Ternary search algorithm

Source: ***https://github.com/rusevrosen/codehub.github.io/tree/main/Algorithms***

1. What is the ternary search algorithm?

Ternary search is an algorithmic technique used to find the minimum or maximum value of a function within a given range. It is a divide and conquers approach, similar to binary search, but with three points rather than two.

The basic idea of ternary search is to divide the search range into three equal parts and check if the desired value lies in the middle of the range or in one of the two outer ranges. If the desired value is in the middle, the algorithm is finished. If it is in one of the outer ranges, the algorithm repeats the process with the corresponding range.

Ternary search is generally used to find the minimum or maximum of a unimodal function, that is, a function that has a single peak or valley. It is more efficient than a linear search because it reduces the search space by one-third with each iteration, but it is slower than a binary search because it requires more comparisons.

Overall, the ternary search can be a useful tool for optimizing the performance of certain algorithms, but it is not as widely used as other search techniques such as binary search or linear search.

2. Code example.

```
#include <iostream>
#include <cmath>

// Function to be minimized
double f(double x) {
  return x*x - 3*x + 1;
}

// Ternary search function
double ternary_search(double low, double high) {
  double mid1, mid2;

  while (low < high) {
    mid1 = low + (high - low) / 3;
    mid2 = high - (high - low) / 3;
    if (f(mid1) < f(mid2)) {
      high = mid2;
    } else {
      low = mid1;
    }
  }

  // Return the minimum value in the given range
  return f(low);
}

int main() {
  // Search for minimum in the range [0, 2]
  double result = ternary_search(0, 2);
  std::cout << "Minimum value: " << result << std::endl;

  return 0;
}
```

This program will find the minimum value of the function f(x) = x^2 - 3x + 1 in the range [0, 2]. The function **f(x)** has a minimum value at **x = 1**, which is the value that the ternary search function will find and return. The output should be:

```
Minimum value: -1
```

3. Code explanation.

Here is how the program works:

1. The function **f(x)** is defined as **x^2 - 3x + 1**.
2. The ternary search function is defined to take two arguments, low and high, which represent the lower and upper bounds of the search range.
3. The ternary search function uses a while loop to repeatedly divide the search range into three parts and check the values of **f(mid1)** and **f(mid2)**, where **mid1** and **mid2** are the middle points of the range.
4. If **f(mid1)** is less than **f(mid2)**, the search range is narrowed to the interval **[low, mid2]**. If **f(mid1)** is greater than or equal to **f(mid2)**, the search range is narrowed to the interval **[mid1, high]**.
5. The while loop continues until the search range becomes small enough, at which point the minimum value in the range is returned.
6. In the main function, the ternary search function is called with the **range [0, 2]**, and the result is printed to the console.

4. Time complexity.

The time complexity of the ternary search algorithm in the C++ example that I provided is **O(log3 n)**, where n is the size of the search range. This is because the search range is reduced by one-third with each iteration of the while loop.

To see why the time complexity is **O(log3 n)**, consider the following:

1. At the start of the algorithm, the search range is **[low, high]**, where **high - low = n.**
2. On the first iteration of the while loop, the search range is divided into three equal parts, and one of the outer ranges is chosen for the next iteration. The size of the search range is reduced to **(2/3) * n.**
3. On the second iteration, the search range is again divided into three equal parts, and one of the outer ranges is chosen for the next iteration. The size of the search range is reduced to **(2/3)^2 * n.**
4. This process continues until the search range becomes small enough, at which point the while loop terminates.

The number of iterations required for the while loop to terminate is equal to the smallest integer k such that **(2/3)^k * n <= eps**, where **eps** is a small positive constant. Solving for **k** gives:

```
k = log3(n/eps)
```

Thus, the time complexity of the algorithm is **`O(log3 n)`.**

I wish you happy learning,

Your Rosen Rusev