Binary Search 2

Bootcamp Track

Gabee De Vera

Computing $\left\lfloor \sqrt{N} \right\rfloor$

Computing $\left\lfloor \sqrt{N} \right\rfloor$

- Suppose that you want to find $\left\lfloor \sqrt{N} \right\rfloor$ quickly, in $O(\log N)$ time *without* using floating point numbers.

Computing $\left\lfloor \sqrt{N} \right\rfloor$

- Suppose that you want to find $\left\lfloor \sqrt{N} \right\rfloor$ quickly, in $O(\log N)$ time *without* using floating point numbers.
- Let $k = \left\lfloor \sqrt{N} \right\rfloor$. Clearly, $k^2 \leq N$. Thus, k is the largest integer such that $k^2 \leq N$.
- However, we can find this using binary search!
- Let L be the set of numbers k such that $k^2 \leq N$, and R be the set of numbers k such that $k^2 > N$. Then, the answer is the largest value in the set L.

Computing $\left\lfloor \sqrt{N} \right\rfloor$

- Here's the implementation in C++

```
int n;
cin >> n;
int l = 0, r = n + 1;
while(r - l > 1) {
    int m = (l + r) >> 1;
    if(m * m <= n) l = m;
    else r = m;
}
cout << l << endl;
```

Computing $\lfloor \sqrt[3]{N} \rfloor$

- Computing $\lfloor \sqrt[3]{N} \rfloor$ can also be done similarly. This time, we want to find the largest integer k such that $k^3 \leq N$

```cpp
int n;
cin >> n;
int l = 0, r = n + 1;
while(r - l > 1) {
    int m = (l + r) >> 1;
    if(m * m * m <= n) l = m;
    else r = m;
}
cout << l << endl;
```

A Generalization

- In general, to find the largest number k that satisfies $f(k) \leq N$, where f is some *monotonically increasing function* (i.e., it does not decrease as k increases), we use,

```
int n;
cin >> n;
int l = 0, r = n + 1;
while(r - l > 1) {
    int m = (l + r) >> 1;
    if(f(m) <= n) l = m;
    else r = m;
}
cout << l << endl;
```

- In fact, $f(m) \leq n$ is known as as *predicate function*.

Predicate Function

Predicate Function

- A **predicate function** is a function that, given a certain value x (usually an integer), returns a boolean.
- For example, $g(x) := x \leq 10$ is a predicate function, since x can be an integer, and g returns a boolean.
- Some other predicate functions that we've seen include,

$$g_1(x) := a_x \leq k$$
$$g_2(x) := x^2 \leq k$$
$$g_3(x) := x^3 \leq k$$

Binary Search with Predicate Functions

- In fact, **binary search** can be generalized to work with *more kinds of* predicate functions.
- That being said, not all types of predicate functions work for binary search.
- The predicate function must return true for inputs $x \leq k$ and return false for inputs $x > k$ for some k. (It could also return false for $x \leq k$ and true for $x > k$. In this case, you just take the logical NOT of your function)
- These kinds of predicate functions are known as **monotonic** predicate functions.

Binary Search with a Monotonic Predicate Function g

- In its full generality, this is what the binary search algorithm looks like:

```
int l = MINV - 1, r = MAXV + 1;
while(r - l > 1) {
    int m = (l + r) >> 1;
    if(g(m)) l = m;
    else r = m;
}
```

- Its time complexity is $O(g \log N)$, where N is the size of the initial interval, and g is the time it takes to execute the function g.

Two-Liner Binary Search

- You can push this even further using the ternary operator. Here's an implementation of binary search in two lines of C++:

```
int l = MINV - 1, r = MAXV + 1;
while(r - l > 1) (g((l + r) >> 1) ? l : r) = (l + r) >> 1;
```

Binary Searching for the Answer

- A common pattern in CompProg is to use *binary search* to find an optimal value in certain optimization problems.
- Instead of directly computing the optimal value v, we instead consider a similar problem: is there a configuration that solves the problem with value v or less?
- Let $\mathrm{good}(v)$ be true if there is a configuration that solves the problem with value v or less
- Then, $\mathrm{good}(v)$ is monotonic, and we can solve this problem with binary search in $O(f(N) \cdot \log N)$, where $f(N)$ is the time complexity of $\mathrm{good}(v)$.

Binary Searching for the Answer

- Consider, for instance, the following optimization problem (taken from https://cp-algorithms.com/num_methods/binary_search.html): You are given an array a_i composed of n integers. What is the largest floored average value over all possible subarrays? Formally, for all $0 \leq l \leq r \leq n-1$ satisfying $r-l+1 \geq x$, what is the largest value of $\left\lfloor \frac{\sum_{i=l}^{r} a_i}{r-l+1} \right\rfloor$?
- Constraints: $1 \leq a_i \leq 10^9$, $1 \leq n \leq 2 \cdot 10^5$, $1 \leq x \leq n$
- First, note that the brute force algorithm, which simply goes through all possible subarrays is $O(n^3)$. This can be further optimized to $O(n^2)$ using the previously discussed prefix sum technique.
- However, our goal is to find something that is faster than quadratic -- i.e., *subquadratic*. Is such an algorithm possible?

Binary Searching for the Answer

- Let $\left\lfloor \frac{\sum_{i=l}^{r} a_i}{r-l+1} \right\rfloor = k$ be the answer. We will binary search for the proper k.
- Start with a candidate value for k, k'. Then, we will attempt to find a subarray of a such that $\left\lfloor \frac{\sum_{i=l}^{r} a_i}{r-l+1} \right\rfloor \geq k'$. Define a function $g(k')$. If there exists a subarray that satisfies the constraint, then $g(k') = \text{true}$. Else, $g(k') = \text{false}$.
- We can use binary search on the answer to find the answer in $O(f(n) \cdot \log(\text{maximum possible sum}))$, where $f(n)$ is the time complexity of running function g. In practice, a value of around 10^{18} suffices for the initial upper bound of your binary search, but this of course *depends on your problem*.

Binary Searching for the Answer

- Now, we've simplified the optimization problem to the following decision problem:

  Is there a subarray of length at least $r - l + 1 \geq x$ that satisfies $\left\lfloor \frac{\sum_{i=l}^{r} a_i}{r-l+1} \right\rfloor \geq k'$?

Binary Searching for the Answer

- Now, we've simplified the optimization problem to the following decision problem:

  Is there a subarray of length at least $r - l + 1 \geq x$ that satisfies $\left\lfloor \frac{\sum_{i=l}^{r} a_i}{r-l+1} \right\rfloor \geq k'$?

- To solve this, we can perform some simplifications:

$$\frac{\sum_{i=l}^{r} a_i}{r - l + 1} \geq k'$$

$$\sum_{i=l}^{r} a_i \geq k'(r - l + 1)$$

$$\sum_{i=l}^{r} \left(a_i - k'\right) \geq 0$$

Binary Searching for the Answer

$$\sum_{i=l}^{r} \left(a_i - k'\right) \geq 0$$

- Thus, we only need to check if there is a subarray of the array that satisfies the constraint above.
- Consider the array $b_i = a_i - k'$ (which could be computed in $O(n)$). Then, the condition above becomes,

$$\sum_{i=l}^{r} b_i \geq 0$$

- Therefore, we only need to determine whether there is a subarray of b of length at least $r - l + 1 \geq x$ whose sum is nonnegative.

Binary Searching for the Answer

- To calculate a subarray sum over b quickly, we can use *prefix sums*. Consider the prefix sum array $\sum b$ of length $n+1$. Here, $(\sum b)[i] := \sum_{k=0}^{i-1} b[k]$.
- Then, $\sum_{i=l}^{r} b[i] = (\sum b)[r+1] - (\sum b)[l]$.
- We thus want to find two indices r and l satisfying $r - l + 1 \geq x$ such that $(\sum b)[r+1] - (\sum b)[l] \geq 0$.
- With the substitution $u = r + 1$, this is the same as finding two indices u and l in the array $(\sum b)$ such that $u - l \geq x$ and $(\sum b)[u] \geq (\sum b)[l]$.
- Therefore, if we can find such a pair of indices quickly, we can solve the original problem quickly!

Binary Searching for the Answer

- The idea is to *reduce* this problem to a problem of range maximums over a static array, as follows:
- For every index l, consider all indices $i \geq l + x$. If there is at least one value i such that $(\sum b)[i] \geq (\sum b)[l]$.
- This value i only exists if the maximum of $(\sum b)[l+x], (\sum b)[l+x+1], (\sum b)[l+x+2], (\sum b)[l+x+3], \ldots, (\sum b)[n]$ is $\geq (\sum b)[l]$.
- If you compute this naively, you will arrive at a $O(n^2 \log n)$ solution for the problem. That's rather unsatisfying, since our best algorithm so far is $O(n^2)$.

Binary Searching for the Answer

- To speed this up, notice that the step where we take the maximum of $(\sum b)[l+x], (\sum b)[l+x+1], (\sum b)[l+x+2], (\sum b)[l+x+3], \ldots, (\sum b)[n]$ is slow.
- Observe, however, that we are taking the maximum over a *suffix* of the original array.
- We fortunately can precompute the maximum for every possible suffix in $O(n)$ using a variation of the previously discussed prefix sum technique!
- Therefore, since we iterate through all values of l (there are $O(n)$ of them), then find the suffix maximum for each l in $O(1)$ through precomputation, the total complexity of the function g is $O(n)$.

Binary Searching for the Answer

- This gives us a final complexity of $O(n \log n)$, which is fast enough, yay! 🥳
- The implementation for this problem is quite long. Check the GitHub for the implementation: Implementation
- You may also want to see the implementation for the brute force for this problem as well 👀

Homework

- Check the Reboot Website for your homework this week. As usual, feel free to ask for help from your fellow trainees or from the trainers through the Discord server. We're always here to help 😄