COS 397 – week 09

PART 1. Compression. Huffman codes
Methods for compressing data. The Huffman greedy algorithm uses character frequencies.
File with 100000 characters:

Frequency		**Fixed-Length Codeword**	**Variable-Length Codeword**
a	45000	000	0
b	13000	001	101
c	12000	010	100
d	16000	011	111
e	9000	100	1101
f	5000	101	1100

3 bits per character → 300000 bits without "coding" Using a variable-length code: Short code for frequent characters, long code for rare characters.

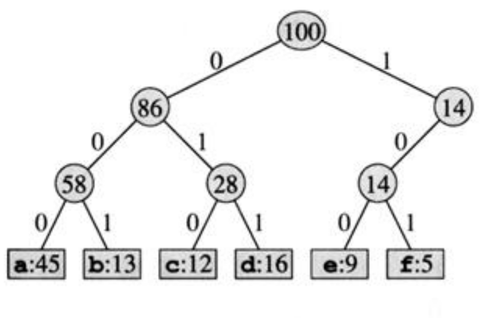
Character	**Frequency**	**Bits**
a	45000	1
b	13000	3
c	12000	3
d	16000	3
e	9000	4
f	5000	4

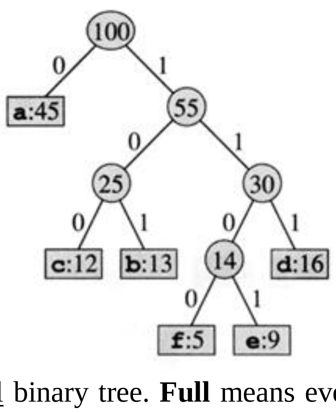
A total of 224,000 bits, much more compact than the original 300,000 bits. How do we choose the variable length code words?
Fixed Codes: Code words are not prefixes of other code words. Can show that optimal compression can always be achieved with a prefix code, so only consider them.

a	b	c
0	101	100

	aaa	b	e
0001011101 =	000	101	1101

a b c = 0 101 100 = 0101100 so binary tree whose leaves are the code words is a convenient tool to aid coding.





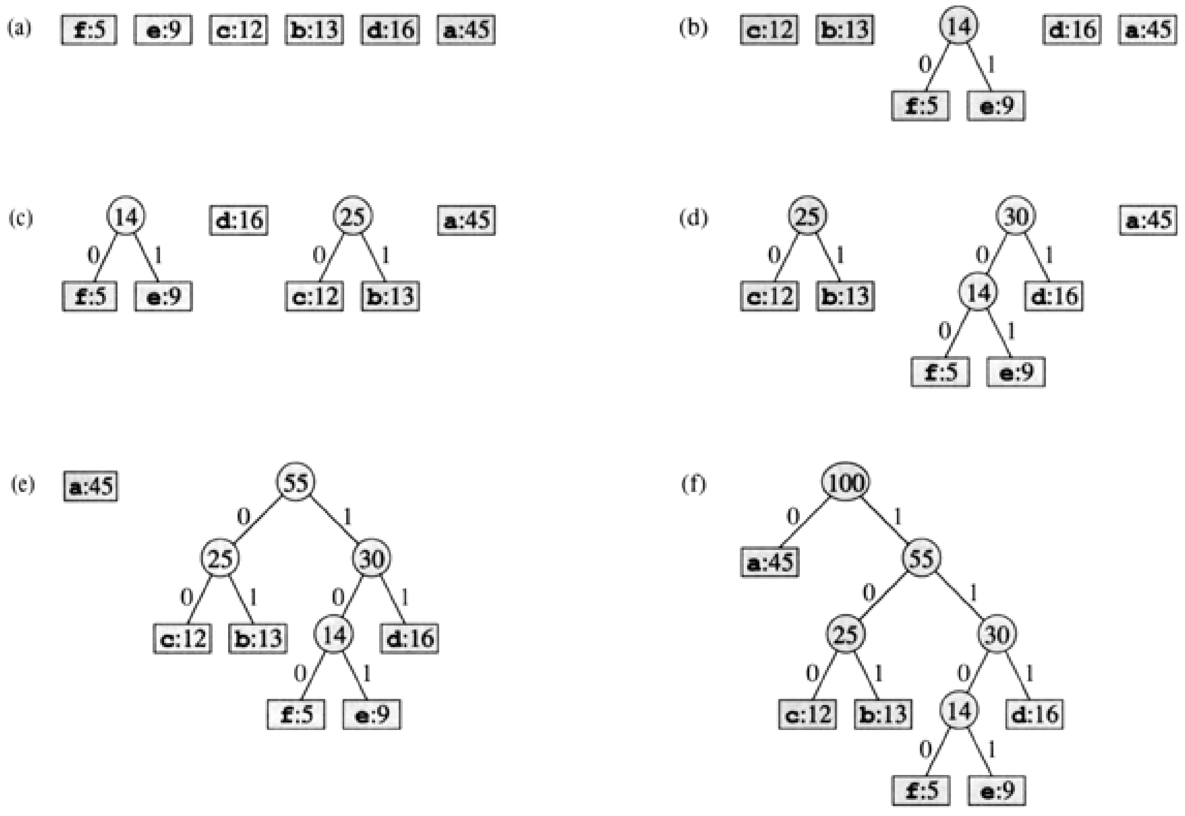
An optimal code for is always represented as a <u>full</u> binary tree. **Full** means every non-leaf child has two children.

Huffman method is a greedy algorithm: Take two least frequent objects and merge them together.

Implementation via Priority queue Q; C is the alphabet; f[c] is the frequency of c.

```
1.    n ← |C|
2.    Q ← C
3.    for i ← 1 to n – 1
4.      do z ← Allocate-Node ()
5.        x left[z] ← Extract-Min (Q) // least frequent
6.        y right[z] ← Extract-Min (Q) // next least
7.        f[z] ← f[x] + f[y] // update frequency
8.        Insert ( Q, z )
9.    return Extract-Min (Q)
```

Running Time: Q into heap = **O** (n), n = |C|, n times through loop, min extract **O** (lg n) so **O** (n lg n)



PART 2. Parsing. Classic algorithms for expression evaluation
The classic approach to writing code to evaluate arithmetic expressions [Donald Knuth, 1962] outlines three steps:

- Parsing an infix expression
- Converting infix expression to a postfix expression
- Evaluating the postfix expression

The three most common forms of notation in arithmetic expressions are *infix, prefix,* and *postfix* notations. Infix notation is a common way of writing expressions, while prefix and postfix notations are primarily used in computer science.

Infix notation
Infix notation is the conventional notation for arithmetic expressions. It is called *infix* notation because each operator is placed between its operands, which is possible only when an operator has exactly two operands (as in the case with binary operators such as addition, subtraction, multiplication, division,

and modulo). When parsing expressions written in infix notation, you need parentheses and precedence rules to remove ambiguity.

Syntax: operand1 operator operand2 Example: (A+B)*C–D/(E+F)

Postfix notation
In postfix notation, the operator comes after its operands. Postfix notation is also known as *reverse Polish notation* (RPN) and is commonly used because it enables easy evaluation of expressions.

Syntax : operand1 operand2 operator Example : AB+C*DEF+/-

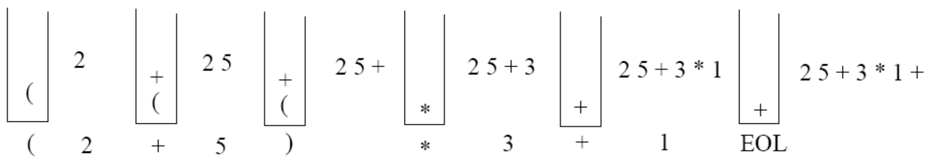
Postfix (and prefix) notation has three common features:
- The operands are in the same order that they would be in the equivalent infix expression.
- Parentheses are not needed.
- The priority of the operators is irrelevant.

Converting infix notation to postfix notation
To convert an expression which in an infix expression to its equivalent in postfix notation, we must know the precedence and associativity of operators. *Precedence* or operator strength determines order of evaluation; an operator with higher precedence is evaluated before one of lower precedence. If the operators all have the same precedence, then the order of evaluation depends on their *associativity*. The associativity of an operator defines the order in which operators of the same precedence are grouped (right-to-left or left-to-right).

Left associativity : A+B+C = (A+B)+C Right associativity : A^B^C = A^(B^C)

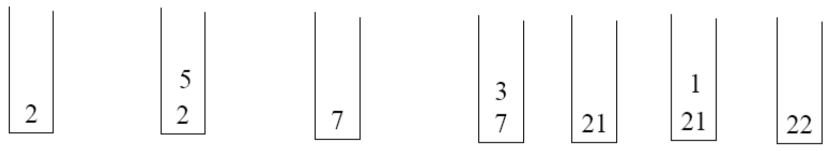
The conversion process involves reading the operands, operators, and parentheses of the infix expression using the following algorithm:
1. Initialize an empty stack and empty result string variable.
2. Read the infix expression from left to right, one character at a time.
3. If the character is an operand, append it to the result string.
4. If the character is an operator, pop operators until you reach an opening parenthesis, an operator of lower precedence, or a right associative symbol of equal precedence. Push the operator onto the stack.
5. If the character is an opening parenthesis, push it onto the stack.
6. If the character is a closing parenthesis, pop all operators until you reach an opening parenthesis and append them to the result string.
7. If the end of the input string is found, pop all operators and append them to the result string.

Example: Infix expression: (2 + 5) * 3 + 1
The operator stack holds just the operators. Operands are sent to the output directly.



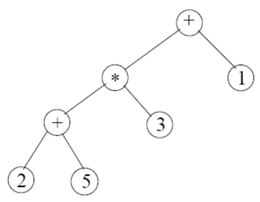
Postfix expression evaluation. Evaluating a postfix expression is simpler than directly evaluating an infix expression. In postfix notation, the need for parentheses is eliminated and the priority of the operators is no longer relevant. You can use the following algorithm to evaluate postfix expressions:
1. Initialize an empty stack.
2. Read the postfix expression from left to right.
3. If the character is an operand, push it onto the stack.
4. If the character is an operator, pop two operands, perform the appropriate operation, and then push the result onto the stack. If you could not pop two operators, the syntax of the postfix expression was not correct.
5. At the end of the postfix expression, pop a result from the stack. If the postfix expression was correctly formed, the stack should be empty.

Example: Postfix expression: 2 5 + 3 * 1 +   The operator stack holds just the operands.

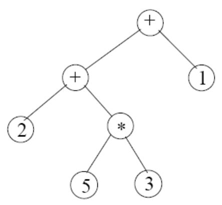


Expression Tree.

(2 + 5) * 3 + 1



2 + 5 * 3 + 1



Context-Free Grammars. Formal rules for production of expressions:

expression :: = *term* {+ *term*}
term ::= *factor* {**factor*}
factor ::= (*expression*) | *operand*
operand ::= *number* | *identifier*

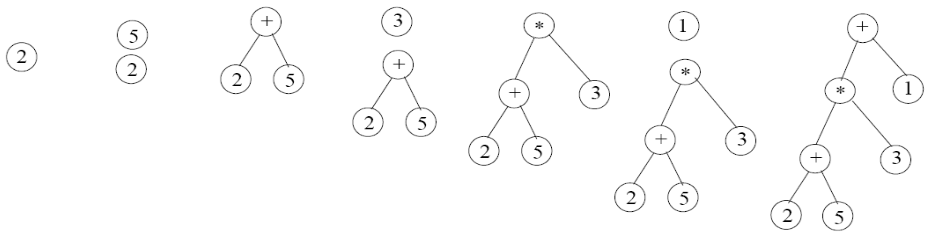
This grammar describes expressions like those that we used, e.g. (A*B+A*C)*D. Each line in the grammar is called a production or replacement rule. The productions consist of terminal symbols (,) , + and * which are the symbols used in the language being described; nonterminal symbols like *expression*, *term*, and *factor* which are internal to the grammar; and metasymbols :: =

```
#include<iostream>
using namespace std;
struct node
{ int tn; /* 0,1,2 */ char op;
  double v;
  node *arg1, *arg2;
};
/********  tree creation ************/
char str[999]; char ch; int spos;
void next() {ch=str[spos++];}
void expression(node*& pt);
void factor(node*& pt)
{ if(ch=='(')
   { next(); expression(pt); next();}
  else
   {pt = new node; pt->tn=0; pt->v=0.0; pt->op=ch;
    pt->arg1=NULL; pt->arg2=NULL; next();
   }
}
void term(node*& pt)
{char op; node *a1, *a2;
 factor(a1); pt=a1;
 while ((ch=='*') || (ch=='/'))
  {op=ch; next(); factor(a2);
   pt=new node; pt->tn=2; pt->v=0.0; pt->op=op;
   pt->arg1=a1; pt->arg2=a2; a1=pt;
  }
}
```

```
void expression(node*& pt)
{ char op; node *a1, *a2;
  term(a1); pt=a1;
  while ((ch=='+') || (ch=='-'))
   {op=ch; next(); term(a2);
    pt = new node; pt->tn=2; pt->v=0.0; pt->op=op;
    pt->arg1=a1; pt->arg2=a2; a1=pt;
   }
}
void create_tree(node*& pt)
{spos=0; next(); expression(pt);}
/********* output tree as a string ***********/
void output_tree(node* pt, char ch)
{char s;
 switch(pt->tn)
 {case 0: cout << pt->op; break;
  case 2:
  { s=pt->op;
    switch(s)
     {case '*':
        if(ch=='/') cout <<'(';
        output_tree(pt->arg1,s); cout << s; output_tree(pt->arg2,s);
        if(ch=='/') cout << ')';
        break;
      case '/':
        if((ch=='*')||(ch=='/')) cout << '(';
        output_tree(pt->arg1,s); cout << s; output_tree(pt->arg2,s);
        if((ch=='*')||(ch=='/')) cout << ')';
        break;
      case '+': case '-' :
        if((ch=='-')||(ch=='*')||(ch=='/')) cout << '(';
        if(ch=='+') output_tree(pt->arg1,s);
        else output_tree(pt->arg1,' ');
        cout << s;
        output_tree(pt->arg2,s);
        if((ch=='-')||(ch=='*')||(ch=='/')) cout << ')';
        break;
     }
  }
 }
}
/********** computing using tree ***********/
typedef char varnameType[21];
typedef double varvalType[21];
varnameType varname;
varvalType varval;
int maxvar;
void initval() /** example only **/
{ maxvar=4;
  varname[1]='a'; varname[2]='b'; varname[3]='c'; varname[4]='d';
  varval[1]=1.0; varval[2]=2.0; varval[3]=3.0; varval[4]=4.0;
}
void argcalc(node* pt)
{ for(int i=1;i<=maxvar;i++) if(pt->op==varname[i])
   {pt->v=varval[i]; return;}
}
void twoargcalc(node* pt)
{if((pt->arg1->tn==0)&&(pt->arg2->tn==0))
```

```
    {switch(pt->op)
      { case '+' : pt->v=pt->arg1->v+pt->arg2->v; break;
        case '-' : pt->v=pt->arg1->v-pt->arg2->v; break;
        case '*' : pt->v=pt->arg1->v*pt->arg2->v; break;
        case '/' : pt->v=pt->arg1->v/pt->arg2->v; break;
      }
     pt->tn=0;
    }
}
void numcalc(node* pt)
{switch(pt->tn)
 {case 0 : argcalc(pt); break;
  case 2 :
    numcalc(pt->arg1); numcalc(pt->arg2); twoargcalc(pt); break;
 }
}
double value(node* pt)
{numcalc(pt); return pt->v;
}
int main()
{node* pt; cin >> str; create_tree(pt);
 output_tree(pt,' '); cout << endl;
 initval(); cout << value(pt) << endl;
}
```

Example of expression tree creation: Infix expression: (2 + 5) * 3 + 1



PART 3. Arithmetic. Cryptology.
Elementary Number Theoretic Notions

Z = {0,±1,±2, . . . } are integers; N = {0, 1, 2, . . . } are natural or counting numbers

d|a "d divides a" means that there exists k, such that a = kd
If d|b then b is **a multiple** of d; If d|b and d ≥ 0, then d is **a divisor** of b
If d is a divisor of b, then 1 ≤ d ≤ |b|, e. g. the divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.
Every integer b is divisible by 1 and b, the trivial divisors.
Nontrivial divisors of b are called factors of b.
Definition: An integer p > 1 with only trivial divisors is a prime number, or prime.
Small primes are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29,.....
There are infinitely many primes. Any positive integer that is not prime is composite.
1 is the "unit" for multiplication and is neither prime nor composite.

Division Theorem: For any integer b, and positive integer n, there exist integers q and r such that 0 ≤ r < n and b = qn + r.

q = [b / n] is called the quotient.
r = b (mod n) is the remainder.
If a (mod n) = b (mod n) we write a ≡ b (mod n) "a is equivalent to b modulo n."

Thus, a and b have the same remainder, w.r.t. n and so n|(b − a).

The equivalence class modulo n and b:
$[b]_n = \{b + kn : k \text{ in } Z\}$, e. g. $[3]_7 = \{..., -11, -4, 3, 10, 17, ...\}$

$Z_n = \{[a]_n : 0 \leq a \leq n - 1\}$ and we often write $Z_n = \{0, 1, 2, 3, ..., n - 1\}$ associate a with $[a]_n$.

$-1 \in [n - 1]_n$ since $n - 1 \pmod{n} \equiv -1$

Common Divisors and Greatest Common Divisors

If d|a and d|b, then d is a common divisor of a and b.

LEMMA: If d|a and d|b, then d|(a + b) and d|(a − b), and also d|(ax + by) for any $x, y \in Z$.

The greatest common divisor of a and b, not both zero, is the largest common divisor:
gcd(24,30) = 6; gcd(5,7) = 1; gcd(0,9) = 9 (all integers divide 0)

$1 \leq \gcd(a, b) \leq \min(|a|, |b|)$

gcd(0,0) = 0 by definition.
Some elementary gcd properties:

gcd(a, b) = gcd(b, a); gcd(a, b) = gcd(–a, b); gcd(a, b) = gcd(|a|, |b|); gcd(a, 0) = |a|
gcd(a, ka) =|a| for any $k \in Z$.

Theorem: If a and b are integers and not both zero, then gcd(a, b) is the smallest positive element of $\{ax + by : x, y \in Z\}$.

Corollary: For a, b in N, if n|ab and gcd(a, n)=1, then n|b.

If gcd(a, b)=1, then a and b are "relatively prime" integers.

Theorem (Unique Factorization): Any integer, *a*, can be written in exactly one way as $a = \prod p_i^{n_i}$, where *pi* is the *i*-th prime.

By prime factorization $\gcd(a,b) = \prod p_i^{\min(n_{ai}, n_{bi})}$

However, factoring integers is HARD, so this is impractical. Instead we will derive a "fast" algorithm: Euclid's algorithm.

Theorem (GCD recursion theorem): gcd(a,b)= gcd(b, a (mod b)); (in C language a(mod b) is a%b)

```
int Euclid(int a, int b)
{ if(b==0) eturn a;
  else return Euclid(b, a%b);}
```

E.g.
gcd(30, 21) = gcd(21, 30 (mod 21)) = gcd(21, 9) = gcd(9, 21 (mod 9)) = gcd(9, 3) = gcd(3, 9 (mod 3)) = gcd(3, 0) = 3

The Running Time of Euclid: What is the worst case?
We can assume $a > b \geq 0$, since it not, the first pass of Euclid fixes his!
Also, if $b = a > 0$, then it returns b after one call. So given this what are the WORST a, b pairs to put into this: successive Fibonacci numbers!

• Lemma: IF $a > b \geq 0$ and Euclid(a, b) performs $k \geq 1$ recursive calls, then $a \geq F_k+2$ and $b \geq F_k+1$
Proof: We prove by induction on k.

If a and b are β-bit numbers you can show it takes only $O(\beta^2)$ bit operations to compute gcd(a, b),

The Extended Euclidean Algorithm: One can try to find x, y so that gcd(a, b) = ax + by

Modular arithmetic. Finite Group
A group $(S,\oplus)$ is a set S, with a binary operation, $\oplus$, with
1. Closure: $\forall a, b \in S, a \oplus b \in S$
2. Identity: $\exists e \in S$ such that $e \otimes a = a \oplus e = a, \forall a \in S$
3. Associativity: $\forall a, b, c \in S, (a \oplus b) \oplus c = a \oplus (b \oplus c)$
4. Inverse: $\forall a \in S, \exists !b \in S$ such that $a \oplus b = b \oplus a = e$

If $(S,\oplus)$ is commutative: $a \oplus b = b \oplus a, \forall a, b \in S$, then $(S,\oplus)$ is called *Abelian*.
If $(S,\oplus)$ has $|S| < \infty$, then it is a finite group.

Consider defining a group on $S = Z_n$ (the integers modulo *n*). We need to define $\oplus$. Consider

$[a]_n +_n [b]_n = [a + b]_n$
$a + b$ is $a + b \pmod n$

$[a]_n \bullet_n [b]_n = [a + b]n$
$a \bullet b$ is $a \bullet b \pmod n$

Theorem: The system $(Z_n,+_n)$ is a finite abelian group, called the additive group modulo *n*.

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

The multiplication group modulo *n* is denoted as $(Z^*_n, \bullet_n)$

$Z^*_n = \{[a]_n \in Z_n : \gcd(a, n) = 1\}$

Since $15=5 \bullet 3$ and $Z^*_{15} = \{1, 2, 4, 7, 8, 11, 13, 14\}$ as 3, 5, 6, 9, 10, and 12 have 3 or 5 as factors.
$\bullet_n$ is multiplication modulo *n*.
• **Theorem:** $(Z^*_n, \bullet_n)$ is a finite Abelian group.

$\times_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

$|Z^*_n| = \varphi(n)$ is Euler's phi, or totient function; we can compute:

$$\phi(n) = n \prod_{p|n} (1 - \frac{1}{p}).$$

Solving Diophantine Linear Equations. Let us solve $ax \equiv b \pmod{n}$ for x, with a, b and n given integers. Consider $<a>$ in Z_n, if $b \in <a> = \{ax \pmod{n} : x > 0\}$, then the equation has a solution.

Theorem: $\forall a,n \in Z_+$, if $\gcd(a, n) = d$, then $<a>=<d>= \{0, d, 2d, ..., (n/d – 1)d\}$ and thus $|<a>| = n/d$

Chinese Remainder Theorem
Around 100 A.D., a Chinese mathematican solved the problem of finding an integer x, that satisfied
$x \equiv 2 \pmod{3}$; $x \equiv 3 \pmod{5}$; $x \equiv 2 \pmod{7}$

$x = 23$ is one solution, but so is $23 + 105k$, for all $k \in Z$. The method of general solution goes under the name the "Chinese Remainder Theorem."
Given $p_1, p_2, \ldots, p_n$, all relatively prime, and remainders $r_1, r_2, \ldots, r_n$, find x such that $x\%p_1=r_1$, $x\%p_2=r_2, \ldots, x\%p_n=r_n$.

Method: Let $q_i= (p_1, p_2, \ldots, p_n)/p_i$. Numbers q_i and p_i are relatively prime, hence there exist a_i and b_i such that $a_ip_i+b_iq_i=1$. It follows that $q_i\%p_j=1$ for i=j, and $q_i\%p_j=0$, otherwise. Now let $x = r_1q_1+r_2q_2+\ldots+r_nq_n$.

Powers of an Element
Consider powers of a modulo n:

i	0 1 2 3 4 5 6 7 8 9 10 11 ...
$3^i \pmod{7}$	1 3 2 6 4 5 1 3 2 6 4 5 ....

i	0 1 2 3 4 5 6 7 8 9 10 11 ...
$2^i \pmod{7}$	1 2 4 1 2 4 1 2 4 1 2 4 ....

• **Theorem:** (Euler) $a^{\varphi(n)} \equiv 1 \pmod{n}$ for all a relatively prime to $n > 1$.

• **Theorem:** (Fermat's little) If p is prime, then $a^{p-1} \equiv 1 \pmod{p}$

Exponentiation via square-and-multiply

Cryptography. Public-Key:

- Alice: P_A, S_A, public and private keys.
- Bob: P_B, S_B, public and private keys

Let D be the set of possible messages.
Let $P_A() : D \to D$ be a permutation, such that for any M in D: $M = S_A(P_A(M)) = P_A(S_A(M))$.
It is important that only Alice can compute $S_A()$. We assume that S_A is kept secret.

Suppose Bob wants to send a message to Alice.
– Bob obtains P_A
– Bob computes the cybertext $C = P_A(M)$ and sends C to Alice
– Alice computes $S_A(C) = S_A(P_A(M)) = M$
This is message encryption.

One can also **digitally sign** a message with a public-key cryptosystem.
Suppose now Alice wants to send Bob a digitally signed message, M
– Alice computes $\sigma = S_A(M)$, her digital signature
– Alice sends (σ,M) to Bob
– Bob computes $M' = PA(\sigma)$ and compares it to the M sent. This verifies that Alice and only Alice could have sent it and that the message was the same, i. e., not altered.

RSA Crytosystem
It seems that a public-key cryptosystem could be quite useful, but to make it a reality, we must find a scheme that satisfies the above given assumptions. Rivest, Shamir, and Adleman came up with such a scheme, that is now called RSA. It's security is based on the difficulty of factoring integers with only two, large sized factors.

• **RSA**

1. Randomly select two large primes p and q
2. Compute $n = pq$
3. Select e small such that $gcd(e, \varphi(n)) = 1$. Note $\varphi(n) = (q-1)(p-1)$
4. Compute $d \equiv e^{-1} \pmod{\varphi(n)}$; this exists and is unique because $\gcd(e, \varphi(n)) = 1$, i.e. there exist d *and* $f > 0$ such that $de - f\varphi(n) = 1$.
5. $P = (e, n)$ is the **RSA public key**
6. $S = (d, n)$ is the **RSA private key**

Encoding of M: compute $P{=}M^{e} \pmod{n}$; Decoding of P: compute $Q{=}P^{d} \pmod{n}$
Theorem (Correctness of RSA): Prove that Q ≡ M ?
$Q \equiv (M^{e})^{d} \equiv M^{ed} \equiv M^{1+f\varphi(n)} \equiv M\,(M^{\varphi(n)})^{f} \equiv M$, because of Euler's Theorem: $M^{\varphi(n)} \equiv 1$, when M and n are relatively prime (what about when M and n are not relatively prime?)

PART 4.
Combinatorial Game Theory

Simple Game: "Squaring the Number"

Beginning with a positive integer (say 75), players A and B alternately subtract a positive *square* integer from it, leaving a *non-negative* integer. In other words, the players can only subtract 1, 4, 9, 16, 25, 36 etc. The player who is unable to make a move loses.

For example, a typical game might proceed as follows:

$$75 \rightarrow 50 \rightarrow 46 \rightarrow 30 \rightarrow 5 \rightarrow 4 \rightarrow 0$$

Here A's moves are indicated by a red arrow, while B's moves are indicated by a green arrow. Hence, in the above game, player B wins since A has no legal move left. There is a nagging suspicion though, that one (possibly both) of the players has not played wisely. If both players had played well, who would win?

Suppose the game begins with the non-negative integer *n*. There are two possible outcomes for the game.

a. the first player can definitely win, by playing a crucial move; or
b. no such move is available to him, so the second player can definitely win.

If (a) holds, then we say that *n* is a **winning number**; otherwise, (b) holds and we call *n* a **losing number**.

Let us look at some simple examples.

i. 0 is definitely a losing number, because no move is available to him at all!
ii. Every square number 1, 4, 9, 16, 25 etc, is a winning number, since the first player can easily reduce it to zero, thereby giving his opponent a losing position.

After some elementary reasoning, we get:

Table 1

0	1	2	3	4	5
☠		☠			☠

where the cross-bones indicate a losing number and a blank space refers to a winning number.
The rules for continuing our analysis may be stated as:

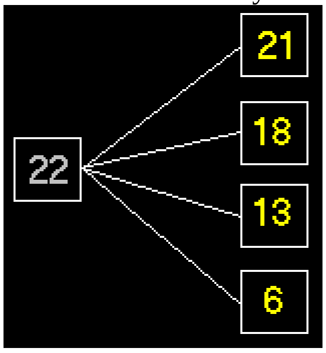
If the first player can make a move, such that the resulting number is losing, then n is a winning number.
Otherwise, each of the first player's move results in a winning number, hence n is a losing number.

If you think about the above rules of thumb, it becomes intuitively obvious why: naturally you want the other player to lose. So, *if you are the first player, you try to leave a losing number to the second player*. If you are unable to do that, you lose since every move you make is a winning number for the other player.
To give a feel of how the above guidelines work, suppose we have continued our analysis for table 1 till *n*=21 :

Table 2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
☠		☠			☠		☠			☠		☠			☠		☠			☠	

Now we wish to analyze *n*=22. From 22, we have 4 possible moves:



Each of the four moves: 6, 13, 18, 21 are winning numbers. Hence, **22 is a losing number**. This gives us an efficient method of finding the status of each number.

Making Good Moves
It is not enough to know whether each number is winning or losing. We must know how to make 'good' moves, so that we can win when placed in an advantageous situation.

Example 1 : In the game with *n*=19, what is the first move you should make?*Answer* : Look at Table 2 above, 19 is a winning number which spells good news for you. Now consider the options you have : *n*=18, 15, 10, or 3. Since you want your opponent to lose, you should give him a losing number : i.e. *n*=10 or 15.

Example 2 : In the game with *n*=18, the first player decided to subtract 4, leaving *n*=14. What would happen?
Answer : Again, from Table 2, we see that 14 is a winning number - hence the first player's move was unwise. The second player may now turn the tables around by subtracting 4, leaving a losing number *n*=10 for the first player. Victory!

Here, we shall describe an even more efficient method to find the outcome of a number *n*. This method is similar to the sieve of Eratosthenes (which finds all primes below a certain range). Suppose we wish to compute the outcomes of all numbers up to 15. We know for sure that 0 is a losing number:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
☠																	

Adding square numbers to 0 gives us 1, 4, 9 (which are winning numbers since there is a choice for player A to leave a losing number).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
☠	☺			☺					☺							☺	

So we know that the first unlabeled number is losing (2 in this case) :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
☠	☺	☠		☺					☺							☺	

Adding square numbers to 2 gives us more winning numbers: (2+1, 2+4, 2+9) :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
☠	☺	☠	☺	☺		☺			☺		☺					☺	

The next unlabeled number 5 must be losing. Continuing in this manner, we eventually obtain :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
☠	☺	☠	☺	☺	☠	☺	☠	☺	☺	☠	☺	☠	☺	☺	☠	☺	☠

Remoteness

Example 3 : I am left with the number *n*=17, which is a losing number. There's no hope for me now, is there?

Answer : Don't panic. Although in theory a losing position is a sure-lose case, you can bet that your opponent isn't well-versed enough to find the exact strategy. Hence, your next move should *serve to confuse your opponent as much as possible*. In your example, it would be daft to take away 1 leaving 16, since 16 is such a clear-cut case for your opponent. In short, we want to find a move to a winning position for the opponent, which is *as remote from the winning goal as possible*.

The remoteness values of *n* are assigned under the following rules:

i. *If n=0, then the remoteness is given 0 by default.*
ii. *If n is a winning position, then the remoteness of n is one more than the smallest possible remoteness among all the possible moves to losing positions.*
iii. *If n is a losing position, then the remoteness of n is one more than the largest possible remoteness among all possible moves.*

The above values follow from the fact that if a player is given a winning number, he'd most definitely want to win as fast as possible to minimize any chance of error. Conversely, a losing player would like to delay his opponent's win for as long as possible, hoping that his opponent would blunder in some move.
Hence the rule:

WIN QUICKLY.
LOSE SLOWLY.

To demonstrate the above rules of thumb, suppose the values of remoteness have already been found for up to *n*=16. We wish to calculate the case for *n*=17.

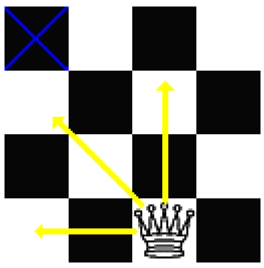
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	1	2	3	1	2	3	4	5	1	4	3	6	7	3	4	1	

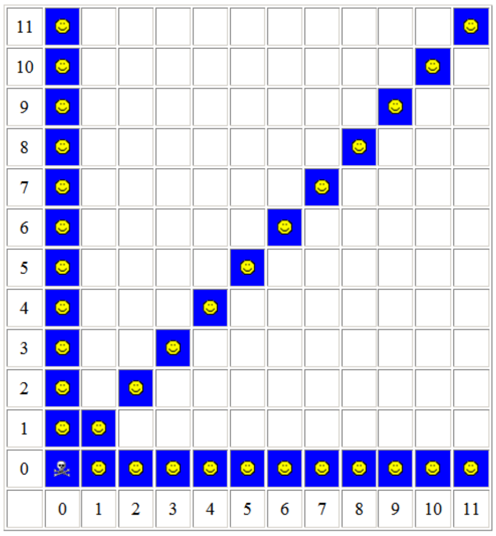
There are 4 possible moves for *n=17*, to values of *n*=16, 13, 8 or 1. The respective values of remoteness are 1, 7, 5, 1. Hence, the remoteness value of 17 is one more than the maximum of these values, which is 8. To answer Question 3, the best move is to make a move to 13, which has a remoteness value of 7, the highest among all possible moves. Finally, observe that *odd values of remoteness correspond to winning positions while even values correspond to losing positions*.

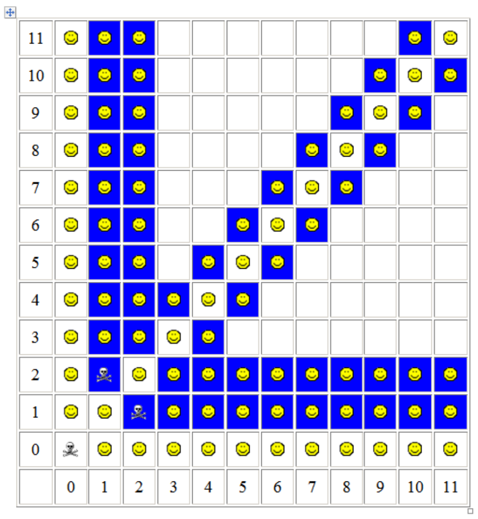
Wythoff's Game

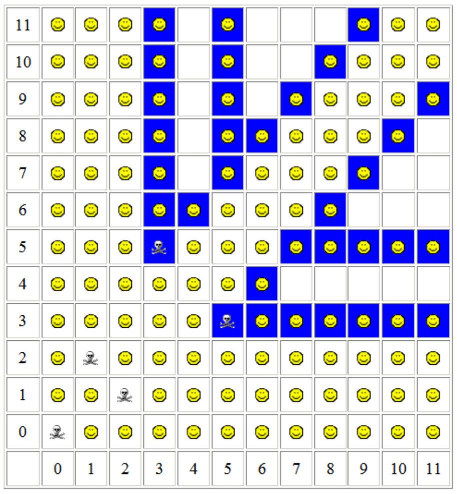
The rules of this game are as follows: given a pair of non-negative integers (which may be distinct), two players alternately make a move. Each move consists of either (a) subtracting a positive integer from one of them, or (b) subtracting the same positive integer from both. The constraint is that none of the integers may be left negative. As before, the player who is unable to make a move loses (i.e. the player who leaves (0,0) for his opponent wins).

This game is equivalent to *Wythoff's Queen* which is played on a checkerboard, with the queen at position (*m*, *n*). Each player may, upon his turn, move a queen in one of the three directions towards the home square (0,0). The player who manages to '*home*' his queen wins.









Continuing in this manner, we find the losing positions (m, n) ($m < n$) : (0,0), (1,2), (3,5), (4,7), (6,10), (8, 13) ... .In general, to find the (i+1)-th losing position, we look for the smallest positive integer k which has not appeared in the sequence (in either the first or second position), and write down (k, k+i).

Game of Nim

The game of **Nim** is played with a few piles of stones. At each player's turn, he/she may remove any number of stones, subjected to the condition that all the stones must be from a single pile. A player who is unable to remove any stones loses. An example of a game of Nim (with red/green highlighting the moves of A/B) is:

(7,8,10) ➡ (3,8,10) ➡ (3,2,10) ➡ (3,2,0) ➡ (2,2,0) ➡ (2,0,0) ➡ (0,0,0)

So the second player wins in this case. As we shall soon see, the above game is actually a win for the first player. We may perform analysis similar to the game earlier to determine whether each configuration is winning or losing. In practice however, it is far too tedious.
Here we offer a straightforward method to compute the outcome of each game of Nim.

Suppose the game consists of piles (7, 8, 10). We arrange the piles as sums of powers of 2 as below:

	8	4	2	1	Sum
Pile 1		★	★	★	7
Pile 2	★				8
Pile 3	★		★		10

The first player A attempts to make a move so that the number of stars in each column is even. This is accomplished by taking away 5 sticks from the first pile, leaving:

	8	4	2	1	Sum
Pile 1			★		2
Pile 2	★				8
Pile 3	★		★		10

Now B is forced to break this parity, so that at least one column must be odd! Suppose for instance, he chooses 3 sticks from the third pile, so that we have

	8	4	2	1	Sum
Pile 1			★		2
Pile 2	★				8
Pile 3		★	★	★	7

Player A may now retaliate by taking 3 sticks from the second pile, so that the number of stars in each column is again even.

	8	4	2	1	Sum
Pile 1			★		2
Pile 2		★		★	5
Pile 3		★	★	★	7

Thus play goes on until player A leaves (0,0,0) and wins. Hence the key to winning Nim is as follows :

A configuration is losing if and only if the number of stars in each column is even. Otherwise, it is winning.

We shall give an explicit method of expressing the above results.
Let *m* and *n* be two non-negative integers. We shall use the **XOR** operation (called the 'exclusive-or' operation) between *m* and *n*, denoted m ^ n. Write down the binary representations of *m* and *n*, one below the other and in the third row, write 0 if the 2 digits in the column are the same, and 1 if the 2 digits are different.

For example, to find 21 ^ 39, we write the binary representations as follows:

```
010101
100111
______
110010
```

Since $(110010)_2 = 50$, we have 21 ^ 39 = 50. Incidentally, the XOR operation is also known as *addition without carry* - it is not hard to see why this is named so. Now in general, we have the following rule:

A configuration is losing if and only if the configuration consists of *k* piles of sticks, containing m_1, m_2 ... m_k sticks respectively, such that
$m_1 \wedge m_2 \wedge \ldots \wedge m_k = 0$

In the above example, we have $7 = (111)_2$, $8 = (1000)_2$, $10 = (1010)_2$. Thus, $7 \wedge 8 \wedge 10 = (101)_2$ which is not zero 0 and (7, 8, 10) must be a winning configuration.

Finding the Winning Move

Here we shall give an algorithm for finding the winning move (to a losing position) in a particular configuration. Suppose the piles consist of m_1, m_2 ... m_k sticks respectively so that $m = m_1 \wedge m_2 \wedge \ldots \wedge m_k$ is non-zero. Our objective is then to reduce one of the values m_i so that their XOR-sum is zero. We shall illustrate the steps using the example, (19, 25, 12), with binary representations $(10011)_2$, $(11001)_2$ and $(01100)_2$. Thus their XOR-sum is $(00110)_2 = 6$.

1. Find the first leading non-zero digit of the XOR-sum, and remove all digits prior to that column.
   Hence, in our example we have the digits in green:

```
10011
11001
01100
_____
00110
```

2. Among the remaining summands, find the one with the largest value.
   In our case, we must take the last summand which is (01100).
3. Change this summand to the desired value.
   In order to obtain (00000) in the final XOR-sum, we must replace the green digits of the third summand by 010. This gives us (01010) which is 10 in binary. Hence, we must 2 sticks from the last pile.

Nim Variants

A game, which is actually a variant of Nim.

- Mom has just baked a large almond cake which is divided into *m*-by-*n* unit squares. Unfortunately, one of the squares is mouldy and must not be consumed. Her two sons, Alfred and Bob, decide to play a game - at each turn one of them would cut the cake into two (by a single horizontal/vertical slice) and consume the part which does not contain the mouldy square. Whoever is left with the mouldy piece is deemed to have lost and must perform forfeit by washing the dishes. Suppose Alfred starts the game for the cake below. Who wins?