Binary Search

Bhavit Sharma and Prateek Karnal

April 5, 2017

Abstract

This is an attempt to explain Binary Search. We'll first try to solve one of the fundamental problems in computer science and will try to abstract or generalize Binary Search by solving various problems.

Contents

1 Introduction

One of the fundamental problems in Computer Science is to retrieve the locate an element i.e. finding it's position in a given ordered list of numbers. More formally the problem can be stated as follows :-

Given an array let's say ***X*** *in which the elements are sorted in ascending order. We have to return the index of an element* ***E*** *if it exists in the array or return -1.*

1.1 Linear Search

This is trivial and left as an exercise for the reader.

1.2 Better Solution aka Binary Search

Instead of Linear Search which works in $O(n)$ time, we can take advantage of the fact that our list is ordered. How? We can see that if we compare given E with any element of the array, we can get a rough idea of where our element might be. Suppose if we compare E with $X[i]$ where $0 \leq i \leq |X| - 1$.
We can have three possibilities while comparing.

1. $X[i] > E$
   We know that if E exists in our array, it can only exists on the left of our array. This is because $\forall j > i,\ X[j] > E$ as well, as our array is sorted in ascending order.
2. $X[i] < E$
   By doing the argument similar to in the last point, we can say that if our E exists, it can only exists on the right of i.
3. **if** $X[i] = E$
   Trivial.

Now we have gotten a general idea regarding the problem. If we choose certain i, then it will be reduce our original problem from array of size $|X|$ to array of size of $|X| - i$ or i depending upon where E lies with respect to i. So the next question which arises is what i to choose. Will it be a random index or some particular element of the array. It turns out that if we take i to be the middle of the array X, then it will lead to the overall best complexity of $O(\log_2 n)$. Proof is left as an exercise for the reader. (*Hint:* Try to write the recursive definition of the complexity function.)

1.3 Code

```
#define _CRT_SECURE_NO_WARNINGS
#include <bits/stdc++.h>

using namespace std;

int main(){
        vector<int> a = {1, 2, 3, 4, 5, 6, 7, 8};
        int E, l = 0, r = (int)a.size() - 1;
        cin >> E;

        while(l <= r){
                int mid = l + r >> 1;
                if(a[mid] > E) r = mid - 1;
                else if(a[mid] == E)r = mid, l = -1;
                else l = mid + 1;
        }

        cout << (a[r] == E ? 1 : -1) << endl;
        return 0;
}
```

2 The Recipe of a Binary Search

Till now we concerned ourselves with finding an element in a range. Also note that in the implementation above, if there are multiple numbers with the desired value in the array, we cannot say for sure which one is returned. It turns out that we can have much more control over binary search which can make our work way easier.

2.1 The ingredients

There are essentialy two ingredients in a binary search.

- A condition on a discrete variable
- A series of discrete variables divided in two halves by the condition

Here the condition can be anything.
The only thing that must satisfy for application of binary search is that the series of variables over which we search should be divided in two proper halves by the condition. One half which satisfies the condition and the other which doesn't.
[**Note:** You might argue that we can apply a binary search on a continuous variable as well. This however involves working with a precision as a terminating conditon. This is the same as dividing the continuous range into discrete values with each subsequent value being greater than the last by the given precision. Although it can sometimes be more practical to consider our variable continuous, but in the end it is not much different from the discrete case.]
There are two possible cases-

- If x satisfies the condition then $x + 1$ also satisfies.
  i.e The right section of the range satisfies the condition.
- If x satisfies the condition then $x - 1$ also satisfies.
  i.e The left section of the range satisfies the condition.

In both cases the range is divided into two halves. Now if we consider the two sections as right and left instead of satisfying and not-satisfying, then both cases become essentially same problems.
Now that we have established our requirements and conventions we will talk about the two sections as left and right sections.
[Note that a section could be empty as well.]

2.2 Our Aim

We have two possible aims in binary search-

- Finding the last element of the left section
- Finding the first element of the right section

2.3 The Method

The basic principle of binary search is that by checking the condition for a value of the discrete variable, we can reduce our search space.
Let us consider the case when we want the last element of the left section. On checking the condition

- if the value belongs to the left section, we can remove all the values which lie further to the left.
- if the value belongs to the right section, we can remove the current value including all the values to the right

Similarly when we want to find the first element of the right section. On checking the condition

- if the value belongs to the left section, we can remove the current value and all the values which lie further to the left.
- if the value belongs to the right section, we can remove all the values which lie further to the right

2.4 The edge case

When we remove all the values that lie further to the left, if the current value is the leftmost value then we end up not reducing our search space at all.
Similarly, when we remove all the values that lie further to the right we wont reduce our search space if the current value is the rightmost.
This normally is not a problem because as we take the middle element or an element closes to middle.
However, when only two elements are left in the search space this becomes a problem as both elements are either leftmost or rightmost.

It is easy to counter this however. We can simply always take the *Ceil* of middle in the first case and the $floor$ of middle in the second.

3 Introductory Problems

Problem 1: *Given a sorted array X, find the first element in X which is not less than a given number Y. Return -1 if no such element exists.*

Solution: First we need to check if we can apply Binary Search on it. If we compare Y with $X[mid]$, we can see that if $Y > X[mid]$, then our answer will lie to right side of mid. But if $Y \leq X[mid]$, then our answer will lie on the left side of mid, with one of the possible answers being $X[mid]$ also, so we'll keep it in our interval.

```cpp
#define _CRT_SECURE_NO_WARNINGS
#include <bits/stdc++.h>

using namespace std;

int main(){
        vector<int> a = {1, 2, 3, 3, 3, 4, 5, 7, 8};
        int E, l = 0, r = (int)a.size() - 1;
        cin >> E;

        while(l < r){
                int mid = l + r >> 1;
                if(a[mid] >= E) r = mid;
                else l = mid + 1;
        }

        cout << (a[r] >= E ? r : -1) << endl;
        return 0;
}
```

Problem 2: *Given a sorted array X, find the first element in X which is greater than a given number Y.*

Solution: First we need to check if we can apply Binary Search on it. If we compare Y with $X[mid]$, we can see that if $Y > X[mid]$, then our answer lie to the right side of mid. But it $Y < X[mid]$, then our answer will lie on left side of mid, with one of the possible answers being $X[mid]$ also, so we'll keep it in our interval.

```cpp
#define _CRT_SECURE_NO_WARNINGS
#include <bits/stdc++.h>
```

```
using namespace std;

int main(){
        vector<int> a = {1, 2, 3, 3, 3, 4, 5, 7, 8};
        int E, l = 0, r = (int)a.size() - 1;
        cin >> E;

        while(l < r){
                int mid = l + r >> 1;
                if(a[mid] > E) r = mid;
                else l = mid + 1;
        }

        cout << (a[r] > E ? r : -1) << endl;
        return 0;
}
```

Problem 3: *Given a sorted array X, find the last index of element which is not greater than Y or return -1 if it doesn't exits*

Solution: Let's say we have a *mid*. Now if $X[mid] > Y$ then we move to the left side of the binary search. But if $X[mid] \leq E$, then we move to the right side of the binary search, with one of the possible solutions being *mid* also. So here we are essentially taking the first element of the right side. So, while writing the code for binary search, we'll take our *mid* to be $\frac{a+b+1}{2}$.

```
#define _CRT_SECURE_NO_WARNINGS
#include <bits/stdc++.h>

using namespace std;

int main(){
        vector<int> a = {1, 2, 3, 3, 3, 4, 5, 7, 8};
        int E, l = 0, r = (int)a.size() - 1;
        cin >> E;

        while(l < r){
                int mid = l + r + 1 >> 1;
                if(a[mid] > E) r = mid - 1;
                else l = mid;
        }

        cout << (a[r] <= E ? r : -1) << endl;
        return 0;
}
```

Problem 4: *Given a function $y = f(x)$ which is a monotonic function with $f(-\infty) \to -\infty$ and $f(\infty) \to \infty$. Find the root of this function correct upto d decimal digits.*

Solution: Let us take two point on this function a and b such that $f(a)f(b) < 0$, then we know that our root (lets say r) lies between a and b. How can we apply binary search on this? We choose $mid = \frac{a+b}{2}$. So if $f(mid)f(a) < 0$, then our root lies between a and mid. If $f(mid)f(a) = 0$, then mid is our root. If $f(mid)f(a) > 0$, then our root lies between mid and b. It turns out that this may take forever to calculate the root to infinite precision. So to calculate it upto d degrees of precision, we let this binary search run for at most $log_2((b - a + 1)(10^d))$ times. (Why?)

Problem 5: **SPOJ Problem : Aggressive Cows**

Solution: First we need to understand how we can apply binary search on this. Lets suppose the minimum distance between 2 cows be d. Then of course, the minimum distance between two cows can be decreased further more by bring cows closer together. So if we can get minimum distance to be d, then we can also get minimum distance $\leq d$. On the another hand, lets say we can't a minimum distance of d between cows. This means that we also can't get $\geq d$ minimum distance between cows(obviously).Hence this satisfies our property of the binary search and hence it's applicable.

How to check if a particular d is valid? We sort all x_i and give x_0 the first cow. Then we assign another cow at the first x_i *such that* $x_i \geq x_0 + d$. And then continue giving the cows this way. You can prove greedily that this ensures the maximum number of cows distribution. Finally we check if the total assigned cows are $\geq c$ for a d to be valid.

Complexity is $O(n \log_2 n)$

```
bool check(const vi & x, const int & mid, const int & c){
        int tot = 0, i = 0, j = 0, n = (int)x.size();
        while(i < n){
                tot++;
                while(i < n && x[i] - x[j] < mid)i++;
                j = i;
        }
        return tot >= c;
}
int main(){
        SYNC;
        int T; cin >> T;
        while(T--){
                int n, c;
                cin >> n >> c;
                vi x(n);

                REP(i, n) cin >> x[i];
                sort(ALL(x));
                int l = 0, r = (int)1e9;

                while(l < r){
                        int mid = l + r + 1 >> 1;
                        if(check(x, mid, c))
                                l = mid;
                        else
                                r = mid - 1;
                }
                cout << l << endl;
        }
}
```

[**Note:** Here we have taken the right *mid* because we have to combine it with the right interval in the case when d is valid.]

Problem 6: **UVA: 11413** Click

Solution: The first thing to see intuitively is that for a given maximum capacity let's say C, we can fill them with the containers satisfying the constraints, then obviously $\forall C' \geq C$ we can fill the containers satisfying the given constraints. But if for a maximum capacity we can't fill the containers with the vessels satisfying the given constraints [**Note: This will happen if the total containers required to be filled by vessels exceed the total containers given in the problem or there is a vessel with capacity $V_i > C$**] then $\forall C' \leq C$ we can't fill the containers as that would require even more containers or we can get a vessel with capacity $> C'$. Hence our condition for binary search satisfies.

Now How to check if a particular maximum capacity C is valid? We start from the first vessel and keep filling the first container until the sum of the capacities of all the vessels $\leq C$. After that we start again greedily to fill another containers. And then we'll check in the end if containers filled are $\leq m$.You can prove that it will be the most optimum way to ensure least numbers of containers filled. [**Proof is left as an exercise for the reader.**].

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

int n, m, V[1000];

int check(int c) {
    int cap = 0, cnt = 0;
    int i = 0;
    for(i = 0; i < n; i++) {
        if(V[i] > c)
            return 1000000;
        if(cap < V[i])
            cap = c, cnt++;
        cap -= V[i];
    }
    return cnt;
}

int main() {
    int i;
    while(scanf("%d %d", &n, &m) == 2) {
        for(i = 0; i < n; i++)
            scanf("%d", &V[i]);
        int l = 1, r = 1000000*n;
        while(l < r) {
            int mid = (l + r)>> 1;
            int cnt = check(mid);
            if(cnt > m)
                l = mid+1;
            else
                r = mid;
        }
        printf("%d\n", l);
    }
    return 0;
}
```

Problem 7: **Codeforces Round 361, Div 2C** : Click

Solution: The first thing to see intuitively is that if for a given n, we can get total number of ways chocolate distribution to be $\geq m$, then obviously $\forall\ N \geq\ n$, we can get total number of ways $\geq m$. But if for some n, we get total number of ways $<\ m$, then $\forall\ N\ \leq n$, we'll get total number of ways $< m$. [**Note the inequalities**]. Hence our condition for binary search satisfies. Now for a particular n, how to calculate the number of ways of distributing chocolates?

Suppose the number of chocolates that the first thief has be X. Now the number of chocolates the last thief is going to have will be k^3X. So we have to find the total solutions of this inequality $k^3X \leq n$. So we'll iterate over $k \geq 2$ to find valid X till $k^3 \leq n$.

```
LL ch(LL mid){
        LL ans = 0;
        for(LL k = 2; k <= mid && k * k <= mid && k * k * k <=
        ↪ mid; k++){
                ans += mid/(k * k * k);
        }
        return ans;
}

int main(){
        SYNC;
        LL m;
        cin >> m;
        LL l = 0, r = (LL)1e16;

        while(l < r){
                LL mid = l + r >> 1;
                if(ch(mid) >= m){
                        r = mid;
                }
                else l = mid + 1;
        }

        if(ch(l) == m) cout << l << endl;
        else cout << -1 << endl;
        return 0;
}
```

Problem 8: CSAcademy: Round 20, Div 2C Click

Solution: We can see that the if our best distance is let's say D then obviously we can also upgrade the cities in such a way that the maximum distance between regular and upgraded city will be $\geq D$ which means it will be monotonic and we can apply binary search on D.

We sort the $x_i's$ first. Now to check if a particular D is valid, we can greedly start from first city and upgrade the city at distance D (We choose the last city with distance atmost D because it minimizes the number of upgraded cities). Now we start from this upgraded city and keep ignoring the cities to right which have distance $\leq D$.

```
vi v;
int ch(int mid, int n){
        int ans = 0, ptr = 0;
        while(ptr < n){
                int i = ptr; ptr++;
                while(ptr < n && v[i] + mid >= v[ptr]){
                        ptr++;
                }
                ans++;ptr--;i = ptr;
                while(ptr < n && v[ptr] - v[i] <= mid)
                        ptr++;
        }

        return ans;
}

int main(){
        SYNC;
        int n , k; cin >> n >> k;
        v.resize(n);

        REP(i, n){
                cin >> v[i];
        }
        sort(ALL(v));

        int l = 0 , r = v[n - 1] + 1;

        cerr << ch(1, n) << endl;

        REP(i , 32){
                int mid = l + r >> 1;
                if(ch(mid, n) > k){
                        l = mid + 1;
                }
                else
                        r = mid;
        }
        cout << r << endl;
        return 0;
}
```

Problem 9: **Codeforces Round 350 D2** Click

Solution:- You should see intuitively that if we can make X dishes then obviously we can make $\leq X$ dishes also. But if we can't make X dishes, then we can't make $\geq X$ dishes also because we won't have sufficient ingredients or the magic powder. So binary search is applicable.

How to check if a particular X is valid?. If we make X dishes, then we'll need $a[i].X$ grams of ingredients of i_{th} ingredient, but we already have $b[i]$ grams of i_{th} ingredient, so we'll need $max(0, a[i].X - b[i])$[0 in case $b[i] > a[i].X$] grams of magic powder for i_{th} ingredient. We'll sum the magic powders needed for all ingredient $0 \leq i \leq n-1$ and check if it's $\leq k$.
Click for code [**Why I've taken the right mid?**]

Problem 10: **Codeforces Round 218 D2** Click

Solution: It is almost similar to the previous problem. Left as an exercise for the reader. Click for code.

[Note: I'll add more introductory and advanced problems later on]