Multe "smenuri" de programare in C/C++... si nu numai!

(Categoria *Limbaje de programare*, Autor *Mircea Pasoi*
http://www.infoarena.ro/multe-smenuri-de-programare-in-cc-si-nu-numai
)

Acest articol vine ca o completare a articolului scris de Alexandru Mosoi (vezi pag. 9), prezentand noi trucuri pe care le-am folosit si m-au ajutat mult. O mare parte din acestea le-am invatat din sursele lui Radu Berinde (cred ca stiti cu totii cine este), asadar ii multumesc!

Array-uri neindexate de la 0

Un dezavantaj fata de Pascal este faptul ca in C nu putem avea expresii de genul `A[-100]` unde `A` este un vector. Dar acest lucru se poate remedia. Spre exemplu, daca vrem sa facem un vector `A` cu elemente de la `-100` la `100` procedam astfel:

```
int A[201];
#define A (A + 100)
```

Fisiere de intrare si iesire

Folositi `freopen()` in loc de `fopen()` deoarece este mai comod, in special la concursurile in care intrarea si iesirea sunt standard.

```
freopen("in.txt", "r", stdin);
freopen("out.txt", "w", stdout);
```

Cautare binara

Urmatorul cod este de aproximativ `4` ori mai rapid (am testat cu cautare binara ca in manual) , mai usor de inteles, mai flexibil si mai scurt... ce ati putea dori mai mult?

```
int N, A[N];
int binary_search(int val)
{
    int i, step;
    for (step = 1; step < N; step <<= 1);
    for (i = 0; step; step >>= 1)
        if (i + step < N && A[i + step] <= val)
           i += step;
    return i;
}
```

Procedura de mai sus face cautarea binara folosind puteri a lui `2` in ordine descrescatoare, practic incerc sa determin fiecare bit al rezultatului.

Impartire in bucati de marime sqrt(n) (cunoscut si ca "smenul lui Bogdan Batog")

Sa presupunem ca avem un vector de lungime n cu numere reale pe care se fac urmatoarele operatii:
`ADUNA(st, dr, x)` - toate elementele cu indicii intre `st` si `dr` isi cresc valoarea cu `x`
`SUMA(st, dr)` - returneaza suma elementelor cu indicii intre `st` si `dr`
Pentru cei ce cunosc arbori de intervale, rezolvarea acestei probleme in `O(lg n)` per operatie este o munca usoara, dar presupunand ca nu stim aceasta structura putem folosi urmatorul truc: vom construi un al doilea vector de lungime `sqrt(n)` care retine suma elementelor pe bucati de lungime `sqrt(n)`. Pentru a face o operatie pe un interval `[st, dr]` vom imparti acest interval in bucati de `sqrt(n)` a caror actualizare o facem in vectorul `B` si elementele care raman in margine in vectorul `A`.

De exemplu pe vectorul $A_{0..15}$ vom avea vectorul $B_{0..3}$. Pentru a actualiza de exemplu [2, 12] vom actualiza B_1 (care corespunde lui [4..7]), B_2 (pentru [8..11]) si A_2, A_3, A_{12} (elementele din margini). Cum sunt maxim sqrt(n) elemente de actualizat in B si pe margini nu vom actualiza niciodata mai mult de 2 * sqrt(n) elemente putem concluziona ca operatiile vor avea complexitate O(sqrt(n)).
Daca am avea aceeasi problema dar in doua dimensiuni, am putea face acelasi "smen" pentru fiecare linie pentru o complexitate O(n*sqrt(n)) per operatie, sau cu arbori de intervale pe fiecare linie O(n*lg n).
Putem, de asemenea obtine o complexitate O(n) folosind urmatoarea impartire:
A - pentru bucati 1 * 1
B - pentru bucati sqrt(n) * sqrt(n)
C - pentru bucati 1 * sqrt(n)
D - pentru bucati sqrt(n) * 1
Acest mod de reprezentare este o extindere directa a aceluiasi smen in doua dimensiuni. Aceasta idee poate fi folosita si pentru alte operatii: inmultire, minim, maxim, etc. In general, orice se poate rezolva cu acest "smen" se poate obtine la o complexitate mai buna cu arbori de intervale, dar merita sa stiti si aceasta ideea deoarece de multe ori scuteste din efortul de implementare, desi se pierde din viteza... alegerea voastra! ;)

LCA in O(sqrt(n))

Daca nu stiti ce este LCA, va recomand sa cititi articolul lui Emilian Miron (vezi pag. 15) din cadrul site-ului pentru a va documenta. In continuare vom prezenta un algoritm mai ineficient, dar foarte usor de implementat. Consideram arborele si atribuim fiecarui nod o inaltime. Vom imparti arborele in sqrt(H) intervale in functie de inaltime, unde H e inaltimea maxima (de exemplu la H=9 nodurile cu inaltimi intre 0 si 2 vor forma un interval, [3..5] alt interval si ultimul interval de inaltimi [6..8]). Astfel, pentru fiecare nod, pe langa tatal sau, vom retine si tatal din intervalul de mai sus, printr-o parcurgere DF. In continuare, codul:
H se calculeaza inainte sau poate fi constant
T sunt tatii nodului si N numarul de noduri

```
void DF(int n, int t2, int lev)
{
    int i;
    T2[n] = t2, Lev[n] = lev;
    if (lev % H == 0) t2 = n;
    for (i = 0; i < N; i++)
        if (T[i] == n) DF(i, t2, lev+1);
}
```

Operatia de LCA se va realiza apoi foarte usor, urcand pe tatii din intervale, pana se ajunge la doua noduri in acelasi interval, apoi folosindu-se metoda clasica. Cod:

```
int LCA(int x, int y)
{
    while (T2[x] != T2[y])
        if (Lev[x] > Lev[y])
            x = T2[x];
        else
            y = T2[y];
    while (x != y)
        if (Lev[x] > Lev[y])
            x = T[x];
        else
            y = T[y];
    return x;
}
```

LCA in $O(\lg^2 n)$

Aceeasi problema, dar o alta rezolvare. Vom construi o matrice $A_{i,j}$ cu semnificatia $A_{i,j}$ = al 2^i-lea tata al nodului j. Folosind aceasta matrice putem cauta binar ($O(\lg n)$) nivelul pe care s-ar putea afla LCA-ul a doua noduri si sa determinam daca nodul ales este corect - adica daca nodul situat la acel nivel este acelasi pentru cele doua noduri pentru care se face LCA ($O(\lg n)$ cu matricea de mai sus). Complexitate finala $O(\lg^2 n)$ si $O(n*\lg n)$ memorie.

For-uri "complicate"

for-ul in C/C++ este foarte flexibil si poate ajuta foarte mult in compactarea codului, deci si a timpului de implementare. In continuare vom prezenta algoritmul merge sort (sortare prin interclasare) scris in cateva linii (putine, zic eu!):

```
int N, A[N], B[N];
void merge_sort(int l, int r)
{
    int m = (l + r) >> 1, i, j, k;
    if (l == r) return;
    merge_sort(l, m);
    merge_sort(m + 1, r);
    for (i=l, j=m+1, k=l; i<=m || j<=r; )
        if (j > r || (i <= m && A[i] < A[j]))
            B[k++] = A[i++];
        else
            B[k++] = A[j++];
    for (k = l; k <= r; k++) A[k] = B[k];

}
```

Recomandari generale

1. Programare dinamica cu memoizare: mult mai simplu si uneori chiar mai rapida cand nu ne trebuie tot array-ul
2. Algoritmi randomizati: de multe ori mai usor de implementat si mai eficienti, mai bine decat cei euristici, dar necesita o analiza mult mai atenta a performantei. Exemple clasice: quicksort, statistici de ordine

"Smenul lui Mars" (Marius Andrei)

Consideram urmatoarea problema: se da un vector A de N elemente pe care se fac M astfel de operatii: `ADUNA(st, dr, x)` - toate elementele cu indicii intre st si dr ($0 \le st \le dr < N$) isi cresc valoarea cu x . La sfarsit trebuie sa se afiseze vectorul rezultat. In continuarea vom descrie o metoda care ne da un timp de rulare de $O(1)$ pentru operatia `ADUNA`si $O(N)$ pentru a determina toate elementele din vector. Vom construi un al doilea vector B de $N+1$ elemente, cu proprietatea ca $A_i = B_0 + B_1 + \dots B_i$. Astfel, o operatie`ADUNA(st, dr, x)` devine:

```
B[st] += x;
B[dr + 1] -= x;
```

Da, este chiar asa de simplu! Pentru a determina un element A_i vom aduna pur si simplu $B_0 + B_1 + \dots B_i$. Incercati pe foaie sa vedeti cum funtioneaza. Aceasta ideea poate fi extinsa si in doua dimensiuni, construind B astfel incat $A_{i,j}$ = suma subtabloului din B cu coltul in $(0, 0)$ si (i, j), astfel (pt. `ADUNA(x1,y1,x2,y2,v)`):

```
B[x1][y1] += v;
B[x1][y2 + 1] -= v;
```

```
B[x2 + 1][y1] -= v;
B[x2 + 1][y2 + 1] += v;
```

Pe cazul general, daca vrem sa facem operatii in d dimensiuni vom avea o complexitate $O(2^d)$. Reamintesc ca aceasta metoda este eficienta doar cand se vrea afisata vectorul/matricea/etc. doar la sfarsitul operatiilor sau sunt foarte putine interogari ale valorilor elementelor, deoarece aflarea unui element este o operatie foarte ineficienta:$O(i)$ pentru a afla valorile elementelor pana la pozitia i.

Grafuri cu liste de adiacenta

Se stie (sau ar trebui sa se stie!) ca lucrul cu pointerii este foarte incet... astfel, cand retinem un graf rar (numar mare de noduri, numar mic de muchii) cu pointeri (vezi mai jos) incetinim foarte mult programul.

```
?
1  struct list
2  {
3      int n;
4      struct list *next;
5  }
6  typedef struct list list;
```

In contiuare vom prezenta o metoda care este de 3-4 ori mai rapida (adica parcurgerile DF , BF sau altii algoritmi ruleaza de 3-4 ori mai rapid cand graful este stocat astfel), dar are ca dezavantaj necesitatea de a citi de doua ori fisierul de intrare.

```
#include <stdlib.h>
#include <stdio.h>

int N, M, *G[N], Deg[N];

int main(void)
{
    int i, j;

    freopen("in.txt", "r", stdin);
    scanf("%d %d", &N, &M);
    for (; M > 0; M--)
    {
        scanf("%d %d", &i, &j);
        Deg[i - 1]++, Deg[j - 1]++;
    }
    for (i = 0; i < N; Deg[i++] = 0)
        G[i] = (int *) malloc(Deg[i]*sizeof(int));
    fseek(stdin, 0, SEEK_SET);
    scanf("%d %d", &N, &M);
    for (; M > 0; M--)
    {
        scanf("%d %d", &i, &j);
        i--, j--;
        G[i][Deg[i]++] = j,
            G[j][Deg[j]++] = i;
    }

    return 0;
```

```
}
```

Sporul de viteza se datoreaza faptului ca se folosesc vectori in loc de pointeri si struct-uri. Daca ne permite memoria putem evita citirea de doua ori a fisierul prin pastrarea muchiilor intr-o lista de muchii si apoi, dupa calcularea gradelor, inserarea muchiilor in liste. Pentru a demonstra eficienta acestei metode faceti urmatorul test: implementati o sursa cu pointeri si struct si implementati un BF, apoi scrieti codul de mai sus cu urmatoarele modificari:

```
...
for (i = 0; i < N; i++)
{
	G[i] = (int *) malloc((Deg[i]+1)*sizeof(int));
	G[i][Deg[i]] = -1;
	Deg[i] = 0;
}

...
```

si implementati BF astfel:

```
void BF()
{
	int Q[N], ql, qr, *p;
	char U[N];
	memset(U, 0, sizeof(U));
	U[Q[ql = qr = 0] = n] = 1;
	for (; ql <= qr; ql++)
		for (p = G[Q[ql]]; *p != -1; p++)
			if (!U[*p]) U[Q[++qr] = *p] = 1;

}
```

Apoi, incercati sa vedeti diferenta de timp intre cele doua programe... impresionant, nu?

Numere mari

In continuare voi prezenta cum se pot realiza operatii pe numere mari cu foarte putine linii de cod. In general, multi programatori se complica la aceste operatii, desi nu este nevoie! Vom considera ca numerele mari sunt vectori in care elementul de indice 0 indica lungimea numarului, iar cifrele sunt retinute in ordinea inversa decat cea a citirii.

Suma a doua numere mari

```
void add(int A[], int B[])
{
	int i, t = 0;
	for (i=1; i<=A[0] || i<=B[0] || t; i++, t/=10)
		A[i] = (t += A[i] + B[i]) % 10;
	A[0] = i - 1;
}
```

Inmultirea unui numar mare cu un numar mic

```
void mul(int A[], int B)
{
	int i, t = 0;
	for (i = 1; i <= A[0] || t; i++, t /= 10)
		A[i] = (t += A[i] * B) % 10;
	A[0] = i - 1;
```

```
}
```

Inmultirea unui numar mare cu un numar mare

```
void mul(int A[], int B[])
{
	int i, j, t, C[NR_CIFRE];
	memset(C, 0, sizeof(C));
	for (i = 1; i <= A[0]; i++)
	{
		for (t=0, j=1; j <= B[0] || t; j++, t/=10)
			C[i+j-1]=(t+=C[i+j-1]+A[i]*B[j])%10;
		if (i + j - 2 > C[0]) C[0] = i + j - 2;
	}
	memcpy(A, C, sizeof(C));
}
```

Scaderea a doua numere mari

```
void sub(int A[], int B[])
{
	int i, t = 0;
	for (i = 1; i <= A[0]; i++) {
		A[i] -= ((i <= B[0]) ? B[i] : 0) + t;
		A[i] += (t = A[i] < 0) * 10;
	}
	for (; A[0] > 1 && !A[A[0]]; A[0]--);
}
```

Impartirea unui numar mare la un numar mic

```
void div(int A[], int B)
{
	int i, t = 0;
	for (i = A[0]; i > 0; i--, t %= B)
		A[i] = (t = t * 10 + A[i]) / B;
	for (; A[0] > 1 && !A[A[0]]; A[0]--);
}
```

Restul unui numar mare la un numar mic

```
int mod(int A[], int B)
{
	int i, t = 0;
	for (i = A[0]; i > 0; i--)
		t = (t * 10 + A[i]) % B;
	return t;
}
```

AVL-uri (implementarea lui Radu Berinde)

AVL-urile sunt arbori de cautare echilibrati care au complexitate O(lg n) pe operatiile de inserare, stergere si cautare. Pentru mai multe detalii cautati cartea "Arbori" pe site-ul doamnei profesoare Emanuela Cerchez. In continuare voi prezenta o metoda destul de simpla de a implementa aceastra structura de date in timp de concurs. Enjoy!

```
#define max(a, b) ((a) > (b) ? (a) : (b))
#define geth(n) (n->h = 1 + max(n->l->h, n->r->h))

struct node
{
	int key, h;
	struct node *l, *r;
```

```
} *R, *NIL;
typedef struct node node;

void init(void)
{
	R = NIL = (node *) malloc(sizeof(node));
	NIL->key = NIL->h = 0,
		NIL->l = NIL->r = NULL;
}

node* rotleft(node *n)
{
	node *t = n->l;

	n->l = t->r, t->r = n,
		geth(n), geth(t);
	return t;
}

node* rotright(node *n)
{
	node *t = n->r;

	n->r = t->l, t->l = n,
		geth(n), geth(t);
	return t;
}

node* balance(node *n)
{
	geth(n);
	if (n->l->h > n->r->h + 1)
	{
		if (n->l->r->h > n->l->l->h)
			n->l = rotright(n->l);
		n = rotleft(n);
	}
	else
		if (n->r->h > n->l->h + 1)
		{
			if (n->r->l->h > n->r->r->h)
				n->r = rotleft(n->r);
			n = rotright(n);
		}
	return n;
}

node* insert(node *n, int key)
{
	if (n == NIL)
	{
		n = (node *) malloc(sizeof(node));
		n->key = key, n->h = 1, n->l = n->r = NIL;
		return n;
	}
	if (key < n->key)
		n->l = insert(n->l, key);
```

```
	else
		n->r = insert(n->r, key);
	return balance(n);
}

node* erase(node *n, int key)
{
	node *t;
	if (n == NIL) return n;
	if (n->key == key)
	{
		if (n->l == NIL || n->r == NIL)
		{
			t = n->l == NIL ? n->r : n->l;
			free(n); return t;
		}
		else
		{
			for (t = n->r; t->l != NIL; t = t->l);
			n->key = t->key,
				n->r = erase(n->r, t->key);
			return balance(n);
		}
	}
	if (key < n->key)
		n->l = erase(n->l, key);
	else
		n->r = erase(n->r, key);
	return balance(n);
}

int search(node *n, int key)
{
	if (n == NIL) return 0;
	if (n->key == key) return 1;
	if (key < n->key)
		return search(n->l, key);
	else
		return search(n->r, key);
}
```

Aici se termina acest articol. Am incercat sa pun accentul pe simplitate si eficienta, si cred ca am reusit acest lucru. Sper ca ati invatat cate ceva din el si recomand sa luati fiecare bucata in parte si sa incercati sa implementati efectiv ca sa intelegi mai bine. Bafta la concursuri tuturor!

12 ponturi pentru programatorii C/C++

(Categoria *Limbaje de programare*, Autor *Alexandru Mosoi*
http://www.infoarena.ro/12-ponturi-pentru-programatorii-cc
)

In urmatoarele cateva randuri am sa incerc sa va arat cateva metode de a scrie.... mai bine. Cea mai mare parte este pentru programatorii C.

Inainte ati putea citi si "Documentation/CodingStyle" aflat in sursa de kernel a Linux-ului. Scuzati-ma daca ma inspir putin. Manualul gcc este si el binevenit.

Pont #0

Prefer C in loc de C++: e mai robust putin ceea ce ma fereste cateodata de greseli. Incercati sa nu folositi un IDE cu debugging inclus (cum ar fi RHIDE sau Borland C++ 3.1). La inceput o sa va vina greu, dar va obisnuiti... si deveniti mai atenti cand scrieti surse. Puteti sa folositi Kate, un editor de text asemanator lui EditPlus de sub Windows. Vim este deasemenea un editor foarte puternic, dar pentru cine stie sa-l foloseasca.

Pont #1

Impartiti programul dumneavoastra in functii, fiecare sa nu depaseasca mai mult de 30-50 de linii (aproximativ 2 ecrane ANSI 80×25). Este important sa aveti mereu o viziune asupra intregii functii. Regula este: complexitatea unei functii trebuie sa fie invers proportionala cu lungimea ei. Puteti sa declarati functiile "inline" (nu pe toate !) pentru a nu pierde din viteza.

Pont #2

Macro-urile nu le recomand. Daca le folositi ca functii aveti grija. Unul dintre colegii mei de la lot a pierdut multe puncte pentru ceva asemanator.

?

```
1   #define MAX(a, b) ((a) < (b) ? (b) : (a))
2
3   int query(int a, int b)
4   {
5       int k, l, r;
6       ...
7       res = MAX(k, query(l, r));
8       ....
9       return res;
10  }
```

Daca observati, exista cazuri cand `query(l, r)` era apelata de `2` ori, ceea ce nu se doreste. In schimb, putea sa declare `MAX` ca o functie inline.

?

```
1   inline int MAX(int a, int b)
```

```
2  {
3      if(a > b)
4          return a;
5      return b;
6  }
```

Pont #3

Cand accesati un element din memorie, procesorul citeste de fapt 32 bytes (sau cat de mare e linia de cache, dar o putere a lui 2). Recomand ca structurile voastre sa aiba de asemenea ca dimensiune o putere a lui 2 pentru a nu forta procesorul sa citeasca de 2 ori. O extensie GNU a standardului ANSI C sunt atributele. Pentru structuri, una din cele mai folosite (de mine) este packed ce instruieste compilatorul se nu mai adauge "padding bytes".

?

```
1  struct foo { int a; char b; int c; };
2  /* sizeof(struct foo) == 12 */
3  struct bla { int a; char b; int c; }
4
5  __attribute__((packed));
6  /* sizeof(struct bla) == 9 */
```

Update

Se pare ca pentru ultimele versiuni ale compilatorului aceasta recomandare nu mai este valabila. Structurile de date de dimensiuni 2^k au tendinta de a incetini executia programului.
Sursa 1
Sursa 2

Pentru mai multe informatii executati consultati manualul gcc. ("info gcc").

De asemenea, e bine sa nu spargeti aceasta line de cache prea des. Uitati un exemplu:

?

```
1   #define maxN 1000
2   #define maxM 1000
3
4   int t[maxN][maxM];
5
6   int f(void)
7   {
8       int i, j;
9       int s = 0;
10      for(i = 0; i < maxM; ++ i)
11          for(j = 0; j < maxN; ++ j)
```

```
12                s += t[j][i];
13        return s;
14    }
```

Pentru `1024` de apelari, pe calculatorul meu, acesta functie consuma cam `18.85`s. In schimb, daca as fi scris

?

```
1   for(i = 0; i < maxN; ++ i)
2       for(j = 0; j < maxM; ++ j)
3           s += t[i][j];
```

... functia s-ar fi executat de ~`3` ori mai repede (doar `6.05`s) iar rezultatul era acelasi. De ce? pentru ca in primul caz la fiecare accesare a `t[j][i]` procesorul era nevoit sa acceseze memoria, iar in cazul al doilea cand citea `t[i][j]`, erau citite de fapt si `t[i][j+1]`, `t[i][j+2]`, `t[i][j+3]`. Si sa nu uitam viteza memoriei este mult mai mica decat cea a procesorului.

Pont #4

Variabilele globale sa nu fie folosite in scop local. Daca as modifica functia astfel

?

```
1    int i, j;
2
3    int f(void)
4    {
5        int s = 0;
6        for(i = 0; i < maxM; ++ i)
7            for(j = 0; j < maxN; ++ j)
8                s += t[i][j];
9        return s;
10   }
```

... timpul de executie s-ar fi marit la `6.44`s. Nu e prea mult... dar se aduna.

Pont #5

Stack-ul (locul unde se pastreaza toate variabilele locale) este foarte rapid. Modificam acelasi program astfel:

?

```
1    #define maxN 1000
2    #define maxM 1000
3
4    int main(void)
5    {
```

```
6       int i, j, k;
7       int N, M, t[maxN][maxM];
8       N = maxN; M = maxM;
9       for(k = 0; k < 1024; ++ k) {
10          int s;
11          for(i = 0; i < N; ++ i)
12              for(j = 0; j < M; ++ j)
13                  s += t[i][j];
14      }
15      return 0;
16  }
```

Ignorand faptul ca `t` nu este initializat (e doar un program de test, nici inainte nu era :D) timpul de executie scade la `1.2`s, Wow! Insa aveti grija sa nu o luati pe urmele lui Silviu: `sizeof(t) ~= 4`Mb care e mult peste limita de `1`Mb ce se impune de obicei in concursuri (si asta daca folositi gcc). Cel mai probabil veti primi "Killed by signal 11".

Pont #6

6.a

`++i` e preferabil in locul lui `i ++` (unde nu complica lucrurile).

6.b

Nu va feriti sa folositi "`const`" si "`static`". "`const`" chiar poate sa faca diferenta ca timp si vizibilitate.

6.c

Utilizati si literele mari pentru anumite variabile mai importante (poate si macro-uri).

Pont #7

O alta extensie GNU sunt "zero-length arrays". Se folosesc in general la skiplist-uri pentru a declara un array de dimensiune variabila intr-o structura.

[?]

```
1  typedef struct bla bla;
2  struct bla {
3      int levels;
4      bla *next[0];
5  };
6  ...
7  bla *temp = (bla *)malloc(sizeof(bla) + no_levels*sizeof(bla *));
```

Pont #8

8.a

Folositi-va de utilitarele puse la dispozitie de sistemul de operare (linux in cazul meu). RTFM :)

- `bc` - pentru calcule cu numere cu precizie multipla (eg. 2^{1024}).

- `octave` - pentru calcule matematice mai complicate.
- `gprof` - determina cat timp a necesitat executia fiecarei functii sau linii.
- `gcov` - determina de cateori a fost apelata o anumita linie.
- `time` - pentru aflarea timpului executiei unui program.
- `factor` - descompune in factori un numar (eg. factor `666`).
- `splint` - o versiune free a programului lint: va da foarte multe warning-uri.
- `bash` - putin scripting

8.b

Compilati-va sursele cu `-W -Wall` (tot pentru warning-uri)

8.c

Generatorul de teste si sursa dumneavoastra trebuie sa fie doua programe diferite !

8.d

Pentru debugging folositi `fprintf(stderr, ...)`. Daca se intampla sa uitati, macar nu primiti "wrong answer" din cauza unui `printf`.

Pont #9

9.a

?

```
1  int t[666];
2  /* toate elementele lui t vor fi -1 */
3  memset(t, 0xff, sizeof(t));
```

9.b

Pentru valoarea infinit folosesc o constanta

?

```
1  #define INFI 0x3f3f3f3f
```

din mai multe motive:

- `INFI + INFI` ramane pozitiv
- in general e destul de mare

?

```
1  /* toate elementele lui t devin INFI */
2  memset(t, 0x3f, sizeof(t));
```

9.c

Daca avem de comparat doua siruri (s_1, s_2) a caror lungime o stim (len_s_1, respectiv len_s_2) este mai rapid

?

```
1  memcmp(s1, s2, MIN(len_s1, len_s2)+1)
```

decat

?

```
1   strcmp(s1, s2);
```

9.d

?

```
1   scanf(" %c", &ch)
```

citeste primul caracter dupa spatiile albe (daca exista).

Pont #10

Daca programati in C++ fara sa folositi STL incercati sa renuntati la C++. Unul dintre motive: clasele (implicit iostream: cin, cout, cerr) incetinesc mult executia programului.

Pont #11

In final, o intrebare pentru cei ce folosesc C++ (asta e un hint). Cum se calculeaza factorial la compilare? (fara a scrie efectiv `1*2*3...*n`)
Raspuns: Utilizand templaturi. Avem nevoie doar de o constanta `N`.

?

```
1   #include <stdio.h>
2
3   template<int N>
4   struct Factorial {
5       enum {
6           value = Factorial<N-1>::value * N
7       };
8   };
9
10  template<>
11  struct Factorial<0> {
12      enum { value = 1 };
13  };
14
15  int main(void)
16  {
17      int i = Factorial<4>::value;
18      char c[Factorial<5>::value];
19      printf("%d ",i);
20      printf("%d ",sizeof(c));
21  }
```

Continut

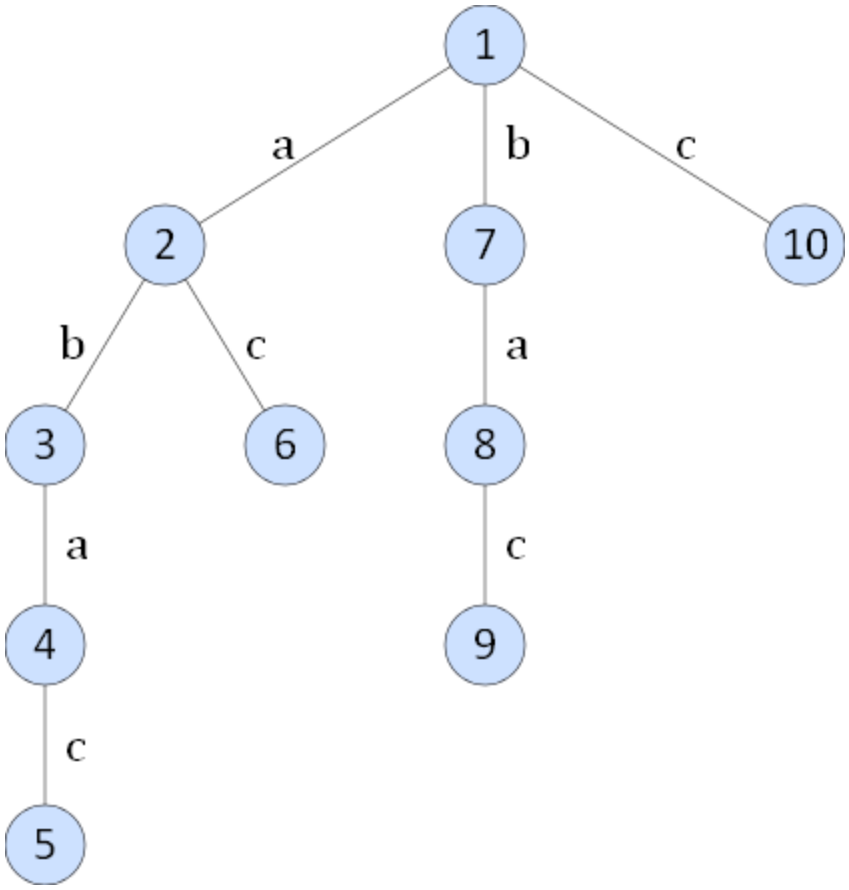
(http://www.infoarena.ro/siruri-de-sufixe)

Introducere

Un domeniu important in algoritmica folosita in practica este acela al algoritmilor pe siruri de caractere. Astfel, la concursurile de programare sunt prezente foarte multe probleme de prelucrare si procesare a unor siruri de caractere. In cadrul concursurilor si antrenamentelor multi dintre noi s-au lovit de probleme ce s-ar fi rezolvat usor daca se reusea in mod eficient determinarea existentei unui cuvant ca subsecventa a unui alt cuvant. Vom prezenta o structura versatila ce permite acest lucru, inlesnind de multe ori realizarea altor operatii utile pe un string dat.

Ce sunt sirurile de sufixe (suffix arrays)?

Pentru a avea o idee mai buna despre *suffix arrays*, vom face inainte o scurta prezentare a structurii de date numita in engleza *trie* si a *arborilor de sufixe* (suffix trees[1]) care sunt o forma speciala a structurii de date trie. Un trie este un arbore menit sa stocheze siruri. Fiecare nod al lui va avea in general un numar de fii egal cu marimea alfabetului sirurilor de caractere care trebuies stocate. In cazul nostru, cu siruri ce contin litere mici ale alfabetului englez, fiecare nod va avea cel mult 26 de fii. Fiecare muchie porneste din tata spre fii si va fi etichetata cu o litera distincta a alfabetului. Etichetele legaturilor de pe un drum de la radacina pana la o frunza vor alcatui un cuvant stocat in arbore. Dupa cum se observa, verificarea existentei unui cuvant in aceasta structura de date este foarte eficienta si se realizeaza in complexitate $O(M)$, unde M e lungimea cuvantului. Astfel, timpul de cautare nu depinde de numarul de cuvinte pe care trebuie sa il gestioneze structura de date, fapt ce face aceasta structura ideala pentru implementarea dictionarelor.
Sa vedem acum ce este un *trie de sufixe*. Dat fiind un string $A = a_0a_1...a_{n-1}$, notam cu $A_i = a_ia_{i+1}...a_{n-1}$ sufixul lui A care incepe la pozitia i. Fie n = lungimea lui A. Trie-ul de sufixe este format prin comprimarea tuturor sufixelor $A_1...A_{n-1}$ intr-un trie, ca in figura de mai jos. Trie-ul de sufixe corespunzator stringului $abac$ este:



Operatiile pe aceasta structura se realizeaza extrem de usor:

- *verificarea daca un string* `W` *este substring al lui* `A` - este suficienta parcurgerea nodurilor, incepand din radacina si urmarind muchiile etichetate corespunzator caracterelor din `W` (complexitate `O(|W|)`)
- *cautarea celui mai lung prefix comun pentru doua sufixe ale lui* `A` - se aleg nodurile `u` si `v` ale trie-ului corespunzatoare sfarsitului celor doua sufixe, iar prin aplicarea unui algoritm de gasire a LCA (Lowest Common Ancestor / cel mai apropiat stramos comun) se gaseste nodul corespunzator sfarsitului prefixului cautat. De exemplu, pentru `abac` si `ac` se gasesc nodurile `5` si `6`. Cel mai apropiat stramos comun al lor este `2`, de unde rezulta prefixul `a`. Autorii va recomanda articolul [2] pentru o rezolvare in `O(sqrt(n))`, [3] pentru o prezentare accesibila a unei rezolvari in `O(log n)` sau `O(1)`, si articolul [4] pentru un algoritm "state of the art".
- *gasirea celui de-al* `k`*-lea sufix in ordine lexicografica* - (complexitate `O(n)`, cu o preprocesare corespunzatoare). De exemplu al `3`-lea sufix al sirului `abac` este reprezentat in trie-ul nostru de a `3`-a frunza.

Desi ideea unui trie de sufixe este incantatoare la prima vedere, implementarea simplista in care inseram pas cu pas sufixele in structura noastra necesita un timp de ordinul $O(n^2)$. Exista o structura numita *arbore de sufixe*[1] care se poate construi in timp liniar fata de lungimea sirului de caractere. Arborele de sufixe este un trie de sufixe in care lanturile din care nu ieseau alte muchii erau comprimate intr-o singura muchie (in exemplul de mai sus acestea ar fi lanturile `2-3-4-5` si `1-7-8-9`algoritmului de complexitate liniara pentru construirea unui arbore de sufixe este anevoioasa, fapt care ne determina sa cautam o alta structura, mai usor de realizat.). Dar implementarea

Sa vedem care sunt sufixele lui A, parcurgand arborele in adancime. Avand in vedere faptul ca la parcurgerea in adancime trebuie sa consideram nodurile in ordinea lexicografic crescatoare a muchiilor care le leaga de tata, obtinem urmatorul sir de sufixe:

abac	A_0
ac	A_2
bac	A_1
c	A_3

Este usor de observat ca acestea sunt ordonate crescator. Pentru memorare, nu este necesar sa pastram un vector ordonat de sufixe, suficienta fiind pastrarea indicilor fiecarui sufix din sirul ordonat. Pentru exemplul de mai sus obtinem vectorul `P = (0, 2, 1, 3)`, acesta fiind array-ul de sufixe pentru stringul `abac`.

Cum construim un sir de sufixe?

Prima metoda care ne vine in minte este sortarea tuturor sufixelor lui A folosind un algoritm de complexitate $O(n \lg n)$. Insa compararea a doua sufixe se face in timp $O(n)$, deci complexitatea finala va fi $O(n^2 \lg n)$. Exista totusi un algoritm relativ usor de implementat si inteles, avand o complexitate de $O(n \lg n)$. Desi este asimptotic mai mare decat cel al constructiei unui arbore de sufixe (suffix tree), in practica timpul de constructie al unui sir de sufixe este mult mai mic, din cauza constantei care apare in fata algoritmul liniar. De asemenea, cantitatea de memorie folosita in cazul implementarii cu memorie $O(n)$ este de la 3 pana la 5 ori mai mica decat in cazul unui arbore de sufixe.
Algoritmul se bazeaza pe mentinerea ordinii sufixelor sirului, sortate dupa prefixele lor de lungime 2^k. Astfel vom executa $m = [\log_2 n]$ (marginit superior) pasi, la pasul $k$stabilind ordinea sufixelor daca sunt luate in considerare doar primele 2^k caractere din fiecare sufix. Se foloseste o matrice P de dimensiune $m \times n$Notam cu . A_i^k subsecventa lui A de lungime 2^k ce incepe pe pozitia i. Pozitia lui A_i^k in sirul sortat al subsecventelor A_j^k $(j=0,n-1)$ se pastreaza in $P_{(k,i)}$.
Pentru a trece de la pasul k la pasul $k+1$ se concateneaza toate secventele A_i^k cu $A_{i+2^k}^k$, obtinandu-se astfel substringurile de lungime 2^{k+1}. Pentru stabilirea ordinii se folosesc informatiile obtinute la pasul anterior. Pentru fiecare indice i se pastreaza o pereche de intregi formata din $P_{(k,i)}$ si $P_{(k,i+2^k)}$. Nu trebuie sa ne preocupe faptul ca $i+2^k$ poate pica in afara sirului, deoarece vom completa sirul cu pseudocaracterul $, despre care vom considera ca este lexicografic mai mic decat oricare alt caracter.
In urma sortarii, perechile vor fi aranjate conform ordinii lexicografice a substringurilor de lungime 2^{k+1} corespunzatoare. Un ultim lucru care mai trebuie notat este ca la un anumit pas k, pot exista doua (sau mai multe) substringuri $A_i^k = A_j^k$, iar acestea trebuie etichetate identic ($P_{(k,i)}$ trebuie sa fie egal cu $P_{(k,j)}$). O imagine spune mai mult decat o mie de cuvinte:

b	o	b	o	c	e	l

Pasul 0:

0	4	0	4	1	2	3
b	o	b	o	c	e	l

Pasul 1:

0	4	0	5	1	2	3
b	o	b	o	c	e	l
o	b	o	c	e	l	$

Pasul 2:

0	5	1	6	2	3	4
b	o	b	o	c	e	l
o	b	o	c	e	l	$
b	o	c	e	l	$	$
o	c	e	l	$	$	$

Pasul 3:

0	5	1	6	2	3	4
b	o	b	o	c	e	l
o	b	o	c	e	l	$
b	o	c	e	l	$	$
o	c	e	l	$	$	$
c	e	l	$	$	$	$
e	l	$	$	$	$	$
l	$	$	$	$	$	$
$	$	$	$	$	$	$

Iata un pseudocod ce sugereaza pasii principali ce trebuie urmati:

```
1   n <- lungime(A)
2   pentru i <- 0, n-1
3       P(0, i) <- pozitia lui Ai in sirul ordonat al caracterelor lui A
    sfarsit pentru
4   cnt <- 1
5   pentru k <- 1, [log2 n] (marginit superior)
6       pentru i <- 0, n-1
7           L(i) <- (P(k-1, i), P(k-1, i+cnt), i)
8       sfarsit pentru
9       sorteaza L
10      calculeaza P(k, i), i = 0, n-1
        cnt <- 2 * cnt
```

```
sfarsit pentru
```

De remarcat ca nu este neaparat necesara o anumita numerotare a substringurilor, atat timp cat intre ele este pastrata o relatie de ordine valida. In vederea atingerii complexitatii $O(n \lg n)$ pentru sortare se recomanda folosirea metodei *radix sort* (de doua ori sortare prin numarare), aceasta avand complexitate $O(n)$. Insa, pentru usurarea implementarii, se poate folosi functia `sort()` din STL (Standard Template Library, o librarie ce contine unele structuri de date si algoritmi in limbajul C++). Desi complexitatea va creste la $O(n \lg^2 n)$ in cazul cel mai defavorabil, implementarea devine simtitor mai simpla, iar in practica diferentele sunt abia sesizabile pentru siruri cu lungime mai mica decat$100\ 000$.
Mai jos puteti vedea o implementare extrem de scurta pentru suffix array in $O(n \lg^2 n)$.

?

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const int MAXN = 65536;
const int MAXLG = 17;

char A[MAXN];
struct entry {
    int nr[2], p;
} L[MAXN];
int P[MAXLG][MAXN], N, i, stp, cnt;

bool cmp(const entry &a, const entry &b) {
    return a.nr[0] == b.nr[0] ? (a.nr[1] < b.nr[1]) : (a.nr[0] < b.nr[0]);
}

int main() {
    gets(A);
    for (N = strlen(A), i = 0; i < N; ++i)
        P[0][i] = A[i] - 'a';
    for (stp = 1, cnt = 1; cnt >> 1 < N; ++stp, cnt <<= 1) {
        for (i = 0; i < N; ++i) {
            L[i].nr[0] = P[stp - 1][i];
            L[i].nr[1] = i + cnt < N ? P[stp - 1][i + cnt] : -1;
            L[i].p = i;
        }
        sort(L, L + N, cmp);
        for (i = 0; i < N; ++i)
            P[stp][L[i].p] = i > 0 && L[i].nr[0] == L[i - 1].nr[0] && L[i].nr[1] == L[
    }
    return 0;
}
```

Sirul de sufixe se va gasi pe ultima linie a matricei `P`. Cautarea celui de-al `k`-lea sufix in ordine lexicografica este acum imediata, deci nu vom reveni asupra acestui aspect.
Cantitatea de memorie folosita poate fi redusa renuntand la folosirea intregii matrice `P` si pastrindu-se la fiecare pas doar ultimele doua linii ale acesteia. In acest caz, insa, structura nu va mai fi capabila sa execute eficient operatia ce urmeaza.

Calcularea celui mai lung prefix comun (LCP)

Se dau doua sufixe ale unui string `A`. Se cere calcularea celui mai lung prefix comun al lor. Am aratat ca un arbore de sufixe poate realiza aceasta in timp `O(1)` cu o preprocesare corespunzatoare. Sa vedem daca un sir de sufixe poate atinge aceeasi performanta.
Fie cele doua sufixe A_i si A_j. Folosind matricea `P`, putem itera descrescator de la cel mai mare `k` pana la `0` si verifica daca $A_i^k = A_j^k$. Daca cele doua prefixe sunt egale, am gasit un prefix comun de lungime 2^k. Nu ne ramane decat sa actualizam `i` si `j`, incrementandu-le cu 2^k si sa verificam in continuare daca mai gasim prefixe comune. Codul functiei care calculeaza LCP este foarte simplu:

```
?
1
2
3
4
5
6
7
8
```

```
int lcp(int x, int y) {
    int k, ret = 0;
    if (x == y) return N - x;
    for (k = stp - 1; k >= 0 && x < N && y < N; --k)
        if (P[k][x] == P[k][y])
            x += 1 << k, y += 1 << k, ret += 1 << k;
    return ret;
}
```

Complexitatea este insa `O(lg n)` pentru un calcul al acestui prefix. Reducerea la `O(1)` se bazeaza pe urmatoarea observatie: `lcp(x, y)` = `min{ lcp(x, x + 1), lcp(x + 1, x + 2), ..., lcp(y - 1, y) }`. Demonstratia este imediata daca ne uitam in arborele de sufixe corespunzator. Asadar, este suficient ca la inceput sa calculam cel mai lung prefix comun intre toate perechile de sufixe consecutive (timp `O(n lg n)`) si sa introducem o structura aditionala ce permite calculul in `O(1)` al minimului dintr-un interval. Cea mai eficienta astfel de structura este cea pentru RMQ (range minimum query), despre care nu vom da detalii aici, dar care este studiata in amanunt in [3], [4] si [5]. Cu inca o preprocesare in `O(n lg n)` ceruta de noua structura putem acum sa raspundem in `O(1)` query-urilor LCP. Structura folosita de RMQ cere tot `O(n lg n)` memorie, asadar timpul si memoria finale necesare sunt `O(n lg n)`.

Cautarea

Deoarece sirul de sufixe ne ofera ordinea sufixelor lui `A`, cautarea unui string `W` in `A` se poate face simplu cu o cautare binara. Deoarece compararea se face in `O(|W|)`, cautarea va avea complexitatea `O(|W| lg n)`. Lucrarea [6] ofera structurii de date si algoritmului de cautare cateva rafinamente ce permit reducerea timpului la `O(|W| + lg n)`, dar autorii nu considera ca acestea sunt folositoare in concursurile de programare.

Probleme de concurs

Autorii au incercat sa adune cat mai multe probleme ce pot fi rezolvate cu ajutorul sirurilor de sufixe. Parcurgerea tuturor problemelor la prima citire, ar putea fi greoaie pentru un cititor care a avut primul contact cu aceasta structura de date citind acest articol. Pentru a usura lectura problemele sunt asezate intr-o ordine crescatoare a dificultatilor.

Problema 1: *Parola ascunsa* (acm 2003, enunt modificat)

Consideram un sir de caractere de lungime $n\ (1 \le n \le 100000)$. Sa se determine rotatia lui circulara lexicografic minima. De exemplu, rotatiile sirului de caractere alabala sunt:

alabala
labalaa
abalaal
balaala
alaalab
laalaba
aalabal

iar cea mai mica dintre ele in ordine lexicografica este aalabal.

Solutie:

De obicei, in probleme unde apare rotatia unui sir dat, putem sa ne folosim de trucul de a concatena sirului de caractere acelasi sir pentru a simplifica problema. Dupa aceasta transformare problema ne cere subsecventa lexicografica minima de lungime n a noului sir. Ordinea subsecventelor de lungime n ale unui sir este egala cu ordinea sufixelor determinate de subsecventele date, deci ne putem folosi de siruri de sufixe pentru a rezolva aceasta problema.

O problema asemanatoare a fost propusa de unul dintre autori la concursul Bursele Agora, editia 2003-2004, care se putea rezolva pe aceiasi idee. Implementarea trebuia facuta atent pentru solutia oficiala era o rezolvare eficienta ce nu folosea aceasta structura de date.

Problema 2: *Sir* (baraj 2004)

Se considera un sir $c_1c_2...c_n$ format din $n\ (1 \le n \le 30\ 000)$ caractere din multimea {A, B}. Concatenam sirul cu el insusi si obtinem un sir de lungime $2n$. Pentru un indice $k\ (1 \le k \le 2n)$ consideram subsecventele de lungime cel mult n, care se termina pe pozitia k, iar dintre acestea fie $s(k)$ subsecventa cea mai mica in ordine lexicografica. Determinati indicele k pentru care $s(k)$ are lungimea cea mai mare.

Nota suplimentara: fie X si Y doua siruri oarecare, iar "$\circ$" operatia de concatenare. In aceasta problema se va considera ca $X > X \circ Y$.

Solutie:

Sirul cautat este prima permutare circulara in ordine lexicografica a sirului dat. Notam cu S_i^k substringul de lungime k ce incepe la pozitia i. Fie S_i^n cel mai mic substring in ordine lexicografica de lungime n Presupunand prin absurd ca al sirului obtinut prin concatenare. $s(i+n-1) < n$ ar insemna ca exista un $i'\ (i < i' \le j)$ astfel incat $S_i^{j-i'+1}$ este lexicografic mai mic decat S_i^n. Dar din conditia impusa de enunt avem $S_i^{j-i'+1} > S_{i'}^n$. Dar $S_{i'}^n > S_i^n$ => contradictie.

Desi exista un algorirm de complexitate $O(n)$ specializat pentru siruri ce contin doar literele A si B, metoda preferata de autor (si cu care a obtinut punctaj maxim in timpul concursului) a fost folosirea sirurilor de sufixe, ca in problema anterioara.

Problema 3: *Substr* (baraj 2003)

Se da un text format din N caractere (litere mari, litere mici si cifre). Un substring al acestui text este o secventa de caractere care apar pe pozitii consecutive in text. Fiind dat un numar K, sa se gaseasca lungimea celui mai lung substring care apare in text de cel putin K ori $(1 \le N \le 16384)$.

Solutie:

Avand sufixele textului sortate, iteram cu o variabila i de la 0 la $N-K$ si calculam cel mai lung prefix comun intre sufixul i si sufixul $i+K-1$. Prefixul maxim determinat in cursul acestei parcurgeri reprezinta solutia problemei.

Problema 4: *Ghicit* (baraj 2003)

Tu si cu Taranul jucati un joc neinteresant. Tu ai un sir de caractere mare. Taranul iti spune un alt sir de caractere, iar tu trebuie sa raspunzi cat mai repede daca sirul respectiv este sau nu o subsecventa a sirului tau.
iti pune multe intrebari si, fiindca esti informatician, te-ai gandit ca ar merge mai repede daca ai sti dinainte toate sirurile despre care te poate intreba. Taranul
Inainte de a face toata acesta munca te-ar interesa numarul total de subsecvente distincte ale sirului tau, ca sa stii daca are sens sa te apuci de acesta treaba sau nu.
Scrieti un program care afla numarul de subsecvente distincte ale unui sir de caractere dat. `(1 ≤ lungimea sirului ≤ 10 000)`

Solutie:

Aceasta problema ne cere, de fapt, sa calculam numarul de noduri (fara radacina) ale trie-ului de sufixe asociat unui string. Fiecare secventa distincta din sir este determinata de drumul unic pe care il parcurgem in trie-ul de sufixe cand cautam acea secventa. Pentru exemplul `abac` avem secventele `a`, `ab`, `aba`, `abac`, `ac`, `b`, `ba`, `bac` si `c`, acestea sunt determinate de drumul de la radacina trieului spre nodurile `2`, `3`, `4`, `5`, `6`, `7`, `8` si `9` in aceasta ordine. Cum constructia trie-ului de sufixe are complexitate patratica, iar construirea unui arbore de sufixe este anevoioasa, este preferabila o abordare prin prisma sirurilor de sufixe. Obtinem sirul sortat de sufixe in `O(n lg n)`, dupa care cautam pozitia in care fiecare pereche de sufixe consecutive difera (folosind functia `lcp`) si adunam la solutie restul caracterelor. Complexitatea totala este `O(n lg n)`.

Problema 5: *SETI* (ONI 2002, enunt modificat)

Se da un string de lungime `N (1 ≤ N ≤ 131072)` si `M` stringuri de lungime cel mult `64`. Se cere sa se numere aparitiile fiecarui string din cele `M` in stringul mare.

Solutie:

Se procedeaza la fel ca in cazul sirurilor de sufixe, numai ca este suficient sa ne oprim dupa pasul 6, unde am calculat relatia de ordine intre stringurile de lungime $2^6 = 64$. Avand substringurile de lungime 64 sortate, fiecare query este rezolvat cu doua cautari binare. Complexitatea algoritmului este `O(N lg 64 + M * 64 * lg N) = O(N + M lg N)`.

Problema 6: *Subsecventa comuna* (Olimpiada poloneza si TopCoder 2004, enunt modificat)

Se considera trei siruri de caractere S_1, S_2 si S_3, de lungimi `m`, `n` si `p (1 ≤ m, n, p ≤ 10000)`. Sa se determine subsecventa de lungime maxima care este comuna celor trei siruri. De exemplu, daca S_1 = `abababca`, S_2 = `aababc` si S_3 = `aaababca`, atunci subsecventa comuna de lungime maxima pentru cele trei siruri este `ababc`.

Solutie:

Daca ar fi vorba doar de doua siruri de lungimi mai mici am putea rezolva usor problema folosind metoda programarii dinamice; astfel, solutia pentru doua siruri ar avea ordinul de complexitate $O(N^2)$.
O alta idee ar fi sa consideram fiecare sufix al sirului S_1 si sa incercam sa ii gasim potrivirea de lungime maxima in celelalte doua siruri.
Potrivirea de lungime maxima rezolvata naiv ar avea complexitatea $O(N^2)$, dar folosind algoritmul KMP[8], putem obtine prefixul maxim al unui sir care se gaseste ca subsecventa in al doilea sir in $O(N)$, iar utilizand aceasta metoda pentru fiecare sufix al lui S_1, am avea o solutie al carei ordin de complexitate este $O(N^2)$.
Sa vedem ce se intampla daca sortam sufixele celor trei siruri:

ababababca	aababc	aaababca
a§ abababca§ ababca§ abca§ bababca§ babca§ bca§ ca§	aababc# ababc# abc# babc# bc# c#	a@ aaababca@ aababca@ ababca@ abca@ babca@ bca@ ca@

Acum interclasam prefixele celor trei siruri (consideram `§ < # < @ < a ...`):

a§
a@
aaababca@
aababc#
aababca@
abababca§
ababc#
ababca§
ababca@
abc#
abca§
abca@
bababca§
babc#
babca§
babca@
bc#
bca§
bca@
c#
ca§
ca@

Subsecventa comuna maxima corespunde prefixelor comune maxime pentru cele trei sufixe `ababca§`, `ababc#` si `ababca@`. Urmariti unde apar ele in sirul sortat al tuturor sufixelor. De aici avem ideea ca solutia se afla ca o secventa `[i..j]` a sirului sortat de sufixe cu proprietatea ca secventa contine cel putin cate un sufix din fiecare sir, iar prefixul cel mai lung comun primului sufix din secventa si ultimul sufix din secventa este maxim; acest cel mai lung prefix este chiar solutia problemei. Alte subsecvente comune ale celor trei siruri ar fi prefixe comune pentru cate o subsecventa a sirului de sufixe sortat, de exemplu `bab` pentru `bababca§`, `babc@`, `babca§`, sau `a` pentru `a§`, `a@`, `aaababca@`, `aababc#`. Pentru a determina aceasta secventa de prefix comun maxim putem folosi o parcurgere cu doi indici (`start` si `end`). Indicele `start` variaza intre `1` si numarul de sufixe, iar `end` este cel mai mic indice mai mare decat `start` astfel incat intre `start` si `end` sa

existe sufixe din toate cele trei siruri. Astfel, perechea `[start, end]` va indica, la un moment dat, secventa optima `[i..j]`. Aceasta parcurgere este liniara, deoarece `start` poate avea cel mult `n` valori, iar `end` va fi incrementat de cel mult `n` ori. Pentru a sorta sirul tuturor sufixelor nu este nevoie sa sortam mai intai sufixele fiecarui sir si apoi sa interclasam sufixele. Putem realiza operatia mult mai simplu concatenand cele trei siruri in unul singur (pentru exemplul considerat avem `abababca§aababc@aaababca#)` si sortand sufixele acestuia.

Problema 7: *Cel mai lung palindrom* (USACO Training Gate)

Se considera un sir de caractere `S` de lungime `n (1 ≤ n ≤ 20000)`. Determinati subsecventa de lungime maxima care este si palindrom (un sir de caractere este palindrom daca este identic cu sirul obtinut prin oglindirea sa).

Solutie:

Daca dorim sa determinam, pentru un indice fixat `i`, care este cel mai mare palindrom centrat in `i` atunci ne intereseaza prefixul maxim al subsecventei `S[i+1..n]` care se potriveste cu prefixul subsecventei `S[1..i]` reflectate. Pentru a rezolva cu usurinta aceasta problema sortam impreuna si sufixele sirului cu prefixele reflectate ale sirului (operatie care se realizeaza usor concatenand sirul `S§` cu sirul `S` oglindit, `S'`) si vom efectua interogari pentru cel mai lung prefix comun pentru `S[i+1]` si `S'[n-i+1] (S'[n-i+1] = S[1..i])`, la care putem raspunde folosind siruri de sufixe in timp `O(1)`. Astfel, putem rezolva problema in timp `O(N log N)`. Sa observam ca am tratat aici doar cazul in care palindromul este de lungime para, dar cazul in care palindromul are lungime impara se trateaza analog.

Problema 8: *Template* (Olimpiada poloneza 2004, enunt modificat)

Pentru un string `A`, sa se determine lungimea minima a unui substring `B` cu proprietatea ca `A` poate fi obtinut prin lipirea intre ele a mai multor stringuri `B` (la lipire doua stringuri se pot suprapune, dar in locurile in care se suprapun caracterele celor doua stringuri trebuie sa coincida).

Exemplu

Pentru string-ul `ababbababbababbababbababbababbaba` rezultatul este `8`. String-ul `B` de lungime minima este `ababbaba`. `A` poate fi obtinut din `B` astfel:

```
ababbababbababbababbababbababbaba
ababbaba
.....ababbaba
............ababbaba
...................ababbaba
.......................ababbaba
```

Solutia 1:

O solutie simpla foloseste siruri de sufixe, un arbore echilibrat si un `max-heap` (se pot folosi structurile `set` si `priority_queue` din `STL`). Este evident ca sablonul cautat este un prefix al lui `A`. Asadar, pentru fiecare prefix `B` al lui `A` vom verifica daca prin lipirea tuturor aparitiilor lui `B` in `A` se obtine chiar cuvantul initial `A`. Pentru a face aceasta verificare este suficient calculul distantei dintre perechile de potriviri consecutive ale lui `B`. Trebuie sa avem grija ca prefixele sa acopere si sfarsitul sirului. Pentru aceasta, cel mai comod este sa mai consideram o aparitie a lui `B` pe pozitia `n+1`. Daca distanta maxima gasita este mai mare decat lungimea lui `B`, acel prefix nu reprezinta o solutie. Pentru o rezolvare eficienta, vom considera initial `B` ca fiind prefixul de lungime `1`, urmand sa introducem la sfarsitul sau, caracter cu caracter, restul caracterelor string-ului `A`. Daca la fiecare pas mentinem multimea `S` a pozitiilor de inceput ale potrivirilor lui `B` in `A`, dupa introducerea unui nou caracter in `B`, multimea `S` va "pierde" anumite elemente (posibil niciunul). Pentru administrare eficienta, vom considera sirul sortat de sufixe ale lui `A` si doi pointeri, `L` si `R`, reprezentand primul, respectiv ultimul sufix din sir care il au ca prefix pe `B`. La adaugarea unui nou caracter in `B`, se incrementeaza, respectiv

decrementeaza `L` si `R` cat timp acestea nu indica sufixe care il au ca prefix pe noul `B`. Arborele echilibrat va contine tot timpul pozitiile de inceput ale sufixelor continute intre `L` si `R`, iar `heap-ul` va contine distantele intre elemente consecutive ale arborelui. La inserarea unui nou caracter in `B`, trebuie sa avem grija de intretinerea acestor structuri. Algoritmul se incheie atunci cand cel mai mare (primul) element din `heap` este mai mic sau egal cu lungimea lui `B`. Lungimea finala a lui `B` ne ofera rezultatul cautat. Ordinul de complexitate este `O(N lg N)`, unde `N` este lungimea lui `A`. Sa consideram un exemplu. `S` este marcat cu `1` in pozitiile in care s-a gasit o potrivire a lui `B` in `A`:

```
1: aab
2: aabaab
3: ab
4: abaab
5: abaabaab
6: b
7: baab
8: baabaab
```

B	A	a	b	a	a	b	a	a	b	§	L	R	distanta maxima
	S.S.	5	8	2	4	7	1	3	6				
a		1	0	1	1	0	1	1	0	1	1	5	2
ab		1	0	0	1	0	0	1	0	1	3	5	3
aba	S	1	0	0	1	0	0	0	0	1	4	5	5
abaa		1	0	0	1	0	0	0	0	1	4	5	5
abaab		1	0	0	1	0	0	0	0	1	4	5	5

Solutia 2 (Mircea Pasoi):

Pentru sirul de caractere `S`, determinam pentru fiecare `i` de la `1` la `n` lungimea celui mai lung prefix al lui `S` cu `S[i..n]`. Aceasta operatie se poate realiza folosind siruri de sufixe. De exemplu, daca `S` este sirul nostru si `T` este sirul de potriviri maxime ale sufixelor, atunci:

S	a	b	b	a	a	b	b	a	a
T	9	0	0	1	5	0	0	1	1

Pentru toate lungimile posibile `k` ale sablonului `(1 ≤ k ≤ n)` verificam daca distanta maxima `d` intre indicii celor mai departate doua elemente de valori mai mari sau egale cu `k` in sirul `T` nu este mai mare decat `k`. Prezentam in continuare un exemplu:

k	S	a	b	b	a	a	b	b	a	a	d	
	T	9	0	0	1	5	0	0	1	1		
9		9	–	–	–	–	–	–	–	–	9	✔
8		9	–	–	–	–	–	–	–	–	9	✘
7		9	–	–	–	–	–	–	–	–	9	✘
6		9	–	–	–	–	–	–	–	–	9	✘
5		9	–	–	–	5	–	–	–	–	5	✔
4		9	–	–	–	5	–	–	–	–	5	✘
3		9	–	–	–	5	–	–	–	–	5	✘
2		9	–	–	–	5	–	–	–	–	5	✘
1		9	–	–	1	5	–	–	1	1	3	✘

Cea mai mica valoare a lui `k` pentru care distanta `d` este suficient de mica reprezinta lungimea sablonului cautat (in cazul precedent `k = 5`). Pentru a obtine un algoritm de complexitate buna trebuie ca acest pas sa fie eficient; putem sa folosim un arbore de intervale, sa folosim un contor cu `k` care variaza de la `1` la `n` si sa eliminam din arbore elemente de marime mai mica decat `k` si, la fiecare pas, sa actualizam arborele pentru a putea raspunde la interogari de genul: *care este distanta maxima intre doua elemente care exista acum in structura*. Algoritmul are complexitatea `O(N log N)`. Pentru o prezentare amanuntita a arborilor de intervale, va recomand [9] si [10].

Problema 9 (Olimpiada Baltica de Informatica[11], 2004)

Un sir de caractere `S` se numeste repetitie `(K, L)` daca `S` se obtine prin concatenarea de `K ≥ 1` ori a unui sir `T` de lungime `L ≥ 1`. De exemplu, sirul `S = abaabaabaaba` este o repetitie `(4, 3)` cu `T = aba`. Sirul `T` are lungimea trei si `S` se obtine repetandu-l pe `T` de patru ori. Avand un sir de caractere `U` format din caractere `a` si/sau `b` de lungime `n (1 ≤ n ≤ 50000)`, va trebui sa determinati o repetitie `(K, L)` care apare ca subsecventa a lui `U` astfel incat `K` sa fie cat mai mare. De exemplu, sirul `U = babbabaabaabaabab` contine repetitia `(4, 3)`, sirul `S` incepand de pe pozitia `5`. Aceasta este si repetitia maxima, deoarece sirul nu mai contine nici o alta subsecventa care sa se repete de mai mult de patru ori. Daca sirul contine mai multe solutii cu acelasi `K`, poate fi aleasa oricare dintre ele.

Solutie:

Dorim ca pentru un `L` fixat sa determinam cea mai mare valoare `K` astfel incat in sirul `U` sa avem o subsecventa `S` care este repetitie `(K, L)`. Vom considera acum un exemplu: `U = babaabaabaabaaab`, `L = 3` si o subsecventa fixata `X = aab` care incepe pe pozitia `4` a sirului `U`. Putem incerca sa extindem secventa `aab` la ambele capete cat mai mult posibil prin repetarea ei asa cum vedem in continuare:

b a b **a a b** a a b a a b a a a b

a a b a a b

a b a a b a a b a a b a **a b a**

Extinzand in acest mod cat mai mult in stanga secventa noastra si apoi extinzand la dreapta prefixul de lungime `L` (in exemplul nostru prefixul de lungime `3`) al secventei obtinute, gasim cea mai lunga repetitie a unui sir de caractere de lungime `L` cu proprietatea ca repetitia contine ca subsecventa sirul `X` (daca repetitia este `(1, L)` afirmatia anterioara nu este adevarata, dar acesta este un caz trivial). Acum observam ca pentru a identifica toate repetitiile `(K, L)` cu `L` fixat din sirul `U`, este suficient sa partitionam sirul in `n/L` bucati si sa extindem aceste bucati. Remarcam ca daca va fi posibil sa realizam acest lucru pentru ficare bucata in `O(1)` algoritmul final va avea ordinul de

complexitate O(n/1 + n/2 + n/3 + .. + n/n) (fiecare bucata se poate repeta in totalitate sau doar partial in stanga sau in dreapta, iar noi nu vom extinde fiecare bucata separat, ci bucatile adiacente le vom reuni intr-o noua bucata; asadar, daca avem p bucati consecutine de aceeasi dimensiune, vom determina extinderile lor maxime in timpO(p)). Dar stim ca sirul 1 + 1/2 + 1/3 + 1/4 + .. + 1/n – ln n converge spre o constanta c, numita constanta lui Euler, si c < 1; de aici tragem concluzia ca O(n/1 + n/2 + n/3 + .. + n/n) = O(n log n), deci algoritmul, in cazul in care extinderile maxime pot fi calculate usor, ar avea ordinul de complexitate O(n log n). Acum intervin in rezolvarea noastra arborii de sufixe. Pentru a determina cu cat putem extinde cel mai mult subsecventa U[i..j] a sirului U la dreapta, practic ne intereseaza cel mai lung prefix comun al subsecventei U[i..j] si al subsecventei U[j+1..n]. Pentru a extinde cat mai mult la stanga este suficient sa inversam sirul U si ajungem sa rezolvam aceeasi problema. Am vazut ca problema celui mai lung prefix comun a doua secvente se rezolva in timp O(1) cu ajutorul sirurilor de sufixe. Astfel, avem nevoie de crearea sirului de sufixe, etapa pe care o rezolvam intr-un timp de ordinul O(n log n) si apoi de aplicarea algoritmului explicat anterior care are complexitatea O(n log n). In concluzie, algoritmul prezent are complexitatea totala O(n log n).

Problema 10 (ACM SEER 2004)

Avand un sir de caractere S dat, se cere ca pentru fiecare prefix al sau sa se determine daca este un sir de caractere periodic. Astfel, pentru fiecare i (2 ≤ i ≤ N) ne intereseaza cel mai mare K ≥ 1 (daca exista un asemenea K) cu proprietatea ca prefixul lui S de lungime i poate fi scris cub forma A^k (sirul A concatenat cu el insusi de k ori) pentru un sir de caractere A. De asemenea, ne intereseaza si valoarea k (avem 0 ≤ N ≤ 1000000).

Exemplu

Pentru sirul aabaabaabaab obtinem rezultatul prezentat in continuare:

```
2 2
6 2
9 3
12 4
```

Explicatii

- prefixul aa are perioada a;
- prefixul aabaab are perioada aab;
- prefixul aabaabaab are perioada aab;
- prefixul aabaabaabaab are perioada aab;

Solutie:

Sa vedem ce se intampla cand incercam sa potrivim un sir cu un sufix al sau. Consideram un sir si il impartim in doua, obtinand un prefix si un sufix:

```
S = aab aabaabaaaab
suf = aab aabaaaab
pre = aab
```

Daca sufixul se potriveste cu sirul initial pe un numar de caractere mai mare sau egal cu lungimea sirului pre, inseamna ca pre este si un prefix al sufixului; deducem ca putem imparti si sufixul in pre si suf1, iar sirul putem sa il impartim in pre, pre si suf1. Daca sirul se potriveste cu sufixul pe un numar de caractere mai mare sau egal cu dublul lungimii sirului pre, atunci sufixul se potriveste cu suf1 pe un numar de caractere mai mare sau egal cu lungimea sirului pre, deci suf1 poate fi scris ca pre si suf2, deci suf poate fi scris ca pre, pre, suf2, iar S poate fi scris ca pre, pre, pre, suf2:

```
S = aab aab aab aaaab
suf = aab aab aaaab
```

`suf1 = aab aaaab`
`pre = aab`
Observam astfel ca daca sirul `S` se potriveste cu sufixul sau pe cel putin `k * |pre|` caractere, atunci `S` are un prefix de lungime `(k+1) * |pre|` care este periodic. Folosindu-ne de structura de date `siruri de sufixe`, putem determina pentru fiecare sufix potrivirea maxima cu sirul initial. Daca al `i`-lea sufix se potiveste cu sirul pe primele `k * (i-1)` pozitii, atunci putem actualiza informatia care indica daca prefixele de dimensiune `j * (i-1)` (unde $2 \leq j \leq k$) sunt periodice. Pentru fiecare sufix `S`, actualizarea tuturor informatiilor are ordinul de complexitate `O(n / (i-1))`. Astfel, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este `O(n log n)`. Trebuie remarcat faptul ca putem obtine o rezolvare in timp `O(n)` folosind o idee similara si algoritmul `KMP`, dar prezentarea acestei rezolvari depaseste scopul acestui articol.

Concluzii

Mentionam ca in timpul concursurilor autorii prefera solutiile ale caror ordine de complexitate sunt $O(n \log^2 n)$, mai lente, dar mai usor de implementat, si care folosesc un spatiu de memorie de ordinul `O(n)`. Din punctul de vedere al timpului real de executie, cele doua tipuri de solutii vor fi comparabile, iar in concurs simplitatea solutiei usureaza foarte mult implementarea si depanarea. Din cele prezentate putem concluziona ca sirurile de sufixe sunt o structura de date usor de implementat si foarte utila. In ultimii ani apar la concursuri tot mai multe probleme care necesita cunoasterea acestora. Mai putem observa si faptul ca polonezii propun probleme destul de grele la olimpiade. Speram ca acest articol va va fi de folos si ca de acum inainte sirurile de sufixe vor fi la indemana oricui are nevoie de ele pentru a le folosi intr-un concurs de informatica.

Bibliografie

1. Mark Nelson, *Fast string searching with suffix trees*
2. Mircea Pasoi, *Multe "smenuri" de programare in C/C++... si nu numai!*
3. Emilian Miron, *LCA - Lowest common ancestor*
4. Michael A. Bender, Martin Farach-Colton, *The LCA Problem Revisited*
5. Erik Demaine, *MIT Advanced Data Structures, Lecture 11, April 2nd, 2003*
6. Udi Manber, Gene Myers, *Suffix arrays: A new method for on-line string searches*
7. Mohamed Ibrahim Abouelhoda, Stefan Kurtz, Enno Ohlebusch, *Replacing suffix trees with enhanced suffix arrays*, Journal of Discrete Algorithms 2, 2004
8. Thomas Cormen, Charles Leiserson, Ronald Rivest, *Introducere in algoritmi*, Editura Computer Libris Agora, 2000
9. Dana Lica, *Arbori de intervale*
10. Cosmin Negruseri, *Cautari ortogonale*, GInfo 15/5 (Mai 2005), Editura Agora Media
11. BOI 2004