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.

Example #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).

On the following page is a function that essentially solves the problem:

```
double findTime(double f1, double f2, double a,
double b, double c) {

    double targettime = (f2-f1)/(f1*f2);
    double low = 0;
    double high = targettime/a;
    double mid = 0;

    while (high - low > EPSILON) {

        mid = (high+low)/2;
        double ans = f(a,b,c,mid);

        if (ans<targettime)
            low = mid;
        else if (ans == targettime)
            break;
        else
            high = mid;
    }

    return mid;
}

double f(double a, double b, double c, double t)
{
    return a*t+b*(1-exp(-c*t));
}
```

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.

How Can We Still Solve 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!

The function that performs this binary search is included on the following page.

```
double getMaxTime(int* perm, struct interval*
times, int length) {


    // Set up our binary search.
    double high = getMaxInterval(times, length);
    double low = 0, mid = (low+high)/2;

    // Keep going until our interval is tiny.
    while (high-low > 1e-9) {

        mid = (low+high)/2;

        // Standard binary search.
        if (works(perm,times,length,mid))
            low = mid;
        else
            high = mid;
    }

    return mid;
}
```