Belgian Olympiad in Informatics

Training Weekend 1 — Sunday

Algorithm Basics and Contest Strategies

Trainer:
Victor Lecomte

February 8, 2015

Contents

1 Complexity

1.1 The idea

When comparing algorithms, the most important criteria (besides correctness) are running time and memory usage. That is, a good algorithm is runs quickly and uses little RAM, even when the problem size increases.

But comparing those values directly is not a good approach. The running time, for example, depends on the programming language, the implementation, the machine that runs the program, and even the weather that day. Therefore, we need some way to compare the quality of the algorithms.

1.2 The big-oh notation

The big-oh notation expresses how the running time or memory usage increase as n grows.

For example, if n doubles, an algorithm with $O(n)$ running time will be two times slower, and an algorithm with $O(n^2)$ memory usage will use up four times as much memory. $O(1)$ means constant running time or memory usage. $O(n)$ is pronounced "big oh of n" or just "oh of n".

In general, it means the running time or memory usage will be proportional to the function in the parentheses.

More rigorously, we say an algorithm runs in $O(g(n))$ if its number of operations $f(n)$ is smaller than $c \cdot g(n)$ for some constant c with large enough n.

In the context of programming contests, only the worst case matters, but other letters, such as Θ and Ω are sometimes used to describe average-case or best-case complexity.

1.3 Determining complexity

1.3.1 Rigorous method

Let us study this simple program that computes the power a^n for some integers a and n:

```
int power(int a, int n)
{
    int p = 1;
    for(int i=0; i<n; i++)
    {
        p *= a;
    }
    return p;
}
```

To calculate the complexity, we will assume every operation runs in the same time. We have to find a function f that gives the number of operations the program will execute.

In our example, the operations are:

- `int p = 1;`
- `int i=0;`
- `i<n;` (n times)
- `i++` (n times)
- `p *= a;` (n times)
- `return p;`

Thus, in total we have $f(n) = 3n + 3$ operations.

To find the time complexity, we just take the fastest-growing term, $3n$, and then remove any constant factor, $3n \to n$. So the complexity is $O(n)$.

To find the space complexity, we can use a similar method with the number of memory locations the program uses.

1.3.2 Intuition

In many cases, we can look at the number of nested loops:

- no loop: $O(1)$,
- one loop: $O(n)$,
- two nested loops: $O(n^2)$, etc.

If the operation in the most nested loop takes non-constant time, just multiply by the complexity of that operation.

We will see some trickier cases in the next sections.

1.4 Running time

Typical time complexities are (in increasing order):

$$O(1) \quad O(\log n) \quad O(\sqrt{n}) \quad O(n) \quad O(n \log n) \quad O(n^2) \quad O(n^3) \quad O(k^n) \quad O(n!)$$

They are often called mentioned with their common names: $O(1)$ is *constant*, $O(\log n)$ is *logarithmic*, $O(n)$ is *linear*, $O(n^2)$ is *quadratic*, $O(k^n)$ is *exponential* and $O(n!)$ is *factorial*.

Modern computers can execute about one billion (10^9) actions per second, a little more if well-optimized. Most contests have a time limit of a few seconds. To determine if your program can run within the time limit, just do the math: if the number of operations in the worst case exceeds one billion or is very close, then most likely your program won't make it.

Let's apply this principle for the most common cases (with a little margin):

- $O(1)$ and $O(\log n)$ work for n as large as they can read.
- $O(n)$ typically works up to $n = 100\,\mathrm{M}$.
- $O(n \log n)$ typically works up to $10\,\mathrm{M}$.
- $O(n^2)$ typically works up to $n = 10\,\mathrm{k}$.
- $O(n^3)$ typically works up to $n = 400$.
- $O(2^n)$, or $O(n \cdot 2^n)$ can hardly pass $n = 20$.
- $O(n!)$ is usually limited to $n = 10$.

1.5 Memory use

Space complexity works the same way as time complexity, although in many cases it is more convenient to allocate the same amount of memory for all test cases, for technical reasons.

Space is rarely the limiting factor in designing algorithms, since you will usually not need more than $O(n)$ space, and any memory allocation also takes time.

You can usually count on 16 MB of space, which can store about 4,000,000 integers.

1.6 Useful examples

1.6.1 Factorization

Factorizing a single integer, or testing if it is prime, can be done using this algorithm:

```
vector<int> factorize(int n)
{
    vector<int> f;
    for(int i=2; i*i <= n; i++)
    {
        while(n%i == 0)
        {
            f.push_back(i);
            n /= i;
        }
    }
    if(n != 1)
        f.push_back(n);
    return f;
}
```

The trick here is to check divisors up to $\sqrt{n}$. If n is divisible by some factor $i > \sqrt{n}$, then it must also be divisible by $n/i < \sqrt{n}$, which has been detected already.

If at the end of the loop n is not 1, it means the remaining value is a prime, and we add it to the factors.

Since i can only take values 2 through $\sqrt{n}$, and n can only have $\log_2 n$ prime divisors, this algorithm has a worst-case running time of $O(\sqrt{n})$.

However, if you need to factorise or test for prime many integers, the Sieve of Eratosthenes is much more efficient.

Let us consider you want to factorise integers in the range $[1, n)$. The idea is to precompute the largest prime factor of every integer, so that by looking at smaller and smaller divisors of n you can factorize it completely in $O(\log n)$ time.

This is the precomputation step, assuming $n = 10^6$:

```
int n = 1000000;
int factor[1000000];

void precompute_factors()
```

```
{
    for(int i=2; i<n; i++)
    {
        if(factor[i] == 0)
        {
            for(int j=1; i*j < n; j++)
            {
                factor[i*j] = i;
            }
        }
    }
}
```

(This implementation uses the fact that arrays declared outside of functions are initialized with zeroes.)

And this is the factorizing function:

```
vector<int> factorize(int a)
{
    vector<int> f;
    while(a != 1)
    {
        f.push_back(factor[a]);
        a /= factor[a];
    }
    return f;
}
```

For obscure reasons linked to the density of primes, the precomputation runs in $O(n \log(\log n))$ time, which is very nearly linear time, while the actual factorizing function runs in logarithmic time.

1.6.2 Euclid's algorithm

The following function, called Euclid's algorithm, gives the GCD (greatest common divisor) of positive integers a and b:

```
int gcd(int a, int b)
{
    if(b == 0)
        return a;
    else
        return gcd(b, a%b);
}
```

Proving that the algorithm is correct requires some knowledge of arithmetic. We will only determine the complexity here.

The usual tricks are uneffective. There is no loop, and the number of times the function will call itself depends on a and b.

However, we notice that in the `gcd(b, a%b)` call, b is always bigger than $a \,\%\, b$. Thus, we can assume $a > b$ after the first recursive call.

There are two cases:

- If $b > a/2$, then $a \,\%\, b = a - b < a/2$.
- If $b \leq a/2$, then $a \,\%\, b < b \leq a/2$, by the definition of modulo.

We see that in both cases, $a \,\%\, b < a/2$. Since $a \,\%\, b$ is the a-parameter of the function two calls later, that parameter is divided by two every two calls. This implies that after at most $2 \log_2 n$ recursive calls, a will drop to zero. But it cannot do so because of the `if` clause.

From this we conclude the function will terminate after less than $2 \log_2 n$ calls, so it has $O(\log n)$ time complexity.

2 Algorithm design

2.1 Complete search

Complete search is the brute-force, try-them-all method of to find the answer. It is based on the "Keep It Simple, Stupid" (KISS) principle. That is, your goal is to find an algorithm that works within the time limit, not the fastest algorithm. If the stupid way works, implement the stupid way.

You should always check if complete search works before anything else. If you can try all possibilities, then just check them all to find the answer. However, it is not always obvious that it is possible.

The $O(\sqrt{n})$ factorization algorithm above is a good example of smart complete search: the naive approach is to check every integer below n to see if it is a divisor of n, but the observation that divisors above $\sqrt{n}$ don't have to be checked dramatically shrinks the number of cases to check.

The next example also shows how one can reduce the pool of different cases using a bit of insight.

2.1.1 Eight queens puzzle

The eight queens puzzle is the problem of placing eight queens on a chess board so that no two queens attack each other. As a reminder, queens in chess can move any distance horizontally, vertically or diagonally.

The naive approach is to test every position for every queen and then check if they attack each other. That gives $64^8 \approx 3 \cdot 10^{14}$ possibilities, way too much to check them all.

But we weren't very clever, we checked many situations multiple times, and sometimes there were multiple queens at the same place. Checking every way to place eight queens in no specific order, at distinct positions takes $\binom{64}{8} \approx 4 \cdot 10^9$ possibilities. Better, but still too much.

Now if we choose only one queen per row, we can take the number of combinations down to $8^8 = 12\,777\,216$ which is still a bit too big since checking a single configuration can take some time.

Finally, we can take this idea further by generating all possible sets of the columns they are placed in, so that no two queens are in the same column. The number of permutations is $8! = 40\,320$, so you can now apply complete search.

2.2 Divide and conquer

Divide and conquer is a technique in which you divide the problem it in two, solve the smaller parts, and then merge the results to solve the larger problem.

The merging step is the core of the algorithm. It has to be very quick for the algorithm to be efficient.

Let us see an example straight away.

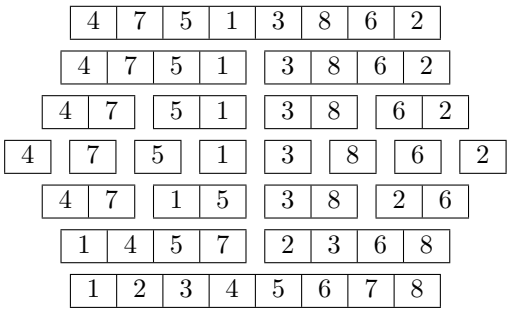
2.2.1 Merge sort

Merge sort is probably the best-known example of divide and conquer. It sorts the elements of an array of numbers in increasing order.

It does so by first separating the array in two halves, then sorting each individual half, and then merging both sorted halves into the full sorted array.

To sort the halves, the function will recursively call itself until the bits all have length one. (A nice property of arrays of one element is that they are always sorted.)

Here is an illustration of how the algorithm splits the array and then merges it back:



We can see that at every step the length of the parts is divided by two, so the algorithm has $\log_2 n$ "levels".

For the actual merging part, the idea is to iterate through both halves at the same time, each time selecting the smaller of the two numbers to go into the merged array:

```
vector<int> merge(vector<int> a, vector<int> b)
{
    vector<int> merged;
    int i=0, j=0;
    while(i < a.size() || j < b.size())
    {
        if(j == b.size() || (i < a.size() && a[i] < b[j]))
        {
```

```
            merged.push_back(a[i]);
            i++;
        }
        else
        {
            merged.push_back(b[j]);
            j++;
        }
    }
    return merged;
}
```

This merging algorithm can be used in other algorithms, and is worth remembering.

Since the merging runs in linear time, and for each of the $\log_2 n$ levels the total length to be merged is n, the algorithm has $O(n \log n)$ time complexity.

2.2.2 Binary search

Although it is not a proper application of divide and conquer, binary search is similar in the approach.

When searching for an element in a list, look at one element in the middle. If you can tell if the element is to the left or to the right of it, then you can look for it in that half and forget about the other one completely.

This way you cut the range in two repeatedly until there is only one element left: the one you are searching for.

If the operation that determines which side to go runs in constant time, then the time complexity of the whole search is $O(\log n)$.

This is binary search applied to finding the position of an element in a sorted array:

```
int find_pos(vector<int> a, int k)
{
    int lower = 0;
    int upper = a.size();

    while(lower+1 < upper)
    {
        int middle = (lower+upper)/2;
        if(a[middle] <= k)
            lower = middle;
        else
            upper = middle;
    }
    return lower;
}
```

(Here, `lower` and `upper` are the inclusive lower bound and exclusive upper bound of the possible location of k.)

Binary search is very powerful in any problem where you can tell in which direction the solution must be.

3 Contest strategies

This section is heavily based on USACO's "Crafting Winning Solutions" module.

Learning how to design algorithms is useful, but it is also important to have strategies to approach problems and perform well in competitions.

3.1 Game plan

Before a contest, you should always plan in your head what you are going to do. That way you will get less nervous, you will know what to do at all times, and you will avoid wasting your time on one problem, thinking or debugging.

You should at least read *all* problems first, before you begin coding anything. Take note of the ideas you have, the algorithms and bits of algorithms you find, the data structures you will need, tricky details, etc.

- Brainstorm many possible algorithms, then pick the stupidest that works.
- Do the math: find the time and space complexity and plug in the numbers to see if it fits the limits.
- Try to break the algorithm: use special or degenerate test cases. It might not be as efficient as you thought it was.
- Order the problems: shortest job first, *in terms of your effort* (shortest to longest; done it before, easy, unfamiliar, hard). That way you can secure some easy points and it leaves you time for the harder problems.

When you have chosen a problem to work on, you should:

- Finalize the algorithm.
- Create test data for tricky cases.
- Write the data structures, if needed.
- Write the input routine and test it (print what you read).
- Write the output routine and test it.
- For more intricate algorithms (e.g. IOI), write and debug the code *one section at a time.*
- Get it working and verify correctness: create many test cases that you solve by hand.
- Submit. If incorrect, go back to the previous step. Don't forget to comment out any debugging output or asserts (do not remove it, chances are you'll need it again).
- If it exceeds the time limit, optimize progressively and keep all versions. If it is still too slow, cry.

3.2 Approaching problems

When first reading a problem, you should always keep in mind all the techniques and approaches you know, and try to apply them all one by one. It's easy to get stuck in one vision of the problem.

Here is what that checklist might look like:

- Read the problem statement *carefully*, read the *samples*, and make sure you *understand* them. Don't spend an hour solving a problem that doesn't exist.
- Focus on the limits, not only for the main variables (N, M...) but also for the quantities themselves. If they are quite small, there might be an algorithm that goes through all possible values.
- Can you use try all possibilities to find the solution? If not, can you use symmetries and patterns to shrink the number of possibilities?
- Can you rephrase the problem in terms of other concepts? Can you reduce it to some simpler problem?
- Does that problem remind you of any other one you solved? Are some parts of the solution common?
- Can you cut the problem in two, solve both halves and merge the results? If so, divide and conquer.
- Can you tell if the solution is on the left or right of some point? If so, binary search.
- Does solving the problem backwards help?
- Can you do some precomputation before to speed up something you do repeatedly?

3.3 Time management

In a contest, there is very often a moment where you have been stuck on a problem for some time, and you start panicking, and you really don't know what the heck to do with your life anymore. And that's okay. In that situation, the central question is "When do you spend more time debugging a program, and when do you cut your losses and move on?".

Consider these issues calmly:

- How long have you spent debugging it already?
- What type of bug do you seem to have?
- Is your algorithm wrong? (Yes, that's hard to admit.)
- Do your data structures need to be changed?
- Do you have any clue about what's going wrong?
- A short amount (20 minutes) of debugging is better than switching to anytheng else; but you might be able to solve another from scratch in 45 minutes.
- When do you go back to a problem you've abandoned previously?

- When do you spend more time optimizing a program, and when do you switch?
- Consider from here out—forget the pain you've been through and focus on the future: how can you get the most points in the hour with what you have?

Then take a decision and hold on to it.

4 Problem Set

The following problems may or may not be related to the course material.

4.1 UVa

- UVa 10487
- UVa 10567
- UVa 11413
- UVa 10892
- UVa 11565

4.2 Others

- `http://codeforces.com/problemset/problem/388/A`
- `http://www.codechef.com/problems/INSOMA3`
- `http://www.codechef.com/problems/PRIME1`