Binary Search

Assume you are reading a book, and you want to read Page 747. How can you reach the page?

One way is to go through all the pages one by one till you reach 747. But this will take up a lot of our time.

What we generally do is, open a random page, and check the page number. If the current page number is less than 747, follow the same process on the right side pages. Similarly, if the current page number is greater than 747, check the left side.

Now, in our case, we can be a bit biased while selecting the page to open first during searching. Because we know the page numbers follow a particular order like 1, 2, 3, … and we also know the last page number. So let’s say the total number of pages is 800. So we know 747 is towards the end. So, we will open up a page towards the end of the book in the first iteration.

However, in real life, we do not always have this information. So picking a pivot is crucial for the search to be faster. It has been found that, the search is fastest in worst case when we choose the middle element as the pivot. This leads us to the concept of binary search.

What is Binary Search?

Binary search is a searching algorithm, that works on a monotonic sequence (increasing or decreasing) and efficiently searches for an element.

Binary Search in Action

How does Binary Search work?

During the binary search, we always discard half of the search space. As I mentioned earlier, it works only on monotonic sequences i.e. sequences that are increasing or decreasing. Let’s take an example:

We are given an array containing the following elements:

```
int arr[] = {1, 1, 2, 3, 4, 5, 6, 7, 8};
```

We need to search for the element 7 in the array.

Following is how the binary search program would execute:

1. Determine the middle element. `4` is the middle element (i.e. index 4)
2. Now we know 7 > 4. And as the array is increasing, it is definitely on the RHS. So we can just discard the left half and continue our search on the right half. So, the new search space we are considering is `[5, 6, 7, 8]`
3. Repeat the same process. The middle element is 6. As 7 > 6, we know it lies on the RHS. So reject the left half. New search space becomes `[7, 8]`
4. The middle element is 7. As it is the same as the requested element, so we return it.

Complexity Analysis

In binary search, as discussed, half of the search space is rejected every iteration. So in the worst case, we will be continuing till we are left with at least one element.

So initially we start with `N` elements.

After the first iteration, we have `N/2` elements.

After the second iteration, `N/4` elements.

And so on.

```
N -> N/2 -> N/4 -> N/8 -> N/16 -> ..... -> 1
```

Copy

We can also write this as,

```
N -> N/2 -> N/(2^2) -> N/(2^3) -> ..... -> N/(2^k)
```

Copy

where `N/(2^k) = 1` and it runs for k iterations.

So `k = log2(N)`

The overall complexity of the binary search algorithm is, `log2(N)` where N is the search space.

Code

Copy

```
int findIndex(int arr[], int n, int x) {
     // lo, hi points to the lowest and highest indices of search space. Curren
     // we are considering the entire array so 0 and n-1 respectively.
   int lo = 0, hi = n-1;
   while (lo <= hi) {
       int mid = (lo + hi) / 2;
       if (arr[mid] == x) return mid;
       else if (arr[mid] < x) {
           // x is on the right hand side, reject left half
           lo = mid + 1;
       } else {
           // x is on the left hand side, reject right half
           hi = mid - 1;
       }
   }

   // If we come here, then we did not find the element.
```

```
    return -1;
  }
```

Question: 2 sum

You are given an array of N elements, and also a number k. Find if there are 2 elements, whose sum is equal to k.

▾ Solution

We can fix every element and try searching for the other element, such that their sum is k. If found, return true, else false.

For that, we can sort the array first. Now go through all the elements one by one. For every element `i` the question becomes, is there another element which is equal to `k - arr[i]`

```
bool bs(vector<int> &nums, int target) {
  int lo = 0, hi = nums.size() - 1;
  while (lo <= hi) {
     int mid = (lo + hi)/2;
     if (nums[mid] == target) return true;
     else if (nums[mid] < target) lo = mid + 1;
     else hi = mid - 1;
  }

  return false;
}
```

```
bool twoSum(vector<int> &nums, int k) {
      sort(nums.begin(), nums.end());
    for(int i=0; i<n; i++) {
        int target = k - nums[i];
        if (bs(nums, target)) return true;
    }

    return false;
}
```

Question: Count Pairs Whose Sum is Less than Target

Link: https://leetcode.com/problems/count-pairs-whose-sum-is-less-than-target/description/

▾ Solution

As discussed earlier, whenever we have problems related to pairs, we can fix one element (as the right element), and search on the LHS of that element.

For each fixed element `j` we want to find the number of elements on the LHS that are less than `target - arr[j]` .

Let's assume the array is sorted. How can we find the number of elements <k in the array?

We can find the first index ≥ k (let's say the index is idx), then the number of elements <k is `idx` (0 - idx-1).

So we will sort, our main array and run the algorithm. That is iterate over all elements and fix them as `j` . For each element identify how many elements are there on the L.H.S i.e (0, j-1) that are < (target - arr[j]) using Binary Search.

This brings us to our next topic, lower-bound searches.

▼ Solution extended

So, in the above case, it is similar to finding lower-bound of `target - arr[j]` .

Copy

```
class Solution {
public:

    int compute(vector<int> &nums, int k, int beg, int end) {
        // Number of elements < k in range [beg, end]
        if (nums[beg] >= k) return 0;
        else if (nums[end] < k) return (end - beg + 1);
        int lo = beg, hi = end;
        while (lo < hi - 1) {
            int mid = (lo + hi) / 2;
            if (nums[mid] < k) lo = mid;
            else hi = mid;
        }

        return hi;
    }
    int countPairs(vector<int>& nums, int target) {
        sort(nums.begin(), nums.end());
        int ans = 0;
        for(int i=1; i<nums.size(); i++) {
            ans += compute(nums, target - nums[i], 0, i-1);
        }

        return ans;

    }
};
```

lower-bound like searches

Firstly, what do I mean by `lower-bound` . It is a function that is available with the algorithm library. It returns the first index ≥ x. In these cases, you are not just searching if an element is present or not, but rather the first index that satisfies a monotonic property.

Let's understand with an example. In the following array, find the first index that is greater than or equal to 5

```
int arr[] = {1, 1, 2, 3, 3, 4, 5, 5, 6, 7, 8};
```

Copy

The answer should be 6.

I see these types of questions as 0-1 questions, where you can generate a 0-1 array based on the property. The property is ≥ 5.

Now elements satisfying the property get a 1, else 0.

If you think about it, as the array is increasing, initially the elements are 0, because they are less than 5. Once it becomes 5, it will always be ≥5 as the array is increasing.

So the 0-1 equivalent array will be

```
[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

Copy

💡 There are multiple ways to solve this. Below I have outlined the approach that I use and find more intuitive.

In these arrays, we need to find the first occurrence of 1. Now traditional binary search doesn't work here as it would return true whenever it sees a one. We specifically want the first occurrence of 1. So we will tweak our binary search implementation a bit to tackle this.

1. We will not terminate when we find the desired number (`1` or `5`).
2. lo will **always** point to something that is not satisfying the property
3. hi will **always** point to something that is satisfying the property.
4. Or vice versa between lo and hi

If we think about the statements above, where should we end the loop?

In a state like this `[0, 1]` , where lo points to the last 0 and hi points to the first 1.

With this in mind, let's write the code

Copy

```
int lower_bound_search(vector<int> &nums, int k) {
  // returns the first index, where elements are >= k
  // first check for 2 extreme cases, if no element >=k or if all >=k
  int n = nums.size();
  if (nums[0] >= k) return 0;
  else if (nums[n-1] < k) return -1; // or n whichever works
  int lo = 0, hi = n-1;
  while(lo < hi - 1) {
      // We always end in this state, lo is one index before hi
      int mid = (lo + hi)/2;
      if (nums[mid] >= k) {
        // all the elements on the right are definitely >= k. On the left is un
        // so let's search on left
        hi = mid; // We cannot set to mid - 1 as we are not sure if nums[mid-1]
      } else {
          lo = mid;
      }
  }

  // Now hi points to the first index >= k
```

```
    return hi;
  }
```

Question: Find Peak Element

Link: https://leetcode.com/problems/find-peak-element/

▾ Solution

Is there a property that is getting satisfied by each array element and is monotonic?

In the first half we have elements arr[i] > arr[i-1]

In the second half, it is arr[i] < arr[i-1]

If we consider this property, do we get a 0-1 array? Yes!

Let's understand with an example:

[1, 2, 3, 4, 3, 2]

Ignore index 0, as -1 is not a valid index. Let's compute from 1

index = 1: Is arr[i] < arr[i-1]? No

index = 2: Is arr[i] < arr[i-1]? No

index = 3: Is arr[i] < arr[i-1]? No

index = 4: Is arr[i] < arr[i-1]? Yes

index = 5: Is arr[i] < arr[i-1]? Yes

So we get [null, 0, 0, 0, 1, 1]

Now we can do binary search on this.

▼ Code

Copy

```cpp
class Solution {
public:
    int findPeakElement(vector<int>& arr) {
        int n = arr.size();
        if (n == 1) {
            return 0;
        }
        int lo = 1, hi = n - 1;
        // Property arr[i] < arr[i-1]
        if (arr[lo] < arr[lo-1]) {
            return 0;
        } else if (arr[hi] > arr[hi-1]) {
            return n-1;
        }

        while (lo < hi - 1) {
            cout<<lo<<" "<<hi<<endl;
            int mid = (lo +  hi) / 2;
            if (arr[mid] > arr[mid-1]) {
                lo = mid;
            } else hi = mid;
        }

        return lo;
    }
};
```

Taking it to the next level

We discussed earlier, that if we have some monotonic array, where each element satisfies a monotonic property, we can binary search on the monotonic property using 0-1 array.

But, this holds true even for a non-monotonic array. In general, if we have some property that is monotonic, we can binary search for it.

We saw that with the peak element example, the array was not monotonic. But we identified a property `arr[i] > arr[i-1]` that is monotonic, and based on that did a binary search. However, often the property that makes it monotonic is not so simple.

Let’s understand with an example: https://leetcode.com/problems/capacity-to-ship-packages-within-d-days/description/

Here the weights in not a monotonic array (i.e. neither increasing nor decreasing).

Let’s say the maximum capacity of the ship is X.

Now for X, if it takes more than D days, obviously if the capacity is < X, then also it takes more than D days. (Think like this, if the ship can carry max capacity of 10 kgs and with this it takes 5 days, then if it has max capacity of 5 kgs, obviously it will take ≥ 5 days)

Similarly, if the ship takes ≤D days with X capacity, it will obviously take ≤D days when its capacity is > X. (Example, if ship can carry 10kgs and it takes 5 days, obviously it will take ≤ 5 days if its capacity was 15 kgs)

So the property maximum capacity of the ship is itself monotonic with the number of days taken.

As max capacity increases, the days taken decrease. As capacity decreases, days taken increase.

Let's say `G(x)` returns the number of days taken when max capacity is x.

`F(x)` is whether the ship can transport all the goods within D days if its max capacity is x.

That is, `F(x) = G(x) <= D`

If we carefully observe, F(x) is monotonic.

For lower values of x, G(x) is more and thus F(x) is false. As x increases, a point comes when G(x) ≤ D, and F(x) becomes true. And from there, it always stays true as x increases.

Now, what do we want. Minimum possible value of the maximum capacity. So we want the first x, where F(x) becomes true. 😳

That's it!

The code becomes something like this

```
Copy
int lo = LOW_VAL, hi = MAX_VAL;

while (lo < hi-1) {
    int mid = (lo + hi)/2;
  if (F(mid)) hi = mid;
  else lo = mid;
}

return hi;
```

Now what is pending to identify? LOW_VAL, HIGH_VAL and how is G(x) calculated

LOW_VAL = Lowest possible value of maximum capacity. That is the maximum capacity cannot be lower than this. So in the worst case, we need to carry all the packages right? And one package / day. So the ship should be at least able to carry the maximum weighted package.

```
Copy
LOW_VAL = *max_element(weights.begin(), weights.end());
```

MAX_VAL = Maximum possible capacity that is needed for our problem. In best case, the ship transfers all the goods in one day itself. So at max, it needs a total capacity of total sum of all the weights.

```
MAX_VAL = accumulate(weights.begin(), weights.end(), 0);
```

Finally G(x). What is it? Given max carrying capacity of the ship is x, how many days would it take. We can just run a linear search for this because it has to transfer all the weights in order.

```
int G(int x, vector<int> &weights) {
  int days = 0;
  int current_weight = 0;
  for(int i=0; i<weights.size(); i++) {
      if (current_weight + weights[i] <= x) {
        // We can take this in the same day
        current_weight += weights[i];
      } else {
         // We need to take this on a new day
         days++;
         current_weight = weights[i];
      }
  }

  return days;
}

bool F(int x, int d, vector<int> &weights) {
   return G(x, weights) <= d;
}
```

Now you might be wondering, wouldn't it take a lot of time, or not meet the time constraints?

Let's analyze the complexity. We are running a binary search. For each binary search iteration, we are calling `F(mid)` which in turn calls `G(mid)` . So each iteration takes O(N) time. But how many iterations in binary search? It's `log2(N)` .

So overall complexity is: `O(Nlog2(N))`

💡 Generally, if you come across problems like minimizing the maximum value of some property or maximizing the minimum value of some property, or minimization / maximization in general, binary search can be one of the ways to solve it.

Q - Vasya and String

Link: https://codeforces.com/problemset/problem/676/C

▾ Solution

We need to find the maximum length substring containing all equal characters, which is also known as the beauty of the string. Let's call it `beauty number`

Also, Vasya cannot change more than K characters of the string.

Okay, so let's assume the beauty number is X. We want to determine what is the minimum number of changes needed to make the string have a beauty number of X.

Let's say G(X) returns this thing.

So $F(X) = G(X) \le K$ as we cannot change more than K characters.

Now is F(X) monotonic. As X increases, i.e. we need a longer substring to have all characters equal, which means more changes.

Similarly, as X decreases, we need to make all characters of a smaller substring equal, which means less changes.

Thus, F(X) is monotonic, so we can binary search on X i.e. the beauty number or the length of substring containing all equal characters.

Now how to calculate G(X)?

We can use a sliding window algorithm. The substring length is fixed, so we can use sliding window on all possible substrings and determine the minimum changes needed for one of them to be all equal. To calculate for a window, we maintain a frequency array of size 26, which gives the frequency of all characters in the current window. To make all of them equal, it is same as determining the one with maximum frequency and changing all the others to this i.e. (X - frequency of char with max frequency).

To slide the window, `freq[(int)(s[i] - 'a')] ++` and `freq[(int)(s[i] - 'a')] --`

▼ Code for G(x)

Copy

```
int calcChanges(vector<int> &freq, int x) {
  int mx = freq[0];
  for(int i=1; i<26; i++) mx = max(mx, freq[i]);
  return (x-mx);
}

int minChanges(string &s, int x) {
  vector<int> freq(26, 0);

  // Calculate frequencies for the first window manually
  for(int i=0; i<x; i++) freq[(int)(s[i]-'a')]++;

  int changes = calcChanges(freq, x);

  // Sliding window start
  for(int i=x; i<n; i++) {
    // Remove i-x as it goes out of window
    freq[(int)(s[i-x] - 'a')]--;
    // Add ith element as it gets added new in window
    freq[(int)(s[i] - 'a])]++;
    changes = min(changes, calcChanges(freq, x));
  }
  return changes;
}
```

Now once G(x) is calculated, we can just write F(x) = G(x) ≤ k. And then do a binary search for X. The answer will be the last occurrence of True of F(x).

▾ Code

Copy

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

using namespace std;

int calcChanges(vector<int> &freq, int x) {
  int mx = freq[0];
  for(int i=1; i<26; i++) mx = max(mx, freq[i]);
  return (x-mx);
}

int minChanges(string &s, int x) {
  vector<int> freq(26, 0);

  // Calculate frequencies for the first window manually
  for(int i=0; i<x; i++) freq[(int)(s[i]-'a')]++;

  int changes = calcChanges(freq, x);

  // Sliding window start
  for(int i=x; i<s.size(); i++) {
    freq[(int)(s[i-x] - 'a')]--;
    freq[(int)(s[i] - 'a')]++;

    changes = min(changes, calcChanges(freq, x));
  }

  return changes;
}

int f(string &s, int x, int k) {
    return minChanges(s, x) <= k;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    int n, k;
    cin>>n>>k;
    string s;
    cin>>s;

    int lo = 0, hi = n+1;

    while(lo < hi-1) {
        int mid = (lo + hi)/2;
        if (f(s, mid, k)) {
            lo = mid;
        } else hi = mid;
    }

    cout<<lo<<"\n";

    return 0;
}
```

Q - Minimum Time to Complete Trips

Link: https://leetcode.com/problems/minimum-time-to-complete-trips/description/