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.”

Problem #1: Crystal Etching
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**

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 in 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)
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.

\[
\begin{align*}
0 &\leftrightarrow 1, \ d = 100 \\
1 &\leftrightarrow 2, \ d = 200 \\
2 &\leftrightarrow 3, \ d = 300 \\
3 &\leftrightarrow 0, \ d = 400
\end{align*}
\]

So we have a graph that goes in a circle.
We find that the min distance traveled from nodes
\[
\begin{align*}
0 &\leftrightarrow 1= 100 \\
0 &\leftrightarrow 2 = 200 \\
0 &\leftrightarrow 3= 300
\end{align*}
\]

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 $(V^2E)*V$. If we try all starting positions $V^2E$ and try all numbers from 1-10⁹ 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: Reconnaissance (Ternary Search)**

The problem basically is you are given n cars, for each car you are given their starting x position and their velocity and the direction of the velocity. Find the minimum distance that will cover all of the cars at some time.

The first observation you can make is that the min distance is clearly the car on the far left and the far right at the given moment of time. Regardless of how many cars you have.

So if we look at just two cars and look at how the distance is affected based on the velocity of two cars we notice the following.

So assume we only have two cars.

We have three cases with two cars:

Both cars are going in the same direction with the same velocity which means their distance will always stay the same.

One car is on the left going left and the other car is on the right going right. As time progresses their distance also increases. Therefore the minimum answer is at time 0 before they started to move.

The final case is if the cars directions are towards each other. This means for some time the distance gap is going down and eventually reaches 0 and then as they pass each other this case reverts to the second case.

If the distance was graphed as a time/distance graph we will notice that the third case resembles a parabola. Here the answer would be the bottom point of a parabola. Which in the case of 2 cars would just be zero.

But of course the problem is not as simple and there are n cars. What is important though is that regardless of how many cars there are if we were to graph the time/distance it would still resemble a parabola.

Well using ternary search we can find the bottom of this parabola and our answer

lowmid = (low+high)/2
highmid = lowmid + 1e-9

if (lowmid < highmid)  high = highmid;
else  low = lowmid

What can we ternary search on?

We'll we can ternary search the time and then with a simple translation function we will find our solution.
**Problem #5: Carpet (Binary search with geometry)**

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.