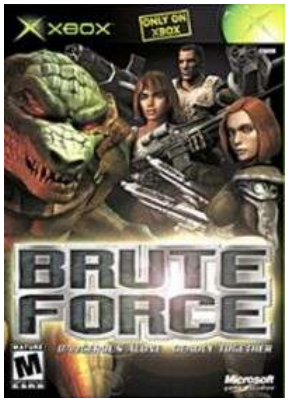
Brute Force

What is brute force?

"Brute-force algorithms are distinguished by approaching the solution of a problem in the most natural, obvious, simple, or direct method in contrast to a more clever or sophisticated way. Exhaustive enumeration is an example of a brute-force algorithm." - http://moonbase.wwc.edu/~aabyan/FAS/book/node392.html

"Describes a primitive programming style, one in which the programmer relies on the computer's processing power instead of using his or her own intelligence to simplify the problem, often ignoring problems of scale and applying naive methods suited to small problems directly to large ones. The term can also be used in reference to programming style: brute-force programs are written in a heavyhanded, tedious way, full of repetition and devoid of any elegance or useful abstraction (see also brute force and ignorance)." - http://www.catb.org/jargon/html/B/brute-force.html



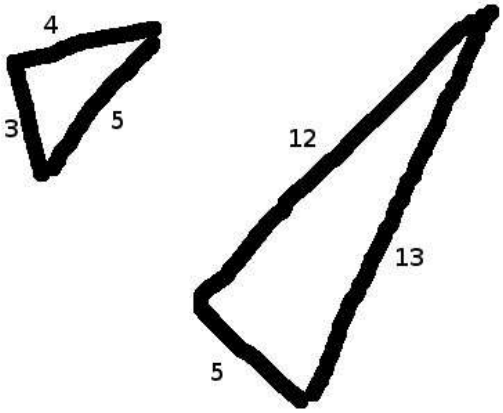
"Brute Force is a new action-packed, sci-fi squad-based shooter played from a third person perspective." - **www.replaygames.com.au/buy/brute-force-xbox.html**

Brute force algorithms are not always as bad choice. Often, brute force methods are the only way to approach a problem. They do have a legitimate place, but should not be misapplied.

Example Problem:

You need to find Pythagorean triples. These are defined as any a, b and c such that a, b, and c are all integers greater than 0, and $a^2 + b^2 = c^2$. Such a triple is considered to be below a number d if each of the three 'sides' of the triple is less than d.

The input consists of the value of d. Output the number of relatively prime triples that satisfy the conditions. d is less than one million.



Extremely naïve brute force idea:

```
for i = 1 to n
  for j = i to n
    see if i and j form the short sides of a triangle
```

Outcome: Gets sample output, but times out when you submit it for some reason?!

The correct solution:

There is a numerical method that can generate all these triangles. Medium difficulty to code, but runs very efficiently.

Brute force solution:

Notice that you don't need to try ALL possibilities. A brute force solution could be finding a way to avoid checking many possibilities. One way to do this is to notice that

$$a^2 + b^2 = c^2$$
$$a^2 = c^2 - b^2$$
$$a^2 = (c – b) * (c + b)$$

Furthermore, since each number has a unique representation as a product of primes, we can factorize the left half and apply it to the right half. If there are F prime factors of a, then there are at most 2F in a^2, and at most 2^{2F} possibilities for c-b and c+b. Finding c-b and c+b is just as good as finding c and b.

But, it is STILL too slow. There are ways to make it faster, and it is possible to speed it up enough to get it under ten seconds. This solution is messy and long.

The moral of the story: brute force is a bad idea in the wrong place.

When is brute force a good idea?

There are many situations in which brute force is well suited. This is generally when there is no better solution. Knowing this isn't so easy, but here are some tips to help decide which way to go.

1) Break the problem down

A complex problem can sometimes be expressed as many simple problems.

For example, say you are running an arranged marriage company on the net. All your customers are monogamous heterosexual xenophobes. That is, each client wants to be matched with a single person of the opposite gender in their own city. Write a program that, given a list of {name, gender, city} data, tells you whether all clients can be matched.

You can use a simple approach to solve this. Look at each individual – if there is a match, give it to them and go on to the next one. This problem is easy to break down, but if each customer had a list of other countries he was willing to consider matches from, it would be more difficult.

In some optimization problems, taking the best local solution will result in a solution that is also optimal. This is the greedy approach.

2) Eliminate impossible solutions

In some problems you can cut off extraneous possibilities. This is usually only a useful approach if you can do so dramatically. The foremost example of this is the branch and bound method. This works by traversing the solutions like a tree, and bounding at every node.


A horrible brute

3) Looking at a different aspect of the problem

Although there may be an impractical amount of possibilities to consider, perhaps you can focus on a single aspect that spans all of them. This is closely connected with dynamic programming, which uses this aspect to construct partial solutions.

If you can't break a problem down, chip sections off of it, or transform it into a new problem, then quite possibly the only option left is vanilla brute force. This sometimes happens. In fact, it happens quite a lot, but we tend to ignore these problems since they aren't very noteworthy.

Backtracking

One common way of solving brute force problems is by checking each possibility in turn, then backtracking and trying the next until a solution is reached.

Example
Your character in a computer game is collecting power ups. Each power-up modifies your points multiplier, ie, you get a 2 power up – now the points you get are doubled. Getting multiple power-ups multiplies the multiplier. Is it possible to get a certain multiplier given a set of power-ups? If not, which is the closest multiplier to it that you can get?

```
int mult;               //number of power-ups
int n;                  //the multiplier you want to make
int pup[];              //the array that holds the power-ups

long long bfind( long long mult, int toTry, long long best ){
  mult *= pup[toTry];
  if( abs( n - mult ) < abs( n - best ))
    best = mult;
  for( int i=toTry+1; i<mults; i++ ){
    best = bfind( mult, i, best );
  }
  return best;
}
```

This is a very simple example, but the principle here is clear. The way possible answers are evaluated is by a depth first search, evaluating at the leaves and saving the best answer as it backtracks towards the root.

This approach can be used for a variety of problems. For example, the traveling salesman problem can

be approached the same way:

```
int best;                    //global best distance
dist( int thisCity, int distSoFar ){
  mark this city visited
  for each city connected to thisCity
     add the distance to go to that city and recurse on it
  if there are no cities to go to, and all cities have been visited
     if dist < best
        best = dist
  mark this city unvisited
}
```

There are a couple things that are important about this example: unlike normal depth first searches, we are marking the node visited only temporarily. This is necessary for the problem, but it is a drastic change. It is what makes the TSP intractable while DFS is relatively trivial.

Another thing to notice is that we can improve this algorithm drastically. In the first example, with the power-ups, it might be the case that we are looking for a small power-up while most of the answers that we check are orders of magnitude larger. It is not very efficient to wait until we reach a leaf to check the answer; we should check at every node. This is called the branch and bound technique.