Ural SU *W*ine Team

Reference materials

0. Configure files.

```
//makefile
%exe: %cpp
   g++ -pg -g -Wall -Wextra -Wconversion -DDBG1 -o $@ $^

//vimrc
map <C-k> :w<cr>:make<cr>
imap <C-k> <esc>:w<cr>:make<cr>

set autoindent
set cindent
set tabstop=4
set shiftwidth=4
set makeprg=make\ %:r.exe\ -f\ /home/user/makefile
set nohlsearch

//autokill.sh
#!/bin/bash
f=0
while [ $f = 0 ]; do
       ./gen.exe
       ./code.exe
       ./brute.exe
       diff output.txt ans.txt
       f=$?
done

//exact time
#include <sys/time.h>
struct timeval s;
gettimeofday(&s, NULL);
int t = s.tv_sec * 1000000 + s.tv_usec;

//debug print
void dbg(const char * fmt, ...)
{
#ifdef DBG1
#if DBG2
  va_list args;
  va_start(args, fmt);
  vfprintf(stderr, fmt, args);
  va_end(args);
#endif
#endif
}
```

1. Математика.

★ 1.1. Интегралы.

$$\int \frac{dx}{a^2+x^2} = \frac{1}{a}\arctan\frac{x}{a}\ ,\ a \neq 0$$

$$\int \frac{dx}{\sqrt{a^2-x^2}} = \frac{1}{2a}\ln\left|\frac{a+x}{a-x}\right|\ ,\ a \neq 0$$

$$\int \frac{xdx}{a^2 \pm x^2} = \pm\frac{1}{2}\ln|a^2 \pm x^2|$$

$$\int \frac{dx}{\sqrt{a^2-x^2}} = \arcsin\frac{x}{a}\ ,\ a > 0$$

$$\int \frac{dx}{\sqrt{x^2 \pm a^2}} = \ln\left|x+\sqrt{x^2 \pm a^2}\right|\ ,\ a > 0$$

$$\int \frac{xdx}{\sqrt{a^2 \pm x^2}} = \pm\sqrt{a^2 \pm x^2}\ ,\ a > 0$$

$$\int \sqrt{a^2-x^2}\ dx = \frac{x}{2}\sqrt{a^2-x^2} + \frac{a^2}{2}\arcsin\frac{x}{a}\ ,\ a > 0$$

$$\int \sqrt{x^2 \pm a^2}\ dx = \frac{x}{2}\sqrt{x^2 \pm a^2} \pm \frac{a^2}{2}\ln\left|x+\sqrt{x^2 \pm a^2}\right|\ ,\ a > 0$$

$$\int \sqrt{\frac{a+x}{b+x}}\ dx = \sqrt{(a+x)(b+x)} + (a-b)\ln\left(\sqrt{a+x}+\sqrt{b+x}\right)$$

$$\int \sqrt{\frac{a-x}{b+x}}\ dx = \sqrt{(a-x)(b+x)} + (a+b)\arcsin\sqrt{\frac{x+b}{a+b}}$$

$$\int \sqrt{\frac{a+x}{b-x}}\ dx = -\sqrt{(a+x)(b-x)} - (a+b)\arcsin\sqrt{\frac{b-x}{a+b}}$$

★ 1.2. Некоторые формулы матана.

Длина плоской кривой $y = f(x)$:

$$l = \int_{x_1}^{x_2} \sqrt{1+f'(x)^2}\,dx = \int_{t_1}^{t_2} \sqrt{{x'_t}^2+{y'_t}^2}\,dt$$

Площадь поверхности вращения кривой $y = f(x)$ вокруг оси OX:

$$S = 2\pi\int_{x_1}^{x_2} f(x)\sqrt{1+f'(x)^2}\,dx$$

Объем поверхности вращения кривой $y = f(x)$ вокруг оси OX:

$$V = \pi\int_{x_1}^{x_2} f(x)^2\,dx$$

Формула Симпсона для приближенного интегрирования:

$$\int_a^b f(x)\,dx = \frac{h}{3}\left(\frac{f(x_0)+f(x_n)}{2} + \sum_{i=1}^{n-1} f(x_i) + 2\sum_{i=0}^{n-1} f\left(x_{i+\frac{1}{2}}\right)\right)$$

★ 1.3. Треугольник.

Радиус вписанной окружности: $r = \dfrac{S}{p}$

Радиус описанной окружности: $R = \dfrac{abc}{4S}$

Формула для медианы: $m_a = \dfrac{1}{2}\sqrt{2(b^2+c^2)-a^2}$

Формула для биссектрисы: $l_a = \dfrac{2\sqrt{bcp(p-a)}}{b+c}$

★ 1.4. Сфера.

Площадь сегмента сферы: $S = 2\pi Rh$

Объем сегмента сферы: $V = \pi h^2\left(R - \dfrac{h}{3}\right)$

Объем сектора сферы: $V = \dfrac{2}{3}\pi h^2 R$

★ 1.5. 3D-Преобразования.

► 1.5.1. Отражение относительно плоскости.

Матрица отражения точки, заданной в однородных координатах, относительно нормированной плоскости $Ax + By + Cz + D = 0$, то есть $A^2 + B^2 + C^2 = 1$:

$$Q = \begin{pmatrix} 1-2A & -2B & -2C & -2D \\ -2A & 1-2B & -2C & -2D \\ -2A & -2B & 1-2C & -2D \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

► 1.5.2. Вращение вокруг начала координат.

Матрица вращения вокруг единичного вектора $\mathbf{u} = (x, y, z)$ на угол θ:

$$Q_{\mathbf{u}}(\theta) = \begin{pmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{pmatrix} \sin\theta + (I - \mathbf{u}\mathbf{u}^T)\cos\theta + \mathbf{u}\mathbf{u}^T =$$

$$= \begin{pmatrix} x^2(1-\cos\theta)+\cos\theta & xy(1-\cos\theta)-z\sin\theta & xz(1-\cos\theta)+y\sin\theta \\ xy(1-\cos\theta)+z\sin\theta & y^2(1-\cos\theta)+\cos\theta & yz(1-\cos\theta)-x\sin\theta \\ xz(1-\cos\theta)-y\sin\theta & yz(1-\cos\theta)+x\sin\theta & z^2(1-\cos\theta)+\cos\theta \end{pmatrix}$$

Матрица вращения через углы Эйлера.

$$Q(\alpha,\beta,\gamma) = \begin{pmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{pmatrix}$$

▶ 1.5.3. Использование кватернионов для вращений.

Каждому единичному кватерниону соответствует ровно одно вращение, а каждому вращению соответствует ровно два единичных кватерниона $\pm q$ (точнее, имеет место следующий изоморфизм групп: $SO(3) \simeq SU(2)/\{\pm E\}$).

Переход от кватерниона $q = w + \mathbf{i}x + \mathbf{j}y + \mathbf{k}z$ к матрице:

$$Q = \begin{pmatrix} 2x^2 + 2w^2 - 1 & 2xy - 2zw & 2xz + 2yw \\ 2xy + 2zw & 2y^2 + 2w^2 - 1 & 2yz - 2xw \\ 2xz - 2yw & 2yz + 2xw & 2z^2 + 2w^2 - 1 \end{pmatrix}$$

Переход от матрицы $Q = \begin{pmatrix} Q_{xx} & Q_{xy} & Q_{xz} \\ Q_{yx} & Q_{yy} & Q_{yz} \\ Q_{zx} & Q_{zy} & Q_{zz} \end{pmatrix}$ к кватерниону $q = w + \mathbf{i}x + \mathbf{j}y + \mathbf{k}z$:

$$w = \frac{1}{2}\sqrt{1 + Q_{xx} + Q_{yy} + Q_{zz}}$$

$$x = \frac{1}{2}\sqrt{1 + Q_{xx} - Q_{yy} - Q_{zz}}\,\operatorname{sign}(Q_{zy} - Q_{yz})$$
$$y = \frac{1}{2}\sqrt{1 - Q_{xx} + Q_{yy} - Q_{zz}}\,\operatorname{sign}(Q_{xz} - Q_{zx})$$
$$z = \frac{1}{2}\sqrt{1 - Q_{xx} - Q_{yy} + Q_{zz}}\,\operatorname{sign}(Q_{yx} - Q_{xy})$$

Вращению вокруг единичного вектора $\mathbf{u} = (x, y, z)$ на угол θ соотвествует кватернион

$$q = \cos\frac{\theta}{2} + x\sin\frac{\theta}{2}\mathbf{i} + y\sin\frac{\theta}{2}\mathbf{j} + z\sin\frac{\theta}{2}\mathbf{k}$$

Кватерниону $q = w + \mathbf{i}x + \mathbf{j}y + \mathbf{k}z$ соответствует вращение вокруг единичного вектора

$$\mathbf{u} = \left(\frac{x}{\sqrt{1-w^2}}, \frac{y}{\sqrt{1-w^2}}, \frac{z}{\sqrt{1-w^2}}\right) \text{ на угол } \theta = 2\arccos w.$$

★ 1.6. Метод Рунге-Кутта.

Наиболее употребительным методом Рунге-Кутта решения уравнения первого порядка $y' = F(x, y)$ является метод четвертого порядка, в котором вычисления производятся по формуле:

$$y_{k+1} = y_k + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

, где

$$k_1 = F_k h = F(x_k, y_k)h$$

$$k_2 = F(x_k + \frac{h}{2}, y_k + \frac{k_1}{2})h$$

$$k_3 = F(x_k + \frac{h}{2}, y_k + \frac{k_2}{2})h$$

$$k4_= F(x_k + h, yk + k3)h$$

$$k = 0, \dots, n-1$$

$$h = \frac{x_f - x_0}{n}$$

★ 1.7. Формула Кардано.

Формула Кардано — формула для нахождения корней канонической формы кубического уравнения

$$y^3 + py + q = 0$$

над полем комплексных чисел. К такому виду может быть приведено любое кубическое уравнение общего вида

$$ax^3 + bx^2 + cx + d = 0$$

при помощи замены:

$$y = x + \frac{b}{3a}$$

$$p = -\frac{b^2}{3a^2} + \frac{c}{a}$$

$$q = \frac{2b^3}{27a^3} - \frac{bc}{3a^2} + \frac{d}{a}$$

Определим Q:

Число действительных корней кубического уравнения зависит от знака Q:

$Q > 0$ — один действительный корень и два сопряженных комплексных корня.

$Q < 0$ — три действительных корня.

$Q = 0$ — один однократный действительный корень и два двукратных, или, если $p = q = 0$, то один трехкратный действительный корень.

По формуле Кардано, корни кубического уравнения в канонической форме равны:

$$y_1 = \alpha + \beta$$

$$y_{2,3} = -\frac{\alpha + \beta}{2} \pm i\frac{\alpha - \beta}{2}\sqrt{3}$$

$$\alpha = \sqrt[3]{-\frac{q}{2} + \sqrt{Q}}$$

$$\beta = \sqrt[3]{-\frac{q}{2} - \sqrt{Q}}$$

Дискриминант многочлена $y^3 + py + q$ при этом равен $\Delta = -108Q$.

Применяя данные формулы, для каждого из трёх значений α необходимо брать такое β, для которого выполняется условие $\alpha\beta = -p/3$ (такое значение β всегда существует).

Если кубическое уравнение действительное, то рекомендуется по возможности выбирать действительные значения α,β.

2. Алгоритмы и исходники.

★ 2.1. Теория чисел.

► 2.1.1. Алгоритм Шермана-Лемана. (Разложение на множители за $O(n^{1/3})$)

```
#define EPS 1e-4

// Finds the greatest common divisor of a and b
inline long long gcd(long long a,long long b) {
  if(b<0) b=-b;
  while(b) {
    long long r=a%b;
    a=b;
    b=r;
  }
  return a;
}

/*
 * Returns number of all prime divisors
 * divz will contain the factorization
 */
int factor(long long n,long long *divz) {
  int cnt=0;
  // Find all small divisors
  for(long long r=2;r*r*r<=n;++r) {
    while(!(n%r)) {
      divz[cnt++]=r;
      n/=r;
    }
  }
  if(n>1) {
    // Now n is a prime or a product of two primes
    long long k,d;
    double pow_n=pow(n+0.0,1/6.0);
    for(k=1;k*k*k<=n;++k) {
      double x=pow_n;
      x=0.25*x/sqrt(k+0.0);
      long long up=(long long)(x+1-EPS);
      for(d=0;d<=up;++d) {
        x=sqrt(4*k*n+0.0);
        long long t=(long long)(x+EPS)+d;
        t=t*t-4*k*n;
        x=sqrt(t+0.0);
        long long s=(long long)(x+0.5);
        if(s*s==t) {
          long long A,B;
          x=sqrt(4*k*n+0.0);
          A=(long long)(x+EPS)+d;
          x=sqrt(A*A-4*k*n+0.0);
```

```
          B=(long long)(x+EPS);
          // Now A^2 ≡ B^2  mod n
          t=gcd(n,A+B);
          if(t==1 || t==n)
            t=gcd(n,A-B);
          if(t>1 && t<n) {
            s=n/t;
            divz[cnt++]=t;
            divz[cnt++]=s;
            return cnt;
          }
        }
      }
    }
    // n is a prime
    divz[cnt++]=n;
  }
  return cnt;
}
```

▶ **2.1.2. Тест Рабина-Миллера проверки на простоту.**

```
/*
 * Returns a · b  mod n
 * a, b, n  ⩽  2^63 − 1
 */
inline unsigned long long mul(unsigned long long a,
                              unsigned long long b,
                              unsigned long long n) {
  unsigned long long res=0;
  for(;b;a=(a<<1)%n,b>>=1)
    if(b&1) res=(res+a)%n;
  return res;
}

// Returns x^k  mod n
unsigned long long pow_mod(unsigned long long x,
                           unsigned long long k,
                           unsigned long long n) {
  unsigned long long a=1,b=x;
  while(k) {
    if(k&1) {
      a=mul(a,b,n);
      --k;
    } else {
      b=mul(b,b,n);
      k>>=1;
    }
  }
  return a;
}
```

```
// Returns random 64-bit number
inline long long rand64() {
  return ((long long)rand())|(((long long)rand())<<15)|
         (((long long)rand())<<30)|(((long long)rand())<<45)|
         (((long long)rand())<<60);
}

#define ITER 20

/*
 * Checks whether n is a prime
 * Probability of error is 1/4^ITER
 */
int rabin_miller(unsigned long long n) {
  if(n<=3) return 1;
  int r;
  unsigned long long t;
  for(r=0,t=n-1;!(t&1);++r,t>>=1);
  for(int iter=0;iter<ITER;++iter) {
    unsigned long long a;
    do {
      a=rand64()%n;
    } while(a<2);
    unsigned long long z=pow_mod(a,t,n);
    if(z!=1) {
      int bad=1;
      for(int j=1;j<=r;++j) {
        if(z==n-1) {
          bad=0;
          break;
        }
        z=mul(z,z,n);
      }
      if(bad) {
        return 0;
      }
    }
  }
  return 1;
}
```

► 2.1.3. Нахождение квадратного корня из вычета.

```
// 2x2 matrices
struct mat2x2 {
  int p,a,b,c,d;

  mat2x2(int p):
    p(p), a(1), b(0), c(0), d(1)
    { }
```

```
  mat2x2(int p,int a,int b,int c,int d):
    p(p), a(a), b(b), c(c), d(d)
    { }

  inline mat2x2 &operator*=(const mat2x2 &A) {
    mat2x2 res(p);
    res.a=((long long)a*A.a+(long long)b*A.c)%p;
    res.b=((long long)a*A.b+(long long)b*A.d)%p;
    res.c=((long long)c*A.a+(long long)d*A.c)%p;
    res.d=((long long)c*A.b+(long long)d*A.d)%p;
    return (*this=res);
  }

  mat2x2 &operator^=(int k) {
    mat2x2 A(p),B(*this);
    while(k) {
      if(k&1) {
        A*=B;
        --k;
      }
      else {
        B*=B;
        k>>=1;
      }
    }
    return (*this=A);
  }
};

// 32-bit random number
inline int rand32() {
  return rand()|(rand()<<15)|(rand()<<30);
}

// Finds symbol Jacobi (m/n)
int jacobi(int m,int n) {
  if(!m) return 0;
  for(;!(n&1);n>>=1);
  for(;!(m&3);m>>=2);
  int s=1;
  if(!(m&1)) {
    if((n&7)==3 || (n&7)==5)
      s=-s;
    m>>=1;
  }
  if(m==1) return s;
  if(((n-1)&3)==2 && ((m-1)&3)==2)
    s=-s;
  return s*jacobi(n%m,m);
}
```

```
/*
 * Returns such x that x^2 ≡ a mod p
 * or 0 if such x does not exist
 */
int sqrt_mod(int a,int p) {
  if(jacobi(a,p)!=1)
    return 0;
  int b;
  // Find such b that polynom x^2 - bx + a is irreducible in Z_p[x]
  do {
    b=rand32()%p;
  }
  while(jacobi((((long long)b*b-4*(long long)a)%p+p)%p,p)!=-1);
  mat2x2 A(p,0,1,p-a,b);
  A^=(p-1)>>1;
  return (((long long)a*A.b)%p+p)%p;
}
```

► 2.1.4. Вычисление квадратного корня (быстро, но неточно).

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;
  i  = * ( long * ) &y;                        // grab bits
  i  = 0x5f3759df - ( i >> 1 );                // do magic
  y  = * ( float * ) &i;
  y  = y * ( threehalfs - ( x2 * y * y ) );    // improve

  return y * number;
}
```

► 2.1.5. Следующая маска с таким-же числом битов.

```
unsigned snoob(unsigned x) {
   unsigned smallest, ripple, ones;
                                  // x =  xxx0 1111 0000
   smallest = x & -x;             //      0000 0001 0000
   ripple = x + smallest;         //      xxx1 0000 0000
   ones = x ^ ripple;             //      0001 1111 0000
   ones = (ones >> 2)/smallest;   //      0000 0000 0111
   return ripple | ones;          //      xxx1 0000 0111
}
```

★ 2.2. Строковые алгоритмы.

► 2.2.1. Алгоритм Дюваля и его применение. (Построение линдонской декомпозиции)

```
/*
 * Finds Lyndon's decomposition str = w1w2...wk,
 * where w1 <= w2 <= ... <= wk - Lyndon's words
 * ends will contain endings of words wi
 * function returns k
 */
int duval(char *str,int *ends) {
  int n=strlen(str);
  int l=0;
  for(int h=0;h<n;) {
    int i=h+1;
    int j=h+2;
    while(str[j-1]>=str[i-1]) {
      if(str[j-1]>str[i-1]) {
        i=h+1;
      }
      else {
        ++i;
      }
      ++j;
    }
    do {
      h+=(j-i);
      ends[l++]=h;
    } while(h<i);
  }
  return l;
}

/*
 * Finds MSP = Minimum Starting Point of str
 * that is lexicographically minimal cyclic shift of str
 * Uses Duval's algorithm to find Lyndon's decomposition of the word str^2
 */
int find_MSP(char *str) {
  int n=strlen(str);
  int prev=0;
  for(int h=0;h<(n<<1);) {
    int i=h+1;
    int j=h+2;
    while(j<=(n<<1) && str[(j-1)%n]>=str[(i-1)%n]) {
      if(str[(j-1)%n]>str[(i-1)%n]) {
        i=h+1;
      }
      else {
        ++i;
      }
      ++j;
```

```
    }
    do {
      h+=(j-i);
    } while(h<i);
    if(h-prev>=n) {
      return prev;
    }
    prev=h;
  }
}
```

▶ **2.2.2. Построение автомата Ахо-Корасик.**

```
const int N = 10100;

int n, k, total = 1;
int price[N];
int lev[N];
int inds[N] = {0, 1};
int next[N][26];
int prev[N];
int f[N];
int prevlet[N];
int link[N];

void put(char* str){
  int i, cver = 1, c;
  for (i=0; str[i]; i++){
    c = str[i] - 'a';
    if (!next[cver][c]){
      total++;
      next[cver][c] = total;
      prev[total] = cver;
      prevlet[total] = c;
      lev[total] = lev[cver] + 1;
      inds[total] = total;
    }
    cver = next[cver][c];
  }
  f[cver] = 1;
  return;
}

bool cmp(int a, int b){
  return (lev[a] < lev[b]);
}

void count_link(int a){
  if (lev[a] <= 1){
    link[a] = 1;
    return;
```

```
    }
    int cver = link[prev[a]];
    int c = prevlet[a];
    while ((cver > 1) && (!next[cver][c])){
      cver = link[cver];
    }
    if (next[cver][c]){
      cver = next[cver][c];
    }
    link[a] = cver;
    return;
  }

  void build_auto(char** strs, int n){
    int i;
    for (i=0; i<n; i++){
      put(strs[i]);
    }
    sort(inds+1, inds+total+1, cmp);
    for (i=1; i<=total; i++){
      count_link(inds[i]);
    }
    return;
  }
```

▶ 2.2.3. Алгоритм Укконена. (Построение сжатого суффиксного дерева за линейное время)

```
// Suffix tree node
struct suffix_node {
  int l,r;
  char c;
  suffix_node *parent; // Parent of the node
  suffix_node *link;   // Suffix link
  suffix_node *next;   // Next node in the current level
  suffix_node *child;  // Link to the leftmost child

  inline int len() {
    return r-l;
  }
};

struct suffix_state {
  suffix_node *v;
  int len;

  suffix_state(suffix_node *v,int len):
    v(v), len(len)
    { }
};

#define MAX 100100

suffix_node buf[MAX<<1];
int used;

suffix_node *root;

char str[MAX];
int n;

// Splits the node according to the state
inline suffix_node *split(const suffix_state &st) {
  suffix_node *v=st.v;
  if(!st.len) {
    return v;
  }
  int len=v->len()-st.len;
  if(!len) {
    return v->parent;
  }
  suffix_node *w=&buf[used++];
  w->l=v->l;
  w->r=v->l+len;
  w->c=v->c;
  w->child=v;
  w->next=v->next;
```

```
    w->parent=v->parent;
    v->parent=w;
    v->next=0;
    v->l+=len;
    v->c=str[v->l];
    if(w->parent->child==v) {
      w->parent->child=w;
    }
    else {
      suffix_node *u=w->parent->child;
      while(u->next!=v)
        u=u->next;
      u->next=w;
    }
    return w;
  }

  // Fast scanning
  inline suffix_state down(suffix_node *v,int l,int r) {
    if(l==r)
      return suffix_state(v,0);
    for(;;) {
      v=v->child;
      while(v->c!=str[l])
        v=v->next;
      if(v->len()>=r-l) {
        return suffix_state(v,v->len()-r+l);
      }
      l+=v->len();
    }
  }

  // Slow scanning
  inline suffix_state go(const suffix_state &st,char c) {
    suffix_node *v=st.v;
    int len=st.len;
    if(len) {
      if(str[v->r-len]==c) {
        return suffix_state(v,len-1);
      }
      else {
        return suffix_state(0,-1);
      }
    }
    suffix_node *w=v->child;
    while(w && w->c!=c) {
      w=w->next;
    }
    if(w) {
      return suffix_state(w,w->len()-1);
    }
    else {
```

```
    return suffix_state(0,-1);
  }
}

// Makes suffix link
suffix_node *link(suffix_node *w) {
  if(!w->link) {
    w->link=split(down(link(w->parent),w->l+(w->parent==root),w->r));
  }
  return w->link;
}

// Processes one character
inline void add_char(suffix_state &st,int k) {
  for(;;) {
    suffix_state nst=go(st,str[k]);
    if(nst.v) {
      st=nst;
      return;
    }
    suffix_node *v=split(st);
    suffix_node *w=&buf[used++];
    w->parent=v;
    w->l=k;
    w->r=n;
    w->c=str[k];
    if(!v->child) {
      v->child=w;
    }
    else {
      suffix_node *u=v->child;
      while(u->next)
        u=u->next;
      u->next=w;
    }
    st.v=link(v);
    st.len=0;
    if(v==root) return;
  }
}

// Constructs the tree
void build_tree() {
  root=&buf[used++];
  root->parent=root;
  root->link=root;
  suffix_state st(root,0);
  for(int i=0;i<n;++i)
    add_char(st,i);
}
```

▶ **2.2.4. Построение суффиксного автомата.**

```
int len[2*N];
int link[2*N];
int next[2*N][ALPH];
int total = 1;

inline void copy(int src, int dest){
  len[src] = len[dest];
  link[src] = link[dest];
  memcpy(next[src], next[dest], sizeof(next[dest]));
  return;
}

void build(char* str){
  int slen = strlen(str);
  int i, last = 1, j;
  for (i=0; i<slen; i++){
    int c = (str[i] - 'A');
    int cpos = last;
    total++;
    int npos = total;
    len[npos] = len[cpos] + 1;
    while ((cpos > 1) && (!next[cpos][c])){
      next[cpos][c] = npos;
      cpos = link[cpos];
    }
    if (!next[cpos][c]){
      next[cpos][c] = npos;
      link[npos] = cpos;
    }
    else{
      int k = next[cpos][c];
      if (len[k] == len[cpos] + 1){
        link[npos] = k;
      }
      else{
        int r = ++total;
        copy(r, k);
        len[r] = len[cpos] + 1;
        link[k] = r;
        link[npos] = r;
        while ((cpos > 0) && (len[next[cpos][c]] >= len[r])){
          next[cpos][c] = r;
          cpos = link[cpos];
        }
      }
    }
    last = npos;
  }
  return;
}
```

▶ 2.2.5. Построение суффиксного массива + LCP.

```
const int NMAX = 256 * 1024;
char str[NMAX];
int c[NMAX],c2[NMAX];
int p[NMAX],p2[NMAX];
int pos[NMAX];
int num[NMAX];
int n;
int lcp[NMAX];

void build_sa() {
  for (int i = 0; i < n; i++){
    c[i] = str[i];
    p[i] = i;
  }
  for (int k = 0, k1 = 1; k < n; k = k1, k1 <<= 1){
    clr(num);
    for (int i = 0; i < n; i++){
      num[c[i]]++;
    }
    pos[0] = 0;
    for (int i = 0; i < NMAX - 1; i++){
      pos[i + 1] = pos[i] + num[i];
    }
    for (int i = 0; i < n; i++){
      int j = (p[i] - k + n) % n;
      p2[pos[c[j]]++] = j;
    }
    int cc=0;
    for (int i = 0; i < n; i++){
      if (i && (make_pair(c[p2[i]],c[(p2[i] + k) % n]) != make_pair(c[p2[i-1]],c[(p2[i-1]
+ k) % n]))){
        cc++;
      }
      c2[p2[i]] = cc;
    }
    for (int i = 0; i < n; i++){
      c[i] = c2[i]; p[i] = p2[i];
    }
  }
  for (int i = 0; i < n; i++)
    p2[p[i]] = i;
}

void calc_lcp(){
  int k = 0;
  int s1 = 0;
  int s2 = p[p2[s1] + 1];
  for (int i = 0; i < n; i++){
    if (k)
      k--;
```

```
    if (p2[s1] == n - 1){
      k = 0;
      s1++;
      continue;
    }
    s2 = p[p2[s1] + 1];
    while (k < n && str[(s1 + k) % n] == str[(s2 + k) % n])
      k++;
    lcp[p2[s2]] = k;
    if (p2[(s2 + 1) % n] < p2[s1 + 1])
      k = 0;
    s1++;
  }
}
```

▶ **2.2.6. Построение префикс-функции и Z-функции.**

```
const int LEN = 100;
char s[LEN];
int n;

int p[LEN];
int z[LEN];

void prefixFunction()
{
  p[0] = -1;
  for (int i = 1; i < n; ++i)
  {
    p[i] = p[i - 1];
    while (p[i] >= 0 && s[i] != s[p[i] + 1])
      p[i] = p[p[i]];
    if (s[i] == s[p[i] + 1])
      ++p[i];
  }
}

void zFunction()
{
  int l = 0, r = -1;
  z[0] = 0;
  for (int i = 1; i < n; ++i)
  {
    if (i <= r)
      z[i] = min(z[i-l], r-i+1);
    else
      z[i] = 0;
    while (s[z[i]] == s[i + z[i]])
      ++z[i];
    if (r < i + z[i] -1)
    {
      l = i;
      r = i + z[i] - 1;
```

```
    }
  }
}
```

★ 2.4. Декартово дерево.

```
const int CNT_NODE = 200000;
const int INF = 1 << 30;

struct Node {
  Node *l, *r;
  int y;
  int data, cnt;
  long long sum;
  int min, max;
  bool rev;
  int add;
};

Node * EMPTY;
int cnt_node;
Node node[CNT_NODE];

inline void addForTree(Node * t, int c)
{
  if (t == EMPTY)
    return ;
  t -> add += c;
  t -> data += c;
  t -> sum += c * (t -> cnt);
  t -> min += c;
  t -> max += c;
}

inline void normalize(Node * t)
{
  if (t == EMPTY)
    return;
  if (t -> rev)
  {
    std::swap(t -> l, t -> r);
    t -> l -> rev ^= 1;
    t -> r -> rev ^= 1;

    t -> rev = 0;
  }
  if (t -> add != 0)
  {
    addForTree(t -> l, t -> add);
    addForTree(t -> r, t -> add);
    t -> add = 0;
  }
```

```
}

inline void recalc(Node * t)
{
  if (t == EMPTY)
    return;
  normalize(t);
  t -> cnt = t -> l -> cnt + t -> r -> cnt + 1;
  t -> sum = (t -> l -> sum + t -> r -> sum + t -> data);
  t -> min = std::min(std::min(t -> l -> min, t -> r -> min), t -> data);
  t -> max = std::max(std::max(t -> l -> max, t -> r -> max), t -> data);
}

void split(Node * t, int x, Node * &l, Node * &r)
{
  if (t == EMPTY)
  {
    l = r = EMPTY;
    return;
  }
  normalize(t);
  if (x <= t -> l -> cnt)
  {
    split(t -> l, x, l, t -> l);
    r = t;
  }
  else
  {
    split(t -> r, x - t -> l -> cnt - 1, t -> r, r);
    l = t;
  }
  recalc(t);
}

Node * merge(Node * l, Node * r)
{
  if (l == EMPTY)
    return r;
  if (r == EMPTY)
    return l;
  normalize(l);
  normalize(r);
  if (l -> y < r -> y)
  {
    l -> r = merge(l -> r, r);
    recalc(l);
    return l;
  }
  else
  {
    r -> l = merge(l, r -> l);
    recalc(r);
```

```
    return r;
  }
}

Node * newTree(int data)
{
  assert(cnt_node < CNT_NODE);
  int res = cnt_node++;
  node[res].l = node[res].r = EMPTY;
  node[res].data = data;
  node[res].sum = node[res].min = node[res].max = data;
  node[res].cnt = 1;
  node[res].y = rand();
  return &node[res];
}

void printTree(Node * root)
{
  if (root == EMPTY)
    return ;
  normalize(root);
  printTree(root -> l);
  printf("%d ", root -> data);
  printTree(root -> r);
}

void init()
{
  EMPTY = &node[0];
  node[0].l = node[0].r = EMPTY;
  node[0].data = -1;
  node[0].cnt = 0;
  node[0].y = INF;
  node[0].sum = 0;
  node[0].min = INF;
  node[0].max = -INF;
  node[0].rev = node[0].add = 0;
  cnt_node = 1;
}
// Indices l and r numbered from 1
int getSum(Node * root, int l, int r)
{
  Node *a, *b, *c;
  split(root, r, b, c);
  split(b, l - 1, a, b);
  int res = b -> sum;
  root = merge(a, merge(b, c));
  return res;
}
```

★ **2.1. Максимальный поток минимальной стоимости (время работы— $O(N^3)$).**

```
const int NN = 310;
const int INF = 10000000;

int N;
int cap[NN][NN];
int flow[NN][NN];
int cost[NN][NN];

int prev[NN];
int dist[NN];
int pi[NN];
char used[NN];

int find_path(int src, int dest){
  int i;
  for (i=0; i<N; i++){
    dist[i] = INF;
    used[i] = 0;
  }
  dist[src] = 0;
  while (1){
    int bdist = INF, bver;
    for (i=0; i<N; i++){
      if ((!used[i]) && (dist[i] < bdist)){
        bdist = dist[i];
        bver = i;
      }
    }
    if (bdist == INF) break;
    used[bver] = 1;
    for (i=0; i<N; i++){
      if (used[i]) continue;
      if (flow[i][bver]){
        int cval = dist[bver] + pi[bver] - pi[i] - cost[i][bver];
        if (dist[i] > cval){
          dist[i] = cval;
          prev[i] = bver;
        }
      }
      if (flow[bver][i] < cap[bver][i]){
        int cval = dist[bver] + pi[bver] - pi[i] + cost[bver][i];
        if (dist[i] > cval){
          dist[i] = cval;
          prev[i] = bver;
        }
      }
    }
  }
  for (i=0; i<N; i++){
    if (pi[i] < INF) pi[i] += dist[i];
```

```
  }
  return (dist[dest] < INF);
}

int mcmf(int src, int dest, int& fcost){
  int res = 0;
  fcost = 0;
  while (find_path(src, dest)){
    int x = dest;
    int topush = INF;
    while (x != src){
      int y = prev[x];
      int tmp = (flow[x][y]) ? flow[x][y] : (cap[y][x] - flow[y][x]);
      if (tmp < topush) topush = tmp;
      x = y;
    }
    x = dest;
    while (x != src){
      int y = prev[x];
      if (flow[x][y]){
        flow[x][y] -= topush;
        fcost -= topush * cost[x][y];
      }
      else{
        flow[y][x] += topush;
        fcost += topush * cost[y][x];
      }
      x = y;
    }
    res += topush;
  }
  return res;
}
```

$Z\ Z\ Z$

$Z\ Z\ Z$

$Z\ Z\ Z$