Courses: Sriram Sankaranarayanan

A Quick Tutorial on Pollard's Rho Algorithm

Pollard's Rho Algorithm is a very interesting and quite accessible algorithm for factoring numbers. It is not the fastest algorithm by far but in practice it outperforms trial division by many orders of magnitude. It is based on very simple ideas that can be used in other contexts as well.

History

The original reference for this algorithm is a paper by John M. Pollard (who has many contributions to computational number theory).

```
Pollard, J. M. (1975), “A Monte Carlo method for factorization”, BIT Numerical Mathematics 15
(3): 331–334
```

Improvements were suggested by Richard Brent in a follow up paper that appeared in 1980

```
Brent, Richard P. (1980), “An Improved Monte Carlo Factorization Algorithm”, BIT 20: 176–184,
doi:10.1007/BF01933190
```

A good reference to this algorithm is by Cormen, Leiserson and Rivest in their book. They discuss integer factorization and Pollard's rho algorithm.

Problem Setup

Let us assume that $N = p * q$ is a number to be factorized and $p \neq q$. Our goal is to find one of the factors p or q (the other can be found by dividing from N).

We saw the trial division algorithm

Naive Trial Division Algorithm

```
int main (int argc, char * const argv []) {
   int N,i;
   // Read in the number N
   scanf("%d", &N);
   printf ("You entered N = %d \n", N);
   if (N %2 == 0) {
      puts ("2 is a factor");
      exit(1);
   }

   for (i = 3; i < N; i+= 2){
       if (N % i == 0) {
          printf ("%d is a factor \n", i);
          exit(1);
       }
   }
   printf("No factor found. \n");
   return 1;
}
```

Let us try an even more atrocious version that uses random numbers. Note that the code below is not perfect but it will do.

I am feeling very lucky today Algorithm

```
int main (int argc, char * const argv []) {
   int N,i;
   // Read in the number N
   scanf("%d", &N);
   printf ("You entered N = %d \n", N);

   i = 2 + rand(); // Gets a number from 0 to RAND_MAX

   if (N % i == 0) {
       printf(" I found %d \n", i);
       exit(1);
   }

   printf ("go fishing!\n");
}
```

The `I am feeling lucky` algorithm (no offense to google) generates a random number between 2 and $N-1$ and checks if we found a factor. What are the odds of finding a factor?

Very simple: we have precisely two factors to find p, q in the entire space and $N-1$ numbers. Therefore, the probability is $\frac{2}{N-1}$. If $N \sim 10^{10}$ (a 10 digit number), the chances are very much not in our favor about $\frac{1}{5000000000}$. Now the odds for winning Lotto are much better than this.

Put another way, we have to repeatedly run the `I am feeling lucky algorithm` approximately N times with different random numbers to find an answer. This is absolutely no better than trial division.

Improving the Odds with Birthday Trick

There is a simple trick to improve the odds and it is a very useful one too. It is called the `Birthday Trick`. Let us illustrate this trick.

Suppose we randomly pick a number uniformly at random from 1 to 1000. What are the chances that we land on the number 42. The answer is very simple: $\frac{1}{1000}$. Each number is equally probable and getting 42 is just as likely as picking any other number, really.

But suppose we modify the problem a little: we pick *two* random numbers i, j from 1 to 1000. What are the odds that $i - j = 42$? Note that i, j need not be different. There are roughly 958×2 possible values of i, j that ensure that $i - j = 42$ and the probability reduces to $\frac{2 \times 958}{1000 \times 1000}$ and it works out to roughly $\frac{1}{500}$.

Rather than insist that we pick one number and it be exactly 42, if we are allowed to pick two and just ask that their difference be 42, the odds improve.

What if we pick k numbers $x_1, \ldots, x_k$ in the range $[1, 1000]$ and ask whether any two numbers x_i, x_j satisfy $x_i - x_j = 42$? How do the odds relate to k? The calculation is not hard but let us write a simple program to empirically estimate the odds.

Computing Birthday Paradox Probability

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, int * argv){
  int i,j,k,success;
  int nTrials = 100000, nSucc = 0,l;
  int * p;

  /* Read in the number k */
  printf ("Enter k:");
  scanf ("%d", &k);
  printf ("\n You entered k = %d \n", k);
  if ( k < 2){
    printf (" select a k >= 2 please. \n");
    return 1;
  }
  /* Allocate memory for our purposes */
  p = (int *) malloc(k * sizeof(int));

  //nTrials = number of times to repeat the experiment.
  for (j = 0; j < nTrials; ++j){

    success = 0;
    // Each experiment will generate k random variables
    // and check if the difference between
    // any two of the generated variables is exactly 42.
    // The loop below folds both in.
    for (i = 0; i < k; ++i){
      // Generate the random numbers between 1 and 1000
      p[i] = 1+ (int) ( 1000.0 *  (double) rand() / (double) RAND_MAX );

      // Check if a difference of 42 has been achieved already
      for (l = 0; l < i; ++l)
        if (p[l] - p[i] == 42 || p[i] - p[l] == 42 ){
          success = 1; // Success: we found a diff of 42
          break;
        }
    }

    if (success == 1){ // We track the number of successes so far.
      nSucc ++;
    }

  }
  // Probability is simply estimated by number of success/ number of trials.
  printf ("Est. probability is %f \n", (double) nSucc/ (double) nTrials);
  // Do not forget to cleanup the memory.
  free(p);

  return 1;

}
```

You can run the code for various values of k starting from $k >= 2$.

k	Prob. Est.
2	0.0018
3	0.0054
4	0.01123

5	0.018
6	0.0284
10	0.08196
15	0.18204
20	0.30648
25	0.4376
30	0.566
100	0.9999

This shows a curious property. Around $k = 30$ there is more than a half probability of success. We have a large set of numbers 1000 but generating about 30 random numbers gets us a half probability of success. Around $k = 100$, we are virtually assured of success. This is an important observation and it is called the *birthday problem* or the *birthday paradox*.

Suppose we pick a person at random and ask what is the probability that their birthday is the April 1st. Well, the answer is $\frac{1}{365}$ assuming that no leap years exist.

We number the days in the year from 1 to 365 and April 1st is roughly day number 91. Since each day is equally probable as a birthday, we get to $\frac{1}{365}$. We can play the same game as before.

Let us take $k \geq 2$ people at random and ask the probability that they have the same birthday. A quick modification of our difference of 42 code yields the answers we want

Exploring the birthday paradox

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, int * argv){
  int i,j,k,success;
  int nTrials = 100000, nSucc = 0,l;
  int * p;
  printf ("Enter k:");
  scanf ("%d", &k);
  printf ("\n You entered k = %d \n", k);


  p = (int *) malloc(k * sizeof(int));
  // We will do 1000 reps
  for (j = 0; j < nTrials; ++j){
    success = 0;
    for (i = 0; i < k; ++i){
      // Generate the random numbers between 1 and 365
      p[i] = 1+ (int) ( 365 *  (double) rand() / (double) RAND_MAX );
      // Check if a difference of 42 has been achieved
      for (l = 0; l < i; ++l)
        if (p[l] - p[i] == 0  ){
          success = 1;
          break;
        }
    }
    if (success == 1){
      nSucc ++;
    }
  }

  printf ("Est. probability is %f \n", (double) nSucc/ (double) nTrials);
  return 1;
}
```

We can see that with $k = 10$ people there is a roughly 11% chance of having two people with the same birthday. With $k = 23$ people, we have a roughly 50% chance. Our class size is $k = 58$ and we have roughly 99% chance of having two people having the same birthday. We get mathematical certainty 100% chance with $k \geq 366$.

If we have N days in a year ($N = 365$ on this planet) then with $k = \sqrt{N}$ people we have a 50% chance of having a birthday collision.

Imagine a version of star trek where the enterprise docks on a strange new planet and they are unable to find out how long a year is. Captain Kirk and Officer Spock land on the planet and walk over to the birth records. They toss coins to pick people at random and look at how many people give them even odds of birthday collision. They can back out the revolution period of the planet divided by its rotational period (i.e, number of days in the year).

This is all well and good, you say. How does this help us at all?

Applying Birthday Paradox to Factoring

Let us go back to the I am feeling lucky algorithm. We are given $N = p * q$ and we randomly picked a number between 1 and N. Chances of landing on either p or q are quite small. So we have to repeat the algorithm many many times to get us to even odds.

We can ask a different question. Instead of picking just one number, we can pick k numbers for some k. Let these numbers be $x_1, \ldots, x_k$. We can now ask if $x_i - x_j$ divides N.

The difference between the former scheme and latter is exactly the difference between picking one person and asking if their birthday falls on April the 1st (a specific day) or picking k people and asking if any two among k share a birthday. We can already see that for k roughly around $\sqrt{N}$, we get even odds. So very roughly speaking, we can improve the chances from roughly $\frac{1}{N}$ to $\sqrt{\frac{1}{N}}$. Therefore, for a 10 digit number, instead of doing 10^{10} reps, we can pick $k = 10^5$ random numbers and do the test above.

But unfortunately, this does not save us any effort. With $k = 10^5$ people, we still do $k^2 = 10^{10}$ pairwise comparisons and divisions. Bah.. there's got to be a better way 🙂

We can do something even better.

We can pick numbers $x_1, \ldots, x_k$ and instead of asking if $x_i - x_j$ divides N, we can ask if $GCD(x_i - x_j, N) > 1$. In other words, we ask if $x_i - x_j$ and N have a non-trivial factor in common. This at once increases the number of chances for successes.

If we ask how many numbers divide N, we have just 2 : p, q.

If we ask how many numbers have $GCD(x, N) > 1$, we have quite a few now:

$$p, 2p, 3p, 4p, 5p, \ldots., (q-1)p, q, 2q, 3q, \ldots (p-1)q$$

Precisely, we have $p + q - 2$ of these numbers.

So a simple scheme is as follows:

- Pick k numbers $x_1, \ldots, x_k$ uniformly at random between 2 and $N-1$.
- Ask if $GCD(x_i - x_j, N) > 1$. If yes, then $GCD(x_i - x_j, N)$ is a factor of N (either p or q).

But there is already a problem, we need to pick a number k that is in the order of $N^{\frac{1}{4}}$ and do pairwise comparisons. This is already too much to store in the memory. If $N \sim 10^{30}$, we are storing about 10^8 numbers in memory.

To get to Pollard's rho algorithm, we want to do things so that we just have two numbers in memory.

Pollard's Rho Algorithm

Therefore, Pollard's rho algorithm works like this. We would like to generate k numbers $x_1, \ldots, x_k$ and check pairwise but we cannot. The next best thing is to generate random numbers one by one and check two consecutive numbers. We keep doing this for a while and hopefully we are going to get lucky.

We use a function f that will generate pseudo random numbers. In other words, we will keep applying f and it will generate numbers that “look and feel” random. Not every function does it but one such function that has mysterious pseudo random property is $f(x) = x^2 + a \mod N$, where we generate a using a random number generator.

We start with $x_1 = 2$ or some other number. We do $x_2 = f(x_1), x_3 = f(x_2), \ldots$. The general rule is $x_{n+1} = f(x_n)$.

We can start off with a naive algorithm and start to fix the problems as we go.

Pollard's Rho Algorithm Scheme

```
x := 2;

while (.. exit criterion .. )

   y := f(x);
   p := GCD( | y - x | , N);
   if ( p > 1)
       return "Found factor: p";
   x := y;
end

return "Failed. :-("
```

Let us take $N = 55$ and $f(x) = x^2 + 2 \mod 55$.

x_n	x_{n+1}	**GCD($\lvert x_n - x_{n+1} \rvert$, N)**
2	6	1
6	38	1
38	16	1
16	36	5 😃

You can see that in many cases this works. But in some cases, it goes into an infinite loop because, the function f cycles. When that happens, we keep repeating the same set of value pairs x_n and x_{n+1} and never stop.

For example, we can make up a pseudo random function f which gives us a sequence like

$$2, 10, 16, 23, 29, 13, 16, 23, 29, 13, \ldots .$$

In this case, we will keep cycling and never find a factor. How do we detect that the cycling has happened?

Solution #1 is to keep all the numbers seen so far $x_1, \ldots, x_n$ and see if $x_n = x_l$ for some previous $l < n$. This gets back to our memory crunch as n is going to be large in practice.

Solution #2 is a clever algorithm by Floyd. To illustrate Floyd's algorithm, suppose we are running on a long circular race track, how do we know we have completed one cycle? We could use solution #1 and remember everything we have seen so far. But a cleverer solution is to have two runners A and B with B running twice as fast as A. They start off at the same position and when B overtakes A, we know that B has cycled around at least once.

Pollard's Rho Algorithm Scheme

```
a := 2;
b := 2;
while ( b != a )

   a = f(a); // a runs once
   b = f(f(b)); // b runs twice as fast.
   p = GCD( | b - a | , N);
   if ( p > 1)
       return "Found factor: p";
end

return "Failed. :-("
```

If the algorithm fails, we simply find a new function f or start from a new random seed for a, b.

Now we have derived the full Pollard's rho algorithm with Floyd's cycle detection scheme. Hope you were able to follow this presentation.

You may find the wikipedia presentation on `cycle detection` quite useful. It also explains the use of Bernt's cycle finding algorithm in this context.

pollard_rho_tutorial.txt · Last modified: 2013/04/07 13:06 by srirams