Binary Search

Strictly speaking, a binary search is a technique for searching for a value in a sorted array, where, instead of looping through the array, you look halfway through your search space, and make a comparison, so you can determine if the value you are trying to find is on the left side of the array or the right. In this manner, you can cut down your search space in half after each comparison, determining if a value is in the sorted array in O(lg n) time, where n is the number of values in the original array. Here is code for a binary search (returns true if val is in the vector a, false otherwise):

```
bool inVector(vector<int> a, int val) {

    int low = 0, high = a.size()-1;

    while (low <= high) {
        int mid = (low+high)/2;
        if (val < a[mid])      high = mid-1;
        else if (val > a[mid]) low = mid+1;
        else return true;
    }

    return false;
}
```

Each time through the loop, either we move high below mid or low above mid, ensuring that the new search space is at most half the size of the previous search space. The invariant is that if val is in the vector, then in must be within the indexes low to high. We exploit the fact that the vector is sorted by immediately only looking in one half of the vector after making a single comparison.

This idea can be adapted in a powerful way to solve many competitive programming problems.

Consider the following problem:

Let $f(t) = \sqrt{t}e^t$. Given a positive real number Y, compute the value of t such that f(t) = Y.

In essence, we are asked to invert the function f(t). A quick analysis will yield that inverting this function is rather difficult. However, one key property of the function is that as t increases, so does f(t). Thus, if we make some guess for t and plug into f, we can quickly see if our guess was too big or too small. For example, consider Y = 100 and t = 3. We can find that f(t) is roughly 34.7. This means that the real value of t for which f(t) = 100 must be greater than 3.

Now, we can see how to solve the problem fairly quickly. Assign low and high to safe values that are below and above the real answer. In this case, assuming Y > e, we can choose low = 0, high = ln Y. Guess halfway in between low and high. See if the guess was too big or too small and adjust high or low accordingly. Each time we guess, we narrow down the range of our answer by one half. Since the answer to this query is a real number, we call this a real-valued binary search.

Here are some tips for a real valued binary search:

1. Run for a fixed number of iterations, usually 60 to 100 suffice, due to the precision of double.

2. In each iteration either low or high will be set to exactly mid.

Let's look at a problem that involves a real-valued binary search.

Kattis Problem: Need for Speed
Here is a link to the problem:

https://open.kattis.com/problems/speed

It's hard to compute the value of c from the input, but if we make a guess for c, then we can calculate the time each segment of the trip would take, and then see if our guess for c was too low or too high. If the time we compute is too high, then we must increase our guess for c. Alternatively, we must reduce it. One key issue is the initial setting for low and high. c could be negative, but we also notice that if we plug in a bad value for c, we could divide by 0. So we must make sure low = -min, where min is the smallest speedometer reading in the input. Also, high could be quite high, since we could be making a long trip with low readings. Consider a situation were t = 1 but the sum of distances is 10^6, the max and each speedometer reading was -1000. We might have to add up to 1,001,000. There's no harm in making high a bit too high because all we're doing is dividing by this, so we'll set high = 2,000,000 to be safe. Now we're ready to look at the code.

Let’s take a look at a solution to this problem:

```
using namespace std;
#include <bits/stdc++.h>

int main() {

    int n, tTime;
    cin >> n >> tTime;

    vector<double> d(n);
    vector<double> s(n);

    for (int i=0; i<n; i++)
        cin >> d[i] >> s[i];

    // Set low to negative of minimum value.
    double low = s[0], high = 2000000;
    for (int i=1; i<n; i++)
        low = min(low, s[i]);
    low = -low;

    // 60 is probably good enough but we'll do 100.
    for (int i=0; i<100; i++) {

        // Our guess for c.
        double mid = (low+high)/2;

        // Add up trip time if c = mid.
        double thisT = 0;
        for (int j=0; j<n; j++)
            thisT += d[j]/(s[j]+mid);

        // Our guess is too low, increase low.
        if (thisT > tTime)
            low = mid;

        // Guess is too high...
        else
            high = mid;
    }

    // Ta da!
    cout << setprecision(9) << low << endl;
    return 0;
}
```

All the hallmarks of a real-valued binary search are here. The function we need to calculate forward (how much time is the trip given c) is easy to calculate and as c increases, this value decreases.

In our solution, we ran our search for a fixed number of iterations, we took some care to set low and high initially, and finally we either set low or high equal to mid after each loop iteration.

Now, let's consider another type of binary search: one where the answer is an integer.

In these, we will do a couple things differently:

1. Use a while loop…something like while (low < high)

2. Be very careful in setting low and high. It's possible to either set low = mid OR low = mid+1, and for high, it's possible to either set high = mid or high = mid-1. It's important to make sure by tracing the test case low = 2, high = 3, that the loop converges to one value for both low and high in the following iteration.

Let's consider the following problem:

Given a list of sorted integers, place them into *k* sets such that the maximum difference between any two values in the same set is minimized. What is this maximum difference?

Here is a sample case:

List: 2, 3, 3, 4, 6, 12, 13, 14, 15, 22, 27, 28, 29, 34
k = 3

It's hard to know how big to make a box, but if we agree that we're going to see if this maximum difference could equal 8, we can count how many boxes we would need:

2, 3, 3, 4, 6, 12, 13, 14, 15, 22, 27, 28, 29, 34

We start our boxes from the left. We cut off the box as soon as our difference with the minimum element exceeds 8. In doing this, we see that four boxes are required. That means the actual answer must be strictly greater than 8. (ie. we would set low = mid+1).

Alternatively consider trying 11:

2, 3, 3, 4, 6, 12, 13, 14, 15, 22, 27, 28, 29, 34

When we see this is possible, we can simply set high = 11. (ie set high = mid.) Unfortunately, there's no way to know if we can go any lower. (This breakdown doesn't prove that 10 is also possible.)

Let's look at a similar problem to this one where we utilized an integer binary search.

Kattis Problem: Financial Planning

Here is the link to the problem:

https://open.kattis.com/problems/financialplanning

If you knew the number of days of investing in advance, you could easily calculate your maximum profit: buy each stock which makes some profit, skipping all where you take a loss. Then, if this value is big enough, then you know that the minimum necessary days to invest is less than or equal to this number of days. Alternatively, if investing k days doesn't give you enough money, the real answer must be strictly greater than k.

Keeping this in mind, here is the solution:

```
using namespace std;
#include <bits/stdc++.h>
typedef long long ll;

int main() {
    int n;
    ll money;
    cin >> n >> money;

    vector<ll> profit(n);
    vector<ll> cost(n);

    for (int i=0; i<n; i++)
        cin >> profit[i] >> cost[i];

    // Set low to negative of minimum value.
    // Worst case is 2 billion and 1 days...
    ll low = 0, high = 210000000011;

    // Run binary search.
    while (low < high) {

        ll mid = (low+high)/2;

        // See how much money we make in this many days.
        // Just buy profiting stocks.
        ll make = 0;
        for (int i=0; i<n; i++) {
            if (mid*profit[i] > cost[i])
                make += (mid*profit[i] - cost[i]);

            // This is key so we don't overflow long long!!!
            if (make > money) break;
        }

        // This wasn't good enough...
        if (make < money)
            low = mid+1;

        // Answer can't be bigger than mid.
        else
            high = mid;
    }

    cout << low << endl;
    return 0;
}
```

A few things to notice here. In the evaluation function, if the profit we are making exceeds money (the amount we are trying to make), then we immediately break out of the calculation, instead of completing it. Had we completed it, we would overflow long long, because there could be 10^5 investments, each which make 10^{18} money. Thus, this single if statement:

```
if (make > money) break;
```

is the difference between a Wrong Answer and Accepted verdict.

Alternatively, setting high too low initially would prevent us from getting a correct answer as well.