algorithm is called a *linear search*. If the problem is that you are given a pile of sockes and you have to sort them all into pairs then a different algorithm is needed.

Exercise: get two packs of cards an shuffle each one. Take one card from the first pack and search through the other pack to find the matching card. Repeat this until you get a feel for how it performs.

If the problem is putting 14 socks away in matching pairs then one algorithm is to

1. clear a place to layout the unmatched socks,
2. for each sock in turn,
    1. compare it to each unmatched sock and if they match put them both away as a pair. But, if the new sock matches none of the

unmatched socks, add it to the unmatched socks.

I call this the "SockSort" algorithm.

Exercise. Get a pack of card and extract the Ace,2,3, .. 7 of the hearts and clubs. Shuffle them together. Now run the SockSort algorithm to pair them up by rank: Ace-Ace, 2-2, ...

If you had an array of 7 numbered boxes you could sort the cards faster than sorting the socks. Perhaps I should put numbered tags on my socks? Note: some times a real problem is best avoided rather than solved.

So, changing what we are given, changes the problem, and so changes the best algorithm to solve it.

Here is another example. Problem: to find a person's name in a telephone directory. Algorithm: use your finger to split the phone book in half. Pick the half that contains the name. Repeatedly, split the piece of the phone book that contains the name in half, ... This is a *binary search* algorithm.

Here is another example problem using the same data as the previous one: Find my neighbor's phone number. Algorithm: look up my address using binary search, then read the whole directory looking for the address that is closest to mine. This is a *linear search*.

If we change the given data, the problem of my neighbor's phone number has a much faster algorithm. All we need is copy of the census listing of people by street and it is easy to find my neighbor's name and so phone number.

How to express problems

A problem best expressed in terms of

- *Givens: What is there before the algorithm?*
- *Goals: What is needed?*
- *Operations: What operations are permitted?*

How to express algorithms

We either use a structured form of English with numbered steps or a pseudocode that is based on a programming language.

Examples below.

What do I need to learn in CSci202 about algorithms?

1. Know the definition of an algorithm above.
2. Know how to express a problem.
3. Recognize well known problems.
4. Recognize well known algorithms.
5. Match algorithms to problems.
6. Describe algorithms informally and in structured English/pseudocode
7. Walk through an algorithm, step by step, for a particular problem by hand.
8. Code an algorithm expressed in structured English.
9. Know when to use an algorithm and where they relate to objects and classes,

Where does the word algorithm come from?

The word is a corruption of the name of a mathematician born in Khwarizm in Uzbekistan in the 820's(CE). Al-Khwarizmi (as he was known) was one of the first inducted into the "House of Wisdom" in Baghdad where he worked on algebra, arithmetic, astronomy, and solving equations. His work had a strongly computational flavor. Later, in his honor, medieval mathematicians in Europe called similar methods "algorismic" and then, later, "algorithmic". [pages 113-119, "The Rainbow of Mathematics", Ivor Grattan-Guiness, W W Norton & Co, NY, 1998]

Should I use the standard algorithms in C++?

If you are using C++ and understand the ideas used in the C++ <algorithm> library you can save a lot of time. Otherwise you will have to reinvent the wheel. As a rule, the standard algorithm will be faster than anything you could quickly code to do the same job. However, you still need to understand the theory of algorithms and data structures to be able to use them well.

Bjarne Stroustroup, the developer of C++, has written a very comprehensive and deep introduction to the standard C++ library as part of his classic C++ book. This book is in the CSUSB library.

As a general rule: a practical programmer starts searching the library of their language for known algorithms before "reinventing the wheel".

Is there an easy way to grasp a new algorithm?

No... unless the algorithm is simple or the author very good at writing.

It helps if you know many algorithms. The same ideas turn up in many different algorithms.

I find that a deck of playing cards is very helpful for sorting and searching algorithms.

A pencil and paper is useful for doing a dry run of an algorithm. With a group, chalk and chalk board is very helpful.

Programming a computer just to test an algorithm tends to waste time unless you use the program to gather statistics on the performance of the algorithm.

When there are loops it is well worth looking for things that the body of the loop does not change. They are called *invariants*. If an invariant is true before the loop, then it will be true afterward as well. You can often figure out precisely what an algorithm does by noting what it does not change...

Are there any common problems that have algorithmic solutions?

Probably the simplest algorithm worth know solve the problem pf swapping the values of two locations or variables. Here you are given two variables of loctations *p* and *q* that hold the same type of data, and you need to swap them. This looks trivial but in fact we need to use an extra temporary variable *t*:

Algorithm to Swap p and q: (
1. *SET t = p*
2. *SET p = q*
3. *Set q = t*

)

The two classic problems are called **Searching** and **Sorting.** In **searching** we are given a collection of objects and need to find one that passes some test. For example: finding the student whose student Id number ends "1234". In **sorting,** we are given a collection and need to reorganize it according to some rule. An example: placing a grade roster in order of the last four digits of student identifier,

Finding the root of an equation: is this a search or a sort?

You can't find things in your library so you decide to place the books in a particular order: is this a search or a sort?

Optimization problems are another class of problems that have attracted a lot of attention. Here we have a large number of possibilities and want to pick the best. For example: what dress is the most attractive in a given price range? What is the shortest route home from the campus? What is the smallest amount of work needed to get a B in this class?

What types of algorithm are there?

- (**linear**): a linear algorithm does something, once, to every object in turn in a collection. Examples: looking for words in the dictionary that fit into a crossword. Adding up all the numbers in an array.
- (**divide_and_conquer**): the algorithms divide the problem's data into pieces and work on the pieces. For example conquering Gaul by dividing it into three pieces and then beating rebellious pieces into submission. Another example is the Stroud-Warnock algorithm for displaying 3D scenes: the view is divided into four quadrants, if a quadrant is simple, it is displayed by a simple process, but if complex, the same Stroud-Warnock algorithm is reapplied to it. Divide-and-conquer algorithms work best when their is a fast way to divide the problem into sub-problems that are of about the same difficulty. Typically we aim to divide the data into equal sized pieces. The closer that we get to this ideal the better the algorithm. As an example, merge-sort splits an array into nearly-equal halves, sorts each of them and then merges the two into a single one. On the other hand, Tony Hoare's Quicksort and Treesort algorithms make a rough split into two parts that can be sorted and rapidly joined together. On average each split is into equal halves and the algorithm performs well. But in the worst case, QuickSort split the data into a single element vs all the rest, and so performs slowly. So, divid_and_conquer algorithms are faster with precise divisions, but can perform very badly on some data if you can not guarantee a 50-50 split.
- (**binary**): These are a special **divide_and_conquer** algorithm where we divide the data into two equal halves. The classic is binary search for hunting lions: divide the area into two and pick a piece that contains a lion.... repeat. This leads to an elegant way to find roots of equations.

  ALGORITHM to find the integer *lo* that is just below the square root of an integer *n* (√*n*).

  (
  1. *SET low = 0 and high = n, (now low<= √ n < high).*
  2. *SET mid = (low + high)/ 2, (integer division)*
  3. *IF mid * mid > n THEN SET high = mid*
  4. *ELSE SET lo =mid.*
  5. *END IF (again low<=√ n < high)*
  6. *IF low < high -1 THEN REPEAT from step 2 above.*

) Note: this algorithm needs checking out!
- (**Greedy algorithms**): try to solve problems by selecting the best piece first and then working on the other pieces later. For example, to pack a knapsack, try putting in the biggest objects first and add the smaller one later. Or to find the shortest path through a maze, always take the shortest next step that you can see. Greedy algorithms don't always produce optimal solutions, but often give acceptable approximations to the optimal solutions.
- (**Iterative algorithms**): start with a value and repeatedly change it in the direction of the solution. We get a series of approximations to the answer. The algorithm stops when two successive values get close enough. For example: the **Newton-Raphson** algorithm for calculating approximate square roots.

  ALGORITHM Given a positive number *a* and error *epsilon* calculate the square root of *a*:

(
1. *SET oldv=a*
2. *SET newv=(1+a)/2*
3. *WHILE | oldv - newv | > epsilon*

   (
   1. *SET oldv =newv*
   2. *SET newv =(a+oldv * oldv)/(2*oldv)*

)
4. *END WHILE*

)

END ALGORITHM (*newv* is now within *epsilon* of the square root of a)

Exercise. Code & test the above.

Can every problem be solved by an algorithm?

NO! Take CSCI546 to see what problems are *unsolvable* and why this is so. As a quick example: No algorithm can exist for finding the bugs in any given program.

Optimization problems often prove difficult to solve: consider finding the highest point in Seattle without a map in dense fog...

How are algorithms expressed?

First they were expressed in natural language: Arabic, Latin, English, etc.

In the 1950s we used flow charts. These where reborn as Activity Diagrams in the Unified Modeling Language in the 2000s.

Boehm and Jacopini proved in the 1960's that all algorithms can constructed using three structures

- *Sequence*
- *Selection -- if-then-else, switch-case, ...*
- *Iteration -- while, do-while, ...*

From 1964 onward we used "Structured English" or Pseudo-code. Structured English is English with "Algol 60" structures. "Algol" is the name of the "Algorithmic Language" of that decade. I have a page [**algorithms.html**] of algorithms written in a C++based Pseudo-code.

Here is a sample of structured English:

```
ALGORITHM Sock_Sort.

    clear space for unmatched socks

    FOR each sock in the pile,

        FOR each unmatched sock UNTIL end or a match is found

            IF the unmatched sock matches the sock in your hand THEN

                form a pair and put it in the sock drawer

            END IF

        END FOR

        IF sock in hand is unmatched THEN

            put it down with the unmatched socks

        END IF

    END FOR

END ALGORITHM
```

Where do algorithms appear in objects and classes?

Algorithms often appear inside the methods in a class. However some algorithms are best expressed in terms of interacting objects. So, a method may be defined in terms of a traditional algorithm or as a system of collaborating objects.

You need an algorithm any time there is no simple sequence of steps to code the solution to a problem inside a function. It is wise to either write at an algorithm or use the UML to sketch out the messages passing between the objects.

Algorithms can be encapsulated inside objects. If you create an inheritance hierarchy, you can organize a set of objects each knowing a different algorithm (method). You can then let the program choose its algorithm at run time.

When should I write an algorithm?

Algorithms are used in all upper division computer science classes.

I write my algorithm, in my program, as a sequence of comments. I then add code that carries out each step -- under the comment that describes the step.

How do you write algorithms?

First there is no algorithm for writing algorithms! So here are some hints.

1. The more algorithms you know the easier it is to pick one that fits, and the more ideas you have to invent a new one. Take CSci classes and read books.
2. Look on the WWW or in the Library for ideas.
3. Try doing several simple cases by hand, and observing what you do.
4. Work out a 90% correct solution, and add IF-THEN-ELSEs to fix up the special cases...
5. Go to see a CSCI faculty member: this is free when you are a student. Once in the real world you have to hire us as consultants.
6. Often a good algorithm may need a special device or data structure to work. For example, in my office I have multi-pocket folder with slots labeled with dates. I put all the papers for a day in its slot, and when the day comes I have all the paperwork to hand. CSCI330 teaches a lot of useful data structures and the C++ library has a dozen or so.
7. Think! This is hard work but rewarding when you succeed.

How are algorithms tested?

First, try doing it by hand. Second, discuss it with a colleague. Third, try coding and running it in a program. Fourth, go back and prove that your algorithm must work.

Are there any good books on algorithms?

Yes. Many.

You should use known algorithms whenever you can and always state where they came from. Put this citation as a comment in your code. First this is honest. Second this makes it easier to understand what the code is all about.

Check out the text books in the Data Structures and Algorithms classes in the upper division of our degree programs.

John Mongan and Noah Suojanen have written "Programming Interviews exposed: Secrets to landing your next job" (Wiley 2000, ISBN 0-471-38356-2). Most chapters are about problem solving and the well known algorithms involved.

Donald Knuth 's multi-volume "Art of Computer Programming" founded the study of algorithms and how to code and analyze them. It remains an incredible resource for computer professionals. All three volumes are in our library.

Jon Bentley 's two books of "Programming Pearls" are lighter than Knuth's work but have lots of good examples and advice for programmers and good discussion of algorithms. They are in the library.

G H Gonnet and R. Baeza-Yates wrote a very comprehensive "Handbook of Algorithms and Data Structures in Pascal and C" (Addison-Wesley 1991) which still my favorite resource for detailed analysis of known algorithms. There is a copy in my office.... along with other resources.

The Association for Computing Machinery (ACM) started publishing algorithms in a special supplement (CALGO) in the 1960s. These are algorithms involving numbers. So they tend to be written in FORTRAN.

Are there any algorithms on the web?

Yes -- lots. The Wikipedia, alone, has dozens of articles on particular algorithms, on the theory of algorithms, and on classes of algorithms.

How are algorithms turned into code?

Step by step you translate each line of the algorithm into your target language. Ideally each line in the algorithm becomes two or three lines of code. I like to leave the steps in my algorithm visible in my code as comments.

Do this in pencil or in an editor. You will need to make changes in it!

How are algorithms rated?

You know that movies are rated by using stars or "thumbs-up". Rating an algorithm is more complex, more scientific, and more mathematical. In fact, we label algorithms with a formula, like this, for example:

1. *The linear search algorithm is order big_O of n*

   or in short

2. *Linear search is O(n)*

   This means: for very large values of *n* the search can not take more than a linear time (plus some small ignored terms) to run.

   On the other hand we can show:

3. *A Binary search is O(log n)*

   The above formulas tell us that if we make *n* large enough then binary search will be faster than linear search.

   As another example, a linear search is in *O(n)* and the Sock-Sort (above) is in *O(n squared)*. This means that the linear search is *better* than the sock_search. But in what way? This takes a little explanation.

   Originally (in the 40's through to the mid-60's) we worked out the *average number of steps* to solve a problem. This tended to be correlated with the time. However we found (Circa 1970) two problems with this measure. First, it is often hard to calculate the averages. Knuth spends pages deriving the average performnce of Euclid's algorithm -- which is only 5 lines long! Second, the average depends on how the data is distributed. For example, a linear search is fast if we are likely to find the target at the start of the search rather than at the end. Worse, certain sorting algorithms work quickly on average but sometimes are very slow. For example the beginner's Bubble Sort is fast when most of the data is in sequence. But Quick Sort can be horribly slow on the same data.... but for other re-orderings of the same data quick sort is much better than bubble sort. So before you can work out an average you have to know the frequencies with which different data appear. Typically we don't know this distribution.

   These days a computer scientist judges an algorithm by its *worst case* behavior, We are pessimistic, with good reason. Many of us have implemented an algorithm with a good average performance on random data and had users complain because their data is not random but in the worst case possible. This happened, for example, to Jim Bentley of AT&T [Programming Pearls above]. In other words we choose the algorithm that gives the best guarantee of speed, not the best average performance. It is also easier to figure out the worst case performance than the average -- the math is easier.

   The second complication for comparing algorithms is how much data to consider. Nearly all algorithms have different times depending on the amount of data. Typically the performance on small amounts of data is more erratic than for large amounts of data. Further, small amounts of data don't make a big delays that the users will notice. So, it is best to consider large amounts of data. Luckily, mathematicians have a tool kit for working with larger numbers. It is called **asymptotic analysis.** This is the calculus of what functions *look like* for large values of their arguments. It simplifies the calculations because we only need to look at the higher order terms in the formula.

   Finally, to get a standard "measure" of an algorithm, independent of its hardware, we need to eliminate the speed of the processor from our comparison. We do this by removing all constant factors out of our formula to give the algorithm its rating.

   We talk about the *order* of an algorithm and use a big letter O to symbolize the rating. To summarize: To work out the order of an algorithm, calculate
   1. the number of simple steps
   2. in the worst case
   3. as a formula including the amount of data
   4. for only large amounts of data
   5. including only the most important term
   6. and ignoring constants

   For example, when I do a sock-search, with *n* socks, in the worst case I would have to layout *n*/2 unmatched socks before I found my first match, and finding the match I would have to look at all *n*/2 socks. Then I'd have to look at the remaining *n*/2 -1 socks to match the next one, and then *n*/2-2, ... So the total number of "looks" would be

4. *1 + 2 + 3 + ... + n/2*

   or

5. *(1 + n/2) * (n/ 4)*

   or

6. *n/4 + n^2/8*

   Simplifying by ignoring the n/4 term and the constant (1/8) we get

7. *O(n^2)*

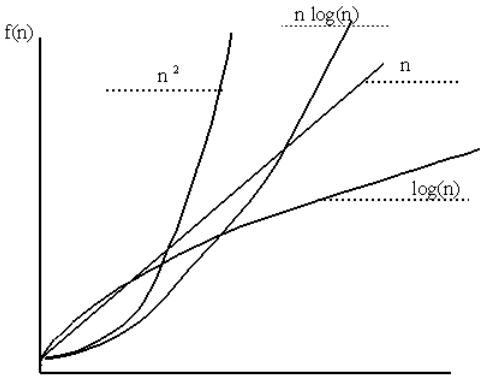
   Here is listing of typical ratings from best to worst...:

Table

Name	Formula	Note
Constant	O(1)	
Logarithmic	O(log *n*)	Good search algorithm
Linear	O(*n*)	Bad Search algorithms
n log n	O(*n* * log *n*)	Good sort algorithms
n squared	O(*n*^2)	Simple sort algorithms
Cubic	O(*n*^3)	Normal matrix multiplication
Polynomial	O(*n*^p)	good solutions to bad problems
Exponential	O(2^*n*)	Not a good solution to a bad problem
Factorial	O(*n*!)	Checking all permutations of *n* objects.

(Close Table)
Note: I wrote *n*^2, *n*^3, *n*^p, 2^*n*, etc. to indicate powers/superscripts. *p* is some power > 1.

Here is a picture graphing some typical Os:



Notice how the *worst* functions may start out less than the better ones, but that they always end up being bigger.

Most algorithms for simple problems have a worst case times that are a power of *n* times a power of *log n*. The lower the powers, the better the algorithm is.

There exist thousands of problems where the best solutions we have found, however, are exponential -- O(2^n)! Why we have failed to improve on this is one of the big puzzles of Computer Science.

Helpful Page on BigO

Please read and study this page: [**000957.html**]

What is this Big-O Notation?

A formula like O(*n^2.7*) names a large family of similar functions that are smaller than the formula for very large *n* (if we ignore the scale). *n* and *n*n* are both in O(*n^2.7*). *n^3* and exp(*n*) are not in O(*n*^2.7). Now, a clever divide and conquer matrix multiplication algorithm is O(*n*^2.7) and so better than the simple *O(n^3)* one for large matrices.

Big_O (asymptotic) formula are simpler and easier to work with than the original formula because we can ignore so much: instead of 2^*n* +*n*^2.7+123.4*n* we write *2^n* or *exponential*.

Timing formula are expressed in terms of the size *n* of the data. To simplify the formulas we remove all constant factors: 123**n*n* is replaced by *n*n*. We also ignore the lower order terms: n*n + n + 5 becomes n*n. So

8. *123*n^2 +200*n+5 is in O(n^2).*

To be very precise and formal, here is the classic text book definition:

9. *f(n) is in O(g(n)) iff for some constant c>0, some number n0, and all n > n0 (f(n) <= c * g(n)).*

This means that to show that *f* is in O(*g*) then you have to find a constant multiplier *c* and an number *n0* so that for *n* bigger than *n0*, *f*(*n*) is less than or equal to *c**g*(*n*).

For example *123n* is in *O(n^2)* because for all *n> 123*, *123n <= n^2*. So, by choosing *n0=123* and *c=1* we have *123n <= 1 * n^2*.

We say *f* and *g* are **asymptotically equivalent** if and only if both (1) *f*(*n*) is in O(*g*(*n*)) and (2) *g*(*n*) is in O(*f*(*n*)).

So *n^2-3* is asymptotically equivalent to *1+2*n+*123*n`^2.

There is another way to look at the ordering of these functions. We can look at the limit *L* of the ratio of the functions for large values:

10. *L=lim[n-> ∞](f(n)/g(n)).*
    - *If L=0 then f(n) in O(g(n)).*
    - *If L=∞ then g(n) in O(f(n)).*
    - *If L is a finite non-zero constant then f(n) is asymptotically equivalent to g(n).*

In CSCI202 I expect you to take these facts on trust. Proofs will follow in the upper division courses.

Exercise: Reduce each formula to one the classic "big_O"s listed.
1. 42n^2+100n+2000
2. 1000+n^3
3. log(n) + 3n
4. 200 n + n * log(n).

Answers (randomized)

1. *n^3*
2. *n*
3. *n log(n)*
4. *n^2*

Where can I learn more about this analysis

Classes like CSCI431 an MATH372. Or hit the stacks and Wikipedia.

Is there an efficient algorithm for every problem?

Probably not.

There are everyday problems that force you to find needles in haystacks. Problems like this force a computer to do a lot of work to solve them. You need to learn to spot these, and try to avoid them if possible.

We normally consider polynomial algorithms as efficient and non-polynomial ones as hard. Computer scientists have discovered a large class of problems that don't seem to have polynomial solutions, even though we can check the correctness of the answer efficiently. A common example is to find the shortest route that visits every city in a country in the shortest time. This is the famous Traveling Salesperson's Problem. Warning: you can waste a lot of time trying to find an efficient solution to this problem.

What are the important algorithms of Computer Science

Each discipline (Mathematics, Physics, . . .) has its own "Classic" algorithms. In computer science the most famous algorithms fall into to two types: sorting and searching. Other classes of algorithm include these involved in graphs, optimization, graphics, compiling, operating systems, etc. Here is my personal selection and classification of Searching and Sorting algorithms.

Searching Algorithms

We give searching algorithms a collection of data and a key. Their goal is to find an item in the collection that matches the key. The algorithms that work best depend on the structure of the given collection.

1. (**Direct Access**): The algorithm calculate the address of the data from a given data value. Example: arrays. Example: getting direct access data from a disk. This avoids the need to look for the data! O(1).
2. (**Linear Search**): The algorithm tries each item in turn until the key matches it. Finds unique items or can create a set of matching items. O(n).
3. (**Binary Search**): The collection of data must be sorted, Look at the middle one, if it is too big, try the first half, but if it is too small, try the other half. Repeat. O(log n).
4. (**Hashing**): The algorithm calculates a value from the key that gives the address of a data structure that has many items of data in it. Then it searches the data structure. Works well (O(1)) when you have at least twice as much storage as the data and the rate of increase is small. For very large n, O(log n).
5. (**Indexes**): An Index is a special additional data structure that records where each key value can be found in the main data structure. It takes space and time to maintain but lets you retrieve the data faster. Indexes may be direct or need searching. If the index is optimal, the time is O(√ n).
6. (**Trees**): These are special data structures that can speed up searches. A tree has nodes. Each node has an item of data. Some nodes are leaves, and the rest have branches leading to another tree. The algorithm chooses one branch to follow to find the data. If the tree is balanced (all branches have nearly the same length) this gives O(log n) time. If the tree is not balanced, the worst case is O(n). There exist special forms that use O(log n) to maintain O(log n) search.
7. (**Data Bases**): These are large, complex data structures stored on disk. They have a complex and rigid structure that takes careful planning. They use all the searching and sorting algorithms and data structures to help users store and retrieve the data they want. Take CSCI350 and CSCI580? to learn about these.

8. Combinations: you can design data so that two or three different searches are used to find data. For example: A hash code gives you the starting point for a linear search. Direct access to an index gives you the address that directs you to the block of disk storage that contains the data, in this block is another index that has the number of the data record, and the actual byte is then calculated directly.

Sorting Algorithms

We give a sorting algorithm a collection of items. It changes the collection so that the data items are (in some sense or other) increasing. The algorithm needs a comparison operation that compares two items and tells whether one in smaller than the other. An example for numbers is "a < b" but in general we might have to create a Boolean function "compare" to do the comparison.

1. (**Slow Sort**): Throw all the data in the air . . . If it comes down in the right order, stop, else repeat. O(n!). Example of a very bad algorithm.
2. (**Bubble Sort**): Scan the data from first to last, if two adjacent items are out of order, swap them. Repeat until they make no swaps. The beginner's favorite sort. Works OK if the data is almost in the right order. There were many clever optimizations that often slowed this algorithm down! O(n*n)
3. (**Cocktail shaker Sort**): a variation on bubble sort.
4. (**Insertion Sort**): Like a Bridge player sorts a hand of cards. Mentally split the hand into a sorted and unsorted part. Take each

unsorted item and search the sorted items to see where it fits. O(n^2). Is simple enough that for small amounts of data (n: 1..10) it is fast enough.
5. (**Selection Sort**): Find the maximum item and swap it with the last item. Repeat with all but the last item. Gives a fixed number of swaps. Easy to understand. Always executes *n* swaps. Time O(n^2).
6. (**Shell Sort**): Mr. Shell improved on bubble sort by swapping items that are not adjacent. Complex and clever. Basic idea: For a sequence of decreasing *ps*, take every p'th item starting with the first and sort them, next every p'th item starting with the second, repeat with 3rd..(p-1)th. Then decrease p and do it again. Different speeds for different sequences of p's. Average speed O(n^p) where 1<*p*<=2. The divide-and-conquer algorithms (below) are better.
7. (**Quick Sort**): Tony Hoare's clever idea. Partition the data in two so that every item in one part is less than any item in the other part. Sort each part (recursively) . . . Good performance on random data: O(n log n) but has a bad O(n^2) worst case performance.
8. (**Merge Sort**): Divide data into two equally sized halves. Sort each half. Merge to two halves. Good performance: worst case is O(n log n). However, needs clever programming and extra storage to handle the merge. For small amounts of data, this tends to be slower than other algorithms because it copies data into spare storage and then merges it back where it belongs.
9. (**Heap Sort**): Uses Robert Floyd's clever data structure called a heap. A heap is a binary tree structure (each node has two children) that has the big items on top of the small ones (parents > children), and is always balanced (all branches have nearly equal lengths). It is stored without pointers. Instead the data is in an array and node *a* is the parent of *2*a* and *2*a+1*. Floyd worked out a clever way to insert new data in the array so that it remains a heap in O(log n). He also found a way to remove the biggest items and *heapify* the rest in O(log n). First insert all *n* items in a heap (O(n log n)) and then extract them (O(n log n)), top down, we get a worst and average case O(n log n) algorithm. However, on random data, Quick sort will often be faster.
10. (**Radix Sort**): The data needs to be expressed as a decimal number or character string. First sort with respect to the first digit/character. Then sort each part by the second digit/character. This is a neat algorithm for short keys. But because size of key is O(log n) the time is O(n * log n).

11. Combinations: We often combine different algorithms to suit a particular circumstance. For example, when I have to manually sort 20 or more pieces of work (2 or more times a week...), I don't have room to handle the recursion needed for Merge or Quick sort, or table space for a heap. So I take each 10 pieces of work and sort using insertion sort, and then merge the resulting sorted stacks. But sometimes I use a manual sort based on techniques developed for sorting magnetic tape data, and now obsolete. Here you take the items and place them into sorted runs by adding them on the top or bottom of sorted piles... and then merge the result. You might call this **Deque Sort** because I use a Double-Ended Queue to hold the runs. This is just a curiosity and not a famous piece of computer science.

. . . . . (end of section **What are the important algorithms of Computer Science**) <<Contents | End>>

Review

Go back to the start of this document and look at the list of questions ... try to write down, from memory, a short, personal, answer to each one?

Review Question

1. Define what an algorithm is.
2. What is a searching algorithm?
3. Name two algorithms often used to search for things. If you have a choice, which is the faster of your two searching algorithms.
4. What is a sorting algorithm?
5. Name four algorithms often used for sorting. If you have a large number of random items which of these sorting algorithm is likely to be fastest?
6. Find the Big_O of *2000+300*n+2*n^2*.
7. What is the Big_O worst time with *n* items for a linear search, binary search, bubble sort, and merge sort.
8. Name a sorting algorithm that has a good average behaviour on randome data but a slow behavior on some data.
9. **TBA**

. . . . . (end of section **Review Questions**) <<Contents | End>>

. . . . . (end of section **Algorithms**) <<Contents | End>>

Acknowledgments

Dr. Botting wants to acknowledge the excellent help and advice given by Dr. Zemoudeh on this document. Whatever errors remain are Dr. Botting's mistakes.

Glossary

1. **accessor**::=`A **Function** that accesses information in an **object** with out changing the object in any visible way". In C++ this is called a "const function". In the **UML** it is called a *query*.
2. **Algorithm**::=*A precise description of a series of steps to attain a goal*, [**Algorithm**] (Wikipedia).
3. **class**::="A description of a set of similar objects that have similar data plus the functions needed to manipulate the data".
4. **constructor**::="A **Function** in a **class** that creates new **objects** in the class".
5. **Data_Structure**::=*A small data base*.
6. **destructor**::=`A **Function** that is called when an object is destroyed".
7. **Function**::programming=*A selfcontained and named piece of program that knows how to do something*.

8. **Gnu**::="Gnu's Not Unix", a long running open source project that supplies a very popular and free C++ compiler.
9. **mutator**::="A **Function** that changes an **object**".
10. **object**::="A little bit of knowledge -- some data and some know how". An object is instance of a class.
11. **objects**::=*plural of **object***.
12. **OOP**::="Object-Oriented Programming", Current paradigm for programming.
13. **Semantics**::=*Rules determining the meaning of correct statements in a language*.
14. **SP**::="Structured Programming", a previous paradigm for programming.
15. **STL**::="The standard C++ library of classes and functions" -- also called the "Standard Template Library" because many of the classes and functions will work with any kind of data.
16. **Syntax**::=*The rules determining the correctness and structure of statements in a language*, grammar.
17. **Q**::software="A program I wrote to make software easier to develop",
18. **TBA**::="To Be Announced", something I should do.
19. **TBD**::="To Be Done", something you have to do.
20. **UML**::="Unified Modeling Language".
21. **void**::C++Keyword="Indicates a function that has no return".

End