# Applications of Binary/Ternary 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).

*Problem #2: Cow Dance Show (USACO January 2017 Silver)*
In this problem, there cows will be dancing on a stage. The first k cows take the stage at the beginning. When one cow finishes dancing, the next takes her place. This continues until all cows have finished dancing. For each of n cows, you are given the amount of time they will dance. You also have a maximum amount of time you want the show to take. The question is: what's the smallest value of k such that the show can get done in time?

In this problem, if we were to know k, then it's easy to figure out how long the show takes (You can simulate the show with a priority queue.) But, we need to solve the inverse problem, given the maximum length of the show, determine the smallest stage we can complete the show on time. The other key fact is that as the stage gets bigger, the time it takes the show to complete gets smaller. Once we nail down these two facts, we see that the value of k is binary searchable.

The solution is as follows:

1) Guess k.
2) Simulate the cow dance process.
3) If it takes too long, then set low = guess + 1.
4) If it the simulation runs in time, then set high = guess.
Stop when the binary search converges on a single integer answer.

*Problem #3: Moo Buzz (2019 December Silver)*
In this problem you start counting positive integers, but if an integer is divisible by 3 or 5, you say "Fizz", or "Buzz". If it's divisible by both you say "FizzBuzz". Given an integer N, where N can go up to $10^9$, you are supposed to determine the Nth **number** you said.

For example, when N = 4, the answer is 7, since previously, you said, 1, 2, Fizz, 4, Buzz, Fizz and 7. (You didn't say 3, 5 or 6 due to the divisibility rules.)

If you know which number you are ending on (say 7), then, using the Inclusion-Exclusion Principle, you can figure out how many numbers were said. However, we are posed the inverse problem, which is more difficult. Notice however, that if N increases, the answer must increase. Thus, once again, we have a binary searchable property and the problem is easier to solve the other way around. Thus, we binary search the answer to the question. We do have to be careful with the low and high bounds, but it should be fairly easy to see that 2N is a sufficient high bound for the search. Also, care must be taken to answer a number that is NOT divisible by 3 or 5, since the binary search could say that the exact correct number of values are "said" up until that point.

## *Problem #4: 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

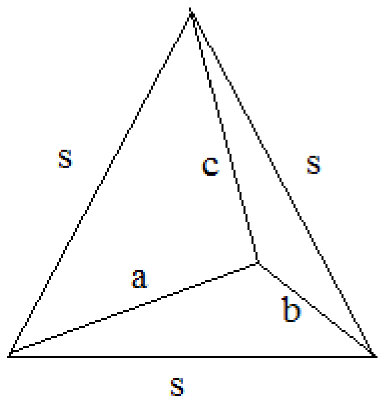
TRYING ALL ORDERINGS OF THE PLANES LANDING!!!

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 #5: Carpet (Binary search with geometry)*
The problem is you are given 3 lengths a, b, and c. They represent the distance from S in any angle or direction. Find the length, s, such that you can form an equilateral triangle and from some point in the triangle the distance from all 3 points of the triangle is a, b, and c. Here is a diagram:



Since we are given a, b and c, one thing we might imagine is seeing if we can check if a given value L is too big or too small for our equilateral triangle.

We know that law of cosines state that c2 = a2 + b2 - 2abCos(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 then 360 degrees then we need longer legs to increase the angle sum, if the angle sum is more then 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.

*Problem #6: 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 = (2*low+high)/3
highmid = (low+2*high)/3

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

What can we ternary search on?

We'll we can ternary search the time. For each time we check, we can evaluate the farthest distance between two cars by evaluating each car at that particular time and taking the min and max.