COS 460 – Algorithms (week 07) Mathematical algorithms (PARTS 1,2,3,4).

PART 1. Linear Programming. Why significant?

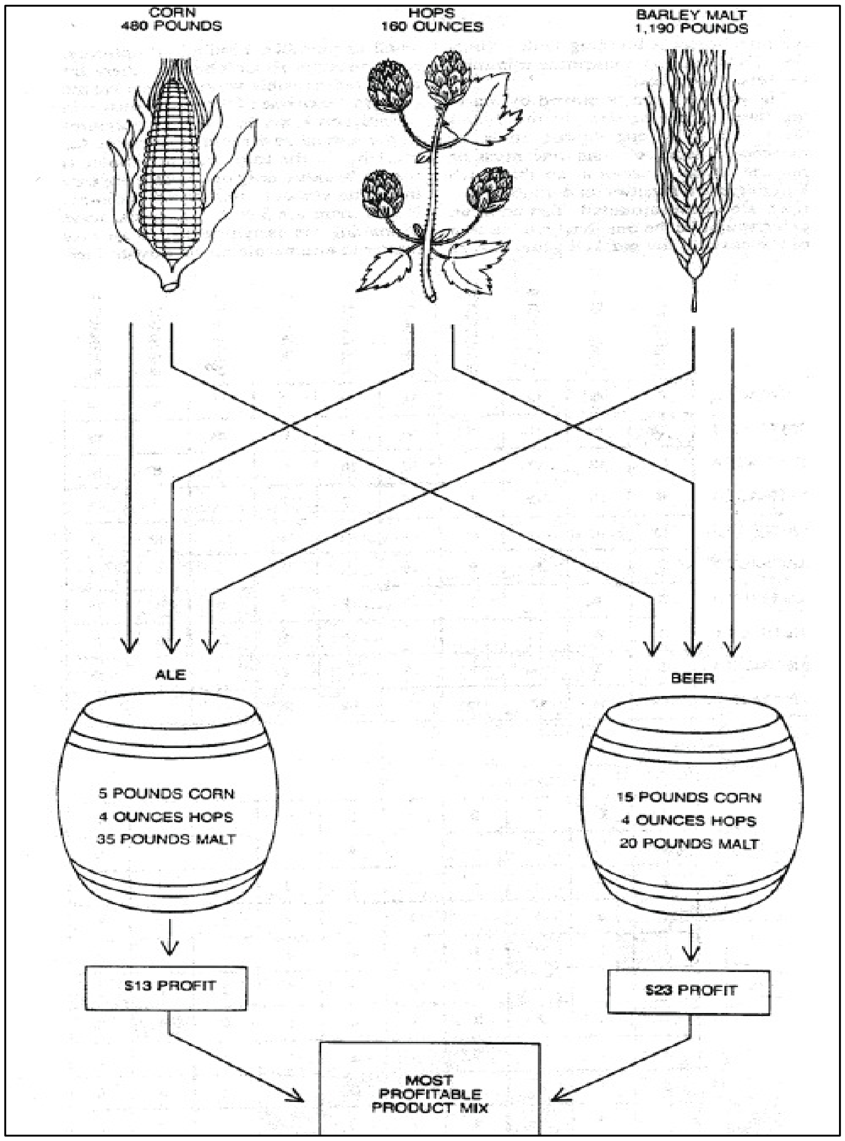
- Fast commercial solvers: CPLEX, OSL, and free solvers: LP_SOLVE .
- Powerful modeling languages: AMPL, GAMS.
- Ranked among most important scientific advances of 20th century.
- Widely applicable and dominates world of industry.

Applications:
Agriculture. Diet problem.
Computer science. Compiler register allocation, data mining.
Electrical engineering. VLSI (very large scale integration) design, optimal clocking.
Energy. Blending petroleum products.
Economics. Equilibrium theory, two-person zero-sum games.
Environment. Water quality management.
Finance. Portfolio optimization.
Logistics. Supply-chain management.
Management. Hotel yield management.
Marketing. Direct mail advertising.
Manufacturing. Production line balancing, cutting stock.
Medicine. Radioactive seed placement in cancer treatment.
Operations research. Airline crew assignment, vehicle routing.
Plasma physics. Optimal stellarator design.
Telecommunication. Network design, Internet routing.

What is it?

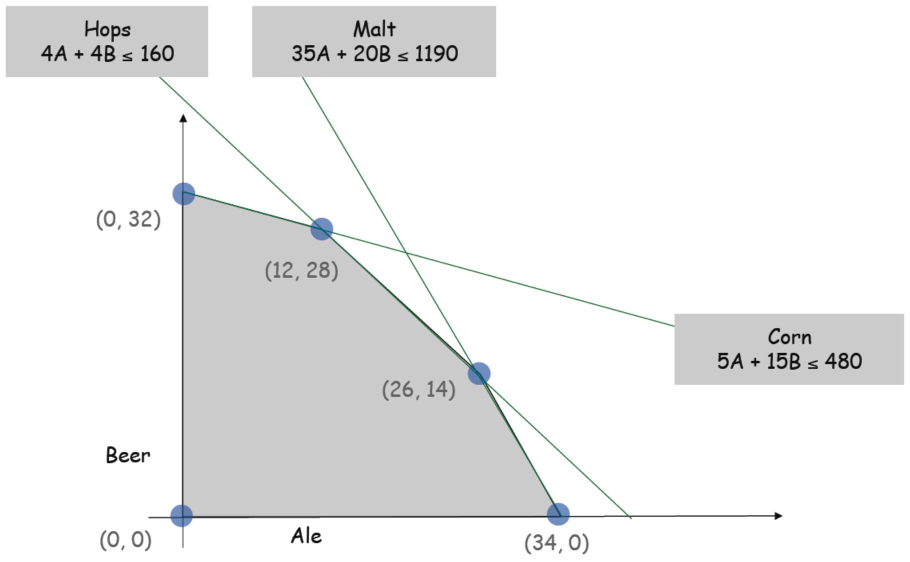
- Tool for optimal allocation of scarce resources, among a number of competing activities.
- Powerful and general problem-solving method that encompasses:
  - shortest path, network flow, MST, matching
  - Ax = b
  - 2-person zero sum games

Example: Brewery Problem:

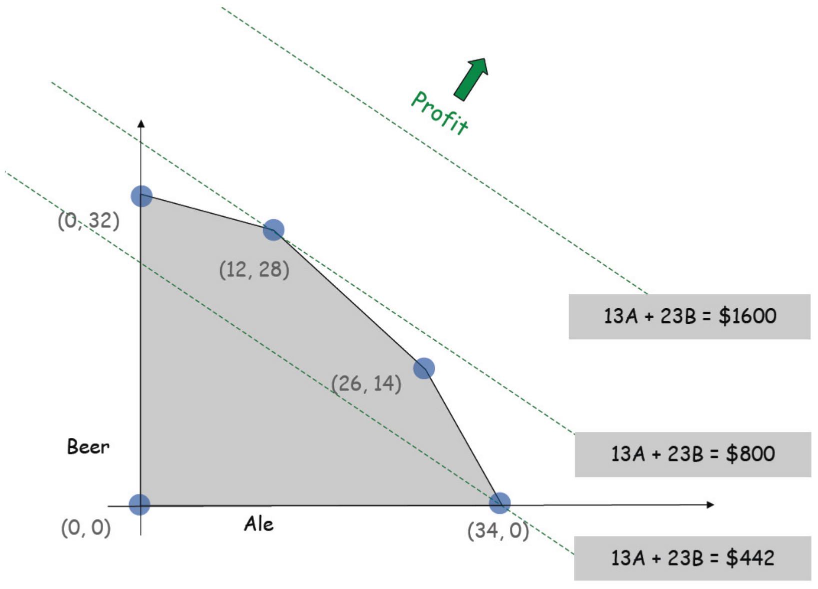


$$\begin{array}{llcccccl}
 & \text{Ale} & & \text{Beer} & & & & \\
\max & 13A & + & 23B & & & & \text{Profit} \\
\text{s. t.} & 5A & + & 15B & \leq & 480 & & \text{Corn} \\
 & 4A & + & 4B & \leq & 160 & & \text{Hops} \\
 & 35A & + & 20B & \leq & 1190 & & \text{Malt} \\
 & A & , & B & \geq & 0 & &
\end{array}$$

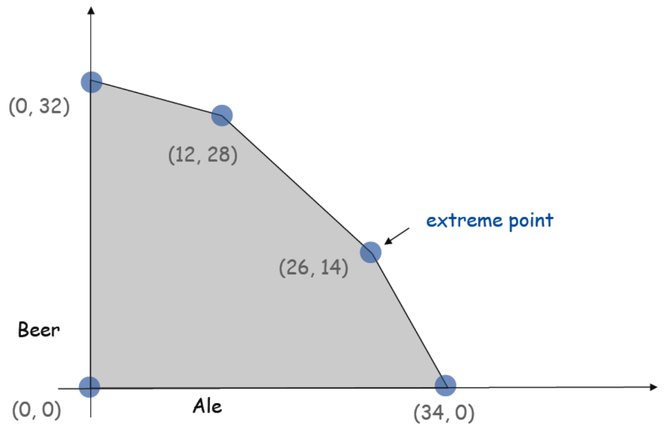
Brewery Problem: Feasible Region:



Brewery Problem: Objective Function:



Geometry: Regardless of objective function coefficients, an optimal solution occurs at an extreme point:



Standard Form LP

"Standard form" LP.

- Input: real numbers a_{ij}, c_j, b_i.
- Output: real numbers x_j.
- n = # nonnegative variables, m = # constraints.
- Maximize linear objective function subject to linear inequalities.

$$
\begin{array}{llrcll}
\text{(P)} & \max & \sum_{j=1}^{n} c_j x_j & & & \\
 & \text{s. t.} & \sum_{j=1}^{n} a_{ij} x_j & = & b_i & 1 \le i \le m \\
 & & x_j & \ge & 0 & 1 \le j \le n
\end{array}
\qquad
\begin{array}{llrcl}
\text{(P)} & \max & c^T x & & \\
 & \text{s. t.} & Ax & = & b \\
 & & x & \ge & 0
\end{array}
$$

It is linear; Programming: Planning (term predates computer programming).

Brewery Problem: Converting to Standard Form:

Original input.

$$
\begin{array}{lrcrcr}
\max & 13A & + & 23B & & \\
\text{s. t.} & 5A & + & 15B & \le & 480 \\
 & 4A & + & 4B & \le & 160 \\
 & 35A & + & 20B & \le & 1190 \\
 & A & , & B & \ge & 0
\end{array}
$$

Standard form.

- Add slack variable for each inequality.
- Now a 5-dimensional problem.

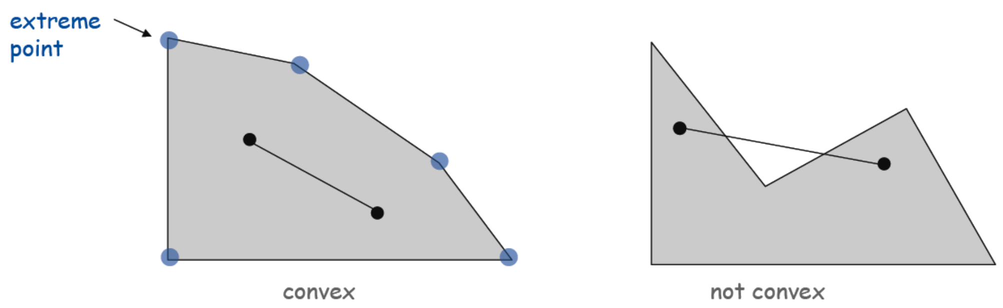
$$
\begin{array}{lrcrcccccccr}
\max & 13A & + & 23B & & & & & & & & \\
\text{s. t.} & 5A & + & 15B & + & S_C & & & & & = & 480 \\
 & 4A & + & 4B & & & + & S_H & & & = & 160 \\
 & 35A & + & 20B & & & & & + & S_M & = & 1190 \\
 & A & , & B & , & S_C & , & S_H & , & S_M & \ge & 0
\end{array}
$$

Geometry.

- Inequality: halfplane (2D), hyperplane (kD).
- Bounded feasible region: convex polygon (2D), convex polytope (kD).

Convex set. If two points a and b are in the set, then so is $\frac{1}{2}(a + b)$.

Extreme point. A point in the set that can't be written as $\frac{1}{2}(a + b)$, where a and b are two distinct points in the set.



Extreme point property. If there exists an optimal solution to (P), then there exists one that is an extreme point.

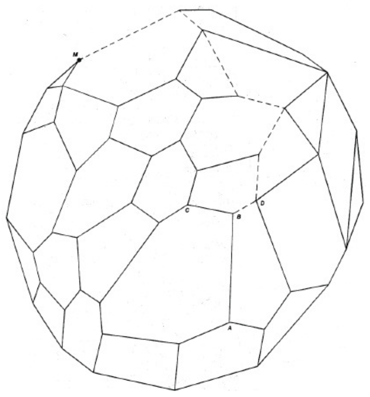
Consequence. Only need to consider finitely many possible solutions.

Challenge. Number of extreme points can be exponential!

↗ n-dimensional hypercube

Greedy property. Extreme point is optimal iff no neighboring extreme point is better.

↗ local optima are global optima



Simplex algorithm. [George Dantzig, 1947]

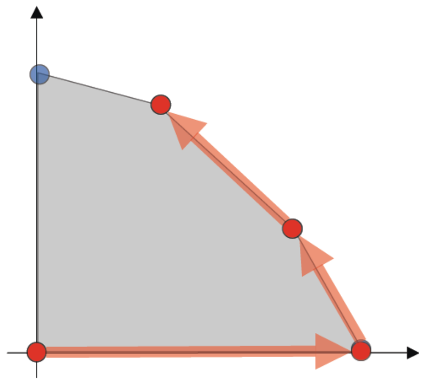
- Developed shortly after WWII in response to logistical problems, including Berlin airlift.
- One of greatest and most successful algorithms of all time.

Generic algorithm.

never decreasing objective function

- Start at some extreme point.
- Pivot from one extreme point to a neighboring one.
- Repeat until optimal.

How to implement? Linear algebra.

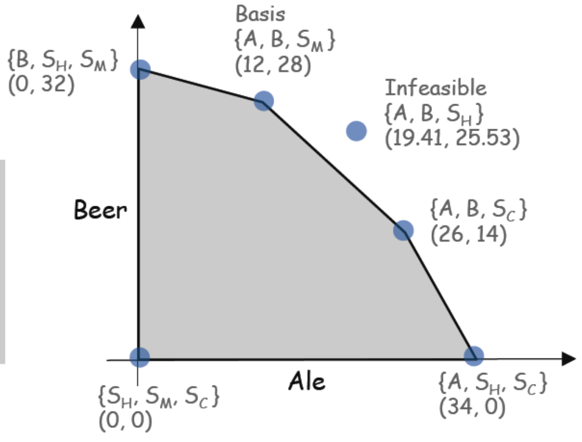


Basis. Subset of m of the n variables.

Basic feasible solution (BFS). Set n - m nonbasic variables to 0, solve for remaining m variables.

- Solve m equations in m unknowns.
- If unique and feasible solution $\Rightarrow$ BFS.
- BFS $\Leftrightarrow$ extreme point.

$$
\begin{array}{llllllllll}
\text{max} & 13A & + & 23B & & & & & & \\
\text{s. t.} & 5A & + & 15B & + S_C & & & = & 480 \\
 & 4A & + & 4B & & + S_H & & = & 160 \\
 & 35A & + & 20B & & & + S_M & = & 1190 \\
 & A & , & B & , S_C & , S_H & , S_M & \geq & 0
\end{array}
$$



Simplex Algorithm: Initialization

max Z subject to

$$\begin{array}{rcrcrcrcrcrcr}
13A & + & 23B & & & & & & & - & Z & = & 0 \\ \hline
5A & + & 15B & + & S_C & & & & & & & = & 480 \\
4A & + & 4B & & & + & S_H & & & & & = & 160 \\
35A & + & 20B & & & & & + & S_M & & & = & 1190 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}$$

Basis = $\{S_C, S_H, S_M\}$
$A = B = 0$
$Z = 0$
$S_C = 480$
$S_H = 160$
$S_M = 1190$

Simplex Algorithm: Pivot 1

max Z subject to

$$\begin{array}{rcrcrcrcrcrcr}
13A & + & 23B & & & & & & & - & Z & = & 0 \\ \hline
5A & + & 15B & + & S_C & & & & & & & = & 480 \\
4A & + & 4B & & & + & S_H & & & & & = & 160 \\
35A & + & 20B & & & & & + & S_M & & & = & 1190 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}$$

Basis = $\{S_C, S_H, S_M\}$
$A = B = 0$
$Z = 0$
$S_C = 480$
$S_H = 160$
$S_M = 1190$

Substitute: $B = 1/15\ (480 - 5A - S_C)$

max Z subject to

$$\begin{array}{rcrcrcrcrcrcr}
\frac{16}{3}A & & & - & \frac{23}{15}S_C & & & & & - & Z & = & -736 \\ \hline
\frac{1}{3}A & + & B & + & \frac{1}{15}S_C & & & & & & & = & 32 \\
\frac{8}{3}A & & & - & \frac{4}{15}S_C & + & S_H & & & & & = & 32 \\
\frac{85}{3}A & & & - & \frac{4}{3}S_C & & & + & S_M & & & = & 550 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}$$

Basis = $\{B, S_H, S_M\}$
$A = S_C = 0$
$Z = 736$
$B = 32$
$S_H = 32$
$S_M = 550$

Simplex Algorithm: Pivot 1

max Z subject to

$$\begin{array}{rcrcrcrcrcrcr}
13A & + & 23B & & & & & & & - & Z & = & 0 \\ \hline
5A & + & 15B & + & S_C & & & & & & & = & 480 \\
4A & + & 4B & & & + & S_H & & & & & = & 160 \\
35A & + & 20B & & & & & + & S_M & & & = & 1190 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}$$

Basis = $\{S_C, S_H, S_M\}$
$A = B = 0$
$Z = 0$
$S_C = 480$
$S_H = 160$
$S_M = 1190$

Why pivot on column 2?

- Each unit increase in B increases objective value by \$23.
- Pivoting on column 1 also OK.

Why pivot on row 2?

- Preserves feasibility by ensuring RHS ≥ 0.
- Minimum ratio rule: min { 480/15, 160/4, 1190/20 }.

Simplex Algorithm: Pivot 2

max Z subject to

$$\begin{array}{rcrcrcrcrcrcl}
\frac{16}{3} A & & & - & \frac{23}{15} S_C & & & & & - & Z & = & -736 \\
\hline
\frac{1}{3} A & + & B & + & \frac{1}{15} S_C & & & & & & & = & 32 \\
\frac{8}{3} A & & & - & \frac{4}{15} S_C & + & S_H & & & & & = & 32 \\
\frac{85}{3} A & & & - & \frac{4}{3} S_C & & & + & S_M & & & = & 550 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}$$

Basis = $\{B, S_H, S_M\}$
$A = S_C = 0$
$Z = 736$
$B = 32$
$S_H = 32$
$S_M = 550$

Substitute: $A = 3/8\ (32 + 4/15\ S_C - S_H)$

max Z subject to

$$\begin{array}{rcrcrcrcrcrcl}
 & & & - & S_C & - & 2\ S_H & & & - & Z & = & -800 \\
\hline
 & & B & + & \frac{1}{10} S_C & + & \frac{1}{8} S_H & & & & & = & 28 \\
A & & & - & \frac{1}{10} S_C & + & \frac{3}{8} S_H & & & & & = & 12 \\
 & & & - & \frac{25}{6} S_C & - & \frac{85}{8} S_H & + & S_M & & & = & 110 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}$$

Basis = $\{A, B, S_M\}$
$S_C = S_H = 0$
$Z = 800$
$B = 28$
$A = 12$
$S_M = 110$

Simplex Algorithm: Optimality

Q. When to stop pivoting?

A. When all coefficients in top row are non-positive.

Q. Why is resulting solution optimal?

A. Any feasible solution satisfies system of equations in tableaux.

- In particular: $Z = 800 - S_C - 2\ S_H$
- Thus, optimal objective value $Z^* \leq 800$ since $S_C, S_H \geq 0$.
- Current BFS has value 800 $\Rightarrow$ optimal.

max Z subject to

$$\begin{array}{rcrcrcrcrcrcl}
 & & & - & S_C & - & 2\ S_H & & & - & Z & = & -800 \\
\hline
 & & B & + & \frac{1}{10} S_C & + & \frac{1}{8} S_H & & & & & = & 28 \\
A & & & - & \frac{1}{10} S_C & + & \frac{3}{8} S_H & & & & & = & 12 \\
 & & & - & \frac{25}{6} S_C & - & \frac{85}{8} S_H & + & S_M & & & = & 110 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}$$

Basis = $\{A, B, S_M\}$
$S_C = S_H = 0$
$Z = 800$
$B = 28$
$A = 12$
$S_M = 110$

Remarkable property: In practice, simplex algorithm typically terminates after at most 2(m+n) pivots.

- No polynomial pivot rule known.
- Most pivot rules known to be exponential (or worse) in worst-case.

History
1939. Production, planning. [Kantorovich]
1947. Simplex algorithm. [Dantzig]
1950. Applications in many fields.
1975. Nobel prize in Economics. [Kantorovich and Koopmans]
1979. Ellipsoid algorithm. [Khachian]
1984. Projective scaling algorithm. [Karmarkar]
1990. Interior point methods.
200x. Approximation algorithms, large scale optimization.

PART 2. 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 $\rightarrow$ 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

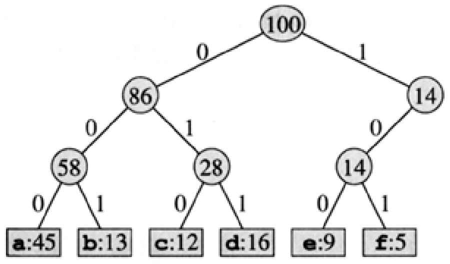
A total of 224,000 bits, much more compact than the original 300,000 bits. How do we choose the variable length code words?

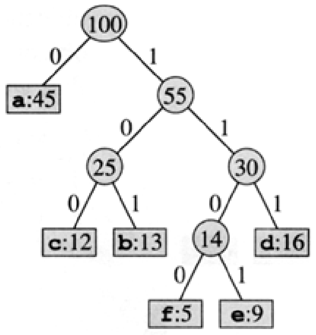
Fixed Codes: Code words are not prefixes of other code words. Can show that optimal compression can always be achieved with a prefix code, so only consider them.

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.





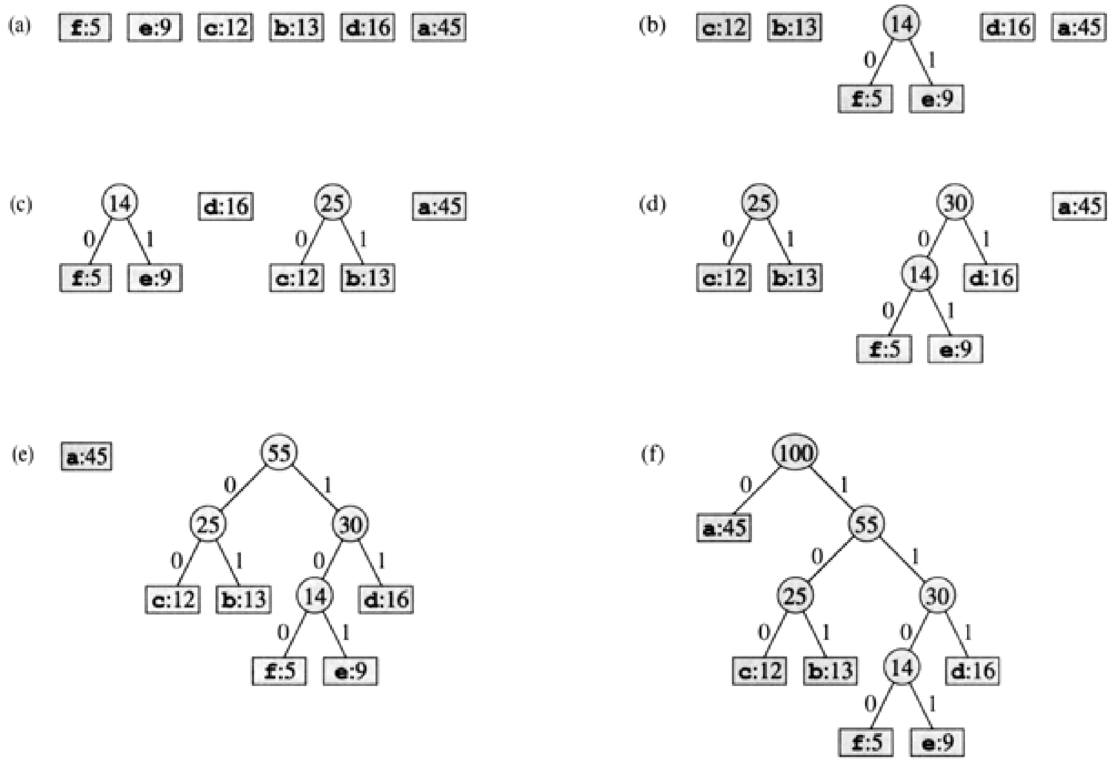
An optimal code for is always represented as a full binary tree. **Full** means every non-leaf child has two children.

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 3. 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.

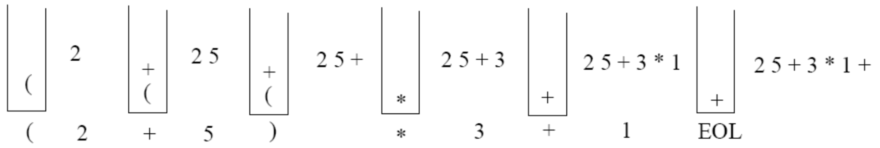
Converting infix notation to postfix notation
To convert an expression which in an infix expression to its equivalent in postfix notation, we must know the precedence and associativity of operators. *Precedence* or operator strength determines order of evaluation; an operator with higher precedence is 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.

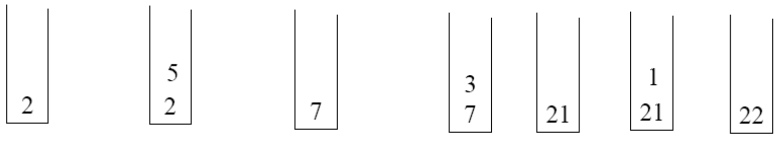
Example: Infix expression: (2 + 5) * 3 + 1
The operator stack holds just the operators. Operands are sent to the output directly.



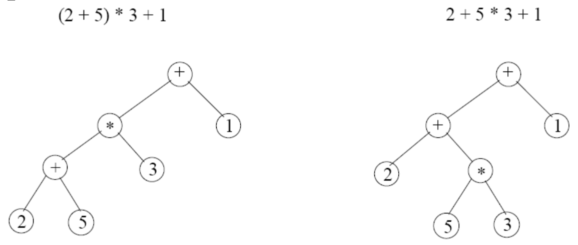
Postfix expression evaluation. Evaluating a postfix expression is simpler than directly evaluating an infix expression. In postfix notation, the need for parentheses is eliminated and the priority of the operators is no longer relevant. You can use the following algorithm to evaluate postfix expressions:

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.



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

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

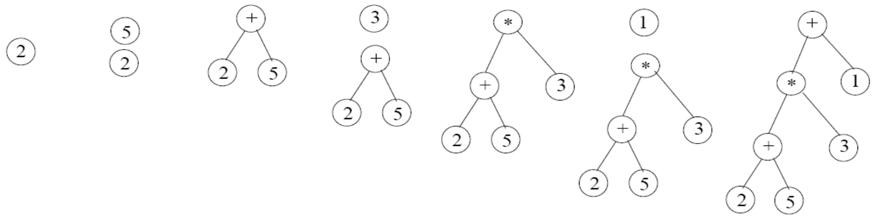
This grammar describes expressions like those that we used, e.g. (A*B+A*C)*D. Each line in the grammar is called a production or replacement rule. The productions consist of terminal symbols (,) , + and * which are the symbols used in the language being described; nonterminal symbols like *expression*, *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 4. 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 in Z}, e. g. $[3]_7$ = {...,–11,–4, 3, 10, 17, ...}

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

–1 ∈ $[n – 1]_n$ since n – 1 (mod n) ≡ –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 ∈ 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 ∈ 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 ∈ 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 ≥ 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 ≥ 0 and Euclid(a, b) performs k ≥ 1 recursive calls, then a ≥ F_k+2 and b ≥ 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 $(S, \cdot)$ is a set *S*, with a binary operation, ·, with
1. Closure: $\forall a, b \in S, a \cdot b \in S$
2. Identity: $\exists e \in S$ such that $e \cdot a = a \cdot e = a, \forall a \in S$
3. Associativity: $\forall a, b, c \in S, (a \cdot b) \cdot c = a \cdot (b \cdot c)$
4. Inverse: $\forall a \in S, \exists ! b \in S$ such that $a \cdot b = b \cdot a = e$

If $(S,\cdot)$ is commutative: $a \cdot b = b \cdot a$, $\forall a, b \in S$, then $(S,\cdot)$ is called *Abelian*.
If $(S,\cdot)$ 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 $\cdot$. 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, …, p_n, all relatively prime, and remainders r_1, r_2, …, r_n, find x such that $x\%p_1=r_1$, $x\%p_2=r_2$, …, $x\%p_n=r_n$.

Method: Let $q_i= (p_1, p_2, …, p_n)/p_i$. 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 $x = r_1q_1+r_2q_2+…+r_nq_n$.

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 (mod 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 (mod 7)	1	2	4	1	2	4	1	2	4	1	2	4	....

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

• **Theorem:** (Fermat's little) If *p* is prime, then $a^{p-1} \equiv 1$ (mod *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 $P_A() : D \rightarrow D$ 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 $d \equiv e^{-1}$ (mod $\varphi(n)$); 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 $P=M^e$ (mod *n*); Decoding of *P*: compute $Q=P^d$ (mod *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?)