Software Science Seminar

Week 5 - Number Theory and Backtracking

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Sciences

2015-05-18

Last updated May 15, 2015

No classes next week!

No classes on 18/5, 22/5, 25/5.

Next class on 29/5

Outline

Today we will see some more classes of algorithms using some of the concepts in the previous classes. You might have seen some of these before.

- Number Theoretic Algorithms
  - Calculating Prime Numbers
  - Greatest Common Divisor
- Backtracking

Calculating Primes

How do we test whether n is prime?

Naive Algorithm

We test the numbers i between 2 and $\sqrt{n}$, to see if $n \bmod i = 0$

```
for(i = 2; i < sqrt{n}+1; i++}
  {
    if (n%i == 0)
      return FALSE;
  }
```

What is the problem in the code above?

How do we test whether *n* is prime? (2)

```
for(i = 2; i < sqrt{n}; i++)
  {
    if (n%i == 0)
      return FALSE;
  }
```

What is the problem in the code above

- The square root function can induce imprecisions in the system;
- We should use $i * i < n$;
- Is there a problem with the use of $i * i$?

How do we test whether n is prime? (3)

```
for(i = 2; i*i < n; i++}
  {
    if (n%i == 0)
      return FALSE;
  }
```

Is there a problem with the use of $i * i$?

- For big enough n, $i * i$ may overflow;
- We can calculate $i * i$ without a multiplication, using a *recurrence*
- $i^2 = (i-1)^2 + 2(i-1) + 1$
- Replace i in the *for* with the recurrence value above;

Calculating primes in the naive way

- We used the idea of a recurrence relation from last class;
- The recurrence eliminated multiplication and exponentiation operations; (Not only for recursion!)
- This kind of thought process is very useful when fine tuning algorithms;

Real-world Primality Testing

Fermat's primality testing

- Most libraries test for primality using an approximated test, such as the Fermat's Primality Test.
- n is *probably* prime if for a random number a:

$$a^{(n-1)} \equiv 1 (mod n)$$

- The more a's you try, the higher the probability of primality;
- There are stronger versions of this test, but the principle is the same;

Calculating the Greatest Common Divisor

Greatest Common Divisor (GDC)

Given two positive integers, a and b, the GDC is the largest integer c so that $a\%c = 0$ and $b\%c = 0$

Useful for reducing fractions (two classes ago), and many other algorithms.

Ideas for algorithms? What are their complexities?

Algorithms for calculating the GDC

Naive Algorithm

- Calculate the factorization of a
- For each factor f_a, calculate $b \% f_a$
- Should we factor a or b? Hard to tell!
- What is the complexity of this algorithm?

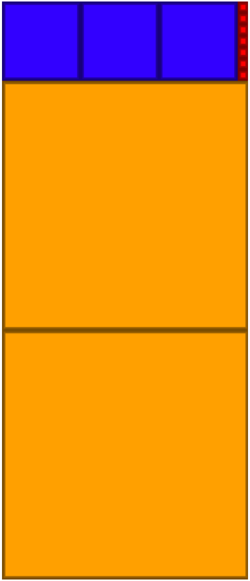
Algorithms for calculating the GDC

Euclid's Algorithm

- if $a = bt$, $GCD(a, b) = b$
- if $a = bt + r$, $GCD(a, b) = GCD(bt + r, b) = GCD(b, r)$
- $GCD(a, b) = GCD(b, a\%b)$ Should we factor a or b? Hard to tell!
- What is the complexity of this algorithm? – O(Factorization(a))

Why is GCD(a,b) = GCD(b,a%b)?

1. Imagine a rectangle with sides *a* and *b*;
2. The *GCD* is the side of largest square that can fill this rectangle;
3. By subtracting *b* from *a*, we have a new rectangle with sides *b*, *a*%*b*;
4. Can you see that a square that would fit the rectangle *b*, *a*%*b*, will also fit the rectangle *b*, *b*?
5. Consequently, it will also fit the rectangle *a*, *b*;



To know more cool facts about numbers

- Proving Concepts using Geometry;
- Interesting facts about prime numbers;
- $1 + 2 + 3 + 4 + ... = \frac{-1}{12}$

- Numberphile channel:
- `https://www.youtube.com/channel/UCoxcjq-8xIDTYp3uz647V5A`

Backtracking

What does the word *Backtracking* means?

To go back on your own steps; To return the same way that you came;

Basic Ideas:

- Try to "assemble" a solution;
- Return, or "undo" some steps if you reach a dead end;
- Avoid dead ends as soon as possible ("pruning");

Backtracking – some important points

Backtracking Algorithm

Backtracking is a systematic method to iterate through all the possible solutions of a problem.

- Represent the solution as a vector $a = (a_1, a_2, ..., a_n)$;
- Each a_i is a step of the solution, taken from a finite set S_i
  - Solution is a set, and a_i is a boolean denoting the presence of i;
  - Solution is a permutation, and a_i is the i_{th} element of the permutation;
- This general technique that must be customized for each application;

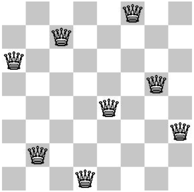
Backtracking

Backtracking Algorithm

Backtracking is a systematic method to iterate through all the possible solutions of a problem.

1. At each step in the algorith, start from a partial solution $(a_1, a_2, ..., a_k)$, and try to extend it, by adding a_{k+1}.
2. After adding a new element, test to see if a is a valid solution.
3. Test to see if a can be extended again. If so, repeat (1).
4. If it cannot, remove a_{k+1} and repeat (1) with another value for a_{k+1}.

Example: The 8-queen Problem



- Put n queens in an n by n square board;
- No 2 queens can be on the same line, row or diagonal;

The n-queen Problem (2)

Thinking about the problem

- Main question: How do we represent one solution?
  - Array of true/false?
  - Array of x,y position of queens?
  - Column Representation?

Important question: How many solutions exist for each representation?

The n-queen Problem (3)

Secondary question: How to search for the solution?

- Sequential search through all possibilities;
- Can we skip some branches that we KNOW are dead ends?
  (this is called "pruning")

When should we use Backtracking?

- Backtracking is powerful, but very expensive;
- It potentially explores all possible solution patterns;

- We usually want to use backtracking when an efficient solution is not known;
- Or when we don't know if an optimal solution exists;

Backtracking Videos

- Sudoku:
  `http://www.youtube.com/watch?v=pd9awN2xBqw`
- Maze:
  `http://www.youtube.com/watch?v=anZlAmtaV1s`
- 8-queens:
  `http://www.youtube.com/watch?v=ckC2hFdLff0`

Backtracking Structure

A backtracking algorithm usually includes these three functions:

- is_a_solution(a,k,input);
  Tests whether the first k elements of a are a complete solution to the problem
- construct_candidates(a,k,input,c,&ncandidates);
  Fills the array c with the complete list of possible candidates for position k;
- process_solution(a,k,input);
  Prints/counts/etc a complete solution;

Constructing All Subsets

- We can construct all the subsets of n items by iterating on all possible 2^n vectors of true/false values.
- How do we define a solution for "any subset"?
- How do we define a solution for "any subset with k items"?

Constructing All Permutations

- To avoid repeating elements, we need to keep track of what elements were already used.
- We test whether an element was already used when adding a new item to a partial solution.

Backtracking Looks a bit like DFS...

- Backtracking is a generalization of Depth First Search (DFS)
- DFS applies to tree-graphs, while backtracking applies to any data structure;
- The graph is built implicitely as the search is performed – the entire graph is not necessary to perform the search;

The Limits of Backtracking

- Backtracking is an exhaustive search; depending on the size of the problem, it can take a LONG time.
- How to calculate the size of the problem? Remember our class on Combinatorics.
  - How big is a subset problem?
  - How big is a permutation problem?

Pruning

Pruning

Problem specific “Tricks” to reduce the number of solutions we have to search for.

- Stopping after the first solution;
- Changing the representation;
- Checking for invalid/unpromising answers;
- etc;

This Week's Problems

- Light, more Light (Number Theory)
- Marbles (Number Theory)
- Queue (Recursion)
- Tug of War (Recursion)