Algorithms Background Notes

1 Vocabulary

"Always use the proper name for things. Fear of a name increases fear of a thing itself."
-Dumbledore

1.1 ACM Scoring Rules

The rules you need to think about are:

1. Given a two teams with the same number of problems solved, the lowest score wins
2. You only get points when you successfully complete a problem within the time limit.
3. Every incorrect submission adds a 20 point penalty when you finally get that problem right
4. When you answer a problem right you get the number of minutes since the contest started plus any accumulated penalty points for that problem added to your score.

There scoring rules mean that participants should have *two* concerns when answering questions. Can they get the largest number of questions right, and can they do it faster than the other teams. Now assuming that a question will not take any less time if you were to do it at the start, that leads to a sensible strategy: do problems in the order of how fast you can solve them. You do the fastest question first, and move on from there. This will minimise your score (given that the time taken to solve a given problem for a given team is does not depend at when you start it) and at the same time make you feel better because you already have some right answers locked in.

1.2 Good Practice vs Speed

Going through your various courses people will have been encouraging you to structure you code in a readable way, to encapsulate functionality etc. In terms of writing reliable and maintainable software that is definitely a good idea. The ACM contest is a slightly different kettle of fish.

First the ACM is a race. You want to get the questions right, and you also want to do it quickly. This itself reduces your ability to think about how to structure your code. But this isn't really the crux of the problem. The issue comes in terms of performance penalties. Maximising performance is not the same as ideally structuring your code. The more abstract your code is, the slower it will go. The more the program needs to jump around in memory the more time you lose. One thing you will notice about algorithms is that inside themselves they will rarely separate things out. Everything will be global and visible to everything else.

This is not a lesson in good practice, or how to structure code, but it is also not trying to encourage you to write poorly structured code either. This is really just a warning to say that good practice and fast code are often at odds, and for this contest we are usually looking at how to make things go fast.

1.3 Numbers

"By all means continue destroying my possessions. I daresay I have too many."
-Dumbledore

We encounter numbers all the time, but problems will often be specifically stated to require a certain set of numbers. There is also a distinction between numbers as mathematicians and people see them compared to how they are in a computer.

1.3.1 Mathematical Numbers

There are really four main kinds of numbers that people concern themselves with:

Integers ($\mathbb{Z}$): These are numbers that have no decimal part either positive or negative (. . . ,-2,-1,0,1,2,. . .). You can specify the positive integers as $\mathbb{Z}^+$, and similarly negative integers as $\mathbb{Z}^-$.

Reals ($\mathbb{R}$): These are all the possible numbers with decimals. The decimal can go on forever or not. Numbers like $\sqrt{2}$, π or just 5.98.

Rationals ($\mathbb{Q}$): $\frac{a}{b}$ where $a, b \in \mathbb{Z}$, i.e., fractions. It is important to note that both numbers that make up the fraction must be integers. Not all possible real numbers can be represented as rationals, but a lot of the time rationals are sufficient. Each rational number has many (infinitely many) different ways of being written e.g., $\frac{1}{2}, \frac{2}{4}, \ldots$. When we are using rational numbers we normally want the two numbers chosen to represent it to be the smallest numbers possible. How to actually do this will be looked at later.

Complex / Imaginary ($\mathbb{C}$): Numbers of the form $a + bi$ where $a, b \in \mathbb{R}$ (i.e. a and b are real numbers) and i is just the letter. These numbers are there for the case where you need to take the square root of a negative number. As there is no number that can actually represent the square root of a negative, we use the symbol i to represent $\sqrt{-1}$. Then we just carry on with maths as normal with the i floating about until (ideally) we can get rid of it and get a proper number once more. That said, even though complex numbers don't seem to make actual sense, they are still very important in making physics work.

1.3.2 Mathematical Symbols

In ($\in$): The mathematical symbol that means in. It is used to specify where things come from, particularly for ranges. There are three main ways of specifying ranges of numbers. The first is simply saying in words what the range is e.g., n is an integer and ranges from 0 to 150. The second way is by setting limits around a variable e.g., $0 \leq n \leq 150, n \in \mathbb{Z}$. And the third way is as a subset from a larger set e.g., $x \in [4, 10], x \in \mathbb{Z}$ which means integers from 4 to 10. The $\in$ symbol means *in*. If you are doing ranges over real numbers then the sort of brackets you use matter. Round brackets '(' mean non inclusive, while square brackets '[' mean inclusive. So $(4, 10] \in \mathbb{R}$, means real numbers bigger than 4 and up to and including 10.

Sum ($\sum$): This is the mathematical short hand for a loop to do addition. The number above the symbol shows the upper limit, the number below shows the starting value and also usually specifies what variable that number represents, and the expression after specifies what to do each time. This is very similar to a for loop in a computer program. Consider the expression $\sum_{i=0}^{50} 10 * i$. This is a loop that goes from $i = 0$ to $i = 50$, each time adding $10 * i$, resulting in $10 * 0 + 10 * 1 + 10 * 2 + \ldots + 10 * 50$.

Product ($\prod$): Mathematical shorthand for a loop doing multiplication. Essentially the same as $\sum$ above, but instead of adding everything you multiply. As before the number above shows the upper limit, the number below shows the variable being modified and the lower limit, and the expression after shows what you do each time round the loop. This lets us write say factorial in the following way $\text{factorial}(n) := \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot n$.

Modulo (mod): This is harking back to division as it was taught in primary school, back before you learnt about decimal numbers and when doing 7/5 would have looked strange. The idea was called a remainder. The amount that is left over after you finish doing the easy bit of the division. So in the example above 5 goes into 7 once, but then there is still 2 left over. As we haven't learnt about decimals yet, we just say that 2 is the remainder. The modulo function gives the remainder after a division. so 7 mod 5 is 2. There are several important things about the modulo function. The result is never less than zero and always less than the number you are diving by. This should actually already be very familiar as this is essentially how we count time. Hours above 12 simply wrap around to give a sensible time. If you have $a \bmod b = 0$ then that means that a/b gives an integer, i.e. it goes in without a decimal part. This will be useful to us in later sections.

Floor and ceil ($\lfloor a \rfloor$ and $\lceil a \rceil$): These are really just fancy names for round up and round down. Floor is round down (towards negative infinity). So floor(3.56) = $\lfloor 3.56 \rfloor = 3$, while floor(-3.45) = $\lfloor -3.45 \rfloor$ = -4. Ceil is round up. So ceil(3.56) = $\lceil 3.56 \rceil = 4$. These are useful as a shorthand. It is also important to remember that integer division (on a computer) gives the floor for positive numbers as it truncates the answer to get rid of the decimal.

1.3.3 Programming Equivalents

Here we compare the mathematical definitions of things to what equivalents there are in programming. For simplicity we only look at what you have in Java though the ideas are the same for C/C++.

Integers (`int`): These are signed 32 bit decimal free numbers that range from -2,147,483,648 to 2,147,483,647. Division done using them results in the result being truncated to be an int instead of a decimal number. Trying to use `int`s to represent anything outside of that range of numbers will not work correctly.

Longs (`long`): These are signed 64 bit decimal free numbers over a larger range than `int`s. They range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. As with `int`s storing anything outside that range will not work sensibly. Division will truncate the result.

BigInteger: This class is supposed to represent a general mathematical integer. It will store any integer value provided that you have enough memory to fit it. It is as a result rather memory intensive and also not as fast as using a `int` or `long`. However, in some problems the numbers you need to store are so large that there is no other practical option.

Floats (`float`): These are signed 32 bit representations of numbers that have decimal numbers. They can represent a vast range of numbers but have limited accuracy. This can lead to many problems. The first is that they are not exact. So doing what should be a chain of operations that should give the starting value can end up with something else. They also have problems with scale. Adding a small number to a big number will not change the large number. As such these are very prone to compounding errors.

Doubles (`double`): These are signed 64 bit representations of number that have decimals. They suffer from all the same problems as floats do, but they have slightly better accuracy. They are still not suitable for computations that need high accuracy.

BigDecimal: These support numbers with a decimal part of any length, similar to BigIntegers. They will work as long as you have sufficient memory to fit the number. However, as they store the entire decimal expansion they can take up a very large amount of space, and cannot exactly represent any number that has an infinite expansion.

Modulo (%): The modulo operator in Java is very similar to mod as it is defined mathematically. However, it is capable of returning a negative result. Converting from the negative to the expected positive is simply a matter of $n + result$. As the result is negative and less than n, this will give a number in the correct range (which will in fact be the right number). So we have the following algorithm to get the mathematical mod.

Algorithm 1 Mathematical mod function

Require: $a, b \in \mathbb{Z}$ and $b > 0$
Ensure: $c \in \mathbb{Z}^+$ and $a \bmod b = c$
  if a$\leq$0 **then**
    return a % b
  else
    return b + (a % b)
  end if

Rationals: There is no default rational implementation in Java. However, for problems where you need high accuracy and only need the basic arithmetic operations implementing your own rational class is the way to go. It is also important to keep the rational always represented in its simplest form (having cancelled out the common factors). This can be done by dividing both the top and bottom numbers by their *greatest common divisor*, the largest number that they can both be divided by without any remainer. Thankfully there is an algorithm for this called Euclid's algorithm which we can use to implement a `Rational` class.

Algorithm 2 Greatest Common Divisor - Euclid's Algorithm

Require: $a, b \in \mathbb{Z}$
Ensure: $c \in \mathbb{Z}$ and $gcd(a, b) = c$
  if b = 0 **then**
    return a
  else
    return gcd(b,a mod b)
  end if

2 Data Structures and Useful Java Classes

"I'll be more enthusiastic about encouraging thinking outside the box when there's evidence of any thinking going on inside it."
-Terry Pratchett

In general writing your own datastructures when ones that fit what you need already exist is a waste of time. Java thankfully has a large number of predefefined collection types for your use. Obviously there are not enough there for all possible situations, but by and large it is best to use one of the existing classes, but equally important to use the correct one.

2.1 Collections Library

Lists: Most people end up using `ArrayList` all the time as the default list that they use. That's not to say it's a bad idea, `ArrayList`s are great, but they are not always what you want to be using. They have fast insert and remove from the end, and access to any part of the list. On the other hand if you want to add to the front of the list or remove from the middle, they have to move all the other elements to make space / make everything fit again. A `LinkedList` has fast access to both the start and ending, allowing for fast addition and removal. On the other hand it has slow access to the middle as it needs to actually follow the links to get there. `Vector`s are essentially just synchronized `ArrayList`s.

2.2 Helper Classes

In general it isn't necesary to write most of the common list or collection modification methods. This is especially pertinent to things like sorting methods, which while commonly needed are already implemented. It is generally better to just use the existing one unless you have a special reason. There are two main helper classes in Java: `Arrays` and `Collections`. These have a large number of helpful methods on them including things such as sort, binarySearch, disjoint, etc. It is highly recomended that the teams using Java look at these classes and see how they can best be used.

2.3 Strings

`String`s are really commonly used and it is probably fair to say that they are really important. However, they also have some properties that need to be carefully thought about when they are being used.

`String`s are immutable. This means that you cannot change the value stored in a Java `String`. If you try, you will simply be given a different one that has the right value. This is much like number types, and isn't actually a bad thing. However, it can be slow. Consider for example concatenating a large number of strings in something like the following which strings together a whole lot of numbers:

```
String s = "0"
for all i in 1 ... 100 do
  s += ","+ i
end for
```

Well, that will result in a new `String`s being created each time round the loop. To get around this, Java has `StringBuilder`s. These are like mutable `String`s. They allow you to quickly modify their contents without the penalty of creating a new object each time. Using a `StringBuilder` the code above would look like:

```
StringBuilder st = new StringBuilder();
st.append('0');
for all i in 1 ... 100 do
  st.append(',');
  st.append(i);
end for
```

String s = st.toString();

While the second method is a lot more verbose, it also takes substantially less time, especially on problem sets that are as large as the ACM ones tend to be. It is important to realise that even though we have been talking about Java `String`s specifically in this section, this is something that is worth thinking about even with other types and in other languages. Despite this, using immutable objects is often necessary and unavoidable as we will look at in the next section.

2.4 Cloning and Forcing Immutability

Now that we have finished discussing why making new objects all the time can be slow, let's look at situations where not doing so will cause problems. Consider a brute force shortest path algorithm.

Algorithm 3 Brute Force Shortest Path

```
Require: target, node, visited
  visited.add(node)
  if node == target then
    return visited
  end if
  bestlist = null
  for all n ∈ node.children do
    if n ∉ visited then
      list = shortestPath(target,n,visited)
      if list ≠ null and cost(list) < cost(bestList) then
        bestlist = list
      end if
    end if
  end for
  return bestlist
```

This is where a lot of problems occur. If you were to simply pass `visited` along then any modifications made would be visible even after the recursive calls return. So you need to make a copy of `visited` to pass down. But wait— what if `visited` has fields which also would be modified? Well, then you would need to copy those as well when you are copying the higher-up object. On the other hand, it may not be necessary to clone the fields of the object being copied. In that case doing the extra copies would be a waste of time. The main point here is that you should always think about whether objects need to be copied, and how much copying you are actually doing. If you don't copy when you should then your program won't work and it will be hard to work out why.

2.5 Graphs

Graphs are used a lot and here we want to look at how to represent them. But first we need a definition of what they are. Usually we say that a graph is composed of two sets V and E. V is the set of vertices. All this means is that there is some set of things and we are calling them vertices. Just to be confusing we also often call them nodes. In this set of notes we will probably be inconsistent with our notation. They don't need any special properties, though often it is nice to represent them with Integers (the mathematical sort). The set E is called the set of edges, and it consists of pairs of vertices $(a, b) \in E, a, b \in V$. Each of these pairs denotes a connection between the two vertices in question. Edges can also have a weight, or cost associated with them. But this is only the case for some graphs.

If you impose a certain structure on a graph then there are cool things you can do – like make a heap. However, one has to consider the problem of how to actually represent the graph in the program. In this next section we will look at different graph representations and their pros / cons. We are limiting ourselves

to only looking at the more common representations that you are likely to encounter, and not trying to cover everything.

Heap This is a technique for storing trees when in general all the nodes have the same number of children and the tree is balanced. A tree is a graph such that there is exactly one path from every vertex to every other vertex. While this is not the most common way of storing graphs - in part due to only really working for trees - it should still be familiar to most people.

You store all the vertices of the tree in a single one dimensional array. Since it is a tree, any vertex can be labelled the *root* vertex. We then consider every vertex connected to it to be its child. These vertices then act as roots for all the vertices connected to them, and so on recursively. All trees can be thought of in this way, and drawn similar to the one below.

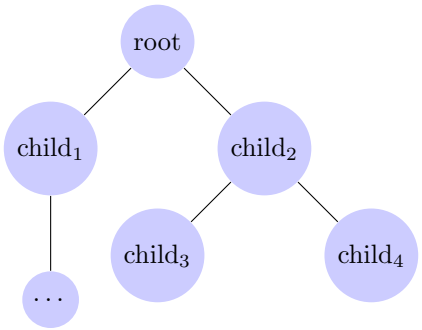


Figure 1: Drawing of a tree

If all the vertices have a fixed number of children (or no more than) called n, then you can effectively do the following to represent the tree as a heap. First you say that the root of the tree is at index 0. The n children of the root are then placed in the n following indices. At the end of these the n^2 children of those nodes are placed all in a row, in the same order as their parents. Then you repeat this recursively. Given a number of levels (d) of a tree with n children per node it is always possible to make an array the size of the tree based on the following pattern.

depth	**Num Nodes**
0	1
1	n
2	$n * n$
3	$n^2 * n$
	...
$d-1$	n^{d-1}

Table 1: Number of nodes at each level of a tree with n children per node

We can then say that the total size of the tree is :

$$NodesInTree(n, d) = \sum_{i=0}^{d-1} n^i \tag{1}$$

You can then extend this to calculate the index of each child of each node which will allow you to traverse the tree, by knowing which index you need to jump to get each node's children. We will not develop equations to find everything else in these notes, as heaps are primarily used for priority queues and Java already has this implemented in the `PriorityQueue` class. Also, given the amount we have done, you should be able to see how you could go about working out the rest of the equations you would want.

Nodes and Adjacency Lists It is also possible to represent a graph as a set of nodes that have connections between them. Here each node has its own list of neighbors. A simple `Node` class might look as follows:

```
public class Node{
  List<Node> Neighbors;
  Object Value;
}
```

But this is not quite sufficient. In many cases, such as in path finding each edge has an associated weight. We then have three ways we can represent this.

```
public class Node{
 List<Pair<Node,Cost>>
     edges;
 Object value;
}
```

```
public class Node{
 Map<Node,Cost> edges;
 Object value;
}
```

```
public class Node{
 List<Node> edges;
 List<Cost> weights;
 Object value;
}
```

Each of these provides a different way of thinking about the graph. In each case, this sort of representation makes it easy to get from the current node to all of its neighbors while being aware of their weights. Of all the representations, this one makes it the easiest to find the direct neighbors for a given node. On the other hand, it makes it hard to find all the edges that specify some given condition.

Edge List A rather different way to store a graph is as a set or list of edges. Here each edge is stored with its weight, in a way that makes it very easy to look at all the edge weights, count edges, or do any other operation that only considers edges. On the other hand, finding all of the edges that connect to a given node is quite slow.

Adjacency Matrix Adjacency matrices are one of the particularly cool ways of representing finite graphs. They are very simple, neat, and for dense graphs compact. They work as follows. Suppose you have a square matrix M with n rows and columns. Suppose you also have a function (bijection) $tonum :: \text{Node} \to [0, n] \in \mathbb{Z}$, which gives each node a unique number in the range from 0 to n. Then we can say that the entry at (i, j) in the matrix M, is the cost of the edge from the node represented by i to j. Consider the following matrix and the graph that it represents.

$$\begin{bmatrix} 1 & 8 & 0 \\ 8 & - & 2 \\ 0 & 2 & - \end{bmatrix}$$

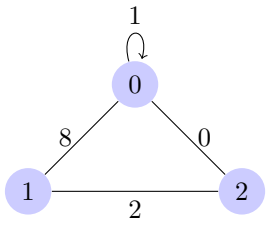


Figure 2: Adjacency matrix and drawn graph

There is, however, still one outstanding issue. In the matrix I used '-' to represent an absent vertex. In a computer program, when you are representing your matrix with something such as an array of `int`s or `double`s, you can't have a value like '-'. In this case you need to decide, in a context specific way, how you will represent absent vertices. Thankfully, in most cases the ACM questions will specify what range of numbers are allowable costs for edge weights, and you can simply pick a fixed invalid value to represent "nothing there".

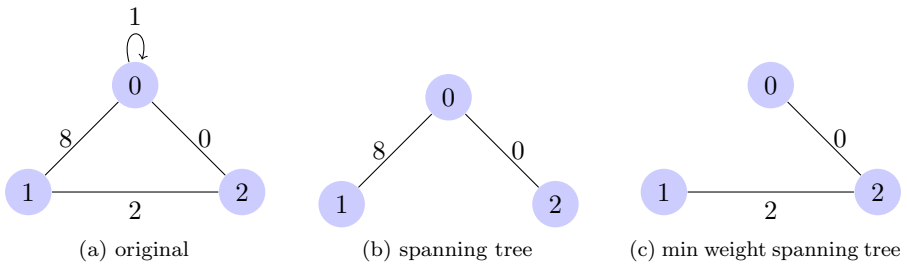
There are also some possible optimisations in terms of saving space. In the case that the graph is undirected then the graph will be symmetrical as the weight from node i to j will be the same as from j to i.

This means that you don't actually need to store the redundant weights, though in practice you probably will not have graphs so large that that becomes a problem.

2.5.1 When Different Representations are Good

We have been looking at a number of different representations of graphs. And it seems like since they are all fundamentally representing the same thing, then you only really need to pick one representation and then run with that. Unfortunately, the different representations have their uses. Depending on what algorithm you are writing, different represenations make sense.

Consider for now the problem of finding a minimum weight spanning tree. The first step here is to ask what is a spanning tree? The answer is that a spanning tree is a subgraph of a graph (a graph which contains only edges and vertices that were in the original graph, but not necessarily all of them) which contains all of the vertices in the original and is also a tree. The minimum weight means that the sum of the weights of all the edges make the smallest possible number compared with all the other possible spanning trees. When thinking about this problem it is important to remember that there may be more than one possible minimum weight spanning tree. And in the case of a general spanning tree, then there are likely to be very many possible valid alternatives. An example is shown in the figure below.



(a) original (b) spanning tree (c) min weight spanning tree

Figure 3: A graph and some of its spanning trees

We shall use the problem of finding a minimum spanning tree to look at how graph representation is useful in implemented real algorithms. For completeness we will look at the three most common minimum weight spanning tree algorithms, though with a note that one of them, while conceptually easy, is actually quite hard to make fast.

The first algorithm is called Kruskal's Algorithm. The idea of this algorithm is that you start with an empty graph. Then you pick the cheapest edge that doesn't make the graph contain a cycle until you are finished. You now have a minimum weight spanning tree.

Algorithm 4 Kruskal's Algorithm

Require: E a list of edges with weights
Ensure: $E' \subseteq E$ such that V, E' is a minimum weight spanning tree
  $E' = \emptyset$
  $E_{sorted} = \text{sort}(E)$ from cheapest to most expensive
  for all e **in** E_{sorted} **do**
    if isATree($E' \bigcup$ e) **then**
      $E' = E' \bigcup$ e
    end if
  end for
  return E'

Note how in the above algorithm, an edge list is the sensible way to store the edges of the graph to make it easy to iterate over them. However, we still have one issue. How do we check `isATree`? We could do a full

depth first search of the tree each time, but that would be slow and inefficient. So we need a better way. Thankfully we can use a union-find datastructure for E' to give fast performance.

Algorithm 5 Prim's Algorithm

Algorithm 6 Reverse Delete

Algorithm 7 Reverse all edges

Require: M an adjacency matrix
Ensure: M' an adjacency matrix where the edges are backwards from M
  return M^T

Algorithm 8 Topological Sort

3 Common Types of Algorithms

- Greedy - Interval scheduling
- Dynamic Programming
- Divide and Conquer
- Search
  - Depth first search
  - Breadth first search
  - Best first search
- Probabilistic
- Approximation

4 Graph Theory

- Flows - Ford-Fulkerson
- Cuts - Max flow => min cut
- Matchings
  - Edmond's matching algorithm - M alternating paths
  - Stable Marriage problem - Gale-Shapley
  - Vertex cover
- Minimum cost assignment - Hungarian algorithm
- Spanning Trees

  - Kruskals
  - Prims
  - Union find data structure
- path finding and search
  - Dijkstra's Algorithm
  - Bellman-Ford
  - Floyd-Warshall
  - A*
  - Heuristics
    * Admissible
    * Euclidean Distance
    * Taxi cab metric
- Planarity
  - 4 color theorem
  - 5 color theorem - Kempe chains
  - Euler's theorem
- Graph coloring
- connectedness / strongly connected components / Bridges
- cycles / trees
- sinks
- topological sort
- Hamiltonian Path / cycle (order 2 approximation)
- Eulerian

5 Linear Algebra

- Matrices
- Gauss-Jordan Elimination
- Vectors and matrices as shapes
- intersection between point / line and line
- point in polygon test (extend ray to infinity, ignore parallel edges)
- Area of a polygon formula

6 Numbers

- Prime Numbers
  - Wilson's Theorem (not so practical, but cool) $(n-1)! = -1 (\text{mod } n) \iff n$ is prime
  - trial division (only need to go up to $\sqrt{n}$, test 2,3 then numbers of the form $6k \pm 1,\ k = 1, 2, \ldots$)
  - BigInteger - isProbablePrime(), nextProbablePrime(int certainty)
  - Sieve of Eratosthenes
- Basic Codes and ciphers
- combinatorics
  - Inclusion - exclusion
  - Balls and boxes with combinations of labeling
  - generating functions
  - twelvefold way

7 Omitted due to not quite fitting in logically

- Relations - transitive, symmetric, anti-symmetric, transitive closure
- minimax