Algorithmique et Programmation

—

Examen écrit / Corrigé

—

G1: monasse*(at)*imagine.enpc.fr G2: david.ok*(at)*imagine.enpc.fr
G3: eric.bughin*(at)*gmail.com G4: renaud.marlet*(at)*enpc.fr
G5: drarenij*(at)*imagine.enpc.fr G6: vialette*(at)*univ-mlv.fr

—

25/03/11

—

1 Programmation : machine à registres

1. L'équivalent de la ligne C++ `while(R[2]!=0){--R[2];++R[1];}` peut s'écrire :

```
1   if R[2]=0 goto 4
2   dec R[2]
3   inc R[1]
4   if R[3]=0 goto 0
5   halt
```

2-12. Voici le programme complet

```
#include <iostream>

const int ID_HALT=0, ID_INC=1, ID_DEC=2, ID_GOTO=3;

class Etat {
    int* reg;
    int nreg;
public:
    int next;
    Etat();
    ~Etat();
    int get(int i) const;
    void set(int i, int v);
    void print() const;
};

Etat::Etat() {
    next = 0;
    nreg = 1;
    reg = new int[nreg];
}

Etat::~Etat() {
    delete [] reg;
}

int Etat::get(int i) const {
    if(i >= nreg)
        return 0;
    return reg[i];
}

void Etat::set(int i, int v) {
```

```
    if(i >= nreg) { // Agrandissement
        int* reg2 = new int[i+1];
        for(int j=0; j<nreg; j++)
            reg2[j] = reg[j];
        for(int j=nreg; j<i; j++)
            reg2[j]=0;
        delete [] reg;
        reg = reg2;
        nreg = i+1;
    }
    reg[i] = v;
}

void Etat::print() const {
    std::cout << "[ ";
    for(int i=1; i<nreg; i++)
        std::cout << reg[i] << ' ';
    std::cout << ']' << std::endl;
}

class Instruction {
    int id;
    int i;
    int p;
public:
    Instruction(int iid=ID_HALT, int iReg=0, int pIns=0);
    void execute(Etat& etat) const;
    void print() const;
};

Instruction::Instruction(int iid, int iReg, int pIns) {
    id = iid;
    i = iReg;
    p = pIns;
}

void Instruction::execute(Etat& etat) const {
    ++etat.next;
    switch(id) {
    case ID_HALT: etat.set(0,0); break;
    case ID_INC: etat.set(i, etat.get(i)+1); break;
    case ID_DEC: etat.set(i, etat.get(i)-1); break;
    case ID_GOTO: if(etat.get(i)==0) etat.next=p; break;
    }
}

void Instruction::print() const {
    switch(id) {
    case ID_HALT: std::cout << "halt"; break;
    case ID_INC:  std::cout << "inc R["<<i<<']'; break;
    case ID_DEC:  std::cout << "dec R["<<i<<']'; break;
    case ID_GOTO: std::cout << "if R["<<i<<"]=0 goto "<< p; break;
    }
    std::cout << std::endl;
}

class Programme {
    Etat etat;
    Instruction* ins;
    int nins;
public:
    Programme();
    ~Programme();
    void ajoute(const Instruction& in);
```

```
    void init(int i, int v);
    void run();
    void print() const;
};

Programme::Programme() {
    nins = 0;
}

Programme::~Programme() {
    if(nins > 0)
        delete [] ins;
}

void Programme::ajoute(const Instruction& in) {
    Instruction* ins2 = new Instruction[nins+1];
    for(int i=0; i<nins; i++)
        ins2[i] = ins[i];
    if(nins > 0)
        delete [] ins;
    ins = ins2;
    ins[nins++] = in;
}

void Programme::init(int i, int v) {
    etat.set(i,v);
}

void Programme::run() {
    etat.set(0,1);
    etat.next = 0;
    while(etat.get(0) != 0) {
        etat.print();
        for(int i=0; i<nins; i++) {
            if(etat.next == i) std::cout << "> ";
            else std::cout << "  ";
            std::cout << i << ' ';
            ins[i].print();
        }
        std::cout << std::endl;
        ins[etat.next].execute(etat);
    }
}

int main() {
    Programme flip;
    flip.ajoute( Instruction(ID_GOTO,1,3) );
    flip.ajoute( Instruction(ID_DEC,1) );
    flip.ajoute( Instruction(ID_GOTO,2,4) );
    flip.ajoute( Instruction(ID_INC,1) );
    flip.ajoute( Instruction(ID_HALT) );

    std::cout << "flip 0 --> 1" << std::endl;
    flip.init(1,0);
    flip.run();
    std::cout << "flip 1 --> 0" << std::endl;
    flip.init(1,1);
    flip.run();
    return 0;
}
```

2 Algorithmique : ensemble sans doublon

13. On peut procéder de la manière suivante :

```
Ensemble union(const Ensemble& e1, const Ensemble& e2) {
    Ensemble u(e2);
    for(int i=0; i<e1.taille(); i++) { // On ajoute e1-e2
        int j=0;
        for(; j<e2.taille(); j++)
            if(e1[i]==e2[j]) break;
        if(j==e2.taille())
            u.ajoute(e1[i]);
    }
}
```

14. Le nombre de comparaisons d'éléments est au pire $m \times n$ s'il s'avère que les ensembles sont disjoints.

15. On exploite le fait que les tableaux sont triés :

```
Ensemble union(const Ensemble& e1, const Ensemble& e2) {
    Ensemble u(e2);
    int i=0, j=0;
    for(; i<e1.taille(); i++) {
        while(j<u.taille() && u[j]<e1[i])
            ++j;
        if(j==u.taille())
            break;
        if(e1[i] != u[j])
            u.insert(j++,e1[i]); // j est incremente car u[j] va etre decale
    }
    for(; i<e1.taille(); i++)
        u.ajoute(e1[i]);
    return u;
}
```

16. On a n comparaisons des éléments de `e2` avec `e1[0]`.

17. On a m comparaisons des éléments de `e1` avec `e2[0]`.

18. Les questions précédantes montrent que c'est au moins $\max(m, n)$ comparaisons. En fait, on en a au plus $m + n - 1$. Mais cela suppose que l'insertion d'un point a un coût constant, donc il faut une structure appropriée, comme une liste, mais du coup l'ajout d'un élément se fait en $O(n)$. Néanmoins, il n'est pas si difficile de modifier l'algorithme ci-dessus pour éviter les insertions.

    Preuve : on observe d'abord que

    $$n_0 + n_1 + \ldots + n_m = n.$$

    On note que dans le `while`, à l'étape $i < m$ on fait $n_i + 1$ comparaisons d'éléments (les premières n_i resultant par un `true`, et la dernière par `false`), mais on n'en fait que n_i au dernier i. Si on épuise le tableau u (c'est-à-dire $n_i = n_{i+1} = \ldots = n_m = 0$), on sort du `for` principal. Notant k le premier indice non nul des n_i en partant de la fin, on a donc :

    $$(n_0 + 1) + (n_1 + 1) + \ldots + (n_{k-1} + 1) + n_k = n + k$$

    comparaisons. Au pire, on se retrouve avec $k = m - 1$ et donc $m + n - 1$ comparaisons. Si par contre $n_m \neq 0$, on arrive à la fin du tableau `e1` avant et on trouve

    $$(n_0 + 1) + (n_1 + 1) + \ldots + (n_{m-1} + 1) = n - n_m + m \leq n + m - 1.$$