Concours National d'Informatique

Questionnaire de sélection

Correction des questions d'algorithmique et rapport des correcteurs

Thomas Deniau *

9 octobre 2009

1 À vos montres !

1.1 Énoncé

En épreuve machine de demi-finale, vous soumettrez des solutions pour divers problèmes. On vous donne six variables entières représentant trois heures de soumission (données sous la forme de deux entiers, l'un pour les heures et l'autre pour les minutes). Écrivez une fonction qui renvoie l'heure de la première soumission.

1.2 Corrigé

Cet exercice très simple visait à nous montrer quels candidats maîtrisaient une structure de données très simple : le test. La manière la plus naturelle de résoudre cet exercice est de convertir tous les temps en minutes, puis de faire deux tests, ce qui peut donner par exemple le code du listing 1.

Une manière un peu plus évoluée de procéder était de remarquer que l'ordre lexicographique suffisait à ordonner des couples de type (h, m). En effet, si $h_1 < h_2$, l'heure (h_1, m_1) précède (h_2, m_2) quelles que soient m_1 et m_2, alors que si $h_1 = h_2$, l'ordre est donné par la comparaison des minutes.

On peut alors, dans les langages qui n'ont pas d'ordre lexicographique intégré, redéfinir cette fonction, avec par exemple en C :

*Pour l'équipe des correcteurs de Prologin 2009 : Cyril Amar, Céline Baraban, Laurent Le Brun, Julien Grall, Thomas Lecomte, Jérémie Marguerie, Alexis Rolland, Thomas Deniau.

Listing 1 – Calcul de la date la plus petite (C)

```c
int earliest(int h1, int m1, int h2, int m2, int h3, int m3)
{
    int t[3] = { m1 + h1*60, m2 + h2*60, m3 + h3*60};
    int min = 0;
    if (t[1] < t[min]) min = 1;
    if (t[2] < t[min]) min = 2;
    return min;
}
```

```
bool earlier(int h1, int m1, int h2, int m2)
{
    return (h1 < h2) || ((h1 == h2 ) && (m1 < m2));
}
```

Des langages comme C++, Caml et Python ont un ordre lexicographique intégré (sur les vecteurs en C++, les listes en Python, les tuples en Caml).

Il suffisait donc par exemple d'écrire en Caml (en renvoyant l'heure et non l'indice ; l'indice était demandé sur le site d'entraînement, mais les deux démarches étaient bien sûr acceptées pour la réponse au questionnaire) :

```
let earliest h1 m1 h2 m2 h3 m3 = min (min (h1, m1) (h2, m2)) (h3, m3)
```

1.3 Solutions proposées par les candidats

Ce problème a été la source d'affres inattendus. Certains candidats ont ainsi écrit plus de 100 lignes de code pour résoudre ce problème étonnamment simple. D'autres ont écrit une fonction de tri pour trouver le minimum d'un tableau de 3 éléments !

Soulignons aussi le nombre de candidats ayant choisi de comparer des quantités du type $100h+m$. Cela est bien entendu correct (tout facteur multiplicatif supérieur ou égal à 60 fonctionnait, comme l'on s'en convaincra aisément en considérant l'ordre lexicographique, puisque m est majoré par 59), mais a étonné les correcteurs qui s'attendaient soit à trouver un réel ordre lexicographique soit une conversion en minutes bien plus naturelle...

Le but de cet exercice simple était de détecter ceux « qui se noient dans un verre d'eau ». Il a bien rempli son rôle. La règle est simple : dès que l'on a l'impression de se répéter en écrivant du code, il faut se demander si on ne peut pas « factoriser » (regrouper du code grâce à une fonction, un tableau, une structure de contrôle...) quelque chose.

Bien sûr, de nombreux candidats ont soumis ici un code correct, ce qui est bien le minimum s'il l'on espère réussir son épreuve régionale. Mais très peu ont su exprimer leur algorithme avec la concision ici possible et souhaitable.

2 GPS

2.1 Énoncé

On vous donne une liste de coordonnées de type (x_i, y_i) (nombres entiers) représentant les coordonnées cartésiennes sur une carte de France des différents centres d'examen pour les demi-finales. Vous vous situez en (x, y). Ecrivez une fonction qui renvoie le centre le plus proche de vous. Vous utiliserez des distances euclidiennes pour vos calculs.

2.2 Solution

Il s'agissait ici de calculer les distances correspondant à chaque centre d'examen, par la formule $(x - x_i)^2 + (y - y_i)^2$ (notons qu'il est inutile de prendre la racine carrée de cette valeur puisqu'on ne fait que des comparaisons, et que des nombres positifs et leurs carrés sont ordonnés de la même manière) puis de trouver le minimum du tableau résultant (ou de réaliser les deux opérations en une).

En Caml, on aurait par exemple pu écrire le code du listing 2 page suivante.

On peut faire encore plus simple avec certains langages qui fournissent des fonctions de plus haut niveau. Par exemple, avec F# :

Listing 2 – Point le plus proche (Caml)

```
let rec nearest (x,y) l =
  let dist (x2, y2) = (x2-x)*(x2-x)+(y2-y)*(y2-y) in
  let d = (List.map dist l) in
  List.fold_left min (List.hd d) (List.tl d)
```

Listing 3 – Point le plus proche (C++)

```
typedef pair<int,int> point;
point nearest(point orig, vector<point> & p)
{
    int minDist = INT_MAX;
    vector<point>::const_iterator minI = p.end();

    for (vector<point>::const_iterator i = p.begin() ;
         i != p.end() ; i++)
    {
        int curDist = (i->first - p->first)*(i->first-p->first) +
        (i->second - p->second)*(i->second-p->second);
        if (curDist < minDist)
        {
            minI = i;
            minDist = curDist;
        }
    }

    return *minI;
}
```

```
let gps (x,y) =
  let sqr n = n * n
  Seq.min_by (fun (xi,yi) -> sqr(xi-x) + sqr(yi-y))
```

En C++, avec une boucle (le même code à peu près qu'en Caml peut aussi être écrit à l'aide de la STL), l'on obtiendrait le code du listing 3.

2.3 Solutions proposées par les candidats

Les candidats ont en général très bien réussi cet exercice. La principale source d'erreur fut le calcul de la distance entre (x, y) et (x_i, y_i), qui n'a pas toujours été bien fait (il a soit été totalement bâclé, soit, plus souvent, basé sur une distance de Manhattan). Il faut croire que le terme « distance euclidienne » n'était pas clair. Une recherche de ce terme sur le Web donne pourtant de bonnes définitions dans les premiers résultats.

Le deuxième écueil fut le calcul du minimum d'un tableau. Ici encore, certains candidats ont cru que la manière la plus efficace d'obtenir le minimum d'un tableau était de le trier, ce qui n'est bien évidemment pas le cas. (Il suffit pour obtenir le minimum d'un parcours simple du tableau, de complexité $O(n)$, alors qu'un tri demande au minimum $O(n \log n)$ sans hypothèses supplémentaires).

Enfin, parmi les candidats qui ont utilisé un tri, certains ont codé leur propre fonction de tri : c'est à éviter. À peu près tous les langages fournissent une fonction de tri, souvent plus performante et moins boguée que ce que tout un chacun est capable de réaliser, et il est donc conseillé de l'utiliser. En C, il y a par exemple qsort. Connaître les méthodes de tri est important car ces dernières recouvrent de nombreux domaines de l'algorithmique, et certains exercices utilisant des algorithmes très particuliers peuvent vous demander de recoder une fonction de tri, mais ce n'était pas le cas ici.

3 Le fil d'Ariane

3.1 Énoncé

TTY, la chatte mythique de Prologin, est perdue dans le campus de Polytechnique. Celui-ci est donné sous la forme d'un labyrinthe rectangulaire, où une case est soit vide, soit un mur infranchissable. Votre tâche est de rassurer, le cas échéant, TTY : écrivez une fonction qui prenne en entrée la position de TTY, la position de sa gamelle, et le labyrinthe, et renvoie si elle peut ou non atteindre sa gamelle. Il n'est pas possible d'aller en diagonale, seuls les déplacements horizontaux et verticaux sont autorisés. TTY et sa gamelle se situent sur une case vide.

3.2 Solution

Il s'agissait d'un parcours simple de graphe. La solution la plus simple était donc ici une recherche en profondeur (DFS) ou en largeur (BFS) d'abord.

Cette correction n'a pas pour but d'expliquer ce qu'est un parcours de graphe. De nombreux sites Web, y compris Wikipédia, qui donne le pseudo-code de ces deux parcours, ainsi que la plupart des ouvrages d'algorithmique, traitent de ce sujet ; nous n'expliquerons donc pas en quoi ces parcours consistent.

Notons juste que pour une DFS appliquée à un labyrinthe, la visite d'une case consiste uniquement en :

- son marquage comme visitée ;
- la visite des quatre cases adjacentes lorsqu'elles existent et qu'elles n'ont pas déjà été visitées.

En déportant la vérification de l'existence et de l'état de la case en début de fonction, on peut écrire le programme extrêmement simple (bien plus que nombre de codes proposés par les candidats) du listing 4 page suivante.

Notons la factorisation de la visite des 4 cases adjacentes dans une boucle. Cela réduit considérablement la longueur du code et donc son temps de conception (et c'est encore plus le cas si l'on peut se déplacer en diagonale).

3.3 Solutions proposées par les candidats

Toutes sortes de solutions ont été ici proposées. Les correcteurs ont ainsi vu passer des solutions de centaines de lignes qui ne fonctionnaient pas : certains tentaient par exemple de gérer les retours sur les intersections précédentes « à la main » dans le cas où l'on était bloqué dans une branche... D'autres ont proposé des solutions fonctionnelles mais trop compliquées. On a ainsi vu des algorithmes s'approchant de calculs de composantes connexes, ce qui était bien évidemment exagéré ici.

De nombreux candidats ont, dans la même veine, tenté d'implémenter un algorithme de plus court chemin de type Dijkstra, voire de plus court chemin intégrant des heuristiques comme A*.

Listing 4 – Parcours d'un labyrinthe en profondeur d'abord (C)

```c
const int dirs[4][2] = {{0,1}, {1, 0}, {0, -1}, {-1, 0}};
bool dfs(int x, int y, char **lab)
{
    if (x < 0 || x >= w || y < 0 || y >= h) return false;
    if (lab[x][y] != '.') return false;
    if (x == x_gam && y == y_gam) return true;

    lab[x][y]='v'; // Marquage comme visitee

    for (int i = 0; i < 4 ; i++)
    {
        if (dfs(x+dirs[i][0], y+dirs[i][1], lab)) return true;
    }
    return false;
}
```

Cela était ici prendre un marteau pour écraser une mouche, puisqu'on ne posait pas la question du *chemin le plus rapide* mais de l'*existence du chemin*. La complexité d'algorithmes de recherche de chemins, que l'on applique ici dans un cas extrêmement particulier (les arêtes ont par exemple toutes le même poids) est inutilement élevée par rapport à celle d'un simple parcours en largeur.

La pertinence de l'utilisation d'heuristiques (A*) pouvait ici se discuter, mais il ne faut pas oublier que le temps de conception d'un algorithme est également un facteur important dans sa pertinence (en général tout comme en épreuves régionales, qui sont en temps limité) et que s'embêter à implémenter un algorithme intégrant des heuristiques comme A* ne peut que faire perdre du temps dans le cas présent.

Par ailleurs, certaines soumissions tentant d'utiliser des heuristiques étant bien faibles par ailleurs, nous recommandons aux candidats débutants de se recentrer sur les fondamentaux (techniques de base de parcours de graphes) avant de tenter d'appliquer des algorithmes évolués. L'algorithmique ne consiste pas en l'application d'algorithmes tout faits mais en l'élaboration de nouveaux algorithmes à partir de briques de base. Choisir ses briques de base est donc important ! La description des parcours simples de graphes se trouvera dans n'importe quel livre d'algorithmique qui se respecte, comme l'irremplaçable *Introduction à l'algorithmique* [1].

4 Avion-cargo

4.1 Énoncé

Un avion cargo peut contenir exactement $2n$ conteneurs, alignés sur deux rangées de n conteneurs chacune. Chaque conteneur a un poids, et le poids d'une rangée est la somme des poids des conteneurs dans celle-ci. Pour équilibrer l'avion, on veut répartir les conteneurs dans les deux rangées de façon à ce que leur différence de poids (en valeur absolue) soit minimale. Étant donnés les poids des $2n$ conteneurs, écrivez une fonction qui renvoie ce minimum.

4.2 Solution

Ce problème à l'énoncé étonamment simple est en fait beaucoup plus compliqué qu'il ne semble y paraître. On le trouvera dans la littérature mentionné sur le nom de « number partitioning

problem » et l'on pourra trouver sur le Web de nombreux articles qui y sont consacrés, comme [2, 3] par exemple.

La plupart de ces articles, en revanche, s'intéressent à la conception d'algorithmes d'approximation pour ce problème. Nous nous intéressons ici à une solution exacte, possible lorsque nous nous intéressons à peu d'éléments ($n \leq 100$ sur le site d'entraînement).

4.2.1 Solution très naïve

La solution la plus naïve essaie tout simplement toutes les possibilités : pour chaque conteneur, on lance deux appels récursifs, l'un testant le cas où le poids est mis à droite, l'autre le cas où le poids est mis à gauche. Cela a une complexité bien trop importante (exponentielle).

L'astuce est alors de se rendre compte que la réponse renvoyée par notre fonction peut uniquement se servir du *poids déjà accumulé à gauche*, de l'indice courant, et du *nombre de conteneurs stockés à gauche* et non de la répartition exacte des conteneurs. On entre alors dans le cadre de la théorie de la *programmation dynamique* (ici encore, nous référons le lecteur à tout bon ouvrage d'algorithmique), ce qui nous permet, en enregistrant les résultats calculés aux appels récursifs précédents, de proposer une solution en $O(nW\,n^2)$ soit $O(Wn^3)$ où W est le poids maximum d'un conteneur (2000 sur le site d'entraînement) (en effet, la somme des poids stockés sur une rangée est au maximum de nW).

Nous obtenons alors le code du listing 5 page suivante (en C++), avec NN une constante valant 200 (valeur maximum de $2n$) et MAXPOIDS une constante valant 2000×100. Les poids sont supposés être contenus dans le tableau poids[NN] et sum contient leur somme (calculée avec une boucle ou la fonction accumulate de la STL). N contient la valeur réelle de N. Enfin, le tableau dyna_tres_naif est supposé être rempli de -1 au premier lancement de la fonction.

4.2.2 Solution naïve

Pour améliorer la complexité de notre algorithme, il fallait sortir de la simple application de la recherche exhaustive et généraliser le problème. Pour trouver la différence minimum atteignable, il faut d'abord savoir quels poids sont atteignables avec n conteneurs. Tel que, ce chiffre est difficilement déductible de la liste des poids atteignables avec $n-1$ conteneurs, car on ne sait pas quels conteneurs ont été choisis (et ne sont donc plus réutilisables) pour engendrer ces poids. C'est alors que l'on a l'idée de répondre à la question « quels poids peut-on atteindre avec k conteneurs pris parmi les l premiers ? ».

Cette question se résout très facilement par programmation dynamique. En effet, ces poids sont ceux atteignables avec $k-1$ conteneurs pris parmi les $l-1$ premiers, augmentés du poids du l^{me}, d'une part, et ceux atteignables avec k conteneurs pris parmi les $l-1$ premiers, d'autre part. Les conditions aux limites s'écrivent elles aussi facilement.

Pour répondre, il suffit alors de trouver tous les poids atteignables avec n conteneurs et de trouver celui qui est le plus proche de la moitié de la somme totale des poids.

Malheureusement, cet algorithme a la même complexité temporelle que le précédent ($O(Wn^3)$). En effet, le nombre de poids atteignables dans chaque configuration est majorable par nW et la recopie de ces ensembles prend donc un temps $O(nW)$ à chaque étape.

En revanche, sa complexité en termes de mémoire est réduite (on passe de $O(Wn^3)$ à $O(Wn\,n)$). On a ici une complexité mémoire de $O(Wn^2)$ et non $O(Wn^3)$, car mémoriser toutes les valeurs est inutile ; lors du calcul de la « ligne » correspondant à l, il suffit de garder la « ligne » précédente).

Nous n'allons donc pas implémenter cet algorithme ; en revanche, il nous donne l'idée pour l'implémentation de l'algorithme suivant.

Listing 5 – Algorithme dynamique très naïf pour le partitionnement équilibré (C++)

```
int dyna_tres_naif[NN+1][MAXPOIDS][NN+1];
int sum, int poids[NN], int N;

int tres_naif(int i, int Pgauche, int Cgauche)
{
  if (i == N && Cgauche == N/2)
  {
    // on a traite tous les conteneurs, renvoi du resultat
    return abs(sum - 2 * Wgauche);
  }
  if (i == N)
  {
    // on a traite tous les conteneurs,
    // il n'y en a pas assez a gauche
    return 1000000000;
  }
  if (Cgauche > N/2)
  {
    // trop de conteneurs a gauche
    return 1000000000;
  }

  int &res = dyna_tres_naif[i][Pgauche][Cgauche];

  // deja calcule
  if (res >= 0) return res;

  res = min(tres_naif(i+1, Pgauche + poids[i], Cgauche+1),
            // on le met a gauche
            tres_naif(i+1, Pgauche,            Cgauche  ));
            // on le met a droite

  return res;
}
```

Listing 6 – Algorithme dynamique subtil pour le partitionnement équilibré (C++)

```
int sum, int poids[NN], int N;
bitset<NN / 2 + 1> reach[MAXPOIDS];

int equilibre() {
  reach[0] = 1;
  const int upper = sum / 2 + 1;
  for (int i = 0 ; i < N ; i++) {
    for (int w = upper ; w >= 0 ; --w) {
      const int nw = w + poids[i];
      if (nw <= upper) {
        reach[nw] |= reach[w]<<1;
      }
    }
  }
  int best = INT_MAX;
  for (int w = upper ; w >= 0 ; w--) {
    if (reach[w][N/2]) {
      best = min(best, abs(sum - 2*w));
    }
  }
  return best;
}
```

4.2.3 Solution subtile

Pour arriver à une complexité optimale, il faut une dernière fois renverser le problème : « avec quels nombres de conteneurs parmi les k premiers un poids donné peut-il être atteint ? ».

Alors, pour le poids W et l'état d'avancement k, il suffit de prendre les nombres de conteneurs parmi les $k-1$ premiers permettant d'atteindre $W - w_k$ et d'y ajouter un, ajoutés bien sûr aux nombre de conteneurs parmi les $k-1$ premiers permettant d'atteindre W. Remarquons ici encore qu'il suffit de stocker la « ligne » actuelle de la matrice, tous les calculs dépendant uniquement de la ligne précédente.

Pour effectuer cette opération (ajouter un à tous les nombres de conteneurs possibles stockés), on peut stocker cette liste sous forme compressée. En supposant qu'on a moins de 64 poids, on peut prendre pour convention qu'un poids est atteignable par le nombre de conteneurs i si son i^{me} bit est positionné à 1. On généralise alors cette opération à plus de 64 poids à l'aide, par exemple, d'un tableau d'entiers. En C++, la structure `bitset` fait tout cela pour nous automatiquement.

L'opération d'ajout d'un à chaque position s'écrit alors `reach[nw] |= reach[w]<<1;` et s'exécute en temps $O(n)$, mais cette complexité théorique est en fait négligeable car la constante cachée dans la notation O est très faible (un bitset permet de multiplier par 64 la vitesse de ce genre d'opérations).

Comme précédemment, il suffit ensuite de chercher le w le plus proche de la somme atteignable avec n conteneurs. Une implémentation de l'algorithme résultant en C++ est donnée en listing 6.

Nous avons ici des complexités théoriques $O(n\,nW\,n)$ en temps et $O(nW\,n)$ en mémoire, soit la même chose que précédemment, mais comme nous venons de l'expliquer, cet algorithme est beaucoup plus performant en pratique. Sur le test 9 (le plus gros, avec $n = 100$), cet algorithme s'exécute en 1,6 s avec l'option de compilation `-O2` contre 9,7 s pour l'algorithme précédent.

4.3 Solutions proposées par les candidats

Les candidats ont ici donné libre cours à leur imagination. Nous avons trouvé toutes sortes de solutions.

Des solutions approchées fondées sur l'heuristique tout d'abord, qui tentaient d'obtenir une solution en se fondant sur un schéma prédéterminé. Les deux algorithmes les plus courants que nous avons trouvés consistent en un tri préalable du tableau des poids, suivi d'une assignation des poids dans l'ordre obtenu à droite et à gauche alternativement (algorithme de base), ou d'une assignation des poids dans l'ordre obtenu successivement à droite ou à gauche, du côté le moins chargé (algorithme un peu plus avancé). Ces algorithmes donnaient des réponses fausses sur bon nombre de tests.

D'autres candidats ont programmé des solutions testant exhaustivement toutes les possibilités. Ils dépassaient donc la limite de temps et/ou de mémoire impartie. Un candidat a tout de même réussi à suffisamment optimiser sa solution naïve (en ne visitant pas les branches inutiles de l'arbre des possibilités) pour passer tous les tests.

Dans les approches les plus originales, nous avons notamment trouvé un recuit simulé et un algorithme génétique. Un candidat a même créé des threads afin de lancer deux algorithmes optimisés dans des cas différents de la distribution d'entrée en parallèle et en donnant le résultat du premier algorithme terminant.

Un très petit nombre de candidats a pensé à la programmation dynamique pour ce problème, dont un seul avec l'algorithme optimal (il y avait à chaque fois une boucle sur n en trop dans les rares copies s'approchant de l'optimal). L'algorithme le plus naïf exposé ci-dessus était pourtant assez facile d'accès. Il est regrettable que si peu de personnes aient pensé à la programmation dynamique alors qu'il s'agit d'une technique de base pour résoudre un très grand nombre de problèmes d'optimisation combinatoire.

Concernant la notation, une heuristique un peu avancée et une solution exhaustive non optimisée ont été notées de la même manière. Pour les algorithmes corrects ou utilisant une heuristique très approfondie, la note était ensuite d'autant plus élevée que l'algorithme trouvé était rapide et correct. Une justification de la validité de l'algorithme en commentaire a également été appréciée.

5 Remarques générales sur le style

Voici, en vrac, quelques remarques qui ressortent de l'examen de vos centaines de copies :

- Nous implorons les candidats programmant en Caml d'utiliser un style de programmation fonctionnel et non impératif. N'utilisez un style impératif que lorsque cela améliore de beaucoup la lisibilité de votre code. N'oubliez pas qu'à chaque fois que vous utilisez le mot-clé `ref`, Dieu tue un poussin... Merci pour les poussins.
- Même si nous aimons le logiciel libre, nous pensons que l'inclusion de la GPL dans chaque exercice est un peu exagérée.
- Recherchez la lisibilité et la concision. Nos exercices ne demandent qu'exceptionnellement plus de cinquante lignes de code (hors lecture de l'entrée).
- Il va sans dire que comme chaque année, les candidats ayant rendu un code correctement indenté, correctement commenté, et facilement compréhensible, ont été récompensés... C'est important pour vos progrès (ce qui se conçoit bien s'énonce clairement ; en d'autres termes, si l'on ne sait pas expliquer ce que l'on fait, c'est que l'on n'a pas compris), votre propre compréhension (qui ne relit pas ses programmes quelques années après les avoir écrits ?), et cela réduit le nombre de cachets de Doliprane consommés pendant la correction.

Merci à tous d'avoir participé, merci à ceux qui ont cherché des algorithmes impressionnants pour la question 4, et bonnes révisions à tous les sélectionnés pour les épreuves régionales ! Et bon courage aux recalés, vous pourrez retenter votre chance en 2010 après avoir travaillé un peu...

Références

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein : *Introduction à l'algorithmique*. Dunod, 2002.

[2] Brian Hayes : The easiest hard problem. *American Scientist*, 90(2):113–117, mars–avril 2002. `http://www.americanscientist.org/issues/pub/the-easiest-hard-problem`.

[3] Stephan Mertens : The easiest hard problem : Number partitioning. *In* A.G. Percus, G. Istrate et C. Moore, éditeurs : *Computational Complexity and Statistical Physics*, pages 125–139, New York, 2006. Oxford University Press. `http://arxiv.org/abs/cond-mat/0302536`.