Ternary Search

Table of Contents

We are given a function $f(x)$ which is unimodal on an interval $[l; r]$. By unimodal function, we mean one of two behaviors of the function:

1. The function strictly increases first, reaches a maximum (at one point or at a segment), and then strictly decreases.

2. The function strictly decreases first, reaches a minimum, and then strictly increases.

In this article we will assume the first scenario, and the second is be completely symmetrical to it.

The task is to find the maximum of function $f(x)$ on the interval $[l; r]$.

Algorithm

Consider any 2 points m_1, and m_2 in this interval: $l < m_1 < m_2 < r$. We evaluate the function at m_1 and m_2, i.e. find the values of $f(m_1)$ and $f(m_2)$. Now, we get one of three options:

- $f(m_1) < f(m_2)$

  The desired maximum can not be located on the left side of m_1, i.e. on the interval $[l; m_1]$, since either both points m_1 and m_2 or just m_1 belong to the area where the function increases. In either case, this means that we have to search for the maximum in the segment $[m_1, r]$.

- $f(m_1) > f(m_2)$

  This situation is symmetrical to the previous one: the maximum can not be located on the right side of m_2, i.e. on the interval $[m_2; r]$, and the search space is reduced to the segment $[l; m_2]$.

- $f(m_1) = f(m_2)$

  We can see that either both of these points belong to the area where the value of the function is maximized, or m_1 is in the area of increasing values and m_2 is in the area of descending values (here we used the strictness of function increasing/decreasing). Thus, the search space is reduced to $[m_1; m_2]$. To simplify the code, this case can be combined with any of the previous cases.

Thus, based on the comparison of the values in the two inner points, we can replace the current interval $[l; r]$ with a new, shorter interval $[l'; r']$. Repeatedly applying the described procedure to the interval, we can get arbitrarily short interval. Eventually its length will be less than a certain pre-defined constant (accuracy), and the process can be stopped. This is a numerical method, so we can assume that after that the function reaches its maximum at all points of the last interval $[l; r]$. Without loss of generality, we can take $f(l)$ as the return value.

We didn't impose any restrictions on the choice of points m_1 and m_2. This choice will define the convergence rate and the accuracy of the implementation. The most common way is to choose the points so that they divide the interval $[l; r]$ into three equal parts. Thus, we have

$$m_1 = l + \frac{(r - l)}{3}$$

$$m_2 = r - \frac{(r - l)}{3}$$

If m_1 and m_2 are chosen to be closer to each other, the convergence rate will increase slightly.

Run time analysis

$$T(n) = T(2n/3) + 1 = \Theta(\log n)$$

It can be visualized as follows: every time after evaluating the function at points m_1 and m_2, we are

essentially ignoring about one third of the interval, either the left or right one. Thus the size of the search space is $2n/3$ of the original one.

Applying Master's Theorem, we get the desired complexity estimate.

The case of the integer arguments

If $f(x)$ takes integer parameter, the interval $[l; r]$ becomes discrete. Since we did not impose any restrictions on the choice of points m_1 and m_2, the correctness of the algorithm is not affected. m_1 and m_2 can still be chosen to divide $[l; r]$ into 3 approximately equal parts.

The difference occurs in the stopping criterion of the algorithm. Ternary search will have to stop when $(r - l) < 3$, because in that case we can no longer select m_1 and m_2 to be different from each other as well as from l and r, and this can cause infinite iterating. Once $(r - l) < 3$, the remaining pool of candidate points $(l, l + 1, \ldots, r)$ needs to be checked to find the point which produces the maximum value $f(x)$.

Implementation

```
double ternary_search(double l, double r) {
    double eps = 1e-9;                  //set the
    while (r - l > eps) {
        double m1 = l + (r - l) / 3;
```

```
        double m2 = r - (r - l) / 3;
        double f1 = f(m1);      //evaluates th
        double f2 = f(m2);      //evaluates th
        if (f1 < f2)
            l = m1;
        else
            r = m2;
    }
    return f(l);                        //return t
}
```

Here `eps` is in fact the absolute error (not taking into account errors due to the inaccurate calculation of the function).

Instead of the criterion `r - l > eps`, we can select a constant number of iterations as a stopping criterion. The number of iterations should be chosen to ensure the required accuracy. Typically, in most programming challenges the error limit is 10^{-6} and thus 200 - 300 iterations are sufficient. Also, the number of iterations doesn't depend on the values of l and r, so the number of iterations sets required relative error.

Practice Problems

- Codechef - Race time
- Hackerearth - Rescuer
- Spoj - Building Construction
- Codeforces - Weakness and Poorness
- LOJ - Closest Distance

- GYM - Dome of Circus (D)
- UVA - Galactic Taxes
- GYM - Chasing the Cheetahs (A)
- UVA - 12197 - Trick or Treat
- SPOJ - Building Construction
- Codeforces - Devu and his Brother