CS 3391 Note

- Data types

Type	Storage	Maximum	Minimum
char	1 byte	127	-128
short	2 byte	32,767	-32,768
long	4 byte	2,147,483,647	-2,147,483,648
int	4 byte	2,147,483,647	-2,147,483,648
float	4 byte	3.4E +/- 38 (7 digits)	3.4E +/- 38 (7 digits)
double	8 byte	1.7E +/- 308 (15 digits)	1.7E +/- 308 (15 digits)
bool (C++)	1 byte	true (if non-zero)	false (if zero)

  - unsigned version to extend
  - Self-defined class for big number:
    - use an array `bit[]` to represent a big number
    - `bit[0]` store the number of digits

```
r = bit[0];
n = bit[1]*1+bit[2]*10+...+bit[r]*(10^(r-1))
```

```
void numberToArray(int n, int bit[]){
    bit[0] = 0;
    while(n){
    bit[++bit[0]] = n%10;
        n/=10;
    }
}
```

Addition: operation, carry-bit, carry flag

```
void add(int a[], int b[], int c[]){
    // operation
    memset(c,0,sizeof(c));
    for(int i=1;i<=a[0];i++)c[i]+=a[i];
    for(int i=1;i<=b[0];i++)c[i]+=b[i];
    c[0] = max(a[0],b[0]);
    // carry-bit
    int flag = 0;
    for(int i=1;i<=c[0];i++){
        c[i]+=flag;
        flag = c[i]/10;
        c[i] = c[i]%10;
    }
```

```
    // carry-flag
    if(flag) c[++c[0]]=flag;

}
```

- Mutiplication: operation, carry-bit, carry flag

```
void mutiply(int a[],int b[],int c[]){
  memset(c,0,sizeof(c));
    for(int i=1;i<=a[0];i++){
        for(int j=1;j<=b[0];j++){
            c[i+j-1]+=a[i]*b[j];  // i+(j-1)
        }
    }
    c[0]=a[0]+b[0]-1;
    int flag = 0;
    for(int i=1;i<=c[0];i++){
        c[i]+=flag;
        flag = c[i]/10;
        c[i] = c[i]%10;
    }
    while(flag){
        c[++c[0]]=flag%10;
        flag = flag/10;
    }
}
```

- Speed up: Base could be larger than 10, int can be large as $2^{31}-1$
- digit compression
- before(base 10): bit[1,...,n]<10
- After compression(base: 1e5): bit[1,...n]<1e5
- **be careful for overflow**

- place holders:

	Meaning	Param type	Example	Output
%d	Decimal	int, short	int a=1; printf("A = %d",a);	A = 1
%u	Unsigned Decimal	unsigned int	unsigned int b=3; printf("B = %u",b);	B = 3
%ld	Long Decimal	long	long c = 123; printf("C = %ld",c);	C = 123
%f	Floating point	float	float d = 3.141592; printf("D = %f",d);	D = 3.141592
%lf	Long Floating point	double	double e = 3.141592; printf("E = %lf",e);	E = 3.141592
%c	Character	char	char f = 'X'; printf("F = %c",f);	F = X
%s	String	char[]	char g[] = "Test"; printf("G = %s",g);	G = Test
%x %X	Hexadecimal	char, short int, long	int h = 255; printf("H = %x, %X,h);	H = ff, FF
%%	The '%' sign	N/A	printf("100%%");	100%

Format	Output	Description
"*%5d*"	* 7*	Totally 5 places for number, right justify
"*%-5d*"	*7 *	Totally 5 places for number, left justify
"*%05d*"	*00007*	Totally 5 places, packed with leading zeroes

Format	Output	Description
"*%.2lf*"	*12.30*	2 places after decimal point.
"*%+.2lf*"	*+12.30*	2 places after decimal point, show sign.
"*%6.2lf*"	* 12.30*	6 places (including decimal point), 2 decimal places, right justify
"*%06.2lf*"	*012.30*	Totally 6 places, 2 decimal places, packed with leading zeroes

- I/O

```
#define SCF(a)                scanf("%d",&a)
#define SCF2(a,b)             scanf("%d%d",&a,&b)
#define IN(a)                 cin >> a
#define FOR(i,a,b)            for(int i=a;i<b;i++)
typedef long long Int;
```

```
// speed up stream I/O
std::ios_base::sync_with_stdio(false);
```

- File I/O

```
FILE* inputfile = fopen("input.txt","r");
FILE* outputfile = fopen("output.txt","w");
freopen("input.txt","r",stdin);
fprintf(outputfile,"%d",num);
char ch = fgetc(inputfile);
fgets(buffer,length,stdin); //will read \n into the buffer
feof()  // if reach the end-of-file, return 0
fclose(inputfile); fclose(outputfile);
```

- String I/O

```
sprintf(buffer,"%d",n);
sscanf(buffer,"%d",&n);
void strcpy(char* src,char* tar){
    sprintf(tar,"%s",src);
}
void strcat(char* tar,char* src1,char* src2){
    sprintf(tar,"%s%s",src1,src2);
}
int atoi(char* buffer){
    int num;
    sscanf(buffer,"%d",&num);
    return num;
}
```

- Arithmetic functions

	Explanation	Example	Result
log(double)	Natural log (base e)	x = log(100)	x = 4.61
log10(double)	Log, using base 10	x = log10(100)	x = 2.00
modf(double, double*)	Return integer & fractional part of a number	y = modf(12.34, &x)	x = 12.0 y = 0.34
pow(double, double)	x raised to the power of y	x = pow(2,3)	x = 8.00
sqrt(double)	square root of a number	x = sqrt(16)	x = 4.00
fabs(double)	absolute for floating-point	x = fabs(-12)	x = 12.0

  - sin(),cos(),tan() accept parameters in radians
  - binary power

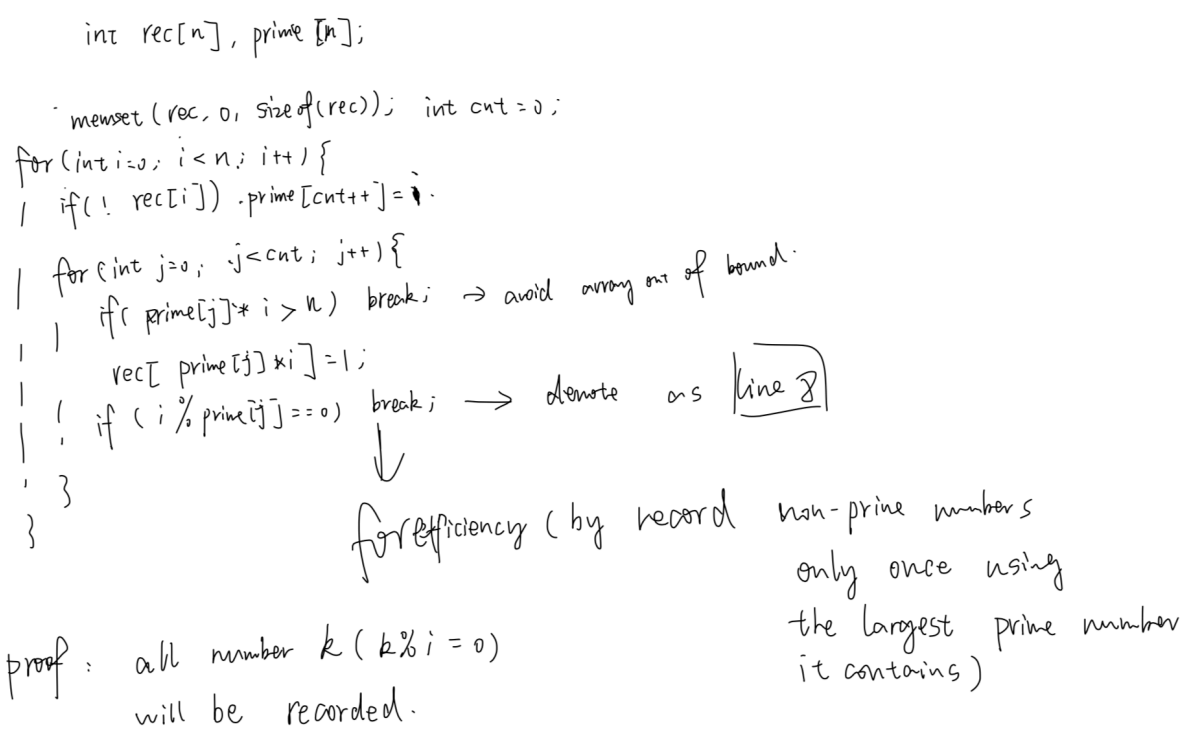
```cpp
long long bi_power(int m,int n){
    if(n==0)return 1L;
    else if(n&1) return bi_power(m,n/2)*bi_power(m,n/2)*(long long)m;
    else return bi_power(m,n/2)*bi_power(m,n/2);
}
```

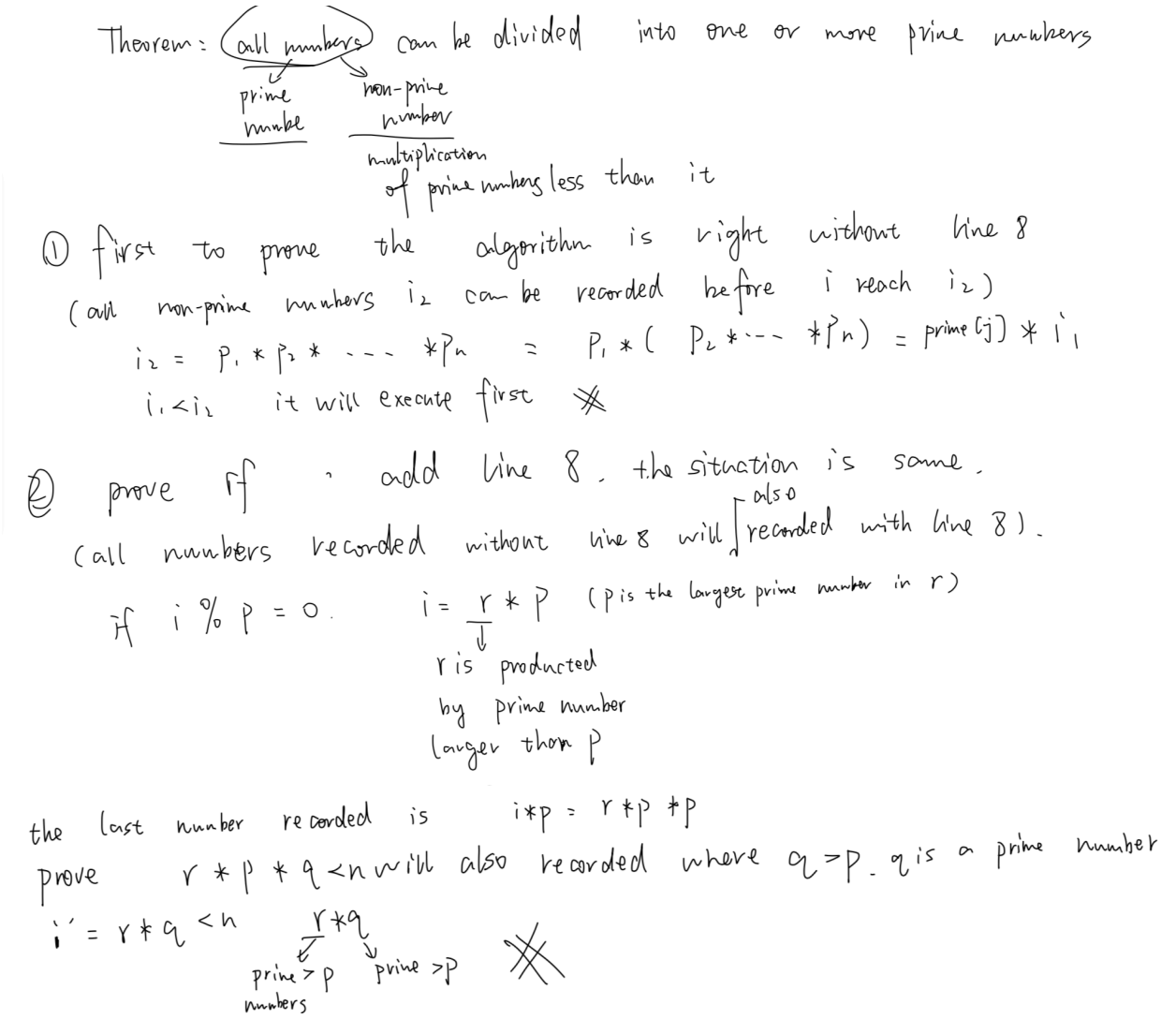
- self defined log

```cpp
double log(int down,int up){
    return log(up)/log(down);
}
```

- Bitwise operation
  - Check odd or even: `if(x&1) printf("x is odd");`
  - Check for divisibility of 2's power: `if(x&7) printf("x is not divisible by 8");`
  - Round down to nearest multiple of 2's power: `x=27; =x&(~7)`

- Counting prime
  - Euler's method
  - Proof:






```cpp
#define MAX 1000000
int prime[MAX];
bool rec[MAX+1];
void init(){
    memset(rec,0,sizeof(rec));
    int cnt = 0;
    for(int i=2;i<=MAX;i++){
        if(!rec[i]) prime[cnt++] = i;
        for(int j=0;j<cnt;j++){
            if(prime[j]*i>MAX)break;
            rec[prime[j]*i] = 1;
            if(i%prime[j]==0)break;
        }
    }
}
```

- Rational Numbers:
  - each rational number $\frac{x}{y}$ can be represented by 2 integers x and y, where x is the numberator and y is the denominator
    - Addition: $x = x_1y_2 + x_2y_1, y = y_1y_2$

    - Subtraction: : $x = x_1y_2 - x_2y_1, y = y_1y_2$
    - Mutiplication: $x = x_1x_2, y = y_1y_2$
    - Division: $x = x_1y_2, y = x_2y_1$
    - Reduce the fractoins to their simplest representation => use **greatest common divisor (GCD)**
  - Also can use two number one is integer part, the other is the fraction part (with a significant digits)
  - How to find such a fraction for representing an infinite decimal number?
    - e.g. Find $\frac{a}{b} = 0.0123123\ldots$
    - Set repeated part $R = 123$ and set length of repeated part $L = 3$
    - consider $10^L * (\frac{a}{b}) - \frac{a}{b} = 12.3.. = R/10$
    - Hence $a/b = R/10/(10^3 - 1) = \frac{123}{9990}$
- Modular Arithmetic: The number we are dividing by is called the modulus, and the remainder left over is called residue

Addition/Subtraction: $(x \pm y) \bmod n = ((x \bmod n) \pm (y \bmod n)) \bmod n$.
Suppose x is a negative number in $[-n+1, -1]$, then $x \bmod n = x + n \in \{0, 1, \cdots, n-1\}$
E.g. $-3 \bmod 5 = 2$
Multiplication: $xy \bmod n = (x \bmod n)(y \bmod n) \bmod n$.
Thus, $x^y \bmod n = (x \bmod n)^y \bmod n$.
E.g. $8 \cdot 5 \bmod 3 = (8 \bmod 3)(5 \bmod 3) \bmod 3$
$= 2 \cdot 2 \bmod 3 = 1$.

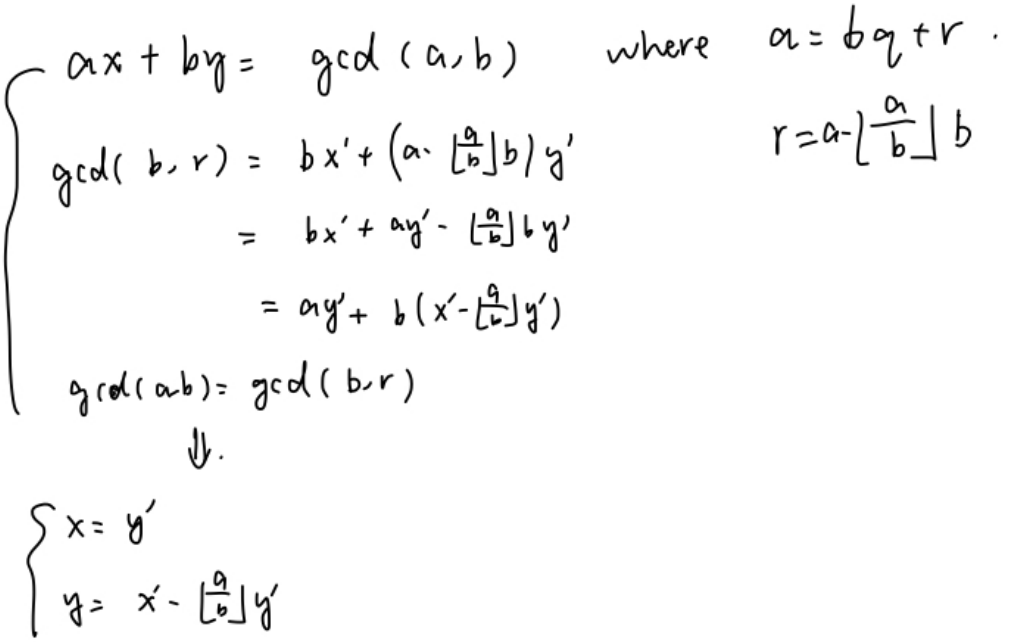
$$(k_1 n + b_1)(k_2 n + b_2)$$

$$= k_1 k_2 n^2 + (k_1 b_2 + k_2 b_1) n + b_1 b_2$$

Finding the Last Digit: What is the last digit of 2^{100}?

$$
\begin{aligned}
2^3 \bmod 10 &= 8 \\
2^6 \bmod 10 &= 8 \times 8 \bmod 10 \to 4 \\
2^{12} \bmod 10 &= 4 \times 4 \bmod 10 \to 6 \\
2^{24} \bmod 10 &= 6 \times 6 \bmod 10 \to 6 \\
2^{48} \bmod 10 &= 6 \times 6 \bmod 10 \to 6 \\
2^{96} \bmod 10 &= 6 \times 6 \bmod 10 \to 6 \\
2^{100} \bmod 10 &= 2^{96} \times 2^3 \times 2^1 \bmod 10 \to 6
\end{aligned}
$$

- GCD: The largest divisor shared by a given pair of integers
  - 2 integers are **relatively prime** if their GCD is 1
  - Important Theorem related to GCD(Euclidean's algorithm) => If $a = qb + r$ for some integer q and r, then $gcd(a, b) = gcd(r, b) = gcd(a \ mod \ b, b)$
  - Extended Euclid's Algorithm:solve the equation $ax + by = gcd(a, b)$
    - Always keep the first parameter of gcd to be the larger one
    - Base case: If the second parameter is zero, that means the previous $gcd(a, b)$ satisfies $a\%b = 0$, and b becomes current a, then a itself is gcd => $gcd * 1 + 0 = gcd$




```
long gcd(long p, long q, long* x, long* y){
    long x1, y1;
    long g;
```

```
    if(q>p) return gcd(q,p,y,x);
    if(q==0){
        *x = 1;
        *y = 0;
        return p;
    }
    g = gcd(q,p%q,&x1,&y1);

    *x = y1;
    *y = (x1 - floor(p/q)*y1);
    return g;
}
```

- LCM (least common multiple)
  - $lcm(x, y) \geq \max(x, y)$ for any x, y
  - $lcm(x, y) \leq x \cdot y$
  - $lcm(x, y) = x \cdot y / \gcd(x, y)$.

- Modular Inverse:
  - Division: The inverse $b^{-1} = 1/b$ of an integer b is an integer such that $bb^{-1} \equiv 1 \pmod{n}$.
  - But this inverse doesn't always exist - try to find a solution to $2x \equiv 1 \pmod{4}$.
  - An inverse only exists if $\gcd(b, n) = 1$.
    - if gcd = 1, always exists an answer

$as + nt = gcd(a, n) = 1$

$(as + nt) \% n = 1$

$as \% n + (nt) \% n = as \% n = 1$

$a = s^{-1}$

- s, t are integers, gcd is less than **a** and **n**, so s and t must have one negative number
- if gcd is not 1, the minus(difference) between several a and n will be some multiple of gcd(a,n)

- Euler's Theorem:
  - The multiplicative group for Z_n, denoted with Z_n, is the subset of elements of Z_n relatively prime with n, less than it and
  - The totient function of $\phi(n)$, denoted with $\phi(n)$, is the size of Z_n
  - Example

$$Z^*_{10} = \{1, 3, 7, 9\} \qquad \phi(10) = 4$$

If p is prime, we have

$$Z^*_p = \{1, 2, \ldots, (p-1)\} \qquad \phi(p) = p - 1$$

  - Euler's Theorem:

Euler's Theorem

For each element x of Z^*_n, we have $x^{\phi(n)} \bmod n = 1$

- Example ($n = 10$)

$3^{\phi(10)} \bmod 10 = 3^4 \bmod 10 = 81 \bmod 10 = 1$

$7^{\phi(10)} \bmod 10 = 7^4 \bmod 10 = 2401 \bmod 10 = 1$

$9^{\phi(10)} \bmod 10 = 9^4 \bmod 10 = 6561 \bmod 10 = 1$

    - where n is **multiplication** of 2 prime number p, q

- Actually x is relative prime with n
- if p,q are prime: $\phi(pq) = (p-1)(q-1) = pq - p - q + 1$
- proof

  欧拉定理：

  对于互质的正整数 a 和 n，有 $a^{\varphi(n)} \equiv 1 \bmod n$ 。

  证明：

  (1) 令 $Zn = \{x_1, x_2, ..., x_{\varphi(n)}\}$ ， $S = \{a * x_1 \bmod n, a * x_2 \bmod n, ... , a * x_{\varphi(n)} \bmod n\}$ ，

  则 Zn = S 。

  ① 因为 a 与 n 互质， x_i $(1 \le i \le \varphi(n))$ 与 n 互质， 所以 $a * x_i$ 与 n 互质，所以 $a * x_i \bmod n \in Zn$ 。

  ② 若 $i \ne j$， 那么 $x_i \ne x_j$，且由 a, n互质可得 $a * x_i \bmod n \ne a * x_j \bmod n$ （消去律）。

  -
    - $a * x_{\phi(i)}$ is different with $a * x_{\phi(j)}$ for any pairs of i, j, therefore if we use all $a * x_{\phi(i)}$, we will fill the set Z_n

  (2) $a^{\varphi(n)} * x_1 * x_2 * ... * x_{\varphi(n)} \bmod n$

  $\equiv (a * x_1) * (a * x_2) * ... * (a * x_{\varphi(n)}) \bmod n$

  $\equiv (a * x_1 \bmod n) * (a * x_2 \bmod n) * ... * (a * x_{\varphi(n)} \bmod n) \bmod n$

  $\equiv x_1 * x_2 * ... * x_{\varphi(n)} \bmod n$

  对比等式的左右两端，因为 x_i $(1 \le i \le \varphi(n))$ 与 n 互质，所以 $a^{\varphi(n)} \equiv 1 \bmod n$ （消去律）。

  注：

  消去律：如果 gcd(c,p) = 1 ，则 $ac \equiv bc \bmod p \Rightarrow a \equiv b \bmod p$ 。

  - Property: for example calculate $20^{10203}\%10403$
    - $10403 = 101(prime) * 103(prime)$
    - 20 is relative prime with 10403
    - $(101-1) * (103-1) = 10200$
    - $20^{10203} = 20^{10200} * 20^3$
    - $20^{10203}\%10403 = 20^3\%10403 = 8000$
    - $(a^{k\phi(pq)+b})\%(pq) = a^b\%(pq)$
- RSA public-key encryption
  - M: message, C: encripted message, (n,e) public key, (n,d) private key
  - $Gcd(e, \phi(n)) = 1$
  - Principle:
    - $M^{k\phi(n)+1}\%n = M$, where $n = pq$ (Euler's theorem)
    - $ed = k\phi(n) + 1$ <=> $ed\%\phi(n) = 1$ <=> d is inverse of e in $\phi(n)$
    - $ed + \phi(n) * k = gcd(e, \phi(n)) = 1$
  - Encrypt: $C = M^e \ mod \ n$
  - Decrypt: $M = C^d \ mod \ n = M^{ed} \ mod \ n$

- Setup:
  - Find two prime that is large enough
  -

- String Parsing

```
char input[80] = "Hi, today is another training of ACM.";
char *p=input, *cur = input;
while(*p){
    if(*p==','||*p==' ') {
        *p=0;
      if(*cur) printf("%s",cur);
        cur = ++p;
    }
    else ++p;
}
printf("%s",cur);
```

```
char** split(char* buffer, char* spliter,int& len){
    char *ptr = strtok(buffer,spliter);
    char **ans = new char*[100];
    int cnt = 0;
    while(ptr){
        ans[cnt++] = new char[100];
        ans[cnt-1] = ptr;
        ptr = strtok(NULL,spliter);
    }
    len = cnt;
    return ans;
}
```

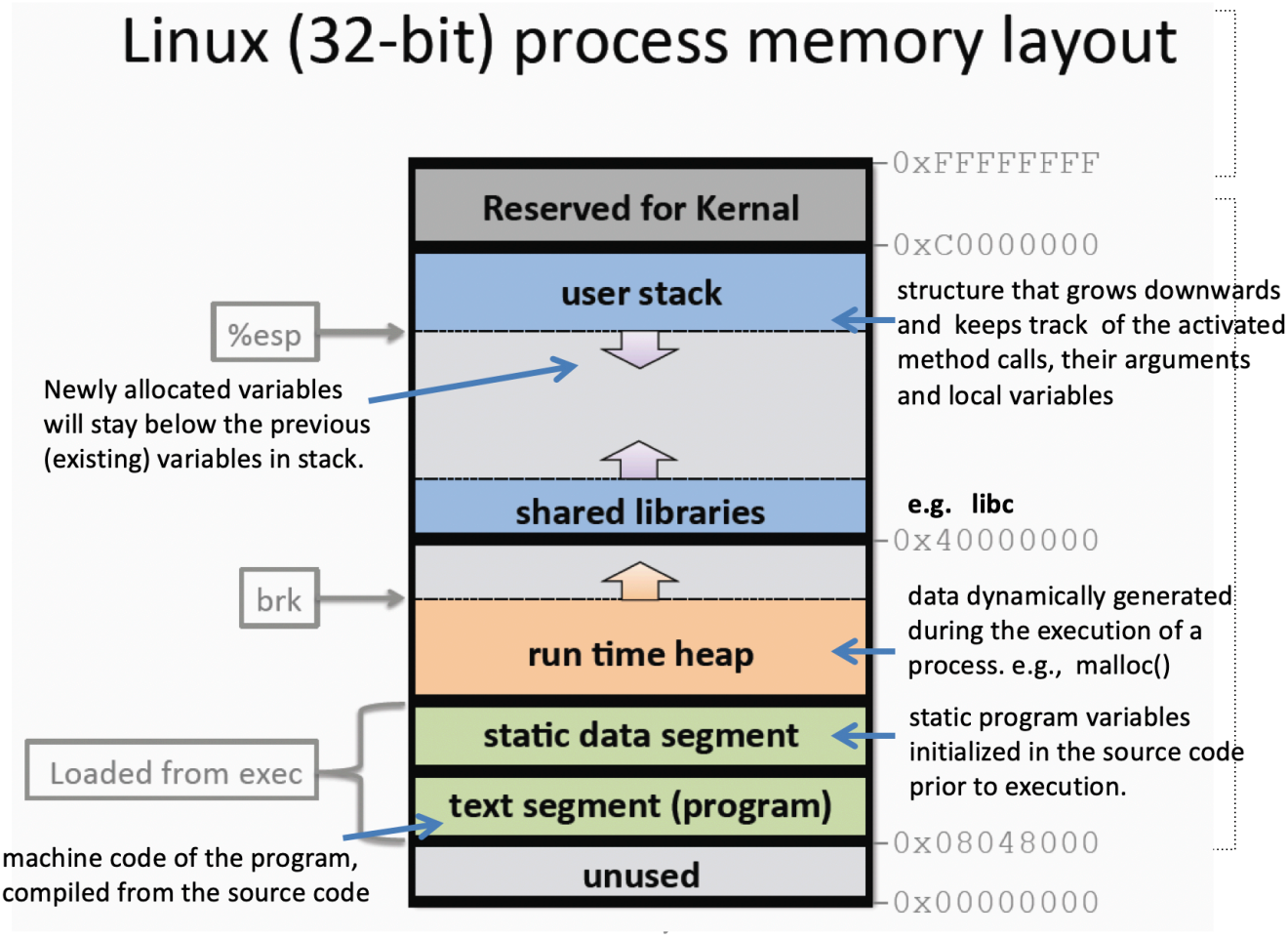
- Stack and Infix, prefix and postfix notation
  - Operator Priority

(	)	+	-	*	/
1	1	2	2	3	3

  - translate infix to postfix
    - Scan from left to right, token by token
    - If current token is an operand -> display token(or push into the number stack)

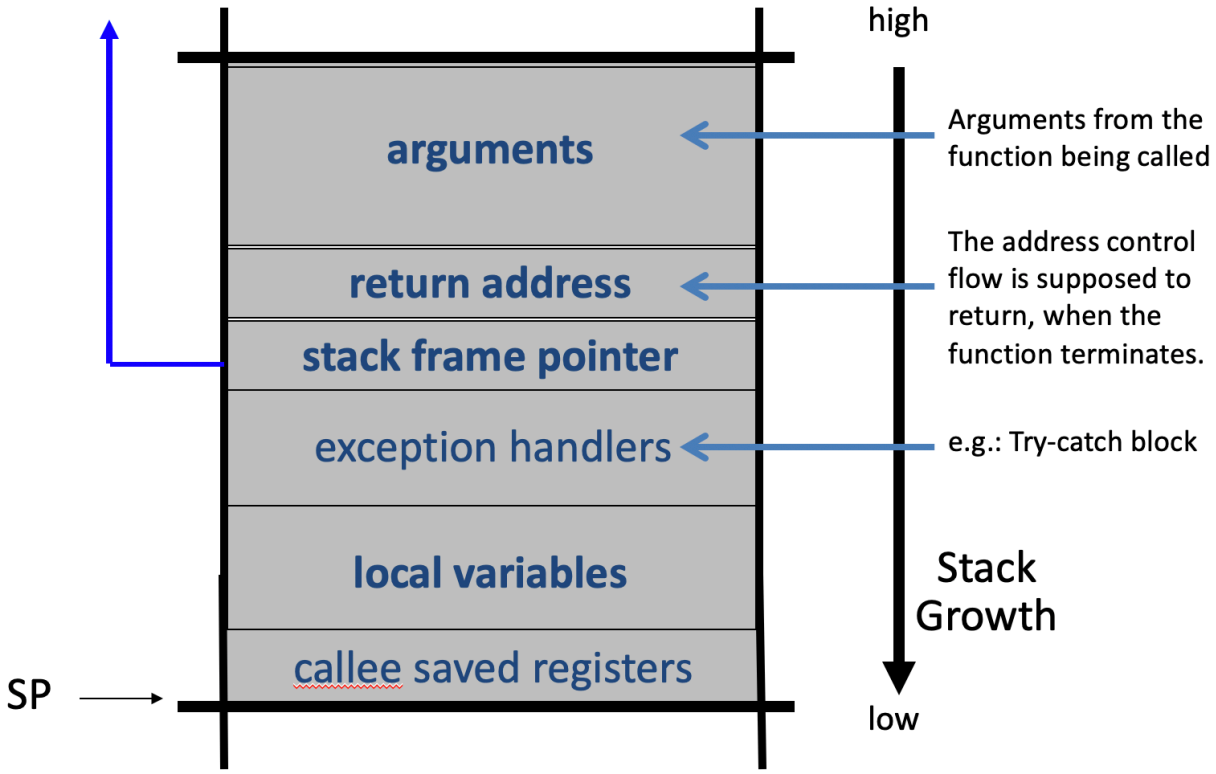
- If current token is an operator
  - case 1: stack is empty / new operator has higher priority -> push the new operator
  - case 2: the priority is same or lower -> pop old operator and display, and check current token again (
- If current token is '(' -> Push
- If current token is ')' -> Pop until '(' and display all operators
- When expression finishes, pop and display remaining operators

- Recursion



Stack Frame

Every time a function call is executed, a new frame is pushed onto the stack.



- nCr: given n characters - a,b,c,d,... => want to choose r characters and display inorder
  - sub problem: i_{th} character should/shouldn't choosed
  - Base case: the last character should/shouldn't choosed

```
combination(set, subset, r) {

    if (r == 0) {// the subset contains enough elements

        print elements in subset; return;
    }

    while (set is not empty) {
        move the first element from set to subset;
        combination(set, subset, r-1);
        remove the last element in subset;
    }
}
```

```
void comb(int r,int cur_pos,char supset[],vector<char>& subset,int& cnt){
    if(r==0){
        subset.push_back(0);
        printf("%s\n",subset.data());
        subset.erase(subset.end()-1);
        cnt++;
```

```
    }
    else{
        for(int i=cur_pos;supset[i];i++){
            subset.push_back(supset[i]);
            comb(r-1,i+1,supset,subset,cnt);
            subset.erase(subset.end()-1);
        }
    }
}
```

- nPr

❑Generate all permutations of set {1,2,…,n}.

- P{1,2,3} = 1-P{2,3} ∪ 2-P{1,3} ∪ 3-P{2,1}

○ {1,2,...,n} perm = ∪ (i, { {1,2,...,n} – {i} } perm)

- Keep set[i] as the first number in a permutations, then forms n sub-problems

❑*f*(set,1,n)

❑swap set[i] set[1], 1<=i <=n

❑*f*(set,2,n)

❑ n=3
- 1 2 3
- 1 3 2
- ○ 2 1 3
- ○ 2 3 1
- ✓ 3 2 1
- ✓ 3 1 2

- BFS

```
#define MAX 1000
bool map[MAX+1][MAX+1];
bool rec[MAX+1];
struct Node{
    int id;
    int depth;
    Node(int i,int l):id(i),depth(l){}
};
// important: record the node before dequeue
int bfs(int src,int tar){
    memset(rec,0,sizeof(rec));
    queue<Node*> path;
    path.push(new Node(src,0));rec[src]=1;
    while(true){
        int id = path.front()->id, depth = path.front()->depth; path.pop();
```

```
        if(id==tar) break;            //break while find target
        for(int j=1;j<=20;j++){
            if(map[j][id]&&!rec[j]){
                rec[j] = 1;
                path.push(new Node(j,depth+1));
            }
        }
    }
    return le;
}
```

- DFS

```
struct Node{
    int val;
    vector<int> next;
};
int cnt = 0;
Node map[10001];
bool rec[10001];
void dfs(int s){
    if(rec[s])return;
    rec[s] = 1;cnt++;
    for(int i=0;i<map[s].next.size();i++){
        int tar = (map[s].next)[i];
        if(!rec[tar]) dfs(tar);
    }
}
```

- Topological sort
  - DFS method
    - Use DFS to calculate the finishing time of each vertex
    - As each vertex is finished, insert it onto the front of a list
    - return the list of vertices
  - Second method
    - Remove all node with in-degree 0
    - Decrease the corresponding nodes' degree
    - repeate until all nodes have been removed
- Dijkstra algorithm
  - without heap

```
int map[MAX][MAX];
int dist[MAX];
bool rec[MAX];
int n; // number of nodes
void dijkstra(int src){
```

```
    memset(dist,INF,sizeof(dist));
    memset(rec,0,sizeof(rec));
    dist[src] = 0;
    while(true){
        int nd = -1;
        for(int i=0;i<n;i++){
            if(!rec[i]&&(nd==-1||dist[nd]>dist[i]))nd = i;
        }
        if(nd==-1)break;
        rec[nd] = true;
        for(int i=0;i<n;i++){
            dist[i]
                = dist[nd]+map[nd][i]<dist[i]?dist[nd]+map[nd][i]:dist[i];
        }
    }
}
```

- With heap -- stored in matrix

```
#define MAX 201;
int map[MAX][MAX];
int dist[MAX];
int n;
struct Node{
    int id,dist;
    bool operator>(const Node& b) const{
        return dist<b.dist;
    }
    bool operator<(const Node& b) const{
        return  !operator>(b);
    }
};

void dijkstra(int src, int tar){
    priority_queue<Node> q;
    fill(dist,dist+n,INF);
    dis[src] = 0;
    q.push((Node){src,0});
    while(!q.empty()){
        Node pre = q.top();q.pop();
        if(pre.id==tar)break;     // important to enhancement
        for(int i=1;i<=n;i++){
            if(i==pre.id) continue;
            if(map[id][i]+pre.dist < dis[i]){
                dist[i] = map[id][i]+pre.dist;
                q.push((Node){i,dist[i]});
            }
```

```
        }
    }
}
```

- With heap -- stored as edges

```
struct edge{int tar,cost;}; // store as edge information
typedef pair<int, int> P; // stored into the heap, first:cost, second:id
int V;
vector<edge> G[MAX];
int dist[MAX];
void dijkstra(int src){
    priority_queue<P,vector<P>,greater<P> >q;
    fill(dist,dist+V,INF);
    dist[src] = 0;
    q.push(P(0,src));
    while(!q.empty()){
        P cur = q.top();q.pop();
        int now = cur.second, cost = cur.first;
        if(dist[now]<cost) continue;    // already updated
        for(int i=0;i<G[now].size();i++){
            int tar = G[now][i].tar;
            if(dist[tar]>cost+G[now][i].cost){
                dist[tar] = cost+G[now][i].cost;
                q.push(P(dist[tar],tar));
            }
        }
    }
}
```

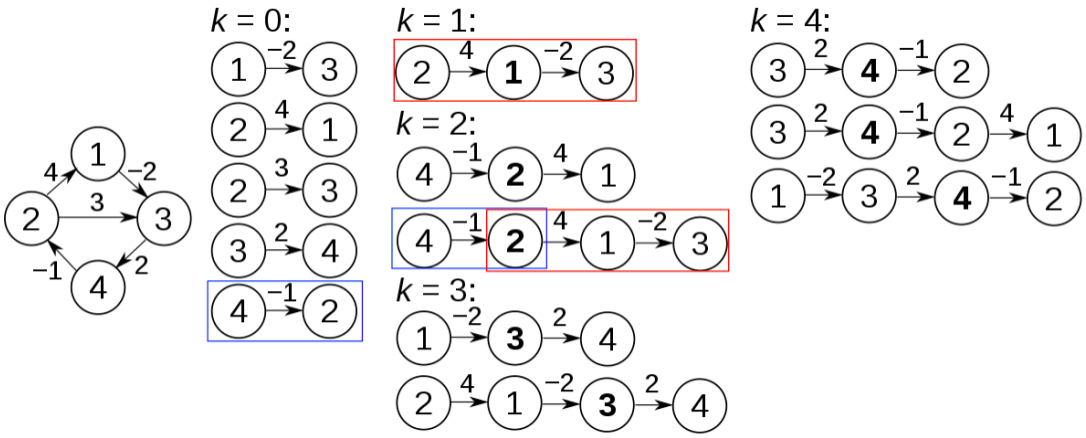
- Using set

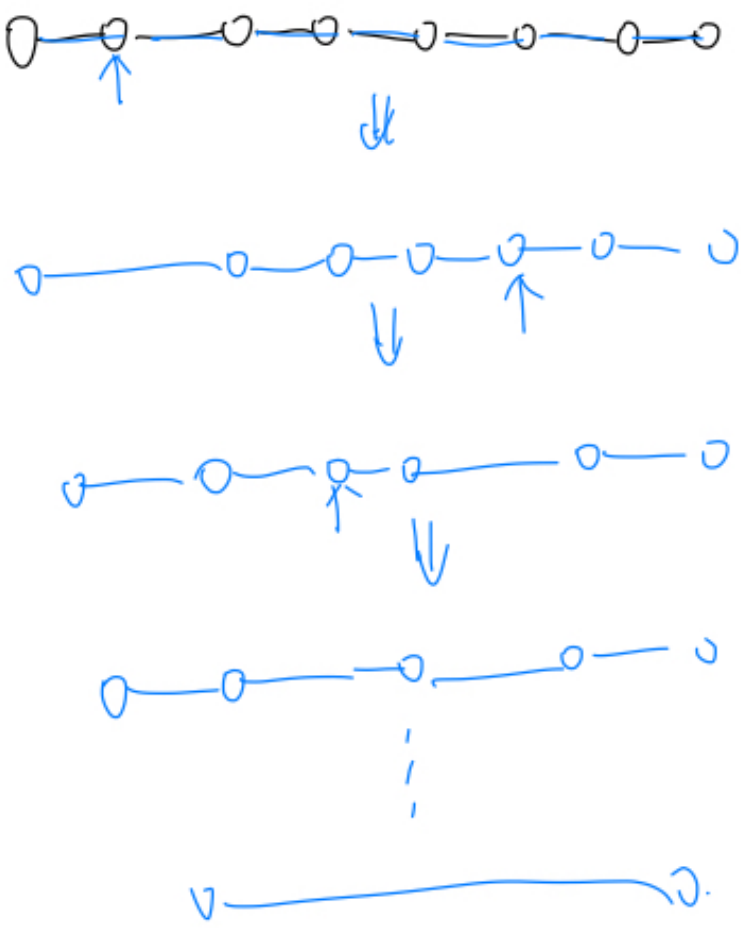
```
struct edge{
 int v,w,nxt;
}E[maxm];
int head[maxn];
int total_edge;
int dist[maxn];
void dij(int src){
   set< pair<int, int> > setds; // <distance,node_id>
   memset(dist,INF,sizeof(dist));
   setds.insert(make_pair(0, src));
   dist[src] = 0;
   while (!setds.empty()){
     pair<int, int> tmp = *(setds.begin());
     setds.erase(setds.begin());
     int u = tmp.second;
     for (int e = head[e]; i != -1; e = E[e].nxt){
```

```
        int v = E[e].v;
        int w = E[e].w;
        if (dist[v] > dist[u] + w){
          if (dist[v] != INF)
          setds.erase(setds.find(make_pair(dist[v], v)));
          dist[v] = dist[u] + w;
          setds.insert(make_pair(dist[v], v));
        }
      }
    }
}
```

- Floyd-Warshall algorithm(Any pairs of shortest path)
  - proof

The algorithm above is executed on the graph on the left below:





- #define black shortest_path; #define blue determined_shortest_path(stored in the matrix);
- Prove: Any arbitrary **black** path(i,j) can be changed to **blue** path, after the loop finished
- Proof:
  - Any segment consists of neighbor nodes on the shortest path is a blue path, because they are on a **black** path
  - As outer for loop go through all nodes, the n-2 nodes between i and j will be covered in some order
  - Each iteration, there will be a **blue** path generated by 2 consecutive blue path
    - Suppose i1, i2, i3 is a subpath between i and j
    - path(i1,i2) and path(i2,i3) are blue path
    - we know that path(i1,i3) is a black path, thus it will be changed to blue

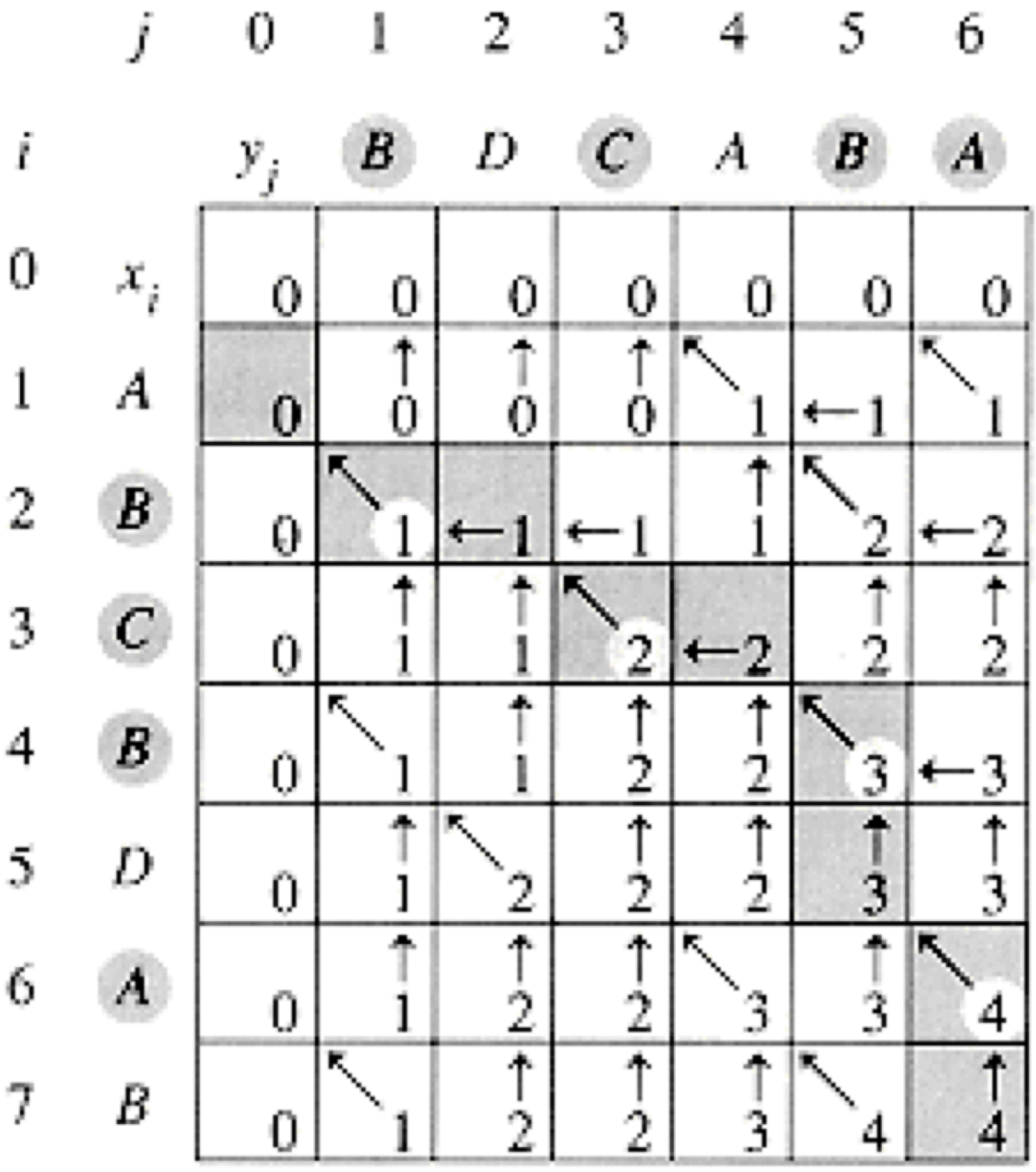
```
int dist[MAXV][MAXV];
int V;

void floyd() {
    for(int k=0;k<V;k++){
        for(int i=0;i<V;i++){
            for(int j=0;j<V;j++){
                dist[i][j]
                    = dist[i][j]>dist[i][k]+dist[k][j];
            }
        }
    }
}
```

- recover the path

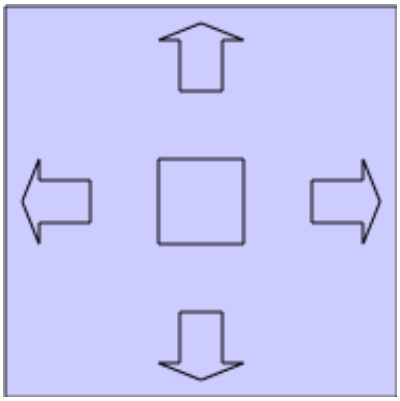
```
int path[MAX_V];
// for dijkstra
memset(path,-1,sizeof(path));
...;
    if(dist[tar]>cost+map[now][tar]){
        dist[tar] = cost+map[now][tar];
        path[tar] = now;
    }
...;
vector<int> get_path(int t){
    vector<int> p;
    for(;t!=-1;t=path[t])p.push_back(t);
    reverse(p.begin(),p.end());
    return p;
}
// for floyd
int path[MAX_V][MAX_V];
for(int i=1;i<=n;i++){
  for(int j=1;j<=n;j++){
    if(map[i][j]!=-1) path[i][j] = i;
        else path[i][j] = 0;
    }
}
for(int k=1;k<=n;k++){
  for(int i=1;i<=n;i++){
    for(int j=1;j<=n;j++){
          if(map[i][j]>map[i][k]+map[k][j]+cost[k]){
              map[i][j] = map[i][k]+map[k][j]+cost[k];
                path[i][j] = path[k][j];
            }
        }
    }
```

```
}
void print(int i,int j){
    if(path[i][j]==i) cout<<i;
    else{
        print(i,path[i][j]);
        cout<<"-->"<<path[i][j];
    }
}
```

- Dp: longest common substring: how to find the substring exactly?(string model)



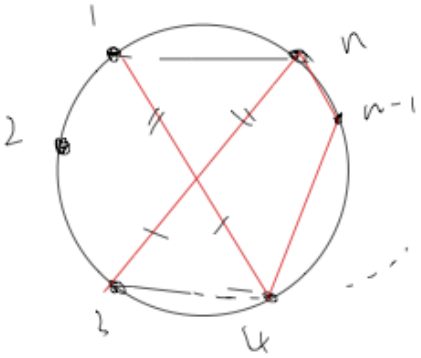
    - Use a matrix path[m][n]
- 背包九讲
- Examples of dynamic programming
    - Dance Dance Revolution: Linear model(consider the number of the)



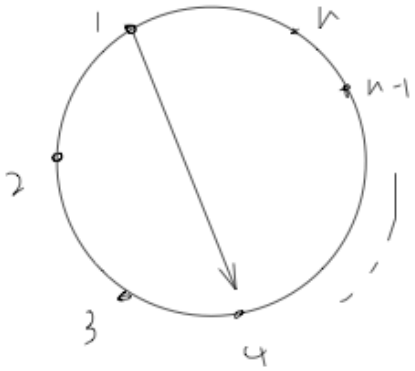
        - Originally, at the middle, input is a set of up,down,left and right.
        - Find the way that correctly follow a melody and use least energy
        - Energy calculation:
            - adjacent move(left->up):3
            - opposite(left->right):4
            - center to any: 2
            - Tap the same position: 1
        - state transaction function

$$dp[i][p_1][p_2] = min\{dp[i+1][x][p_2] + cost(x, p_1), \\ p[i+1][p_1][x] + cost(x, p_2)\} \tag{1}$$

    - Interval Model: Jumping frog
        - There are n stones (1<n<1000) in the pond. All the stones are on the same circle
        - A frog wants to jump to every stone exactly once
        - Try to design a route so that the total distance covered by the frog is minimized



Cross is impossible to be the shortest



Must jump to adjacent, otherwise cut the circle into 2 parts

- use the start and end points of interval as the states, each time can choose left or right
- state transaction function

$$dp[1][n] = min(dp[1][n-1] + dist[1][n] \\ dp[2][n] + dist[1][2]) \tag{2}$$

- Tree Model: (1) Find Maximum comfortable group on tree
  - input: tree
  - output: a comfortable group of maximum size
  - comfortable group: A set of nodes with no edge between any pair
  - A node can be either selected or not selected => create a dimension called selected, another dimension is node id
  - for binary tree:

```
dp[0][0] = max(dp[1][0],dp[1][1])+
       max(dp[2][0],dp[2][1]);
dp[0][1] = dp[1][0]+dp[2][0];
```

  - for a general tree:

```
struct Node{
    int id;
    vector<Node*> childs;
    ...
};

for(int i=0;i<nodes[x].childs.size();i++){
    Node* nd = nodes[x].childs[i];
  dp[x][0]+=max(dp[nd->id][0],dp[nd->id][1]);
    dp[x][1]+=dp[nd->id][0];
}

```

- Tree Mode: (2) Minimize longest path in a tree
  - A program can be branched to a tree by flow control statement
  - there is a worst case for the program, suppose each block takes some times, we need to put some blocks into the cache, such the **longest(worst) case** of the code can be **minimized**(by using cache)
  - First, if we consider each block takes same memory and time, and we have a cache with **finite** memory(says k blocks) that can reduce access time to 0
    - In this case, the best solution is for each k, push the node that has the maximum height
  - Second, we consider each cache can reduce the time from t to t', and each block has different memory
    - In this case, the best solution can be obtained by using dp
    - each node can be considered whether selected into the cache or not
    - we use an array `dp[node][avaliable_cache_mem]` to save the minimized max execution time of the program below(include) the node, use `mem[x],t[x],t_ptime[x]` to represent the memory, time and reduced time of the node x

```
#define child(x,i) x*2+i
dp[x][k] = min(
  max_for_any(k1+k2==k)(
        dp[child(x,1)][k1],
        dp[child(x,2)][k2]
    )+t_prime[x],
    max_for_any(k1+k2==k-mem[x]){
        dp[child(x,1)][k1],
        dp[child(x,2)][k2]
    }+t[x]
);
```

- Greedy, Enumeration and Guess
  - Enumeration: Simplest method, try all the combinations
  - greedy: every time pick the current"best" choice

- Coin Flipping:
  - there are N(<10000) rows of coins each of them consists of 9 coins
  - Some coins are heads up and some are tails up
  - we can flip a whole row or a whole column every time
  - Enumerate all the column flipping conditions and greedy to each row
- Discrete Function:

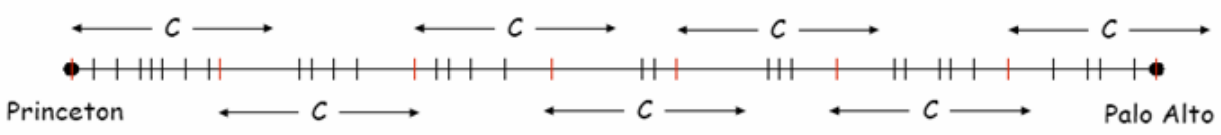
- Discrete Function
  - A discrete function is defined on {1,2,3,…,N}
  - Function values are between 2^{-32} and 2^{32}
  - Find two points on the function curve such that
    - Any function point between these two points are below the line connecting these two points
    - The slope of the line connecting these two points are as large as possible
  - $O(n^3)$ → $O(n^2)$
  - $O(n^2)$ → $O(n)$

  - n^3: enumerate all possible pairs of points and compare n-2 other points with it
  - n^2 : enumerate all points and for each points calculate slope between it to others, choose the line with highest slope(impossible to exist some points above the largest slope one)
  - n : enumerate all adjacent points(because for a biggest slope, if the start point and the end point are not adjacent and there are some points under the line, you always can use points under the line as the starting point to get a larger value)
- Selecting Breakpoints

Selecting breakpoints.

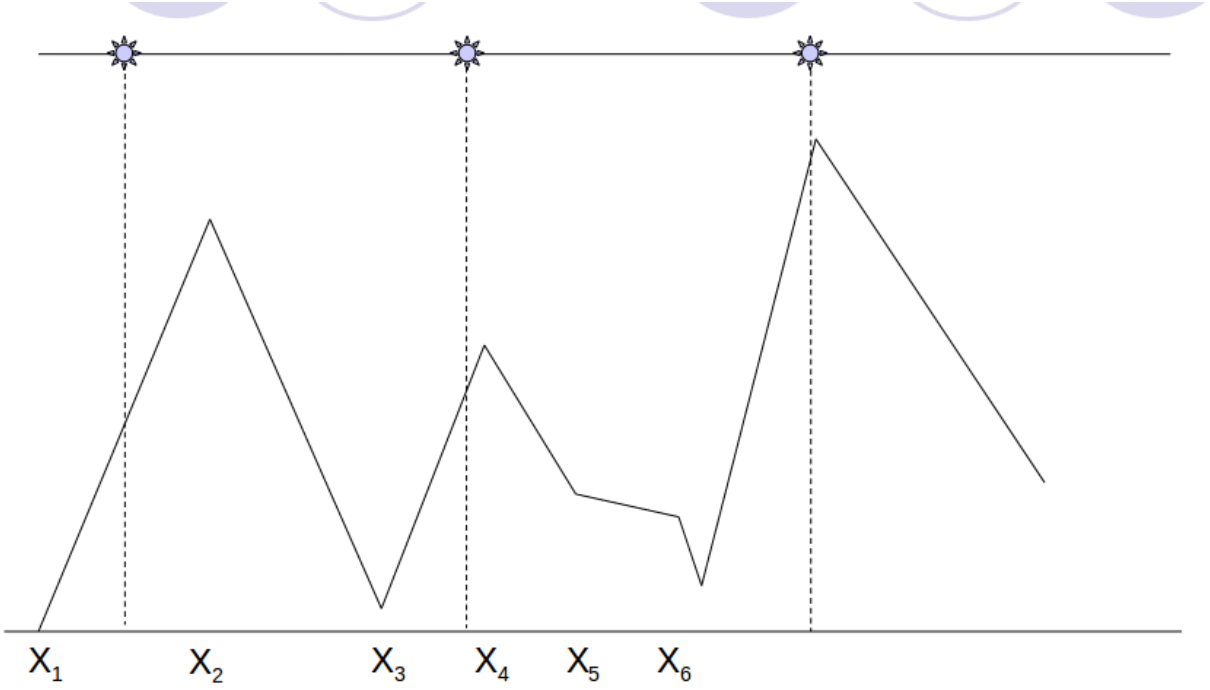
- Road trip from Princeton to Palo Alto along fixed route.
- Refueling stations at certain points along the way.
- Fuel capacity = C.
- Goal: makes as few refueling stops as possible.

Greedy algorithm. Go as far as you can before refueling.

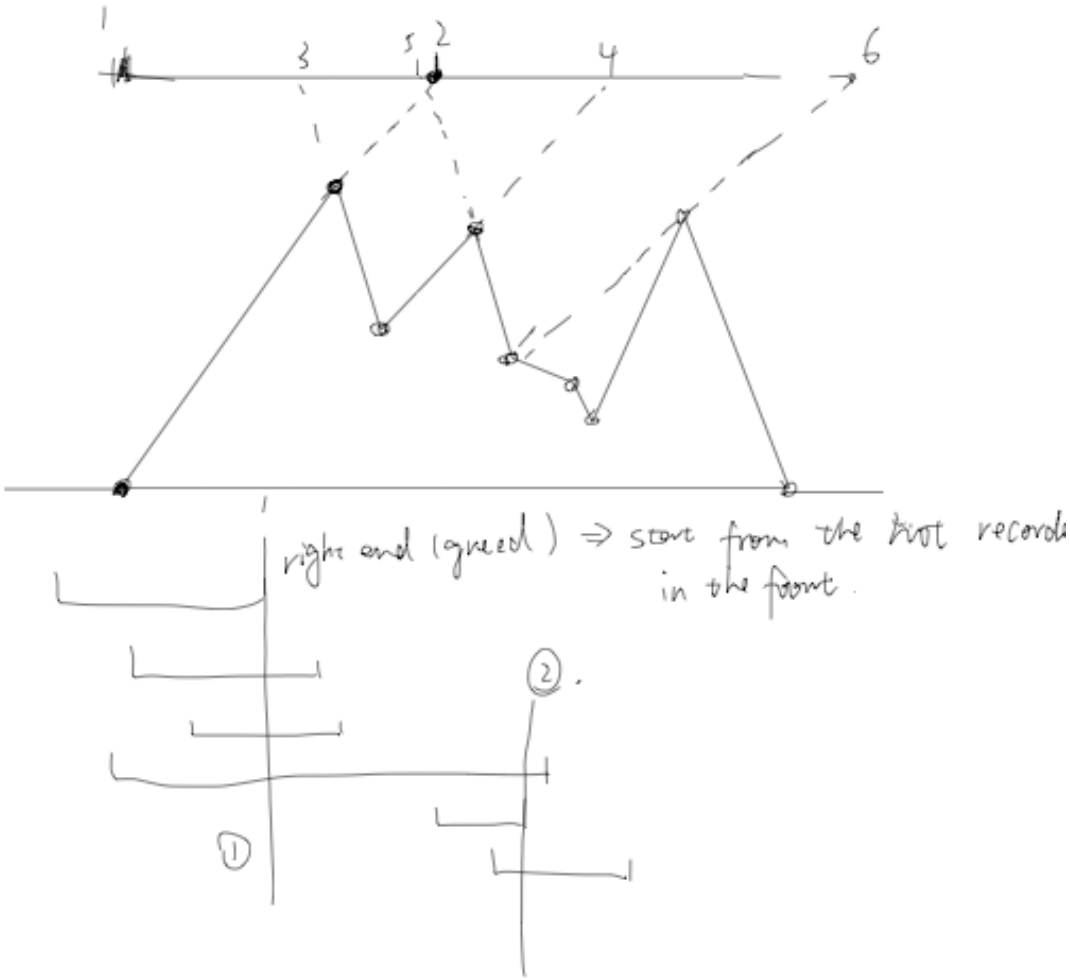


1	2	3	4	5	6	7

- Gone Fishing
  - There are n fishing lakes on a line labeled 1, 2, 3, ..., n from left to right
  - H hours' fishing time in total, and **you must be come back home(i.e. you have to repeat some roads**
  - Start from lake 1
  - 5*T_i minutes to reach lake i+1 from lake i
  - At the same lake i
    - The first 5 minutes can produce F_i fishes
    - Every 5 minutes when you are fishing, there will be a decrease D_i to the profit F_i
  - How to get the maximum number of fishes
- Enlightened Landscape: Make sure all the landscape can be enlightened:



  - All turned points are enlightened is enough for this problem
  - look from down to up and determine an interval
  - scan and greedy, to find minimum vertical line that can cut the interval

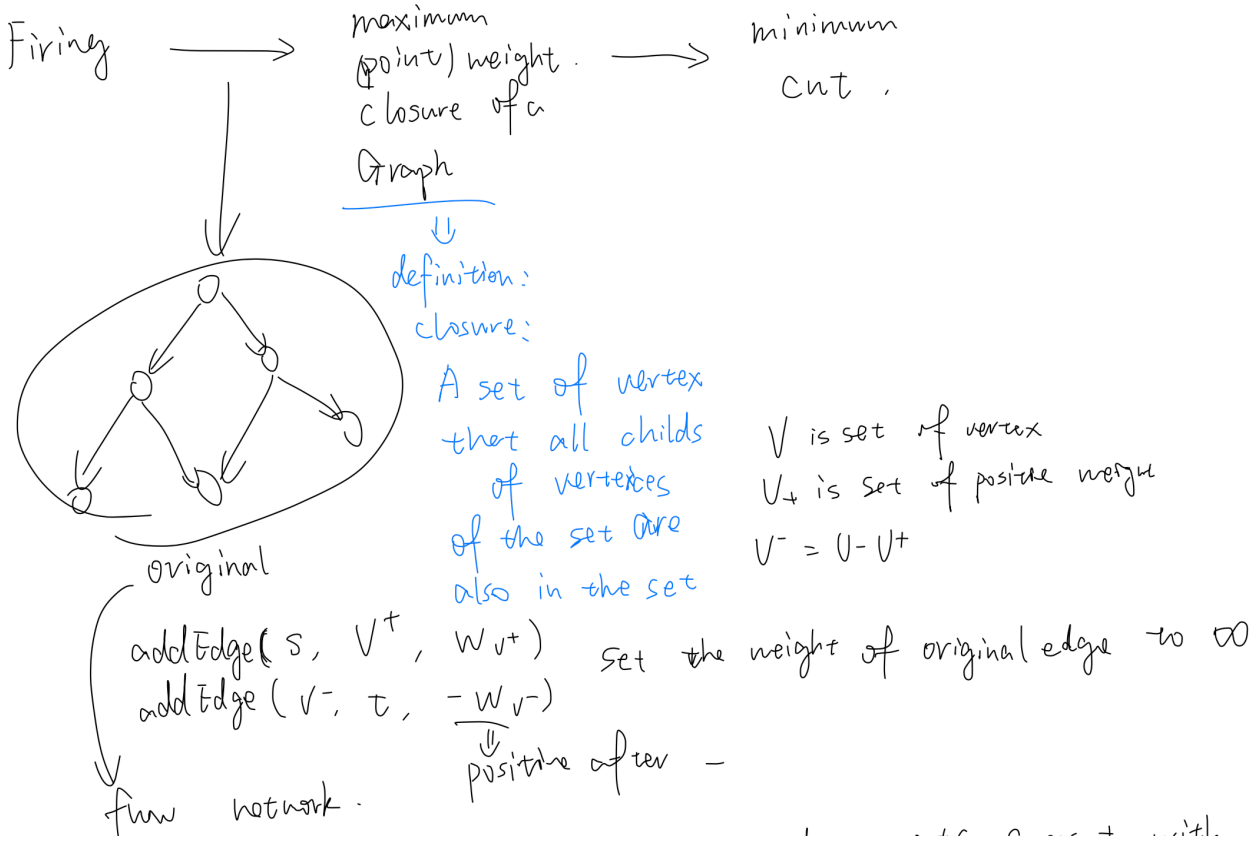


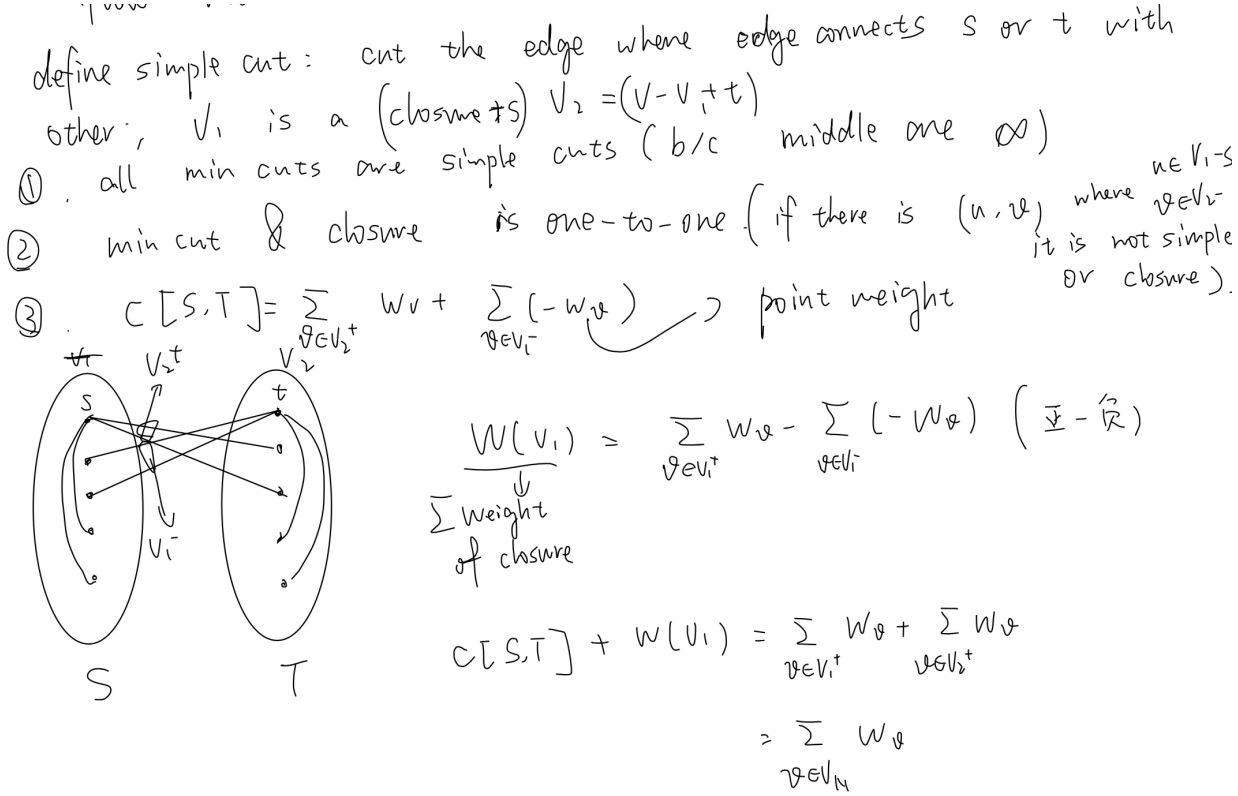
- Ford-Fulkerson algorithm(**max flow==min cut**)

```
// (target, capacity, reversed edge(store the index of the entry where it will be stored in the target node))
struct edge{int tar,cap,rev;};

vector<edge> G[MAX_V];
bool rec[MAX_V];
void add_edge(int src, int tar, int cap){
    int pos_index = G[src].size(), neg_index = G[tar].size();
    G[src].push_back((edge){tar,cap,neg_index});
    G[tar].push_back((edge){src,0,pos_index});
}
//  (start point, end point, minimum redundent flow of reached path)
int dfs(int nd, int tar, int f){
    if(nd==tar)return f;
    rec[nd] = true;
    for(int i=0;i<G[nd].size();i++){
        edge& e = G[nd][i];
        if(!rec[e.tar]&&e.cap>0){
          int d = dfs(e.tar,tar,min(f,e.cap));
            if(d>0){
```

```
                e.cap-=d;
                G[e.tar][e.rev].cap += d;
                return d;
            }
        }
    }
    return 0;
}

int max_flow(int src, int tar){
    int flow = 0;
    while(true){
        memset(rec,0,sizeof(rec));
        int f = dfs(src, tar, INF);
        if (f)flow+=f;
        else return flow;
    }
    return flow;
}
```

- Convertion of max flow, min cut and max (point) weight closure of a Graph





  - Conclusion: Max closure weight = sum of point weight - Max flow
- Point weight:

```cpp
#include <stdio.h>
#include <iostream>
#include <string.h>
#include <vector>
#include <limits.h>
#include <math.h>
using namespace std;
#define INF INT_MAX
#define MAX 210
#define in(i) i
#define out(i) n+i

struct Edge{int tar, cap, revp; };
// use 0 represent North, 2*n+1 represent South
// use i and n+i represent i_th in and out
vector<Edge> G[MAX];
bool rec[MAX];
void add(int src, int tar, int cap){
    G[src].push_back((Edge){tar,cap,G[tar].size()});
    G[tar].push_back((Edge){src,0,G[src].size()-1});
}

int dfs(int nd, int tar, int f){
    if(nd==tar) return f;
```

```
    rec[nd] = 1;
    for(int i=0;i<G[nd].size();i++){
        Edge& e = G[nd][i];
        if(!rec[e.tar]&&e.cap>0){
            int d = dfs(e.tar,tar,min(f,e.cap));
            if(d>0){
                e.cap-=d;
                G[e.tar][e.revp].cap+=d;
                return d;
            }
        }
    }
    return 0;
}

int max_flow(int src, int tar){
    int flow = 0;
    while(true){
        memset(rec,0,sizeof(rec));
        int f = dfs(src,tar,INF);
        if(f>0) flow+=f;
        else return flow;
    }
    return flow;
}


int pts[101][2];
int l, w, n, d, c=0,south,north;

inline double dist(int m, int n){
    int dx = pts[m][0]-pts[n][0], dy = pts[m][1]-pts[n][1];
    return sqrt(dx*dx+dy*dy);
}
int main(){
#ifdef DEBUG
    freopen("input.txt","r",stdin);
#endif
    while(~scanf("%d %d %d %d\n",&l, &w, &n, &d) && l && w && n && d)
    {
        printf("Case %d: ",++c);
        for(int i=0;i<MAX;i++) G[i].clear();
        south = 2*n+1; north = 0;
        for(int i=1; i<=n; i++){
            scanf("%d %d\n",pts[i],pts[i]+1);
            add(i,n+i,1);
            if(pts[i][1]<=d) add(out(i),south,INF);
            if(w-pts[i][1]<=d) add(north,in(i),INF);
        }
```

```
        for(int i=1;i<n;i++){
            for(int j=i+1;j<=n;j++){
                if(dist(i,j)<=2*d){
                    add(out(i),in(j),INF);
                    add(out(j),in(i),INF);
                }
            }
        }
        printf("%d\n",max_flow(north,south));
    }
    return 0;
}
```

- Minimum spanning tree
  - Minimum spanning tree contains at least one minimum edge, a minimum edge at least contained by at least one minimum spanning tree
  - Prim's algorithm: Similar principle with Dijkstra, but store different meaning of distance (each time, grow the tree by connecting the nearest vertex to the group, where the distance is defined by the minimum distance between vertex and another vertex in the group)
    - Start with a random vertex, push all of it's connected edges into the priority queue
    - Pop the shortest edge each time, and push all edges connected to the target vertex of the current edge
  - Kruskal's algorithm: Start with the shortest edge
    - simple proof: if an edge is small enough, and will construct a cycle, you must delete one edge from the cycle, and the edge causes cycle must be the one with maximum weight
    - Sort all the edges
    - Start with an empty disjoint set, each time add one edge according to the sorted order
    - An edge is valid if it will not construct a cycle
    - Do until there are n-1 edges, where n is the number of nodes
    - A template

```
#define MAX 101
int root[MAX];  // record the parent
int find(int a){
    if(root[a] < 0) return a;
    // path compression
    else return root[a] = find(root[a]);
}
void join(int a, int b){
    int r1 = find(a), r2 = find(b);
    // follow the principle of union by size
    if(root[r1] < root[r2]){
        root[r1] += root[r2];
        root[r2] = r1;
```

```
    }
    else{
        root[r2] += root[r1];
        root[r1] = r2;
    }
}
struct Edge{
    int src, tar, cost;
    bool operator<(const Edge& o) const{
        return cost < o.cost;
    }
} G[MAX*(MAX-1)/2];
int main(){
    ...;
    memset(root, -1 , sizeof(root));
    sort(G,G+num_of_edge);
    int connected = 0, ans = 0;
    for(int i=0;i<num_of_edge;i++){
        int x = find(G[i].src), y = find(G[i].tar);
        if(x!=y){
            join(x,y);
            connected++;
            ans += G[i].cost;
        }
        if(connected == n - 1) break;
    }
    return 0;
}

```

- Disjoint Sets & minimum spanning trees
  - Definition of **spanning tree**: Acyclic, connect all the vertices and is generated from the given graph(it is a subset of the original graph)
  - **minimum spanning tree**: The spanning tree formed from a weighted undirected graph in which the total sum of values on edge has the minimum total weight
  - Prim's algorithm: (dijkstra)
    - start with a set of known vertices with only one element, the initial element(vertex) can be any vertex
    - The set of known vertices is grown by one vertex for each stage
      - for each node in the set, iterate all edges connect vertex inside the set with unknown vertex
      - find the edge with minimum length
      - this step can be optimized by a priority queue or set hold the edges
    - Find an unknown vertex which is nearest to the set of known vertices
    - similar principle with dijkstra => greedy
    - implementation can use dijkstra

- Kruskal's algorithm: (hierachical clustering)
  - **sort** the edges in non descending order by weights(can use priority queue at first)
  - Each time pop an edge: if it does not form a cycle, this edge is accepted
  - if the edge forms a cycle, this edge is rejected => repeat until all vertices are connected
- **A graph can have more than one minimum spanning trees**
- Implementation details
  - Sort in C : `qsort()` functoin in `<stdlib.h>` can be used to sort integers, chars and stuctures

```
void qsort(Type* arr,int len,int sizeof(Type),CMP);
// where CMP is a function name
int CMP(const void* a, const void* b){
    if(a less than b) return -1;
    if(a greater than b) return 1;
    return 0;
}
#define MAX 1000
struct Edge{
    int src,tar,len;
} G[MAX];
const int CMP(const void* a,const void* b){
    Edge* A = (Edge*)a, B = (Edge*)b;
    return A->len - B->len;
}
```

  - use disjoint set to record the spanning tree(following is disjoint set with union by size and path compression)

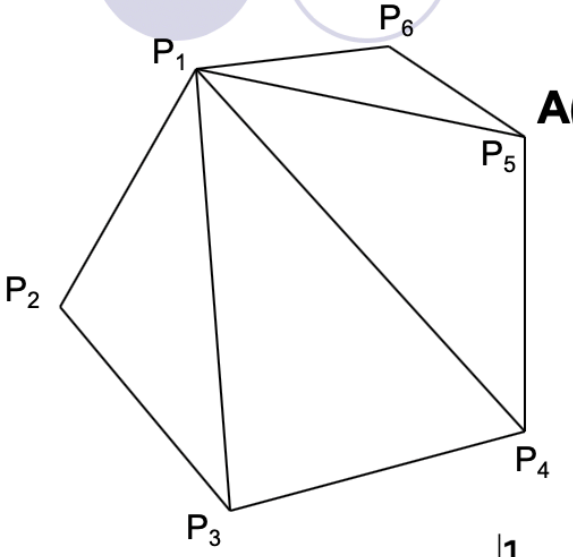
```
int root[MAX];
void Union(int e1, int e2){
    int root1 = find(e1), root2 = find(e2);
    // since root stores -(member count)
    if(root[root1]<root[root2]){
        root[root1] += root[root2];
        root[root2] = root1;
    } else {
        root[root2] += root[root1];
        root[root1] = root2;
    }
}
int find(int e){
    if(root[e]<0)return e;
    else retur root[e] = find(root[e]);
}
```

- Line segment intersection

```cpp
#define MAX 105
struct Point{
  int x, y;
} line[MAX][2];

int clock(Point& p1, Point& p2, Point& p3){
    int val = (p2.x - p1.x)*(p3.y - p2.y) - (p2.y - p1.y)*(p3.x - p2.x);
    if(!val) return val;
    return val < 0 ? -1 : 1;
}
bool online(Point& p1, Point& p2, Point& p3){
    return p3.x <= max(p1.x, p2.x) && p3.x >= min(p1.x, p2.x)
        && p3.y <= max(p1.y, p2.y) && p3.y >= min(p1.y, p2.y);
}
bool checkIntersection(int i, int j){
    int o1 = clock(line[i][0], line[i][1], line[j][0]);
    int o2 = clock(line[i][0], line[i][1], line[j][1]);
    int o3 = clock(line[j][0], line[j][1], line[i][0]);
    int o4 = clock(line[j][0], line[j][1], line[i][1]);
    if(o1 * o2 < 0 && o3 * o4 < 0) return 1;
    if (o1 == 0 && online(line[i][0], line[i][1], line[j][0])) return 1;
  if (o2 == 0 && online(line[i][0], line[i][1], line[j][1])) return 1;
  if (o3 == 0 && online(line[j][0], line[j][1], line[i][0])) return 1;
  if (o4 == 0 && online(line[j][0], line[j][1], line[i][1])) return 1;
  return 0;
}
```

- The Area of Polygon (convex)
  - For concave, the part over calculated will be decreased later => same equation

Triangulation of Convex Polygon



$$A(P_1,P_2,P_3) = \frac{1}{2} * \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix}$$

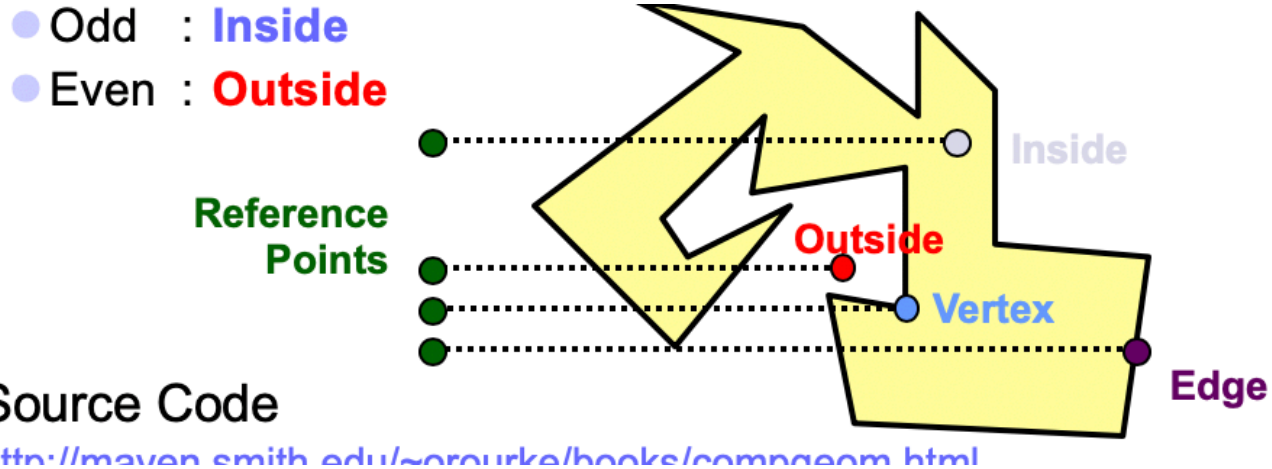
$$= \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} + \begin{vmatrix} x_2 & y_2 \\ x_3 & y_3 \end{vmatrix} + \begin{vmatrix} x_3 & y_3 \\ x_1 & y_1 \end{vmatrix}$$

$$A = \sum_{i=1}^{N-2} A(P_1,P_{i+1},P_{i+2}) = \frac{1}{2}\sum_{i=1}^{N-2} \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_{i+1} & y_{i+1} \\ 1 & x_{i+2} & y_{i+2} \end{vmatrix} = \frac{1}{2}\sum_{i=1}^{N} \begin{vmatrix} x_i & y_i \\ x_{i+1} & y_{i+1} \end{vmatrix}$$

Note: $P_{N+1} = P_1$

31

- Point Inside Polygon
  - Using a reference point (far enough that it is outside of the polygon): Form a **horizontal line** with testing point
  - Count the number of intersections for the horizontal line and all edges



Source Code

http://maven.smith.edu/~orourke/books/compgeom.html

The code

- Graham scan

```
struct Point {
  int x, y, dist;
} pts[MAX];

int t, n;
```

```cpp
int cp(const Point& a, const Point& b, const Point& c) {
  // if counter_clockwise, return larger than 1 => rescale to make sure not overflow
  int val = (b.x - a.x)*(c.y - b.y) - (b.y - a.y)*(c.x - b.x);
  return val?(val<0?-1:1):val;
}

bool point_cmp(const Point& a, const Point& b) {
  return a.y == b.y ? a.x < b.x : a.y < b.y;
}
bool slope_cmp(const Point& b, const Point& c) {
  int dir = cp(pts[0], b, c);
  return dir ? dir>0 : b.dist<c.dist ;
}
void Graham() {
  sort(pts, pts + n, point_cmp);
  for (int i = 1; i < n; i++)
    pts[i].dist = (pts[i].y - pts[0].y)*(pts[i].y - pts[0].y)
                + (pts[i].x - pts[0].x)*(pts[i].x - pts[0].x);
  sort(pts + 1, pts + n, slope_cmp);

  int top = 1;
  // use pts as a stack, because top <= i after each iteration
  for (int i = 2; i < n; i++) {
    while (top > 0 && cp(pts[top - 1], pts[top], pts[i]) <= 0) --top;
    pts[++top] = pts[i];
  }

  // delete the points in the middle of the segment
  pts[++top] = pts[0];
  int now = 1;
  for (int i = 2; i <= top; i++) {
    // move now to the next point on the same segment
    if (!cp(pts[now - 1], pts[now], pts[i])) pts[now] = pts[i];
    else pts[++now] = pts[i];
  }
  printf("%d\n", now + 1);
  for (int i = 0; i <= now; i++) {
    printf("%d %d\n", pts[i].x, pts[i].y);
  }
}
```

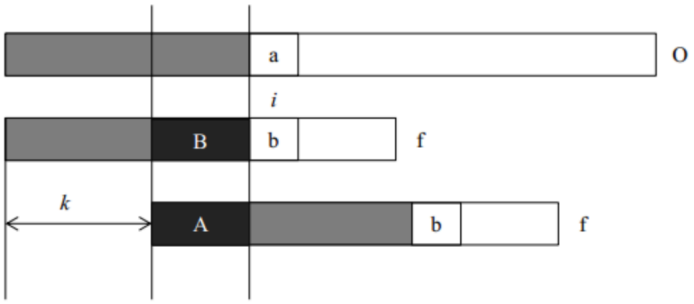
- KMP algorithm

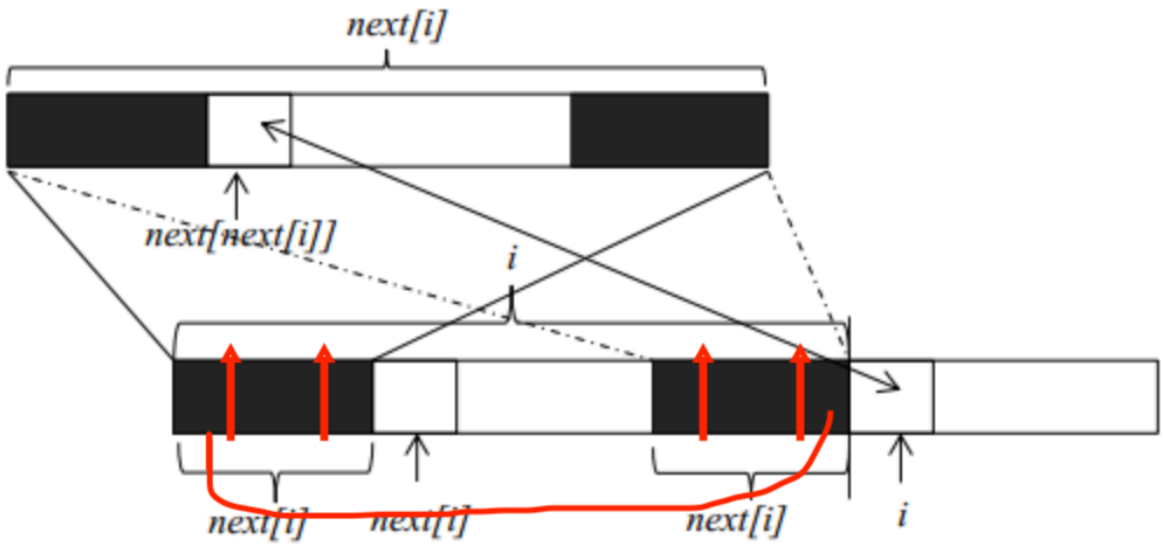
kmp算法思想

我们首先用一个图来描述kmp算法的思想。在字符串O中寻找f，当匹配到位置i时两个字符串不相等，这时我们需要将字符串f向前移动。常规方法是每次向前移动一位，但是它没有考虑前i-1位已经比较过这个事实，所以效率不高。事实上，如果我们提前计算某些信息，就有可能一次前移多位。假设我们根据已经获得的信息知道可以前移k位，我们分析移位前后的f有什么特点。我们可以得到如下的结论：

- A段字符串是f的一个前缀。
- B段字符串是f的一个后缀。
- A段字符串和B段字符串相等。

所以前移k位之后，可以继续比较位置i的前提是f的前i-1个位置满足：**长度为i-k-1的前缀A和后缀B相同**。只有这样，我们才可以前移k位后从新的位置继续比较。



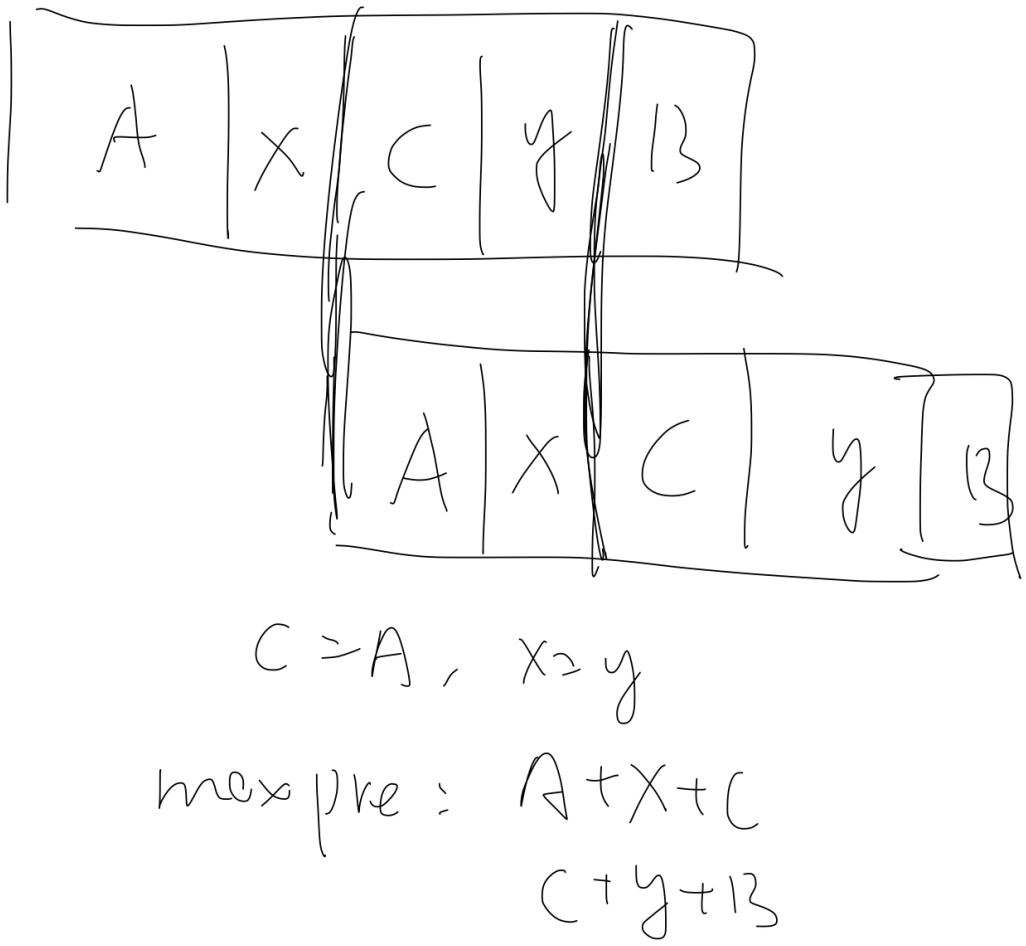


- next[i]: maximum common length when we consider the $0->(i-1)$ substring
- we calculate next array for the "moved" string
- Code

```
int nxt[100010];
char str[100001],tmp[100001];
void getNext(int n){
    int prenxt=0;
    nxt[0] = nxt[1] = 0;
    for(int i=1;i<n;i++){
        while(prenxt>0&&tmp[i]!=tmp[prenxt])
            prenxt = nxt[prenxt];
```

```
        if(tmp[i]==tmp[prenxt]) prenxt++;
        nxt[i+1] = prenxt;  // i_th iteration, judge the length of
i+1(str[0]->str[i])
    }
}
void KMP(int n){
    int pos = 0;
    for(int i=0;i<n;i++){
        // if not match, search for the previous smaller substring
        while(pos>0&&str[i]!=tmp[pos])pos = nxt[pos];
        if(str[i]==tmp[pos])pos++;
    }
    printf("%s%s\n",str,tmp+pos);
}
```

- Proof:
  - if there is a block before B and equals to A, it is skipped



- Suffix array
  - 后缀数组：后缀数组SA 是一个一维数组，它保存1..n 的某个排列SA[1]，SA[2]，……，SA[n]，并且保证Suffix(SA[i]) < Suffix(SA[i+1])，1≤i<n。也就是将S 的n 个后缀从小到大进行排序之后把排好序的后缀的开头位置顺次放入SA 中 (store the sorted suffix)
  - 名次数组：名次数组Rank[i]保存的是Suffix(i)在所有后缀中从小到大排列的“名次”。简单的说，后缀数组是“排第几的是谁？”，名次数组是“你排第几？(index to suffix array)”。容易看出，后缀数组和名次数组为互逆运算。如图1 所示。
  - height 数组：定义height[i]=suffix(sa[i-1])和suffix(sa[i])的最长公共前缀，也就是排名相邻的两个后缀的最长公共前缀

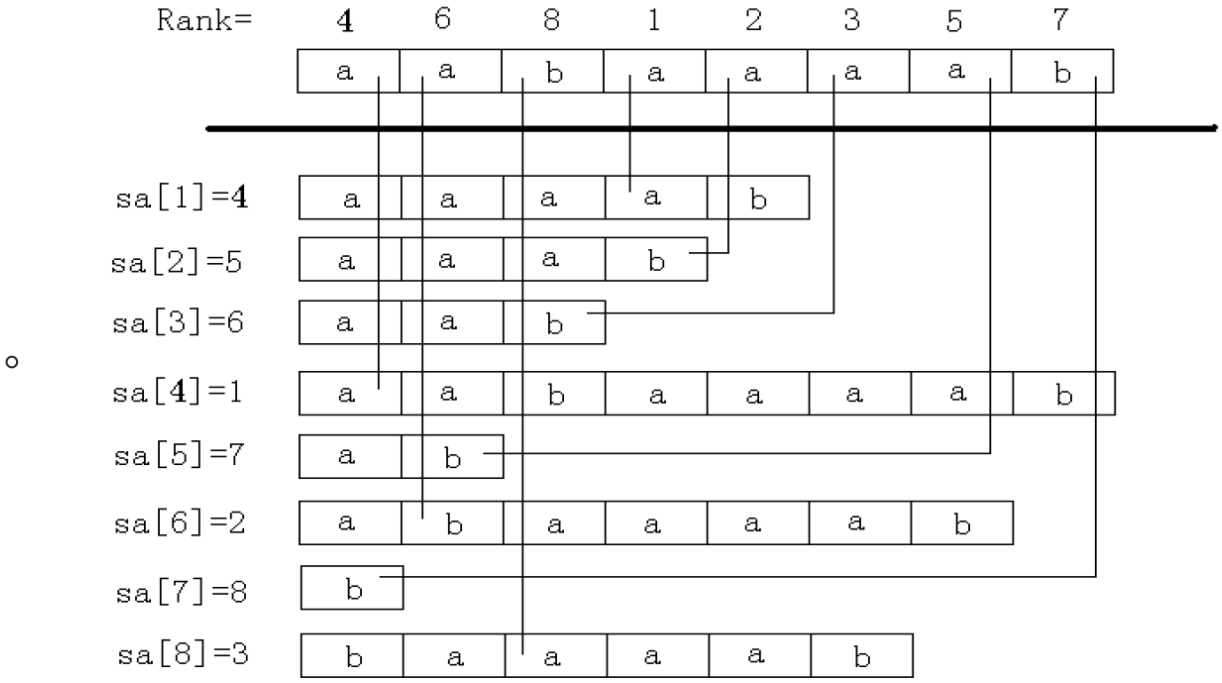


图1

- Example: Find the longest substring and the occurrence

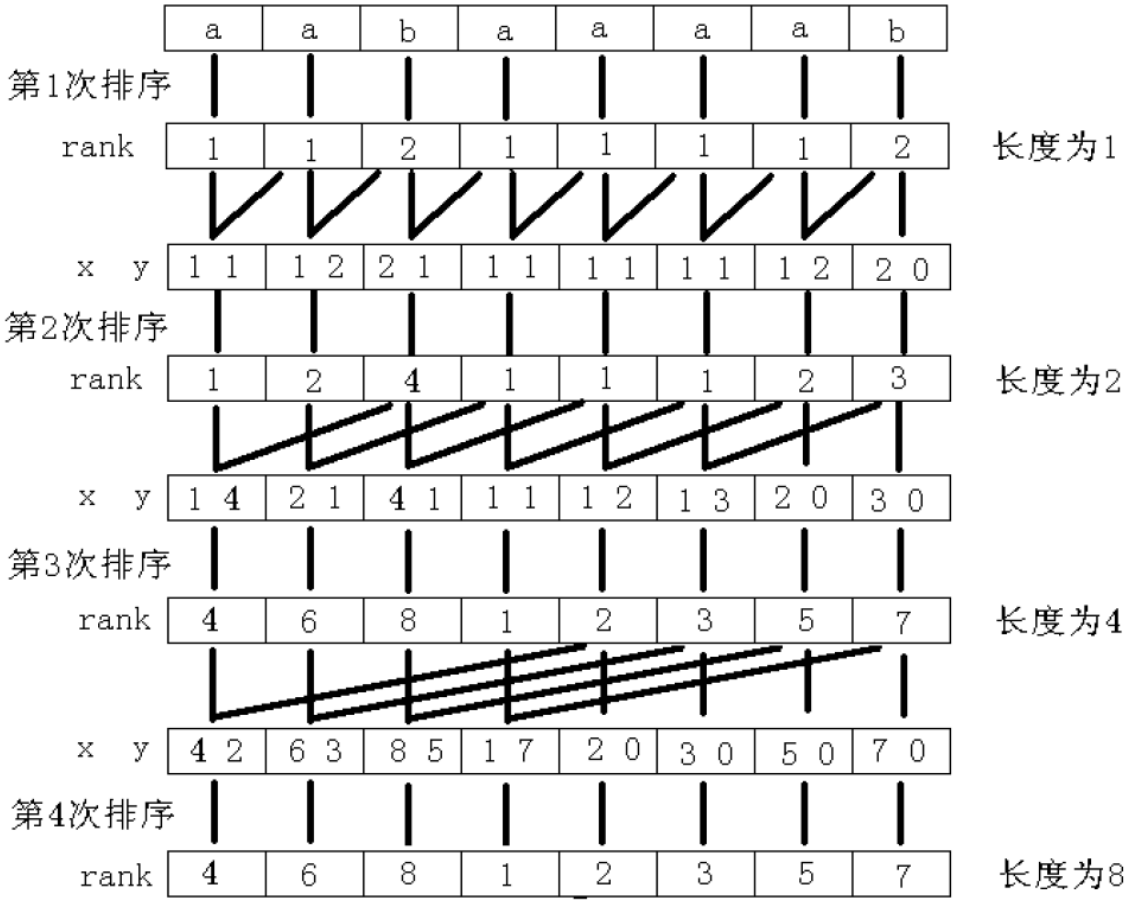


图2

```cpp
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <algorithm>

#define MAX 1005

int sa[MAX], height[MAX], n, bucket[MAX], Rank[MAX], map[MAX], t;
char str[MAX];
void sort(int* x, int* y, int m) {
  // x: key to sort
  // y: index of x as second key to sort(point to some other places of x)
  // w: the bucket
  // m: number of buckets
  for (int i = 0; i < m; i++) bucket[i] = 0;
  for (int i = 0; i < n; i++) bucket[x[y[i]]]++;
  for (int i = 1; i < m; i++) bucket[i] += bucket[i - 1];
  for (int i = n - 1; i >= 0; i--) sa[--bucket[x[y[i]]]] = y[i];
}

bool cmp(int *x, int a, int b, int l) {
  if (x[a] == x[b]) {
    if (a + l >= n || b + l >= n)
      return false;
    return x[a + l] == x[b + l];
  }
  return false;
}

void build() {
  // i: index
  // k: the length of sorted prefix
  // p: number of distinct key(compare value)
  int i, k, m = 128, p;
  int *x = Rank, *y = map;
  n = strlen(str);
  x[n] = 0;
  // first time: sort on itself
  for (int i = 0; i < n; i++) x[i] = str[i], y[i] = i;
  sort(x, y, m);
  for (k = 1, p = 1; p < n; k <<= 1, m = p) {
    // sort suffix that shorter than k on the first character (sort on itself)
    for (p = 0,i = n-k; i < n; i++) y[p++] = i;
    // sort remain suffix according to the first character
    // 从后往前
    for (i = 0; i < n; i++) if (sa[i] >= k) y[p++] = sa[i] - k;
    sort(x, y, m);
    int* temp = x;
    x = y;
```

```
    y = temp;
    // record rank, rank may duplicate
    // y is the previous rank
    for (i = 1, p = 1, x[sa[0]] = 0; i < n; i++)
      x[sa[i]] = cmp(y, sa[i - 1], sa[i], k) ? p - 1 : p++;
  }
}

void build_h()
{
  int i, j, k;
  for (i = 0; i < n; i++)
    Rank[sa[i]] = i;
  for (i = 0; i < n; i++)
  {
    if (Rank[i] == 0)
    {
      height[Rank[i]] = 0;
      continue;
    }
    if (i == 0 || height[Rank[i - 1]] <= 1)
      k = 0;
    else k = height[Rank[i - 1]] - 1;
    while (str[sa[Rank[i] - 1] + k] == str[sa[Rank[i]] + k]) k++;
    height[Rank[i]] = k;
  }
}

int main() {
#ifdef _DEBUG
  freopen("input.txt", "r" , stdin);
#endif
  scanf("%d\n", &t);
  while (t--) {
    gets(str);
    build();
    build_h();
    int ans = 0, count = 0;
    for (int i = 0; i < n; i++)
      ans = ans > height[i] ? ans : height[i];
    if (ans == 0)
      printf("No repetitions found!\n");
    else {
      for (int i = 0; i < n; i++)
      {
        if (ans == height[i])
        {
          for (int j = 0; j < ans; j++)
            printf("%c", str[sa[i] + j]);
```

```
            while (i < n && ans == height[i])
              count++, i++;
            break;
          }
        }
        printf(" %d\n", count + 1);
      }

    }
    return 0;
}
```

-