Binary Search

Sebastian Jeon, Sameer Pai

September 12, 2018

"log factors don't really matter though" - Famous Last Words

§1 Theory and Examples

Consider the following problem:

Example 1.1

Let $a_1, a_2, \cdots, a_n$ be an increasing sequence of integers. You are given an integer k that is in this sequence, but you don't know where it is in the sequence. Find the index i such that $a_i = k$.

What is naive solution?????
A naive solution to this problem is to simply iterate through the sequence and check if each a_i is equal to k. Since every element is checked in the worst case, the time complexity of this algorithm is $\mathcal{O}(n)$.

However, as the array is already sorted, there is a much faster way of answering this question. Suppose we compare k to some index i, and we find that $a_i < k$. Then we know that everything before a_i in the array is also less than k, and so no longer need to consider them. An analogous statement holds if $a_i > k$.

This motivates a **divide and conquer** algorithm: By comparing the midpoint of the array to k, we can determine whether k is in the first half or the second half of the array, and repeating this will lead us to the answer. Formally, we implement this as algorithm as follows:

```
1  int binary_search(int l, int r, int k) {
2      // we know k is among a[l], a[l+1], ... , a[r]
3
4      // arrays of size 1 or 2 (which can't be reduced further)
5      if (r-l <= 1) {
6          if (a[r] == k)
7              return r;
8          else
9              return l;
10     }
11     int mid = (l+r)/2; // middle index
12     if (a[mid] == k) // found
13         return mid;
```

```
14      if (a[mid] > k) // in left half
15          return binary_search(l, mid, k);
16      if (a[mid] < k) // in right half
17          return binary_search(mid, r, k);
18  }
```

Now, what is the time complexity of binary search? It seems a lot better than linear search, but how much better? Every time we call the function, there is an $\mathcal{O}(1)$ comparison, and the size of the array is split in half. So how many times can we split the array in half before its size becomes 1 or 0? If the size of the original array is N, then splitting it in half one gives us $\frac{N}{2}$. Splitting it in half twice gives us $\frac{N}{2^2}$. Splitting it in half t times gives us $\frac{N}{2^t}$. Now, we want to find the t such that $\frac{N}{2^t} \approx 1$. Solving this equation, we get $t \approx \log_2 N$. Therefore, the complexity of this algorithm is $\mathcal{O}(\log N)$.

Binary Search is a very common technique in USACO Silver contests. For example, consider USACO 2016 December Silver #1. This problem asks for the number of haybales in a large array that are between a specified left and right bound. For this problem, searching through the entire list of haybales would take way too long. However, if we first sort the haybales in increasing order, then binary search for the leftmost haybale to the right of A and the rightmost haybale to the left of B, we can answer each of the queries in $\mathcal{O}(\log n)$ time.

§2 Implementation Details

Sometimes, binary searching can be used to find elements in a list other than those equal to k. For example, in the previous problem, we said that we would binary search for the least element greater than A. So how do we do this? This is where two very useful functions in C++ come in: `lower_bound()` and `upper_bound()`. `lower_bound(s.begin(), s.end(), k)` returns a pointer to the first element in vector s that is at least k. On the other hand, `upper_bound(s.begin(), s.end(), k)` return a pointer to the first element in s that is strictly greater than k. So to find the first haybale that is to the right of A, we can write

```
auto l = lower_bound(bales.begin(), bales.end(), A)
```

And to find the last haybale that is to the left of B, we can write

```
auto r = upper_bound(bales.begin(), bales.end(), B) - 1
```

Finally, to find the total number of valid haybales, we can just use `r - l + 1`.

§3 Binary Search Over Answer

Binary searching has uses other than searching in a list. It can also be used to find the smallest value satisfying a condition. Let's say we have a boolean function $P(x)$ that satisfies the following conditions:

- If $P(x)$ is true, then $P(x+1)$ is true (and therefore so is $P(x+2)$, and so on).
- We know that $P(0)$ is false and $P(N)$ is true for some N.
- We can compute $P(i)$ quickly (usually $\mathcal{O}(1)$ or $\mathcal{O}(N)$). For the sake of this example, we'll assume we can do it in $\mathcal{O}(1)$.

Now, how quickly can we find the first positive value of x such that $P(x)$ is true? The naive solution would be to start at $x = 1$ and iterate to N, checking $P(x)$ every time. However, this is $\mathcal{O}(N \cdot \text{time to check } P(i))$. Can we do it faster?

Using the principles of binary search, we can do it in $\mathcal{O}(\log N \cdot \text{time to check } P(i))$. We start by considering the entire range from 0 to N. We then check if $P(\frac{N}{2})$ is true. If it is, then we know that our desired value has to be in the range from 0 to $\frac{N}{2}$. Otherwise, the answer has to be between $\frac{N}{2} + 1$ and N. Either way, we cut the range in half each time, meaning that the overall time complexity is again $\mathcal{O}(\log N \cdot \text{time to check } P(i))$. Here's the code:

```
int bin_search(int l, int r) {
    if(r - l <= 1) {
        if(check(l)) return l;
        else return r;
    }
    int mid = (l + r) / 2;
    if(check(mid)) {
        return bin_search(l, mid);
    }
    else {
        return bin_search(mid + 1, r);
    }
}
```

Here, `check(int i)` is a boolean function that checks whether $P(i)$ is true. To binary search from 0 to N, we have to call `bin_search(0, N)`.

Let's look at an example to demonstrate this concept:

Example 3.1 (USACO Silver, January 2017)
Problem Link

In this problem, we are asked to find the smallest value of K that satisfies some constraint. Any problem asking for the smallest value of something should immediately make you think of binary search. Let's verify the three conditions we need in order for a binary search to be successful:

- If a stage of size K is fast enough, then a stage of size $K + 1$ is obviously at least as fast.
- More on this later, but we can do this in $\mathcal{O}(N \log N)$.
- Clearly a stage of size 0 is invalid, and we are guaranteed in the problem that if $K = N$, then the show will be fast enough.

So, all three constraints are valid, and so we can find the answer in $\mathcal{O}(\log(N) \cdot (N \log N)) = \mathcal{O}(N \log^2 N)$.

Now, how do we check a value of K? Well, we can just simulate the dance show on a stage of size K. At every step, we want to find and remove the cow that ends earliest.

This motivates the use of a priority queue, which supports both of these operations. We keep track of the current time, and insert the next cow with the time that it will end. The implementation details are out of the scope of this handout, but the important part is the binary search.

§4 Practice Problems

1. USACO 2016 January Silver #1

2. USACO 2017 Open Silver #2

3 (hard). (2014 MIT PRIMES Problem Set) John's secret number is between 1 and 2^{16}, and you can ask him "yes or no" questions, but he may lie in response to one of the questions. Explain how to determine his number in 21 questions.

4 (hard + graph). USACO 2008 January Silver #3