Belgian Olympiad in Informatics

Easter Training — Day 3

Competitive programming algorithms and Contest Strategies

Trainer:
Floris Kint

April 16, 2014

Contents

1 Introduction **1**

2 Carnival Review **2**
2.1 STL 2
2.2 More 2

3 Bruteforcing **2**
3.1 Backtracking 2
3.2 Pruning 3
3.3 Exercises 4

4 Knapsack **4**
4.1 Integer single knapsack 4
4.2 Multiple knapsack 4
4.3 Exercises 5

5 Computational Geometry **5**
5.1 Vectors 5
5.2 Basic Geometry 5
5.3 Convex Hull 6
5.4 Exercises 7

6 Number Theory **7**
6.1 Algorithms 7
6.2 Primes 8
6.3 Modulo 8
6.4 Exercises 9

1 Introduction

During today's training we'll study a few specific topics that can be useful at programming contests. We'll start with a quick review of what we've seen about C++ during the Carnival training. Then, there's a section about bruteforcing. Sometimes you can get away with a bruteforce algorithm, but you'll probably need to optimize it a bit. The fourth section is about the Knapsack problem. To finish, there's a section about computational geometry and a section about number theory. Both of these don't occur frequently at IOI problems, but they are interesting to know and useful to compete at other programming contests (during your training).

2 Carnival Review

2.1 STL

The STL library provides lots of useful datastructures and algorithms. It is very important to know what's in it and where to find it.

Make sure you've spent some time at `http://www.cplusplus.com/reference/stl` to get familiar with the structure of STL datastructures. You should know the important methods and functions by heart. Also read through `http://www.cplusplus.com/reference/algorithm/`. Keep in mind that the g++ compiler at IOI might not support all functionalities mentioned in this documentation. Know which version of the compiler and C++ language you'll use and make sure you know how to find out whether that version supports a certain datastructure/function/...

2.2 More

Make sure you understand all of these concepts: STL I/O, C-style I/O, pointers, bitmasks, datatypes, C math functions, references. The best way to get familiar with these topics is to make as many exercises as possible. During a contest, you should not lose time on trivial stuff like this. Also, know how to find information about those topics as fast as possible. Again, don't lose time searching for information. Be well-prepared and be able to find the information you need very fast.

3 Bruteforcing

For some problems, even an 'optimized bruteforce' solution can be sufficient to score (at least some) points. When optimizing a bruteforce algorithm, the search space should be minimized.

3.1 Backtracking

Consider the 8 queens problem (Carnival training Day 1). A naive way to find a suitable board configuration would be to generate 2^{8*8} board configurations (each consisting of booleans representing whether a queen is on a certain field of the board) and check all boards until a good configuration has been found. A smarter way is to (recursively) put 8 queens on the board. Consider the following pseudo-code:

```
N = 10
attacked_by = [[0]*N for i in range(N)]
def add_attacker(start_row, start_col, val):
  for j in xrange(N-start_col):
    attacked_by[start_row][start_col+j] += val
    if start_row+j < N:
      attacked_by[start_row+j][start_col+j]+=val
    if start_row-j >= 0:
```

```
      attacked_by[start_row-j][start_col+j]+=val
result = [0]*N
def queens(current_col):
  global result
  if current_col == N:
    return True
  for i in xrange(N):
    if attacked_by[i][current_col] > 0:
      continue
    result[current_col]=i
    add_attacker(i, current_col, 1)
    if queens(current_col+1):
      return True
    add_attacker(i, current_col, -1)
  return False
queens(0)
```

This algorithm only starts putting a queen in a certain column if all previous columns are filled in a good way (that is, the previously fixed queens can't attack one another). If the algorithm finds a field in the current column that is not under attack of one of the other queens yet, it puts the current queen on that spot and recursively calls itself to fill the other columns. As soon as the algorithm can't find a good spot to put the next queen, it terminates itself to reposition the previous queens and to try again. This technique is called backtracking. It's useless to look further, so as soon as you know you won't find the answer in a branch of the search tree, you can abandon that branch.

3.2 Pruning

Pruning is the technique to determine if your algorithm can backtrack. Consider a problem where you have to maximize the value of a function, within a set of constraints. Two frequent ways of pruning are:

- When trying to maximize a function, check if it is still theoretically possible to get a higher value than the current best achieved value, using the current partial solution proposal. If not, backtrack!
- When searching in the search tree, check if it is still possible to fulfill all requirements

VPW 2014 - sticks: `http://www.vlaamseprogrammeerwedstrijd.be/current/opgaven/cat3/memorysticks/memorysticks.pdf`

- **Constraints**: all files must be put on sticks
- **Function to maximize**: (minimize in this case, so maximizing its opposite): free space on used memory sticks

The following two techniques are particularly known:

$\alpha-\beta$ pruning Alpha-Beta pruning can prune a minmax-tree. A minmax-tree can simulate a turn-by-turn game between two players.

We consider a game where the actions of the two players affect a number. Player A tries to maximize the resulting score, player B tries to minimize the resulting score at the end of the game.

The $\alpha - \beta$ pruning technique keeps track of the lower bound α and upper bound β of the resulting score in a certain branch. As soon as for a certain node $\beta \leq \alpha$, the algorithm can backtrack.

`http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning`

Branch and Bound Consider a function f to maximize. Branch-and-bound pruning first calculates upper bounds and lower bounds for each of the children of the current node. These bounds are sorted in order of "most promising". Children with an upper bound lower than the lower bound of another child, don't need to be investigated as they won't return the final value and hence can be omitted.

`http://en.wikipedia.org/wiki/Branch_and_bound`

3.3 Exercises

- `http://www.vlaamseprogrammeerwedstrijd.be/current/opgaven/cat3/memorysticks/memorysticks.pdf`

4 Knapsack

4.1 Integer single knapsack

Given:

- a knapsack strong enough to contain a weight W
- a set of S objects, each having a weight w_i and a value v_i

Find: the maximal value you can achieve by putting items of the set in the knapsack, without exceeding a total weight of W

As the weights and values are all integers, dynamic programming can be used as long as the total weight W is not extremely large.

```
best = [0]*W
for i from 0 to S-1:
  for j from W-w[i] to 0 step -1:
    best[j+w[i]]=max(best[j+w[i]], v[i]+best[j])
```

Time complexity: O(S*W) Additional space complexity: O(W)

4.2 Multiple knapsack

When there are multiple knapsack the items can be divided into, the search space usually gets too big for dynamic programming. Complete search with

pruning can be used in this case. Example: Section 3: Bruteforce (sticks)

4.3 Exercises

- `http://www.vlaamseprogrammeerwedstrijd.be/current/opgaven/cat4/cupcakes/cupcakes.pdf`

5 Computational Geometry

5.1 Vectors

Points in 2D-space can be represented using a coordinate tuple (x, y).

Points in 3D-space can also be represented using a coordinate tuple (x, y, z).

Vector addition: $(x_1, y_1, z_1) + (x_2, y_2, z_2) = (x_1 + x_2, y_1 + y_2, z_1 + z_2)$

Product with a scalar: $\lambda \cdot (x_1, y_1, z_1) = (\lambda \cdot x_1, \lambda \cdot y_1, \lambda \cdot z_1)$

Scalar product:

$$\begin{aligned} (x_1, y_1, z_1) \cdot (x_2, y_2, z_2) &= x_1 * x_2 + y_1 * y_2 + z_1 * z_2 \\ \|(x_1, y_1, z_1) \cdot (x_2, y_2, z_2)\| &= \|(x_1, y_1, z_1)\| \cdot \|(x_2, y_2, z_2)\| \cdot \cos\alpha \end{aligned}$$

Cross product:

$$\begin{aligned} (x_1, y_1, z_1) \times (x_2, y_2, z_2) &= (y_1 * z_2 - y_2 * z_1, x_2 * z_1 - x_1 * z_2, x_1 * y_2 - x_2 * y_1) \\ (x_1, y_1, z_1) \times (x_2, y_2, z_2) &= \|(x_1, y_1, z_1)\| \cdot \|(x_2, y_2, z_2)\| \cdot \sin\alpha \cdot \vec{n} \end{aligned}$$

*Where $\vec{n}$ is the unity vector perpendicular to both vectors in the direction of the middle finger of your right hand, when your thumb is pointing along the first vector's axis and your index finger is pointing along the second vector's axis.

5.2 Basic Geometry

Same side of a line Consider a line through the points $A(x_a, y_a, 0)$ and $B(x_b, y_b, 0)$. Also consider two points $C(x_c, y_c, 0)$ and $D(x_d, y_d, 0)$. We want to know if C and D are on the same side of the line. Being on the same side of a line only makes sense in 2-D, that's why the third coordinate number is zero for all four points. We don't just write them as points in 2-D space because we need the cross product, which has not been defined in 2-D.

C and D are on the same side of the line iff the angles $A\hat{B}C$ and $A\hat{B}D$ are both positive or both negative (in the range $(-\pi, \pi]$). In that range, the sines of the angles have the same sign as the angles. When looking at the definition of the cross product, it is clear that we can use the cross product of $(A - B) \times (C - B)$ and compare it to the cross product of $(A - B) \times (D - B)$. If both have the same sign, the points are on the same side of the line through A and B. Else, they're not.

Area of triangle Consider the parallelogram defined by $O(0, 0, 0)$, $A(x_a, y_a, z_a)$, $B(x_b, y_b, z_b)$ and $(A + B)(x_a + x_b, y_a + y_b, z_a + z_b)$. From the definition of the cross product, we see that the area of this parallelogram is the norm of the cross

product $A \times B$. The area of the triangle defined by $O(0,0,0)$, $A(x_a, y_a, z_a)$ and $B(x_b, y_b, z_b)$ is exactly half of the area of the parallelogram, so $\frac{1}{2}(A \times B)$

Line segment crossing Given: a line segment between A and B and another line segment between C and D. Determine whether the two line segments cross. The line segments cross iff A and B are on different sides of the second line segment and C and D are on different sides of the first line segment. Using twice the formula from the first paragraph in this section, this problem can easily be solved.

More useful geometry techniques can be found at the USACO Training Program, section 3.4.

5.3 Convex Hull

Find the smallest convex polygon containing all points on a plane.

```
def cross_product(v1, v2):
  return v1[0]*v2[1]-v1[1]*v2[0]
def subtract(p1, p2):
  return [p2[0]-p1[0], p2[1]-p1[1]]

points.sort();
def convex_hull(points, cp_val):
  current_connected = [0,1]
  for i in range(len(points)):
    current = points[i]
    previous = points[current_connected[-1]]
    if previous[0] == current[0]:
      if (current[1]-previous[1])*cp_val >= 0:
        del current_connected[-1]
      else:
        continue
    while True:
      if len(current_connected) == 1:
        break;
      v1 = subtract(points[i], points[current_connected[-1]])
      v2 = subtract(points[current_connected[-2]],\
          points[current_connected[-1]])
      cp = cross_product(v1, v2)
      if cp * cp_val >= 0:
        del current_connected[-1]
      else:
        break
    current_connected.append(i)
  return current_connected
top = convex_hull(points, 1)
bottom = convex_hull(points, -1)
result = [points[i] for i in top] +\
```

```
[points[bottom[i]] for i in range(len(bottom)-1, -1, -1)]
```

5.4 Exercises

- http://codeforces.com/contest/406/problem/D

6 Number Theory

6.1 Algorithms

GCD (http://en.wikipedia.org/wiki/Euclidean_algorithm)

```
function gcd(a, b)
    if b = 0
        return a
    else
      return gcd(b, a mod b)
```

LCM The least common multiple of two positive integer numbers a and b is the smallest positive number that is divisible by both a and b.

```
function lcm(a, b)
  return a*b/gcd(a,b)
```

Power CMath contains a built-in power function (pow) that returns a floating-point number. However, in many cases we need to calculate integer powers. In the subsection about modulo arithmetic, we'll study a slightly different algorithm.

```
function pow(base, exp)
  if exp == 0
    return 1
  tmp = pow(base, exp/2)
  tmp *= tmp
  if base % 2 == 1
    tmp *= base
  return tmp
```

Make sure that you're not calling the C Math pow function by not choosing pow as function name.

Datatypes limitations Always be careful to prevent overflow when multiplying large numbers. In many problems, you have to return the number modulo a certain number (see subsection about primes).

6.2 Primes

During the Carnival training, the Sieve of Eratosthenes was studied. We quickly revise the algorithm.

```
MAXN = 1000000
non_primes = [False]*MAXN
primes = []

for i in range(2, MAXN):
  if non_primes[i]:
    continue
  primes.append(i)
  for j in range(i*i, MAXN, i):
    non_primes[j] = True
print primes
```

6.3 Modulo

In modulo arithmetic, numbers are compared to each other using only their remainder after division with a certain number. Consider $a \equiv b \,\%\, k$. That means that a and b have the same remainder when divided by k. These numbers are considered equivalent modulo k.

Addition and multiplication are easy in modulo arithmetic. The remainder of the sum (resp multiplication) of the two when divided by k can easily be calculated.

```
function mod_sum(a, b, k)
 return (a+b)%k
function mod_mult(a, b, k)
 return (a*b)%k
```

Beware of integer overflow! Make sure $0 \leq a, b < k$ and k^2 doesn't exceed the maximum value of the used datatype.

```
function mod_sum(a, b, k)
 return (a%k+b%k)%k
function mod_mult(a, b, k)
 return ((a%k)*(b%k))%k
```

The power function in modulo arithmetic can be written as

```
function mod_pow(base, exp, p)
 if exp == 0
   return 1
 tmp = pow(base, exp/2)
 tmp *= tmp
 tmp = tmp % p
 if base % 2 == 1
   tmp *= base
 return tmp % p
```

Division is somewhat more difficult in modulo arithmetic.

The inverse of $5 \equiv 2 \,\%\, 3$ is 2 because $2 * 5 \equiv 10 \equiv 1 \,\%\, 3$.

Let's try to find the inverse of $12 \equiv 3 \,\%\, 9$. We need to find a number k such that $k * 3 \equiv 1 \,\%\, 9$. There doesn't exist an integer number that can fulfill this requirement.

By Fermat's little theorem, we know that when k is a prime number $a^n \equiv a \,\%\, k$. When dividing both sides by a^{-2}, we get $1/a \equiv a^{k-2} \,\%\, k$. As we've seen in a previous paragraph, this can be calculated in $\mathcal{O}(\log N)$. Many problem statements on Codeforces, Codechef ask you to calculate a number modulo a prime. That means that you can use this theorem to do modulo divisions.

6.4 Exercises

- `http://www.codechef.com/problems/ANUCBC`