Savoir et savoir-faire en algorithmique

Christian.Kauth@epfl.ch, EPFL - président de PolyProg & Jonas.Wagner@epfl.ch, EPFL- membre du comité de PolyProg

Problem analysis, conception of a solution approach, followed by an efficient implementation – a job for real engineers. Three EPFL teams, trained by PolyProg, recently took up this challenge and proved their skills at a top level.

Décortiquer le problème, chercher une approche conduisant à la solution, implémenter celle-ci de manière efficace – un vrai travail d'ingénieur de A à Z. Ce fut le défi auquel se sont livrées récemment 3 équipes de l'EPFL entraînées par PolyProg, en faisant preuve d'excellente maîtrise.

Ce furent Andrei Giurgiu, Anton Dimitrov et Christian Kauth qui ont su assurer une respectable 18e place parmi 1'500 équipes autour du globe, pendant un marathon de 24 heures de concentration absolue dans le cadre du IEEEXteme 5.0 [1]. Juste un mois plus tard, deux équipes d'étudiants de l'EPFL, Johannes Wüthrich, Mihai Moraru, Przemyslaw Pietrzkiewicz, Cheng Zhong, Le Hung Tran et Cristian Talau, ont décroché des médailles de bronze au SWERC [2] à Madrid. Elles y furent préparées par le formateur de PolyProg, Christian Kauth et ses assistants Jonas Wagner et Andrei Giurgiu.

Sans pouvoir émuler ici l'atmosphère sérieuse, pleine de suspense et ardente d'un vrai concours, nous vous proposons de suivre le parcours qui mène d'un problème populaire du SWERC à sa solution. Une pensée analytique, l'ingéniosité et une bonne maîtrise d'un langage de programmation (nous choisissons le C++ ici) sont indispensables au succès. En voici l'énoncé:

> Ce fut le weekend des élections en Espagne et il s'agissait de distribuer les B *(B<2'000'000)* boîtes de vote parmi les N *(N<500'000)* villes du pays, de sorte à ce que la pire densité d'électeurs par boîte soit minimisée. À part N et B, on vous donne aussi le nombre d'électeurs par ville a_i, ne dépassant pour aucune ville M *(M≤5'000'000)*. Appelons b_i le nombre de boîtes attribuées à la ville i. Un minimum d'une boîte par ville est requis.

Ce problème, extrait de la vraie vie, admet plusieurs approches correctes d'efficacités différentes. Testez jusqu'où vous auriez su pousser la barre !

La force brute en *O(BN)*

Le problème est facile à résoudre : il faut juste donner les boîtes aux villes qui en ont le plus besoin:

```
Initialiser le nombre de boîtes: b_i ← 1 pour
chaque ville
Tant qu'il nous reste des boîtes:
    trouver la ville avec densité a_i/b_i maximale
    ajouter une boîte à cette ville:  b_i ← b_i + 1
```

Cette stratégie est gloutonne. Elle choisit à chaque étape la solution optimale, sans considérer les prochaines étapes. Les boîtes déjà attribuées ne sont jamais redistribuées. Malgré sa simplicité, elle donne le résultat correct. Pour prouver cela, constatons d'abord que le résultat est correct si $B=N$, car il faut donner une boite à chaque ville dans ce cas. Si nous ajoutons la boîte suivante à la ville avec le pire rapport habitants/boîtes, nous parlons de *densité* dans la suite, la solution reste optimale. Par induction, il s'ensuit que la solution est optimale pour chaque $B>N$.

Malheureusement, cette solution est trop lente pour marquer des points dans un concours: pour chaque boîte, il faut parcourir toutes les villes et trouver celle qui recevra la boîte. L'algorithme effectue donc $B*N$ opérations, soit un trillion d'opérations !

g.à.d. Anton Dimitrov, Christian Kauth, Andrei Giurgiu

Encore brut, mais structuré *O(B log(N))*

Nous pouvons améliorer notre algorithme, si à chaque distribution de boîte, nous trouvons la ville la plus dense plus rapidement. Il nous faut une structure de données qui permet d'accéder rapidement au plus grand élément d'une collection. Une telle structure s'appelle un **tas** *(heap)* ou une **file de priorité** *(priority queue)* [3]. Voilà notre algorithme modifié:

```
Initialiser le nombre de boîtes: b_i ← 1 pour
chaque ville
Insérer chaque ville dans une file de priorité Q
Tant qu'il nous reste des boîtes:
  extraire la ville avec densité a_i/b_i
  maximale de Q
  ajouter une boîte à cette ville: b_i ← b_i + 1
  réinsérer la ville dans Q
```

Les opérations d'extraction et d'insertion prennent un temps logarithmique en taille de la file. Du coup, le coût total de cet algorithme est d'ordre *O(B log(N))*. Ceci est nettement plus rapide, mais s'élève encore à une centaine de millions d'opérations.

Se poser la bonne question *O(N log(M))*

Faisons *tabula rasa*. Au lieu de créer la distribution optimale des boîtes pas par pas, nous allons simplement deviner quel est le nombre de personnes par boîte dans une distribution optimale. Cette stratégie n'est pas juste plus élégante, mais s'avère aussi plus efficace.
Appelons X_0 notre densité devinée. Se pose ensuite la question si une distribution respectant cette densité existe ou non ! La réponse à cette question est facile à calculer: Connaissant X_0, chaque ville nécessite $b_i = \left\lceil \frac{a_i}{X_0} \right\rceil$ boîtes. Si le nombre total de boîtes nécessaires ne dépasse pas B, il existe effectivement une telle distribution. Soyons plus ambitieux alors et essayons une nouvelle densité $X_1 \leq X_0$. Dans le cas contraire, tentons notre chance avec un objectif plus modeste $X_1 \geq X_0$.
Ceci nous donne une séquence des bornes pour lesquelles nous savons si elles sont trop ambitieuses ou faisables. Si nous faisons une recherche binaire sur X, chaque réponse coupe la taille de notre intervalle de recherche en deux et nous convergeons vers la plus petite densité qui admet une distribution faisable! Une fois l'idée trouvée, l'étape suivante consiste à formaliser l'algorithme sous forme de pseudocode.

```
Initialiser la borne inférieure X_min ← 0
Initialiser la borne supérieure X_max ← M+1
Tant que X_max - X_min > 1:
    X ← ceil(avg(X_min, X_max))
    Calculer le nombre de boîtes qu'il faudrait:
    B_necessaire = sum ceil(a_i/X)
    Si B_necessaire > B: X_max ← X
    Sinon  X_min ← X
X_min est maintenant la solution recherchée.
```

Vu qu'à chaque itération, la moitié des solutions possibles peut être écartée et qu'il faut calculer le nombre de boîtes pour chaque ville en chaque itération, la complexité totale de cette approche est *O(N log(M))* et elle vous aurait amené des points au SWERC ! Il est vrai que la limite inférieure aurait pu être initialisée à la population de toute l'Espagne $P=\sum a_i$, divisée par B. Est-ce que vous trouvez aussi une meilleure estimation pour la borne supérieure ?

Par hasard *O(N log(N))*

Terminons notre excursion des approches sur une solution extravagante, qui met à profit le hasard et déclasse toutes les approches précédentes ! L'observation de base est la suivante: si P est la population totale de l'Espagne, il faudrait idéalement attribuer 1 boîte par ville (c'est le minimum selon les règles du jeu) et distribuer les *(B-N)* boîtes restantes sur les villes, proportionnellement à leur population. Etant donné que les boîtes constituent des objets discrets, cette stratégie n'est pas applicable telle quelle, mais nous en tirons néanmoins le nombre minimal de boîtes entières pour chaque ville, $b_i = \left\lfloor \frac{a_i(B-N)}{P} \right\rfloor + 1$. Après cette distribution, il nous reste $R=(B-\sum b_i)$ boîtes pour améliorer la situation dans R villes au maximum. Il est garanti que $R<N$, puisqu'autrement, nous pourrions donner une boîte de plus à chaque ville ! Nous en concluons que la ville avec la $(R+1)$–ième plus grande densité (et toutes les villes avec densité inférieure) ne se verra attribuée plus aucune boîte supplémentaire et nous donne une borne inférieure à la pire densité. Ramassons ensuite de nouveau toutes les boîtes des R villes les plus denses et redistribuons-les, avec les R boîtes restantes, selon ce même schéma ! La fonction `solve` accomplit ceci de manière récursive. Dans le pire des cas, $R=N-1$, et `solve` est appelé N fois ! En pratique, R suit néanmoins une distribution uniforme, entre 0 et $N-1$ et l'espérance des appels récursifs de solve vaut *O(log(N))*. Après le pseudo-code vient l'implémentation en langage de programmation. Une somme $s = s_0 + \sum_0^N x_i$ s'écrit en C++ `S=accumulate(&x0,&xN,s0)` et la partie entière supérieure C d'une fraction de deux entiers a et b s'exprime comme `C = a/b + (a%b>0)`.

```
int64 solve(int64 N, int64 B) {
   P = accumulate(&a[0],&a[N],011);
   for (int i=0; i<N; i++)
      b[i] = (a[i]*(B-N))/P+1;
   if (N<=0) return 0;
   if (N==1) return a[0]/b[0]+(a[0]%b[0]>0);
   R = B-accumulate(&b[0],&b[N],011);
   N = nth_largest(0,N,R+1);
   B = accumulate(&b[0],&b[N],R);
   return
      max( a[N]/b[N]+(a[N]%b[N]>0), solve(N,B) );
}
```

Pour mener à bien cette approche, `solve` doit trouver de manière efficace la n-ième densité par appel à la fonction `nth_largest`. L'intuition de trier les villes selon leur densité en *O(N log(N))* et d'en choisir la n-ième ensuite nous tend un piège, car cette solution fait plus de travail que nécessaire. La bonne approche fonctionne comme suit: sélectionnons un pivot et réarrangeons les villes de sorte à ce que celles avec les densités supérieures à celle du pivot se trouvent à sa gauche et celles inférieures, à sa droite. Maintenant seulement la ville pivot est garantie d'être dans sa position ordonnée, que nous appelons m. Si $n=m$, nous avons trouvé ce que nous cherchions! Si m est inférieur à n, le n-ième élément est dans la partie de droite et à gauche si m est supérieur à n. Par récursivité sur la partie en question, nous trouvons finalement la n-ième ville la plus dense sans opportunité d'amélioration ! Le code ci-dessous rend les choses claires.

```
int64 partition(int64 left, int64 right, int64
pivot) {
   int64 swappos(left), i;
   swap(a[right-1],a[pivot]);
   swap(b[right-1],b[pivot]);
   for (i=left; i<right; i++)
      if (a[i]*b[right-1] > a[right-1]*b[i]) {
       swap(a[i],a[swappos]);
       swap(b[i],b[swappos]);
       swappos++;
      }
   swap(a[right-1],a[swappos]);
   swap(b[right-1],b[swappos]);
   return swappos;
}

int64 nth_largest(int64 left, int64 right, int64
N) {
   int64 pivot, rank;
   if (left==right-1) return left;
   pivot = left + rand()%(right-left);
   pivot = partition( left, right, pivot );
   rank = pivot-left+1;
   if (rank>N)
     return nth_largest( left, pivot, N);
   if (rank<N)
     return nth_largest( pivot+1, right, N-rank );
   return pivot;
}
```

g.à.d. Cheng Zhong, Cristian Talau, Le Hung Tran

L'utile théorème de Master [4] prédit une complexité *O(N)* en moyenne et *O(N^2)* dans le pire des cas pour la fonction `nth_largest`. D'une part, cinq grandes têtes (Blum, Floyd, Pratt, Rivest et Tarjan) y ont trouvé un remède élégant [5] qui garantit toujours une complexité linéaire, et d'autre part le fait d'inclure le hasard dans l'algorithme, rend l'occurrence du pire cas très improbable (1 chance sur *N* factorielle). Comme les juges du concours ne peuvent aucunement anticiper notre suite de pivots, il n'y a pas de test [6] pour lequel `nth_largest` ne serait pas de complexité linéaire, donc `solve` de complexité *O(N log(N))*. Dans l'hypothèse des éventualités, notre code dépasse le temps d'exécution limite. Il suffit alors de re-soumettre le même code une seconde fois, avec une initialisation aléatoire différente.

Comme vous avez pu constater, il existe souvent des approches très différentes à un même problème, se distinguant par leur facilité d'implémentation et efficacité. Dans le cadre d'un concours, le temps est une ressource limitée et les programmeurs sont toujours face au dilemme entre l'efficacité de leur approche et sa difficulté d'implémentation. C'est précisément à ce niveau qu'une excellente maîtrise du langage de programmation est un énorme atout. Les programmeurs expérimentés et rusés savent même tirer avantage des propriétés bénéfiques du hasard.

N'hésitez pas à faire vous-mêmes l'exercice de l'implémentation et testez votre solution sur un juge automatique [7]. Si vous avez trouvé du plaisir à stimuler vos neurones avec des problèmes similaires, faisant appel à votre créativité et à votre savoir-faire en algorithmique, formez vite votre équipe pour le **Helvetic Coding Contest** [8] et inscrivez-vous d'ores et déjà sous hc2.ch. PolyProg organisera cette 3e édition avec le soutien de l'EPFL, OpenSystems [9] et AdNovum [10]. Une centaine d'étudiants (de l'école secondaire jusqu'au doctorat) afflueront des quatre coins de la Suisse pour vous défier le **17 mars 2012**, ici au campus, dans une atmosphère conviviale et chacun trouvera des problèmes de son niveau !

g.à.d. Johannes Wüthrich, Mihai Moraru

Références

[1] IEEEXteme 5.0: concours algorithmique de 24 heures. www.ieee.org/membership_services/membership/students/competitions/xtreme/index.html.

[2] SWERC: Finales régionales sud-ouest européennes de l'International Collegiate Programming Contest. swerc.eu.

[3] File de priorité: fr.wikipedia.org/wiki/File_de_priorité

[4] Théorème de Master : Livre de cuisine pour la solution de relations récursives en termes de complexité.

[5] Un algorithme très similaire à celui présenté, qui garantit une réduction d'au moins 30% de l'espace de recherche à chaque itération en utilisant la médiane des médianes.

[6] Dans les concours, un juge automatique compile le code source et le teste contre différentes entrées. Des points sont ensuite attribués en fonction de l'exactitude de la réponse et la mémoire et temps CPU accaparés.

[7] Il s'agit du problem 12390 sur Uva online judge, uva.onlinejudge.org.

[8] Le premier concours de programmation suisse depuis 2010. hc2.ch.

[9] OpenSystems, sponsor de l'hc2 2012. open.ch.

[10] AdNovum, sponsor de l'hc2 2012. www.adnovum.ch.