# Binary Search and Prefix Sums

10 February 2016

# Binary search

You have a monotone function f : N→R (increasing or decreasing)

Given y, find x so that f(x) = y

Possible variations:

- find smallest x so that $f(x) >= y$
- find largest x so that $f(x) <= y$

Examples of f :

- f(x) = x^3/2+10
- A sorted vector [0, 1, 2, 4, 4.2, 4.2, 10, 12]
- Functions that are harder to evaluate but we can prove that they are increasing

# “Brute force” solution

Just check for every $x$ if $f(x) = y$

- Complexity O(N) (where N is the size of the domain)
- Easy to write, but very, very slow
- Doesn’t even use the fact that f is increasing

# Binary search solution

Check for $f(mid)$ where mid is the middle of $f$'s domain: [left,right]

- If $f(mid) = y$, good you solved it
- If $f(mid) < y$, then y can't be in the range [left,mid], so our search interval becomes [mid+1,right]
- If $f(mid) > y$, then y can't be in the range [mid,right], so our search interval becomes [left,mid-1]

Repeat until $f(mid)=y$. If this doesn't happen and the length of our search interval gets smaller than 1 (left > right), then x doesn't exist

Complexity O(log(N)) (N is the size of the domain, so right-left+1)

# Largest x with $f(x) <= y$

Check for $f(mid)$ where mid is the middle of $f$'s domain: [left,right]

- If $f(mid) <= y$, store mid as a potential candidate, then check whether there are even bigger solutions by searching in the interval [mid+1,right]
- If $f(mid) > y$, then y can't be in the range [mid,right], so our search interval becomes [left, mid+1]

Repeat until left > right, then return the value of the variable storing the potential candidate (since the latest update will correspond to the largest value that respected $f(mid) <= y$). If the value is null (or is still equal to its assigned initial value that is out of the domain), then there is no solution.

# Problem 1

Given an array v with N elements, answer M queries of the type:

How many numbers are between x and y?

Elements in vector and number of queries: about 100,000

# Solution 1

We start by sorting the array: N*log(N)

For each query:

- We find the smallest position posLeft in which v[posLeft] >= x
- We find the largest position posRight in which v[posRight]<= y
- If posRight >= posLeft the answer will be posRight – posLeft + 1
- Otherwise, the answer is 0
- O(log(N) ) per query

# Problem 2

You are working at a GPS company and they want you to add a new feature for truck drivers. The truckers like having really heavy trucks, so when going from city A to city B, they don’t care about how much they have to drive, they care only about how heavy their truck can be, while still abiding by the city weight laws. (Each city will have a weight limit).

More formally, you have N cities and M roads between them. Each of the N cities has an integer representing the max weight. You want for a fixed pair of cities A and B to find a route that maximizes the minimum weight.

# Solution 2

The key observation here is that if we can get from city A to city B with a weight W then we can also do that with weight W-1. So, our function f(W) will look something like: [1,1,1,1,1,1,1,1,0,0,0,0,0,0]. 1 = possible.

So we need to find the rightmost 1. We do a binary search for it, and with a fixed weight W, we'll remove the nodes with limit smaller than W, and check if A and B are still connected.

This will take O(logN * (N + M) ).

# Problem 3

Given a number $y$, find its cubic root, with an error of maximum 10^(-6).

The number is around 10^18.

# Solution 3

Let's create the function $f(x)$ = $x$^3, and let eps = 10^(-6)

We can now do a slightly modified binary search on $f$, for $f(x) = y$

Check for $f(mid)$ where mid is the middle of $f$'s domain: [left,right]

- If $f(mid)$ >= $y$, store mid as a potential candidate, then check for smaller solutions by searching in the interval [left,mid-eps]
- If $f(mid)$ < $y$, then y can't be in the range [left , mid], so our search interval becomes [mid + eps, right]

Stop when right - left <= eps

Observation: In practice, precision doesn't allow for correct evaluation of expressions like f (mid) >= y, so instead we should check f(mid) >= y - val, with val really small (e.g. 10^(-12))

# Problem 4

You have a sorted array of distinct integers (let’s call it v).

Find a position x so that v[x] = x.

# Solution 4

Let’s binary search!

We check the usual mid = (left+right) /2.

- If v[mid] = mid , we are done
- If v[mid] > mid, then v[x] > x for any x > mid, so we search in [left,mid-1]
  Explanation: since v is a sorted array of DISTINCT integers, we have:
  v[x] >= v[mid] + x - mid > mid + x - mid = x

- If v[mid] < mid, then v[x] < x for any x < mid, so we search in [mid+1, right]
  Explanation: v[x] <= v[mid] - (mid - x) < mid - mid + x = x

# Problem 5

You have a sorted array with distinct elements that has been shifted an unspecified amount of times, find if x is in the array

Example : [7,8,11,13,14,1,2,5]

# Solution 5

Example : [7,8,11,13,14,1,2,5]

We can binary search for the point the array changes (in this example v[4] = 14).

- If v[mid] > v[mid+1], mid is the point we are looking for.
- If v[mid] < v[0], then the change happened to the left of mid
- If v[mid] > v[0], then the change has not happened yet

We now have 2 sorted arrays where we can do a usual binary search.

# Problem 6

You have 2 sorted arrays. Find the k-th element in their sorted merge.

Example:

a = [1, 4, 6, 10]
b = [2, 2, 5, 9]
Sorted merge: [1, 2, 2, 4, 5, 6, 9, 10]

# Solution 6

Idea 1: Actually merge them

- O(N+M) (where N is the size of a, M is the size of b)

Idea 2: Binary search the result in a, then to check if we need to go to the left or right, binary search how many numbers in b are smaller than the value. If a solution is not found, swap a and b in the above reasoning

- O(log N * log M)

Idea 3: When looking at a value a[mid], check if it's bigger or smaller than the k-th by comparing it to b[k-mid] and b[k-mid+1]. For the desired solution we should have b[k-mid] <= a[mid] <= b[k-mid+1]. Repeat for b.

- O(logN + logM)

# Prefix sums

Given a vector x[1], x[2], … x[N], the prefix sums are:

s[i] = x[1] + x[2] + …. + x[i], with i from 1 to N

Very easy concept, but extremely powerful and useful.

Without losing information, we can represent a vector through its prefix sums.

| indices | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| x | 2 | -1 | -3 | 4 | 1 |
| s | 2 | 1 | -2 | 2 | 3 |

# Problem 1

Given a vector v of N elements, answer M questions of the type:
What is the sum of the elements between v[i] and v[j]? (inclusive)

N and M are both around 1,000,000

# Solution 1

Before the queries, compute the prefix sums: s[1], s[2], …., s[N].

For each query:

- if i = 1 return s[ i ]
- otherwise return s[ j ] - s[i - 1]

Considering s[0] = 0 removes the need for special handling of the case i = 1.

# Problem 2

You are given a vector v with N elements.
Find the subsequence (interval) with the maximum sum.

N is around 1,000,000

Example: 5, -6, 3, 4, -2, 3, -3
Answer: [3, 4, -2, 3] with sum 8

# Solution 2

If all elements are negative, the answer is 0, and this is the only case where the answer corresponds to an empty subsequence.

We want to maximize s[ j ] - s[i - 1], with 1<= i <= j <= N. In other words, we want to maximize f( j ) with 1 <= j <= N, where f( j ) = max (s[ j ] - s[ i ]) with 0 <= i < j, meaning that f( j ) = s[ j ] - min (s[ i ]) with 0 <= i < j.

We can do this with a single pass through the array, where at each step:

- we update the result if s[ j ] - minSum is greater than the current result
- we update minSum if s[ j ] is smaller than minSum

Complexity: O(N)

# Problem 3

Given a vector v of N elements and a value val:
How many non-empty subsequences of v have the sum equal to val?

N is around 200,000

This was actually problem C from the contest you worked on last week

# Solution 3

We can translate this into a prefix sum problem:

For how many $0 \le i < j \le N$ do we have $s[j] - s[i] = val$?

For a fixed j, the problem becomes: For how many $0 \le i < j$ is $s[i] = s[j] - val$? which can be answered with a hash map of (value $\rightarrow$ number of occurrences), or even a frequency vector if the range of s[ j ] is small enough.

Solution is in O(N) with a single pass through the vector, where at each step:

- we add to the result the number of occurrences of s[ j ] - val
- we update the map / frequency vector to take into account s[ j ]


That's all Folks!