Logarithms in the Running Time

There is a group of algorithms that require \( O(\log N) \) operations. They are based on subsequently reducing the problem by a factor of two. Such algorithms are called divide-and-conquer algorithms.

Why \( \log N \)?

A complete binary tree with \( N \) leaves has \( \log N \) levels. Each level in the divide-and-conquer algorithms corresponds to an operation, hence the number of operations is \( O(\log N) \).

1. Binary Search

Given an integer \( X \) and integers \( A_0, A_1, \ldots A_{N-1} \), which are presorted and in memory, find \( i \) such that \( A_i = X \), or return \( i = -1 \) if \( X \) is not in the list \( A_0, A_1, \ldots A_{N-1} \).

**Solution 1:** Scan all elements from left to right, each time comparing with \( X \). This algorithm requires \( O(N) \) operations. It does not take advantage of the fact that the list of integers is sorted.

**Solution 2:**

Find the middle element \( A_{\text{mid}} \) in the list and compare it with \( X \)

- If they are equal, stop (the element is found)
- If \( X < A_{\text{mid}} \) consider the left part of the list and repeat the above operations
- If \( X > A_{\text{mid}} \) consider the right part of the list and repeat the above operations.

When the list is reduced to one element not equal to \( X \), return NOT FOUND.

**Example:**

Look for 7 in the following list: 1, 7, 8, 10, 13, 17, 20, 21

1. \( A_{\text{mid}} = 13 \), \( 7 < 13 \), consider 1, 7, 8, 10
2. \( A_{\text{mid}} = 8 \), \( 7 < 8 \), consider 1, 7
3. \( A_{\text{mid}} = 7 \), \( 7 = 7 \), the element is found

Each next list whose middle element is compared with \( X \) is half the size of the previous list, thus the number of the comparisons is \( O(\log N) \), where \( N \) is the size of the original list.

Other examples of divide and conquer algorithms are the Euclid's algorithm for finding the greatest common divisor and the algorithm for computing \( X^N \).
2. **Euclid's algorithm** for finding the greatest common divisor.

Based on the observation that the GCD (greatest common divisor) of two integer numbers \( M \) and \( N, M > N, \) is the same as the GCD of \( N \) and the remainder of the integer division \( M / N. \)

**Recursion:**

```c
int gcd (long M, long N) {
    if (M % N == 0), return N;
    else return gcd (N, M % N);
}
```

The algorithm works by computing remainders. The last non-zero remainder is the answer. Here is a non-recursive implementation of the algorithm:

```c
long gcd (long m, long n) {
    long rem;
    while (n != 0) {
        rem = m % n;
        m = n;
        n = rem;
    }
    return m;
}
```

**Example:**

\( M = 24, N = 15 \)

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
We can prove that given $M > N$, the remainder $M \mod N$ is at most $M/2$.

a. case 1: $N \leq M/2$. Since the remainder is less than $N$, it would be less than $M/2$.
b. case 2: $N > M/2$. In this case the remainder would be $M - N$, and since $N > M/2$, $M - N$ would be less than $M/2$.

After two iterations the remainder appears in the first column, so after two iterations the remainder would be half of its original value. Hence the number of iterations is at most $2\log N$, i.e. $O(\log N)$

3. The Prime example: What is the probability for two numbers less than $N$ to be relatively prime (e.g. 7 and 9 are relatively prime, $\gcd(9, 7) = 1$)?

$$\text{Probability} = \frac{\text{number of prime pairs}}{\text{number of all pairs}}$$

Count the operations in the following function:

```c
double probRelPrime (int n)
{
    int rel = 0, tot = 0; // rel - number of prime pairs
                             // tot - number of all pairs
    int i, j;
    for (i = 0; i <= n; i++)
        for (j = i+1; j <= n; j++)
        {
            tot++;
            if (gcd(i,j) == 1) rel++;
        }
    return (rel/tot);
}
```

Two nested loops each runs up to $N$. The body contains a function with complexity $O(\log N)$. Hence the complexity of probRelPrime is $O(N^2 \log N)$

4. Computing $X^N$

The algorithm is based on the recursive definition of $X^N$:

$$X^N = X^*((X^2)^{N/2}) \quad \text{if } N \text{ is odd}$$
$$X^N = (X^2)^{N/2} \quad \text{if } N \text{ is even}$$
```c
long pow (long x, int n) {
    if ( n == 0) return 1;
    if ( n == 1) return x;
    if (isEven(n)) return pow ( x * x, n/2);
    else return x * pow ( x * x, n/2);
}
```

In each next recursive call we reduce the power by a factor of two. The operations are at most $2\log N$, accounting for the cases when N is odd, i.e. $O(\log N)$

Note, that there is another recursive definition that reduces the power just by 1:

$X^N = X*X^{N-1}$

Here the operations are $N-1$, i.e. $O(N)$ and the algorithm is less efficient than the divide and conquer algorithm.

5. Summary: how to count operations:

   a. single statements (not function calls) : constant $O(1) = 1$.
   b. sequential fragments: the maximum of the operations of each fragment
   c. single loop running up to N, with single statements in its body: $O(N)$
   d. single loop running up to N, with the number of operations in the body $O(f(N))$:
      $O(N*f(N))$
   e. two nested loops each running up to N, with single statements: $O(N^2)$
   f. divide-and-conquer algorithms with input size N: $O(\log N)$

6. Things to consider in algorithm analysis

   1. Correctness:
      The algorithm has to be correct, the program implementation has to be correct

      Correctness can be proved, but it is a formidable task

   2. Complexity - as discussed above, counting the basic operation.
      Worst and average cases might be considered for a refined analysis

   3. Simplicity of the algorithm

      Simple algorithms are usually not efficient.
      However, they are easy to write, debug and modify.

      If the algorithm is to be used often, then the efficiency is more important.
4. Optimality

An algorithm may be correct, but not optimal. Optimality means that the algorithm does not use more operations than necessary to solve the problem.

For example, to find the maximal number in an array with $n$ elements, we need at least $n-1$ comparisons, thus an algorithm that uses not more than $n-1$ comparisons is optimal.

For other problems we still don't have optimal algorithms. For example, to multiply two matrices the usual algorithm uses $O(n^3)$ operations, though it has been proved theoretically that the necessary operations are at least $n^2$.

Here is a table that compares the run time of algorithms of different complexity depending on the size of input (the results are obtained empirically):

<table>
<thead>
<tr>
<th>Input size</th>
<th>$33n$</th>
<th>$46n\log n$</th>
<th>$13n^2$</th>
<th>$3.4n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.0033 sec</td>
<td>0.0015 sec</td>
<td>0.0013 sec</td>
<td>0.0034 sec</td>
<td>0.001 sec</td>
</tr>
<tr>
<td>100</td>
<td>0.003 sec</td>
<td>0.03 sec</td>
<td>0.13 sec</td>
<td>3.4 sec</td>
<td>$4 \times 10^{16}$ yr</td>
</tr>
<tr>
<td>1,000</td>
<td>0.033 sec</td>
<td>0.45 sec</td>
<td>13 sec</td>
<td>56.4 min</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>0.33 sec</td>
<td>6.1 sec</td>
<td>22 min</td>
<td>39 days</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>3.3 sec</td>
<td>1.3 min</td>
<td>1.5 days</td>
<td>108 yr</td>
<td></td>
</tr>
</tbody>
</table>

Run time greater than an hour has not been tested. The listed results are estimated.