**Four Useful Algorithms: GCD, Subsets, Permutations and Combinations**

These four algorithms are good ones to know for contest problem solving.

Be careful when applying Subsets, Permutations and Combinations. They are all exponential algorithms so they are only applicable for relative small list sizes. Experiment with all four algorithms and make sure you understand how they work and what their limits are.

**Euclidean Algorithm to find the Greatest Common Divisor (GCD) of two integers**

This algorithm computes the greatest common divisor of two integers using Euclid’s algorithm. Note that this algorithm expects that the two integers are at least 0 (the calling routine will have to handle negative numbers).

**Theory:**

Given two integers, *a* and *b*, find the largest integer that divides both evenly. In other words, find the largest integer, *g*, that divides *a* and *b* with no remainder ( and ). Alternatively, there exist integers, *x* and *y*, then when multiplied by *g* will equal a and b, respectively ( and ).

Note that we can express *a* in terms of *b*. By dividing *a* by *b*, we will obtain the quotient and . We can now express *a* as .

We can replace *a* and *b* as:

Then we arrange to solve for *r*:

So here we can see that *r* is divisible by *g*. Furthermore, we know that .

Therefore, we know that *g* will divide *r* in addition to *a* and *b*. We will use this information to speed up our computation of *g*, the GCD.

We compute GCD(*a*, *b*) by computing GCD(*b*, *r*), or noting that it would be the GCD(*b*, ). If we make sure *b* is the smaller of *a* and *b*, the computation may be even faster.

Finally, note that the GCD(x, 0) is x since and .

**Algorithm:**

// Recursive

int gcd(int a, int b)

{

if (b == 0)

return a;

else

return gcd(b, a % b);

}

*OR:*

// Non-recursive

int gcd(int a, int b)

{

if (a < b)

swap(a, b);

while (b > 0)

{

t = a % b;

a = b;

b = t;

}

return a;

}

**Note:**

The least common multiple, the smallest positive integer that is a multiple of both *a* and *b*, can be computed using the GCD. It is computed as follows:

**Extended Euclidean Algorithm**

Given that GCD(a,b) = c, it turns out that there is always a solution (in fact, an infinite number of solutions) to the equation

ax + by = c

Since c is a common factor in this equation, solving this equation is equivalent to solving the equation

a’x + b’y = 1, where a’ = a/c and b’ = b/c.

In the form of the algorithm presented below, our aim is to determine b-1 ≡ t mod n. In particular, we want to determine the value t such that:

bt ≡ 1 mod n

By definition of mod, this means we seek an integer t such that there exists an integer x with

bt + nx = 1 (notice the similarity in form to what is shown above.)

Modular inverses are very useful in solving a number of different problems in number theory.

Here is the algorithm (taken from Cryptography Theory and Practice by Stinson):

int modInv(int b, int n) {

int n0 = n;

int b0 = b;

int t0 = 0;

int t = 1;

int q = n0/b0;

int r = n0 – q\*b0;

while (r > 0) {

int temp = t0 – q\*t;

if (temp >= 0) temp = temp%n;

else temp = n – ((-temp)%n);

t0 = t;

t = temp;

n0 = b0;

b0 = r;

q = n0/b0;

r = n0 – q\*b0;

}

if (b0 != 1)

return -1; // To indicate that no inverse exists.

else

return t;

}

**Subsets**

Given a set (represented as an array of elements), produce the power set of the set (i.e. all of the subsets of the set). For example:

Topping options for a pizza: {Pepperoni, Sausage, Mushroom}

{}, {P}, {S}, {M}, {P,S}, {P,M}, {S,M}, {P,S,M}

Or as a table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Pepperoni** | **Sausage** | **Mushroom** | **Subset** |
| T | T | T | {P,S,M} |
| F | T | T | {S,M} |
| T | F | T | {P,M} |
| F | F | T | {M} |
| T | T | F | {P,S} |
| F | T | F | {S} |
| T | F | F | {P} |
| F | F | F | {} |

Note: The algorithm below goes throw the subsets in the order presented above.

**Theory:**

At each recursive level, this algorithm first generates all subsets that *include* the current element, and then generates all subsets that *exclude* the current element.

**Algorithm:**

inSubset : array[0..N-1] of boolean;

void subsets(int level)

{

if (level < 0)

{

Process Subset

}

else

{

// Order is important!

inSubset[level] = true; // This item is in the subset.

subsets(level-1); // Try all subsets of the rest of the items.

inSubset[level] = false; // Now, this item is NOT in the subset.

subsets(level-1); // Try all subsets of the rest of the items.

}

}

Call this procedure with the statement: subsets(N-1);

**Permutations**

Given a list of objects, generate all rearrangements of the objects. For example:

For a family of three (Father, Mother, Child) that wants to take a picture (all standing up), here are the different ways of lining them up:

F M C

F C M

M F C

M C F

C F M

C M F

**Theory:**

At each recursive level, this algorithm selects an unused element for the current position in the current permutation. The recursion (not the algorithm!) halts when all positions have been filled in the current permutation. The algorithm halts when all permutations have been generated.

**Algorithm:**

// Before starting, *used* should be initialize to all false

used : array[0..N-1] of boolean;

perm : array[0..N-1] of int;

void permute(int level)

{

int i; // Note that *i* is declared locally

if (level < 0)

{

Process the permutation (It is stored in *perm*)

}

else

{

for (i=0; i < N; i++)

{

if (used[i] == false) // Only place I if it’s not used yet.

{

used[i] = true; // Mark that we’ve used i.

perm[level] = i; // Put it in this place.

permute(level-1); // Permute what’s left.

used[i] = false; // Unmark i so we can try the next one.

}

}

}

}

Call this procedure with the statement: permute(N-1);

**Combinations**

Generate all combinations of *n* things taken *r* at a time. For example:

Let's assume there are four items (A,B,C,D) each costing $50. Let's say you have only $150, i.e., you can get only 3 of these 4.

A B C

A B D

A C D

B C D

If you have only $100, i.e., can get 2 of 4

A B

A C

A D

B C

B D

C D

Or let’s say 5 items, take 3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **D** | **E** |
| A | B | C |  |  |
| A | B |  | D |  |
| A | B |  |  | E |
| A |  | C | D |  |
| A |  | C |  | E |
| A |  |  | D | E |
|  | B | C | D |  |
|  | B | C |  | E |
|  | B |  | D | E |
|  |  | C | D | E |
|  |  |  |  |  |

Note: The algorithm on the following page goes through these combinations in a different order. In particular, it lists all the ones that start with E first, then without. Here is the order for this example: CDE, BDE, ADE, BCE, ACE, ABE, BCD, ACD, ABD, ABC.

**Theory:**

At each recursive level, this algorithm first generated all combinations that *include* the current element. It then generates all combinations that *exclude* the current element. The recursion halts when *r* elements have been included in the combination. The algorithm halts when all combinations have been generated.

**Algorithm:**

used : array[0..N-1] of boolean;

void combinations(int n, int r)

{

if (r == 0)

{

Process the combination (given by *used*)

}

else if (n < 0) // Here we have no more items to choose from

{

return; // so no need to process anything…

}

else

{

used[n] = true; // Use item n.

combinations(n-1, r-1); // Now we want r-1 items from the rest.

used[n] = false; // Don’t use item n.

combinations(n-1, r); // We still need r items from the rest.

}

}

Call this procedure with the statement: combinations(N-1, r);