Algorithms and Data Structures II

Lecture 13:

Randomized Algorithms

https://elms.u-aizu.ac.jp

Randomized Algorithms

- An algorithm is called randomized if it uses
  - a random number to make a decision at least once during the computation and
  - its computation time is determined not only by the input data but also by the values of a random number generator.

Example

- As an example, we first introduce a randomized quicksort algorithm.
- Quicksort is a divide-and-conquer method of sorting. It works by partitioning a file into two parts, then sorting the parts independently. The algorithm has the following general structure:

quicksort

```
quicksort(int a[], int l, int r){
  int i;
  if (r>l){
    i=partition(l, r);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
}
```

quicksort(contd)

- The parameters ℓ and r delimit the subfile within the original file that is to be sorted; the call $quicksort(1, n)$ sorts the whole file. The crux of the method is the partition procedure, which must rearrange the array to make the following three conditions hold:
  1. the element $a[i]$ is in its final place in the array for some i,
  2. $a[j] \leq a[i]$ for $1 \leq j \leq i - 1$, and
  3. $a[i] \leq a[k]$ for $i + 1 \leq k \leq r$.

quicksort(contd 2)

- The efficiency of quicksort algorithm depends on the selection of $a[i]$ in each recursion.
- If the chosen $a[i]$ divides the file into two subfiles with the similar sizes then the algorithm sorts the sequence of n data in $O(n \log n)$ time.
- However, if the sizes of the two subfiles are very biased (e.g., one has the size 0 and the other has size n-1) in each recursion then quicksort takes $O(n^2)$ time.

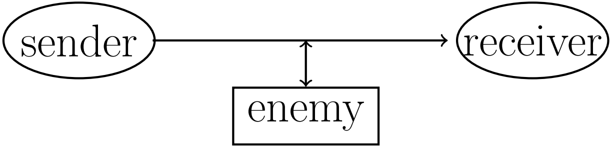
quicksort(contd 3)

- For many practical applications, we can choose the $a[i]$ randomly between $a[\ell]$ and $a[r]$. This can be done by generating a random number i with $\ell \le i \le r$. This results in a randomized quicksort algorithm.

- Now, we introduce another randomized algorithm for primality testing.

Encription

- Finding prime numbers of hundreds digits has important applications in designing cryptography schemes and so on.



Sender: encrypt plaintext to cryptotext, $E(pt) = ct$ we may call $E()$ encryption key for simplicity
Receiver: decrypt cryptotext to plaintext, $D(ct) = pt$
$D()$ decryption key

Encription(contd)

- A cryptosystem with n individuals needs $n(n-1)/2$ pairs of encryption key and decryption key.
- Public-key cryptosystem, each individual has a secrete decryption key $D()$, but the encryption key $E()$ is open to everyone. This system only needs n pairs of keys.
- Computation of pt from $E(pt)$ is likely to be intractable if $D()$ is not known.

RSA public-key cryptosystem

- Encryption key $E()$ is a product of two prime numbers ($N = pq$). Decryption key $D()$ is the prime factors p and q of the product N. Given primes p and q, it is easy to compute $N = pq$, while given N, it is hard to find its prime factors.
- In RSA system, large prime numbers are needed.

Prime number test

- For testing whether an n-digit number P is prime or not, no polynomial time (in n) deterministic algorithm is known.
- An obvious approach is to divide P by odd numbers from 3 to $\sqrt{P}$. This method requires $O(\sqrt{P}) = O(2^{(n/2)})$ time which is exponential in n.

Prime number test(contd)

- We can test if P is prime by a randomized algorithm (which runs in polynomial time in n) in such a way:
  - If the algorithm says that P is not prime then P is definitely not a prime.
  - If the algorithm says that P is a prime then with a high probability (but not a 100% certainty) P is prime.

Prime number test(Fermat)

- Such an algorithm can be developed based on the following theorem:

  Fermat's *Little* Theorem

  If P is prime and $0 < A < P$ then $A^{p-1} \equiv 1 (\mathrm{mod}\ P)$.

  Given a number P, we can choose a particular A (e.g., 2) with $0 < A < P$ and calculate $A^{p-1} (\mathrm{mod}\ P)$:
  If $A^{p-1} \not\equiv 1 (\mathrm{mod}\ P)$ then P is NOT prime.
  If $A^{p-1} \equiv 1 (\mathrm{mod}\ P)$ then P is probably prime.

Prime number test(contd 3)

- For example, for $A = 2$ and $P = 67$, $2^{P-1} \equiv 1 (\bmod P)$ and P is prime, while for $P = 341, 2^{P-1} \equiv 1 (\bmod P)$ but P is not prime($341 = 11 \cdot 31$).
- We can randomize the above algorithm by choosing $1 < A < P - 1$ at random.
- For an A chosen at random if $A^{p-1} \not\equiv 1 (\bmod P)$ then we say P is not prime otherwise we accept P is prime.

Prime number test(contd 4)

- Notice that we may make a mistake by some chance in the second case, i.e., P is in fact not a prime but we accept it as a prime based on $A^{p-1} \equiv 1 (\bmod P)$ for the chosen A.
- However, we can reduce the chance of making a mistake by repeating the above computation many times. We accept P as a prime only if all the computations accept it as a prime.

Prime number test(contd 5)

- Assume that the probability that the algorithm makes a mistake is p in each computation. If the algorithm answers that P is prime in all the m independent computations then with the probability at least $1 - p^m$ that P is a prime.

Prime number test(contd 6)

- Now, we have got a randomized algorithm for testing if P is a prime. Does this algorithm work well for all integers? Unfortunately, it does not.
- There are composite numbers P such that for all A which is relatively prime to P (i.e. $\gcd(A, P) = 1$), $A^{P-1} \equiv 1 \pmod{P}$. Such numbers are known as Carmichael numbers.

Prime number test(contd 7)

- The smallest Carmicheal number is $561 = 3 \times 11 \times 17$. For $P = 561$, we will make a mistake for all the choices of A except only three, $3, 11$, and 17. That is, the probability p of making a mistake based on the evaluation of $A^{P-1} \equiv 1 (\bmod\ P)$ could be very close to 1 and thus, $1 - p^m$ could be very small.
- Notice that there are infinite many Carmicheal numbers. To reduce the probability p that the algorithm makes a mistake in one computation, we can include some other testings.

Prime number test(Miller-Rabin)

- The following theorem provides an efficient one:

> If P is prime and $0 < X < P$ then the only solutions to $X^2 \equiv 1 (\text{mod } P)$ are $X = 1, P - 1$.

- Based on the above theorem, we can do the following check for the value of $X = A^i (\text{mod } P)$ for some $1 \leq i \leq P - 1$.
- If $X^2 (\text{mod } P) = 1$ and $X \neq 1$ and $X \neq P - 1$ then P is not a prime and we stop the evaluation of $A^{P-1} (\text{mod } P)$. Otherwise, we continue the evaluation.

Prime number test(contd 9)

- To get the algorithm, we still need to know how to evaluate A^{P-1} in polynomial time in n for an n-bit P. Below is an algorithm for computing A^{P-1}.

/* Assume that the binary expression of $P-1$ is $(b_{n-1}, b_{n-2}, \ldots, b_0)$ with $b_{n-1} = 1$ and $n > 1$.*/

```
x=A;
for (i:=n-2;i>=0;i--){
   x=x*x;
   if (b_i==1) x=x*A;
}
```

Prime number test(contd 10)

- This algorithm computes A^{P-1} in $O(n)$ time.
- Now, we are ready to give the algorithm for the primal testing.

Prime number test(contd 11)

```
/* Assume that the binary expression of P − 1
is (b_{n−1}, b_{n−2}, ..., b_0); with b_{n−1} = 1 and n > 1.

choose A between 2 and P − 2 at random;
```

Prime number test(contd 12)

```
x=A;
for (i:=n-2;i>=0;i--){
   y=x*x%P;
   if (y==1 && x!=1 && x!=P-1)
      return ``P is not prime'';
   if (b_i==1) y=y*A% P;
   x=y;
}
if (y==1) then return ``P is prime'';
else return ``P is not prime'';
```

Prime number test(contd 13)

- For large P the above algorithm makes a mistake with probability at most $1/4$.
- Thus, if we run the algorithm independently for m times, the probability that we make a mistake (i.e., P is not a prime but we claim it is) is at most 2^{-2m}.
- For example, take $m = 50$, the probability of making a mistake is at most 2^{-100} which is small enough for many applications.