Camp προετοιμασίας
Πανελλήνιος Διαγωνισμός Πληροφορικής
Αθήνα 11-15 Μαρτίου, 2013

C++ STL,

Στρατηγική ανάπτυξης προγράμματος την ώρα του διαγωνισμού

Επικοινωνία: greekpdpcontests@gmail.com

Συνοπτικά

- Δε θα ασχοληθούμε με πολύπλοκα ζητήματα της γλώσσας.
- Ασχολούμαστε με τα στοιχεία αυτά που θα μας βοηθήσουν να γράψουμε μια λύση σωστή κι εύκολη στην αποσφαλμάτωση.
- Κύριος άξονας : Πώς θα κερδίσω και τον τελευταίο πόντο που μπορώ στον ***περιορισμένο χρόνο του διαγωνισμού***.

Επιτρεπόμενες Γλώσσες

- Pascal :

(+) Σχετικά απλή

(-) Πεπαλαιωμένη, μηδενική υποστήριξη βιβλιοθήκης

- C :

(+) Πολύ γρήγορες υλοποιήσεις

(-) Εύκολο να κάνεις λάθος

- C++ :

(+) Βιβλιοθήκη STL (Standard Template Library)

(-) Υπερβολικά πολύπλοκη

Προτεινόμενη Γλώσσα

- Αν γράφουμε σε Pascal, μαθαίνουμε C.
- Αν ξέρουμε C, μαθαίνουμε πώς να γράφουμε C σε περιβάλλον C++ ώστε να έχουμε πρόσβαση στην STL.
- Αν ξέρουμε C++, μαθαίνουμε μερικά χαρακτηριστικά της C++11 που μας σώζουν από τζάμπα κόπο και σφάλματα:
  - Αυτόματοι τύποι με **`auto`**
  - Range-based **`for`** loop

Χρήσιμες Συμβουλές

Πόσο γρήγορο αλγόριθμο πρέπει να κάνω;

Στους διαγωνισμούς έχουμε έναν εμπειρικό κανόνα που λειτουργεί άψογα:

- Βρίσκουμε την πολυπλοκότητα του αλγορίθμου μας και αγνοούμε σταθερές (χαίρω πολύ…).
- 100.000.000 πράξεις ↔ 1 δευτερόλεπτο

Χρήσιμες Συμβουλές

- Καμία θυσία σαφήνειας για χάρη της ταχύτητας.
- Μεγάλα κενά μεταξύ διαφορετικών blocks του κώδικά μας.
- Πολλά, πολλά υποπρογράμματα.

Αν ένα υποπρόγραμμα ξεπεράσει τις 20 γραμμές κώδικα, συνήθως πρέπει να σπάσει σε μικρότερα.

Χρήσιμες Συμβουλές

- Καμία θυσία σαφήνειας για χάρη της ταχύτητας.
- Η μόνη μεταβλητή με όνομα μεγέθους 1 χαρακτήρα είναι η **`i`** κι η **`j`**.

Πιο εύκολα αποσφαλματώνεται το
`curr_node = par[ neib ];`
παρά το
`x = A[y];`

(Πόσο μάλλον το **`u += v;`**)

Λίγα πρώτα λόγια για την STL

- Με τις παρακάτω δύο γραμμές εισάγουμε **όλες** τις βιβλιοθήκες!

```
#include <bits/stdc++.h>
  using namespace std;
```

- Δηλώνουμε έναν container χρησιμοποιώντας

container<type> container_name

  - **vector<int> nodes;**
  - **set< pair<int,int> > distances;**
  - **map< int, int > compressed;**

STL – Pairs

- Δηλώνεται ως pair<type1, type2>. Π.χ.

```
pair<int,int> val;
```

- Πρόσβαση στο πρώτο στοιχείο με **`val.first`**, στο δεύτερο με **`val.second`**
- Δημιουργούμε καινούριο pair με τη χρήση του

```
val = {5, 10};
```

STL – Pairs

- Για ευκολία ορίζουμε στην αρχή του προγράμματος :

```
#define X first
#define Y second
typedef pair<int,int> pii;
```

- Πλέον μπορούμε να γράφουμε πιο γρήγορα

```
val.X = 5;
val.Y = “moxos”;
```

STL – Pairs

- Γιατί όχι δικιά μας **`struct`** με 2 στοιχεία;

Τα pairs έχουν έτοιμους τους operators για σύγκριση.

Συγκρίνουν πρώτα τα pair1.X και pair2.X και επιστρέφουν το αποτέλεσμα, εκτός αν είναι ίσα, οπότε επιστρέφουν το αποτέλεσμα της σύγκρισης του pair1.Y και pair2.Y (λεξικογραφική σύγκριση).

- Παράδειγμα χρήσης pair ο αλγόριθμος του dijkstra, όπου κρατάμε ταξινομημένους τους κόμβους με βάση τις αποστάσεις απ' την αρχή.

Κρατάμε απλά **`pii dist`**, όπου **`dist.X`** η απόσταση και **`dist.Y`** ο κόμβος.

STL – Tuples

- Δηλώνεται ως tuple<type1, type2, … type N>. Π.χ.

```
tuple<int,int,char> val;
```

- Πρόσβαση στο πρώτο στοιχείο με get<0>(val), στο δεύτερο με **get<1>(val)...**
- Δημιουργούμε καινούριο tuple με:

```
val = {5, 10, 'a'};
```

Αντί για:

```
get<0>(val) = 5; get<1>(val) = 10;
  get<2>(val) = 'a';
```

Vectors

- Πρόκειται ουσιαστικά για τα γνωστά arrays.
- Δήλωση ενός vector που περιέχει 30 integers, αρχικοποιημένους στο 0:

```
vector<int> example(30); // Όχι αγκύλες
```

- Πρόσβαση στο πρώτο στοιχείο:

```
printf (“%d\n”, example[0]);
```

- Δήλωση ενός vector με 10 char, αρχικοποιημένους στο `'z'`:

```
vector<char> example2(10, 'z');
```

Vectors

- Πλεονέκτημα : Μπορούμε να αυξήσουμε το μέγεθός τους κατά την ώρα της εκτέλεσης!

```
vector<int> example(30);
example.push_back(5);
example.push_back(10);
printf (“%d\n”, example[31]); // Τυπώνει 10
```

Vectors

- Μπορούμε να χρησιμοποιήσουμε ακόμα και άδειο vector, και να τον γεμίσουμε αργότερα με **push_back**.
- Χρησιμοποιείται ιδιαίτερα για δημιουργία adjacency list. Την διατρέχουμε με
  for (int i = 0; i < e.size(); ++i)

Ακριβέστερα
  i < e[v].size

αφού κάθε κόμβος έχει τη δικιά του adjacency list – vector.
  Μια λίστα γειτνίασης είναι ένα vector από vectors:

typedef vector<vector<int>> adjlist;

Set

- Κρατάνε τα στοιχεία οργανωμένα (ταξινομημένα) με τρόπο που κάνει την αναζήτηση απλή και γρήγορη.
- Κρατάνε μία φορά το κάθε στοιχείο, όσες φορές κι αν το εισάγουμε.
- Αν θέλουμε πολλαπλότητες, δηλώνουμε **`multiset`**.

Set

- Δηλώνονται ως *set<type> S;*

set<int> S;

multiset<int> multiS;

- Προσθέτουμε στοιχεία με την insert.
  S.insert(5);
- Για πρόσβαση χρειαζόμαστε έναν *iterator* (κάτι αντίστοιχο του pointer) που δηλώνεται ως
  set<int>::iterator pos;
  (μη φοβάστε, δε θα χρειαστεί ποτέ η παραπάνω σύνταξη, δόξα τω **auto**)

Set

- Διασχίζουμε όλα τα στοιχεία με

```
for (auto pos = S.begin(); pos != S.end(); ++pos) {
   printf (“%d\n”, *pos); // pos is an iterator/pointer
}
```

- Το **`S.end()`** δείχνει μια θέση “μετά το τέλος” του **`set`**.
- Εναλλακτικά (C++11):

```
for (auto& elem : S){
   printf (“%d\n”, elem); // elem is a reference
}
```

Set

- Εύρεση τιμής :

```
pos = S.find(5);
if (pos == S.end()) {
  printf (“Value not found \n”);
}
```

Set

Λειτουργούν επίσης (όπως σε κάθε container):

- **`S.size()`** που γυρνάει έναν ακέραιο, το πλήθος των στοιχείων που περιέχει το set
- **`S.empty()`** που γυρνάει μια bool τιμή, αν είναι άδειο το set.

Map

- Μπορούν να λειτουργήσουν ως πίνακες, όμως για index δίνουμε ό,τι θέλουμε. Π.χ.

```
my_map[ “moxos” ] = “Way of debugging”;
my_map[ 999999999999LL ] = 'A';
```

- Σε κάθε key (στην περίπτωσή μας “moxos”) αντιστοιχεί μία τιμή (“Way of debugging”).
- Δηλώνονται με:
  map<key_type, value_type> map_name;

Map

- Για να ελέγξουμε ΑΝ υπάρχει ένα key, δε χρησιμοποιούμε την έκφραση

```
my_map[key]
```

γιατί έτσι θα δημιουργούνταν εκείνη τη στιγμή!

- Αντ' αυτού:

```
if (my_map.find(key) != my_map.end())
```

Map

- Ιδιαίτερα χρήσιμη δομή σε περιπτώσεις που δε μπορούμε να χρησιμοποιήσουμε απλούς πίνακες.

Π.χ. Αν θέλουμε να δούμε πόσες φορές εμφανίζεται ένας αριθμός, **`frequency[A[i]]++`** δε θα ήταν σωστό αν ο frequency δεν είχε τόσο πολλές θέσεις όσο ο **`A[i]`**.

- Παρόλα αυτά, η πρόσβαση γίνεται σε O(logN), και μάλιστα με πολύ μεγάλο factor. Συνεπώς, όπως κάθε τι της STL, μπορεί να βοηθήσει πάρα πολύ, αρκεί να χρησιμοποιείται με φειδώ.

Strings

Ο ευκολότερος τρόπος να χειριστούμε strings!

string a=”Haha”, b=”Baba”, c;

if (a<b) //false

  c = **a.substr(2,2)**;

  //c=”ha” (starting at pos 2 and having length 2)

a[0] = 'a';

if (a<b) //true

  printf(“%d\n”, **a.length()**); //prints 4

Strings

Για τους λάτρεις της C:

```
char a[1005];
scanf(“%s”, a);
string str(a); //str equals a
printf(“%s”, str.c_str());
```

Algorithms

- Πολλοί χρήσιμοι αλγόριθμοι, με σημαντικότερη την **`sort`**.
- Βοηθάει να γράφουμε γρήγορο και καθαρό κώδικα.
- Χρησιμοποιούμε σχεδόν παντού ημιανοικτά διαστήματα (half-open ranges). Για κάθε container, ένα τέτοιο διάστημα παράγεται από τα αντίστοιχα **`begin()`** και **`end()`**.

Algorithms

- `min(var1, var2)`

  `max(var1, var2)`
- Επιστρέφουν αντίστοιχα το μικρότερο και το μεγαλύτερο των δύο τιμών.
- Πολύ βολικό, λειτουργεί για κάθε τύπο δεδομένων.

Algorithms

- *`sort(begin_pointer, end_pointer);`*
- Για να ταξινομήσουμε τα στοιχεία ενός vector :

```
sort (my_vec.begin(), my_vec.end());
```

- Για να ταξινομήσουμε τα στοιχεία 0 έως N-1 (ή 1 έως N) ενός πίνακα :

```
sort(Array, Array+N); // 0 έως N-1
sort(Array+1, Array+N+1); // 1 έως N
```

Algorithms

- Αλλιώς για να συγκρίνουμε με δικό μας τρόπο :

```
sort(intervals, intervals+N, smaller);
```

- Όπου η **comp** ορίζεται από εμάς ως :

```
bool smaller (Interval a, Interval b) {
   return a.left < b.left;
   // Για φθίνουσα σειρά return a.left > b.left
}
```

Algorithms

```
next_permutation(first, last);
prev_permutation(first, last);

next_permutation(first, last, smaller);
prev_permutation(first, last, smaller);
```

- Γεννά την επόμενη / προηγούμενη λεξικογραφική διάταξη σε O(N).

Algorithms

- Επιστρέφει **`false`** όταν ο πίνακας δεν έχει επόμενη (ή προηγούμενη) λεξικογραφική διάταξη.
- Για να γεννήσουμε όλες τις διατάξεις :

```
sort (A, A+N, comp);
do {
  function(A);
} while (next_permutation(A, A+N, comp));
```

Άλλες χρήσιμες συναρτήσεις

- __gcd(a,b) : Επιστρέφει τον μέγιστο κοινό διαιρέτη των a και b.
- __builtin_popcount(a) : Επιστρέφει το πλήθος των άσσων στη δυαδική αναπαράσταση του a.

DOs and DON'Ts

Χρησιμοποιούμε STL όταν μας γλιτώνει από περιττό κώδικα. Δεν υπάρχει κανένας λόγος να ξαναγράψουμε quicksort όταν υπάρχει έτοιμη.

Αποφεύγουμε την χρήση της STL όταν μπορούμε με τον *ίδιο κόπο* να γράψουμε standard C/C++.

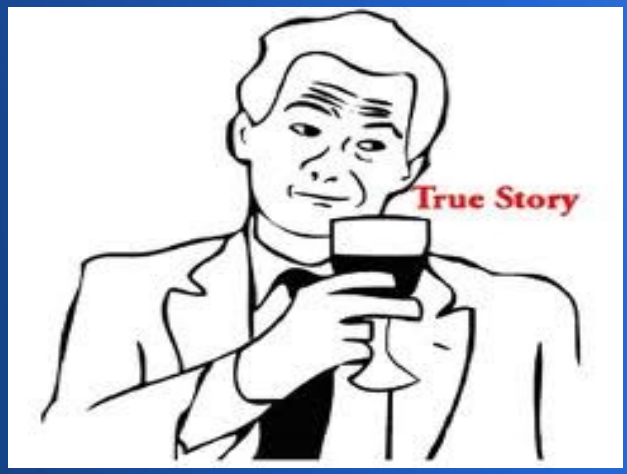
Χρησιμοποιούμε προχωρημένες δομές και features όταν ξέρουμε ότι μας δίνουν *σημαντικό* πλεονέκτημα σε ταχύτητα.

Δεν αναλωνόμαστε σε low-level λεπτομέρειες: Ένας καλός αλγόριθμος υλοποιημένος μέτρια κερδίζει ένα κακό αλγόριθμο υλοποιημένο άψογα.

Δε χρησιμοποιούμε ένα feature με το οποίο δεν έχουμε εμπειρία. Θα χάσουμε πολύτιμο χρόνο διαβάζοντας ακατάληπτα κατεβατά με compilation errors ή αποσφαλματώνοντας κάποιο edge case.

Μερικές τελευταίες συμβουλές

- Δηλώνουμε τους πίνακες με λίγα παραπάνω στοιχεία απ' αυτά που χρειαζόμαστε. Έχουμε παιδιά που χάσανε μετάλλια λόγω μικρού πίνακα!



Μερικές τελευταίες συμβουλές

- Όπου μπορούμε χρησιμοποιούμε αρίθμηση 1 έως Ν που καταλαβαίνουμε εύκολα, όχι 0 έως Ν-1.

Μερικές τελευταίες συμβουλές

- Σπάνια χρήση του Debugger, συνήθως βολεύει αλλά απαιτεί πολύ χρόνο.
- Ο πιο απλός και γρήγορος τρόπος στα πλαίσια του διαγωνισμού είναι η printf.
  Γράφουμε μια χαρακτηριστική λέξη κλειδί ώστε να ξέρουμε ότι πρόκειται για γραμμή debug.

  Π.χ.
  printf (“moxos %d\n”, steps);

Μερικές τελευταίες συμβουλές

- Δεν παρασυρόμαστε επειδή ο χρόνος είναι περιορισμένος!

  Δεν γράφουμε κώδικα αν δεν έχουμε ολόκληρη τη λύση στο μυαλό μας.

Τι να μας μείνει;

- Σαφήνεια, σαφήνεια, σαφήνεια!
- Πολλά υποπρογράμματα για να πετύχουμε σαφήνεια.

  Ακόμα κι αν χρειαστούμε μία φορά κάποιο υποπρόγραμμα, κάνοντας το ξεχωριστή συνάρτηση το debugging γίνεται απείρως πιο εύκολο.