Applications of Binary Search

The basic idea of a binary search can be used in many different places. In particular, any time you are searching for an answer in a search space that is somehow “sorted”, you can simply set a low bound for the value you’re looking for, and a high bound, and through comparisons in the middle, successively reset either your low or high bound, narrowing your search space by a factor of 2 for each comparison. This is especially useful in situations where you can calculate an increasing function forwards easily, but have difficulty calculating its inverse directly. Since the function is increasing, guessing allows you to narrow down your search range for the possible answer by half. In essence, after each guess, you know which direction to “go.”

Binary searches can either be in a continuous space (answer is a real number) or a discrete space (answer is an integer). While the high level concepts are the same for both search domains, there are some specific issues that come up only for continuous searches and a different set of issues that come up for discrete searches. Thus, each problem is labeled with which type of search it is.

The text to each of the problems in this lecture can be found at the end of this document.

**Problem #1: Crystal Etching (Continuous)**

Consider the problem of calculating how many seconds a crystal should be “etched” until it arrives at a given frequency. (This is actually a real problem I worked on at a summer job…)

In particular, the crystals start at an initial frequency, let’s call this $f_1$ and they must be placed in an etch bath until they arrive at a target frequency, $f_2$. Both of these values, $f_1$ and $f_2$ are known.

Furthermore, you are given constants $a$, $b$ and $c$ that can be used to calculate the relationship between $f_1$ and $f_2$. The formula is as follows:

\[
\frac{f_2 - f_1}{f_1 f_2} = at + b(1 - e^{-ct})
\]

The only unknown in this formula is $t$, the number of seconds for which the crystal must be etched.

The difficulty with this problem is solving the equation for $t$. No matter what you try, it’s difficult to only get one copy of $t$ in the equation, since $t$ appears in both an exponent and a linear term.

But, a quick analysis of this specific function, along with a bit of common sense, indicates that as $t$ rises, the value on the right-hand side of the equation also rises. In particular, since the constants $a$, $b$ and $c$ are always positive, that function on the right is a strictly increasing function in terms of $t$. 
This means that if we make a guess as to what t is and plug that guess into the right-hand side, we can compare that to what we want for our answer on the left, and correctly gauge whether or not our guess for t was too small, OR too big.

This is a perfect situation for the application of binary search, so long as we can guarantee an upper bound. Luckily, in the practical setting of this problem, I knew that no crystal would ever be etched for more than 10000 seconds. (This was WAAAY over any of the actual times and a very safe number of use as a high bound.) I also knew that each crystal had to be etched for at least one second. (Actually, if we ignore the second term on the right, we can very easily get a nice upper bound as well.)

From there, we successively try the middle point between high and low, resetting either high (if our guess was too high) or low (if our guess was too low).

**Example #2: A Careful Approach (Continuous)**

This problem is taken from the 2009 World Finals of the ACM International Collegiate Programming Contest that was held in Stockholm, Sweden.

The essence of the problem is that you are given anywhere from 2 to 8 planes that have to land. Each plane has a valid “window” within which it can land. The goal is to schedule the planes in such a way that the gap between all planes’ landing times is maximized.

For example, if Plane1 has a window from t = 0 to t = 10, Plane2 has a window from t = 5 to t = 15 and Plane3 has a window from t = 10 to t = 15, then Plane1 could land at t=0, Plane2 could land at t = 7.5 and Plane3 could land at t = 15. If Plane2 moves its time any earlier, then the gap between Plane1 and Plane2 gets below 7.5 and if it moves its time later, then the gap between Plane2 and Plane3 goes below 7.5. Thus, 7.5 is the largest gap we can guaranteed between each of the planes.

Two Problem Simplifications

First, let’s just assume we knew which order the planes were going to land.

A second simplification will help us as well:

Rather than write a function that returns to us the maximum gap between plane landings, why don’t we write a function that is given an ordering of the planes AND a gap value and simply returns true or false depending on whether that gap is achievable or not.

Here’s how to do it:

1) Make the first plane land as early as possible.
2) Make the subsequent plane land exactly gap minutes later (if this time is within its range), if it is not, then make it land after that time, as soon as possible. If this can’t be done, then the arrangement is impossible. If it can, then continue landing planes.
3) Repeat step two if there’s another plane to land.

This is what is known as a greedy algorithm. If a method exists to land all the planes with the given gap, then this method will work, since we land each plane as EARLY as possible given the constraints. Any alternate schedule gives less freedom to subsequent landing planes.

**Solving the Original Problem**

Now, the question is, HOW can we solve the original problem, if we only know how to solve this easier version.

We can deal with simplification number one by simply

**TRYING ALL ORDERINGS OF THE PLANES LANDING!!!**

Now, if we have a function that returns true if a gap can be achieved and false otherwise, can’t we just call that function over and over again with different gaps, until we solve for the gap within the nearest second? (This is what the actual question specified. Furthermore, the numbers in the input represented minutes, thus, 7.5 should be expressed as 7:30, for 7 minutes and 30 seconds.)

Thus, once again, we have the binary search idea!!!

Set our low gap to 0, and our high gap to something safe, and keep on narrowing down the low and high bounds on the maximum gap until they are so close we have the correct answer to the nearest second!

**Problem #3: Bones (Binary Search with APSP, All Pairs Shortest Paths) (Discrete)**

So the problems is, given an undirected weighted graph, find some number R such that if you travel from one node to another the distance is less than or equal to R and you may reset,charge, R only K-1 times and only at nodes. One charge is used when you leave the node. You are given K, you need to find R. N denotes how many nodes and M denotes how many edges and d denotes the distance of the edge.

\[2 \leq N \leq 100 \text{ and } 1 \leq K \leq 100 \text{ and } 1 \leq d \leq 10^9\]

Example:

Let's say we have a graph with 4 nodes,4 edges, and K = 2.

\[0\leftrightarrow1, \ d = 100\]
\[1\leftrightarrow2, \ d = 200\]
\[2\leftrightarrow3, \ d = 300\]
\[3\leftrightarrow0, \ d = 400\]

So we have a graph that goes in a circle.
We find that the min distance traveled from nodes
0↔1= 100
0↔2 = 200
0↔3= 300

If we leave from node 0 we need to use a charge so we only have 1 more charge left and we can only use a charge at a node.
If R = 300 we can go from node 0 to 1,2 without having to use another charge and then at node 2 we can use a charge to go to 2↔3 since that only cost 300 therefore 0 ↔ 3 in 2 charges.

If R = 200, we would only be able to go to node 0↔1 before we had to use a charge to go to node 2. But then we don’t have enough gas or charges to go to node 3 from 2.

Since the graph can be at most 100 nodes if we try to DFS from each node as a starting point
We get (V*E)*V. If we try all starting positions V^2E and try all numbers from 1-10^9 as R until we find an answer that works we would be waiting for a very long time.

So the first problem we notice is that we have no idea what R could possibly be and it is too large to brute force. Let’s assume that we can come up with a fancy algorithm that finds for all pairs Node i and Node j how many charges it takes to get there given R. What is interesting about the property of R is that if let’s say for some example we knew the minimum value of R such that our fancy algorithm returns true this R is possible. Then we also know that R+1 is also valid because if we can reach all pairs with just R max gas we can obviously do the same with R+1 max gas. This is an important because if R was minimum we know that R-1 is impossible otherwise it would have been our minimum. Therefore R-1 and so on will always be impossible. Because of this duality we can actually run a binary search on R to find the minimum R that is valid, assuming we have a function that takes R and outputs whether this is valid or not. That would be the next subproblem.

So what you notice is that since we need to know for all pairs we need to know if we can go to another node in less than k charges. So we also need to know the min distance from node i -> j.
Luckily we have a fancy algorithm called Floyd Warshall's Algorithm that runs in O(V^3) time where V is the number of vertices there are and finds for us the shortest path for all pairs of nodes in the graph. As you can see V is only 100 so this will be fine. Using this idea we can then make another graph of similar nature but instead of using weights we use the min amount of charges it takes to go from node i ↔ node j.

So all we have to do is binary search R and run Floyd Warshall's Algorithm on a graph that has either a 0 for node i ↔ node i since it takes 0 charges to get to itself, a 1 if from node i ↔ j can we get there in less then equal to R, or K+1 charges otherwise to represent our infinity. We run floyd on this new graph and then search for every pair if it can get there in less than or equal to k charges.

If so return true, else return false.
**Problem #4: Carpet (Binary search with geometry) (Continuous)**

The problem is you are given 3 lengths $a$, $b$, and $c$. They represent the distances from the three points of an equilateral triangle. The goal is to find the length of the side of the equilateral triangle.

So assuming we had a function that tells us if given these 3 variables and some $L$ can we form an equilateral triangle with length $L$ that satisfies $a$, $b$, and $c$. Then we can just binary search on $L$ and if $L$ is too small then search higher else if $L$ is too big search lower. The problem is how do we know if $L$ is too small or too big?

We know that law of cosines state that $c^2 = a^2 + b^2 - 2ab\cos(C)$ where $a$, $b$, $c$ are legs of the triangle, unrelated to the above mentioned $a$, $b$, and $c$, and $C$ is the angle formed at the point opposite of the leg $c$.

So to determine if a current leg length is too big or too small we can determine this by summing up the inner angle formed by the legs using the law of cosine. If the angle sum is less than 360 degrees then we need longer legs to increase the angle sum, if the angle sum is more than 360 we need smaller legs.

There is a small problem we need to consider when using acos in java. It returns a range of 0 to PI. So we have to check if the length of the 3 legs qualify as being a triangle.
Taking Care with Coding Binary Search

Last but not least, I’ve found that almost all errors in binary search code happen in one of four places. Here is a cheat sheet I made for where you should really pay attention when coding binary search.

**Binary Search Debugging Cheat Sheet**

<table>
<thead>
<tr>
<th>Item</th>
<th>Discrete Bin Search</th>
<th>Continuous Bin Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting low and high</td>
<td>Make sure low &lt;= ans, high &gt;= ans, make sure they are ints/longs and overflows don’t occur, and no array out of bounds issues</td>
<td>Make sure low &lt;= ans, high &gt;= ans, but also make sure that low isn’t too low and high isn’t too high, if they are, the could crash math functions</td>
</tr>
<tr>
<td>Loop</td>
<td>while (low &lt; high)</td>
<td>for (iter=0; iter&lt;100; iter++)</td>
</tr>
<tr>
<td>Assign mid</td>
<td>mid = (low+high)/2 or mid = (low+high+1)/2, plug in low=2, high =3 after code is written to determine which will terminate</td>
<td>mid = (low+high)/2.0</td>
</tr>
<tr>
<td>Updating</td>
<td>After each iteration, either low OR high should be updated, but not both. Make sure you update the correct one!!!</td>
<td>After each iteration, either low OR high should be updated, but not both. Make sure you update the correct one!!!</td>
</tr>
<tr>
<td>What to update to</td>
<td>For low, we have two choices: low = mid or low=mid+1. For high, we have two choices: high = mid or high = mid-1. We will typically pair (low=mid with high =mid-1) or pair (low=mid+1 with high=mid)</td>
<td>Always either low = mid or high = mid</td>
</tr>
</tbody>
</table>
UCF Local Contest — September 2, 2006

Crystal Etching (filename: etch)

When one usually thinks of crystals, they think of chandeliers and other ornamental items. Yet, there are crystals that are components in our every day appliances, such as microwaves. These crystals have to be produced at very specific thicknesses to obtain the desired frequency of oscillation. Part of this process involves creating a generic crystal that is too thick, and then calculating exactly how long that crystal needs to be placed in a chemical etch bath to reduce it to the proper thickness. In particular, if the current frequency of oscillation of a crystal is $f_1$, and its target frequency is $f_2$, then $t$, the number of seconds necessary to etch the crystal to achieve the desired frequency satisfies the following equation:

$$\frac{f_2 - f_1}{f_1 f_2} = at + b(1 - e^{-ct})$$

where $a$, $b$, and $c$ are all dependent upon the actual chemical composition of the etch bath. (Note: The units of frequency are 1/sec.)

**The Problem:**

Given the current frequency of a crystal ($f_1$), its target frequency ($f_2$), and the values of $a$, $b$, and $c$, for a particular etch bath, calculate the amount of time ($t$, in seconds), rounded to the nearest hundredth that is necessary to etch the crystal to obtain its desired frequency.

**The Input:**

The first line of the input file will contain a single positive integer $n$, the number of test cases in the file. Each of the next $n$ input lines will contain information for a single test case. On each line, the following values, all positive real values, will be separated by spaces: $f_1$, $f_2$, $a$, $b$, and $c$. It is guaranteed that $100000 \geq f_2 > f_1 \geq 10$.

**The Output:**

For each input case, print a heading followed by the number of seconds (rounded to the nearest hundredth) necessary to etch the corresponding crystal to achieve its target frequency. Assume that no etch time will exceed 1000000.00 seconds. Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)
Sample Input:

2
500 1000 .001 .001 1
1000 2000 .00025 .0005 6931472

Sample Output:

Crystal #1: 0.57
Crystal #2: 1.00
Problem A
A Careful Approach
Input: approach.in

If you think participating in a programming contest is stressful, imagine being an air traffic controller. With human lives at stake, an air traffic controller has to focus on tasks while working under constantly changing conditions as well as dealing with unforeseen events.

Consider the task of scheduling the airplanes that are landing at an airport. Incoming airplanes report their positions, directions, and speeds, and then the controller has to devise a landing schedule that brings all airplanes safely to the ground. Generally, the more time there is between successive landings, the “safer” a landing schedule is. This extra time gives pilots the opportunity to react to changing weather and other surprises.

Luckily, part of this scheduling task can be automated — this is where you come in. You will be given scenarios of airplane landings. Each airplane has a time window during which it can safely land. You must compute an order for landing all airplanes that respects these time windows. Furthermore, the airplane landings should be stretched out as much as possible so that the minimum time gap between successive landings is as large as possible. For example, if three airplanes land at 10:00am, 10:05am, and 10:15am, then the smallest gap is five minutes, which occurs between the first two airplanes. Not all gaps have to be the same, but the smallest gap should be as large as possible.

Input
The input file contains several test cases consisting of descriptions of landing scenarios. Each test case starts with a line containing a single integer \( n \) (\( 2 \leq n \leq 8 \)), which is the number of airplanes in the scenario. This is followed by \( n \) lines, each containing two integers \( a_i, b_i \), which give the beginning and end of the closed interval \([a_i, b_i]\) during which the \( i \)th plane can land safely. The numbers \( a_i \) and \( b_i \) are specified in minutes and satisfy \( 0 \leq a_i \leq b_i \leq 1440 \).

The input is terminated with a line containing the single integer zero.

Output
For each test case in the input, print its case number (starting with 1) followed by the minimum achievable time gap between successive landings. Print the time split into minutes and seconds, rounded to the closest second. Follow the format of the sample output.

<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Output for the Sample Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 0 10 5 15 10 15 2 0 10 10 20 0</td>
<td>Case 1: 7:30 Case 2: 20:00</td>
</tr>
</tbody>
</table>
**Problem B — limit 5 seconds**

**Bones’s Battery**

Bones is investigating what electric shuttle is appropriate for his mom’s school district vehicle. Each school has a charging station. It is important that a trip from one school to any other be completed with no more than \( K \) rechargings. The car is initially at zero battery and must always be recharged at the start of each trip; this counts as one of the \( K \) rechargings. There is at most one road between each pair of schools, and there is at least one path of roads connecting each pair of schools. Given the layout of these roads and \( K \), compute the necessary range required of the electric shuttle.

**Input**

Input begins with a line with one integer \( T \) (\( 1 \leq T \leq 50 \)) denoting the number of test cases. Each test case begins with a line containing three integers \( N \), \( K \), and \( M \) (\( 2 \leq N \leq 100 \), \( 1 \leq K \leq 100 \)), where \( N \) denotes the number of schools, \( K \) denotes the maximum number of rechargings permitted per trip, and \( M \) denotes the number of roads. Next follow \( M \) lines each with three integers \( u_i \), \( v_i \), and \( d_i \) (\( 0 \leq u_i, v_i < N \), \( u_i \neq v_i \), \( 1 \leq d_i \leq 10^9 \)) indicating that road \( i \) connects schools \( u_i \) and \( v_i \) (0-indexed) bidirectionally with distance \( d_i \).

**Output**

For each test case, output one line containing the minimum range required.

<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
</table>
| 2
| 4 2 4
| 0 1 100
| 1 2 200
| 2 3 300
| 3 0 400
| 10 2 15
| 0 1 113
| 1 2 314
| 2 3 271
| 3 4 141
| 4 0 173
| 5 7 235
| 7 9 979
| 9 6 402
| 6 8 431
| 8 5 462
| 0 5 411
| 1 6 855
| 2 7 921
| 3 8 355
| 4 9 113 | 300
| 688  |
B: Stained Carpet

The Algebraist Carpet Manufacturing (ACM) group likes to produce area carpets based upon various geometric figures. The 2014 ACM carpets are all equilateral triangles. Unfortunately, due to a manufacturing defect, some of the carpets are not as stain-resistant as intended. The ACM group is offering to replace each defective carpet that contains a stain.

The web form used to report the stained carpet requests the three distances that the stain is away from the corners of the rug. Based upon these three numbers, you need to compute the area of the rug that is to be sent to the customer, or indicate that the customer’s carpet doesn’t come from ACM.

Input
Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. Each test case will consist of a single line with three floating point numbers \(a, b\) and \(c\) (\(0 < a, b, c \leq 100\)) representing the distances from the stain to each of the three corners of the carpet. There will be a single space between \(a\) and \(b\), and between \(b\) and \(c\).

Output
Output a single line with a single floating point number. If there is a carpet that satisfies the constraints, output the area of this carpet. If not, output \(-1.000\). Output this number to exactly three decimal places, rounded. Output no spaces.

<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1.732051</td>
<td>1.732</td>
</tr>
<tr>
<td>1 1 3.0</td>
<td>-1.000</td>
</tr>
<tr>
<td>1.732051 1.732051 1.732051</td>
<td>3.897</td>
</tr>
</tbody>
</table>