N° d'anonymat:

Année Universitaire 2021-2022
Session 2 de printemps

Parcours/Étape: L3 Informatique **Code UE:** 4TIN602U
Épreuve: Techniques Algorithmiques et Programmation
Date: 19/04/2022 **Heure:** 11h30 **Durée:** 1h30
Documents: une seule feuille A4 recto-verso autorisée.
Épreuve de M. Cyril Gavoille

université de BORDEAUX

Collège Sciences et Technologie

RÉPONDRE DIRECTEMENT SUR LE SUJET
QUI EST À RENDRE DANS LA FEUILLE DOUBLE D'EXAMEN

Questions de cours

Pour chacune des questions suivantes, vous devez indiquer si l'affirmation est « vraie » ou « fausse » tout en justifiant votre réponse par une explication, un exemple ou un contre-exemple. Sans justification, pas de point.

Question 1. *Une instance particulière d'un problème indécidable peut être résolue par un programme* `C`.

Solution. **[3 pts]** Vrai. Par exemple, un programme `C` peut très bien lancer un autre programme (commande `system()`) et déterminer qu'il s'arrête avant 1 minute (ou avant avoir exécuté un certain nombre d'instructions), et en le bloquant après. Pour certaines instances du problème de la HALTE, cela répond à la question posée. De manière plus violante, chaque instance possède évidemment un programme `C` qui la résout : `return true` si elle est vraie, et `return false` sinon.

Question 2. *Un algorithme pour un problème donné peut boucler à l'infini sur certaines de ses instances.*

Solution. **[3 pts]** Faux. Selon la définition du cours, un algorithme doit fournir pour chaque instance d'un problème donné un résultat répondant à la question posée. S'il bouclait, il violerait cette définition.

Question 3. *Pour un problème donné, une heuristique a toujours une complexité en temps plus faible qu'un algorithme exact.*

Solution. **[3 pts]** On aimerait bien, mais c'est faux en général. Pour une heuristique, il n'y a aucune garantie ni pour sa qualité ni pour sa complexité en temps. En résumé, une heuristique peut être tout et n'importe quoi. Sa pourrait être bien plus grande que l'algorithme exact. Par exemple, pour trier un tableau on pourrait permuter aléatoirement les éléments jusqu'à ce que le tableau soit trié (c'est le *bogo sort* ou « tri-stupide »). S'il peut arriver par chance que cette heuristique soit plus rapide (meilleur cas), tout algorithme en $O(n \log n)$ sera plus efficace en pratique.

Question 4. *L'heuristique h dans A* peut être monotone et sous-estimer toutes les distances.*

Solution. **[3 pts]** Vrai. Si, par exemple, l'on pose $h(x, y) = 0$, c'est monotone et sous-estime toutes les distances. La distance vol-d'oiseau a aussi cette double propriété, mais cela concerne des graphes particuliers.

À propos du voyageur de commerce

Dans cette partie on supposera que pour le voyageur de commerce les points sont des points du plan et que la distance d entre deux points est la distance euclidienne.

Question 5. *Rappelez en quelques lignes le principe de l'algorithme* ApproxMST *vu en cours permettant de calculer une tournée pour toute entrée (V, d) du voyageur de commerce.*

SOLUTION. **[3 pts]** On construit un arbre T couvrant V et de poids minimum. Puis on effectue un parcours de T en profondeur d'abord depuis un sommet quelconque. La tournée est définie selon l'ordre de première visite de ces sommets lors de ce parcours.

Question 6. *Donner le facteur d'approximation de l'algorithme* ApproxMST.

SOLUTION. **[1 pts]** D'après le cours, c'est 2.

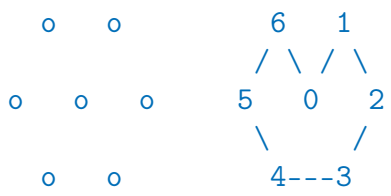
Question 7. *Ce facteur d'approximation est-il valable pour d'autres fonctions de distance $d' \neq d$, c'est-à-dire différente de la distance euclidienne ? Justifiez.*

SOLUTION. **[2 pts]** Oui, mais à condition que d' vérifie l'inégalité triangulaire. C'est le cas par exemple de la distance de Manhattan ou encore la distance du roi sur un échiquier. Cela ne sera plus forcément le cas si, par exemple, $d' = d^2$.

On considère maintenant un ensemble V_7 particulier contenant 7 points, $V_7 = \{v_0, v_1, \ldots, v_6\}$, placés aux sommets d'un hexagone régulier de côté unité avec un point placé au centre. Plus précisément, on construit V_7 à partir d'un point arbitraire v_0, en traçant un cercle de centre v_0 et de rayon unité, et en plaçant successivement et de manière régulière sur le cercle les 6 points $v_1, v_2, \ldots, v_6$ dans cet ordre. Il est important de remarquer que chaque triplet de points $(v_0, v_i, v_{(i \bmod 6)+1})$, $1 \leqslant i \leqslant 6$, forme un triangle équilatéral.

Question 8. *Dessinez les points de V_7 ainsi qu'une tournée de longueur optimale pour (V_7, d). Prenez ce que vous voulez comme unité. On demande bien sûr un dessin approximatif.*

SOLUTION. **[3 pts]**



Comme tournée on peut prendre le cycle $v_0 - v_1 - \cdots - v_6 - v_0$.

Dans la suite on notera $\text{OPT}(V_7, d)$ la longueur optimale de la tournée pour (V_7, d).

Question 9. *Calculez la longueur théorique de votre tournée et prouvez quelle est de longueur optimale, c'est-à-dire de longueur* $\text{OPT}(V_7, d)$.

SOLUTION. **[3 pts]** La tournée possède 7 arêtes de longueur unité, ce qui fait une longueur de 7 au total. Montrons qu'elle est optimale. On a $d(v_i, v_j) \geqslant 1$ pour tout $v_i, v_j \in V_7$, $i \neq j$, et aussi que $|V_7| = 7$. La tournée optimale est donc de longueur $\text{OPT}(V_7, d) \geqslant |V_7| \times \min_{i \neq j} d(v_i, v_j) = 7$. La tournée choisie est donc de longueur optimale.

Question 10. *Donnez le principe d'un algorithme permettant de calculer un arbre couvrant de poids minimum.*

SOLUTION. **[3 pts]** On trie par ordre croissant les arêtes du graphe complet valué par la longueur des arêtes. Puis on les ajoute dans l'ordre, en sautant celles formant un cycle, jusqu'à former un arbre couvrant. C'est l'algorithme de Kruskal.

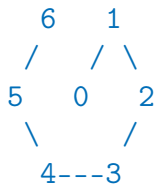
Question 11. *Donnez deux arbres (différents) couvrant V_7 et qui soient de poids minimum. Dessinez les et précisez leurs poids.*

SOLUTION. **[3 pts]** Par exemple, (1) une étoile de centre v_0 ; ou (2) un chemin parcourant les points $v_0 - v_1 - \cdots - v_6$ dans cet ordre. Dans les deux cas, leurs poids est de 6 ce qui est le minimum car il faut 6 arêtes et le poids de toute arête $v_i - v_j$ est toujours au moins 1.

Étant donné un arbre T couvrant V_7 et un point $v_i \in V_7$, on note $\ell(T, v_i)$ la longueur de la tournée pour (V_7, d) obtenue en visitant tous les points de V_7 selon l'ordre de leur première visite lors d'un parcours en profondeur d'abord quelconque de T à partir du point v_i. Attention ! Il y a plusieurs parcours possibles de T depuis v_i, notamment si v_i à plusieurs voisins dans T. La longueur $\ell(T, v_i)$ représente donc la tournée obtenue selon le pire des parcours en profondeur possibles de T en partant de v_i.

Question 12. *Montrez qu'il existe un arbre T tel que $\ell(T, v_0) = \text{OPT}(V_7, d)$, c'est-à-dire un arbre T couvrant V_7 tel que l'ordre de première visite de T depuis v_0 selon n'importe quel parcours en profondeur d'abord donne toujours une tournée de longueur optimale. Dessinez T et la tournée correspondante. Précisez l'ordre de visite des points. Justifiez.*

SOLUTION. **[3 pts]** On considère le chemin $v_0 - v_1 - \cdots - v_6$ couvrant V_7 comme ci-dessous.



Le parcours en profondeur depuis v_0 est unique car chaque sommet de T, enraciné en v_0, n'a qu'au plus un fils. La tournée résultante est $v_0 - v_1 - \cdots - v_6 - v_0$, soit une longueur de $6 + 1 = 7$ ce qui est optimale d'après les questions précédentes.

On admettra sans preuve que la plus grande diagonale d'un losange obtenu en collant deux triangles équilatéraux de côté unité, vaut $\sqrt{3} > 3/2$.

Question 13. *Montrez qu'il existe un arbre T' de poids minimum tel que $\ell(T', v_0) \geqslant 8 + 2\sqrt{3}$. Justifiez.*

SOLUTION. **[3 pts]** Choisir T' comme une étoile de centre v_0. Il est de poids 6 ce qui est le minimum. On visite les points de T' suivant le DFS suivant : $v_0 - v_1 - v_4 - v_6 - v_3 - v_5 - v_2 - v_0$. Ce parcours n'est pas « planaire » dans le sens où il comporte des croisements. La longueur de la tournée résultante est de $1 + 2 + \sqrt{3} + 2 + \sqrt{3} + 2 + 1 = 8 + 2\sqrt{3}$, ce qui montre que $\ell(T', v_0) \geqslant 8 + 2\sqrt{3}$.

Question 14. *En déduire qu'il existe une instance (V, d) pour laquelle l'algorithme ApproxMST a un facteur d'approximation $> 11/7$. Justifiez.*

SOLUTION. **[3 pts]** Lorsqu'on exécute ApproxMST sur (V_7, d) il est possible que l'arbre de poids minimum soit T' et que son parcours produise une tournée de longueur $\ell(T', v_0) \geqslant 8 + 2\sqrt{3}$. Or $\text{OPT}(V_7, d) = 7$. Donc le facteur d'approximation est au moins $\ell(T', v_0)/\text{OPT}(V_7, d) \geqslant (8 + 2\sqrt{3})/7 > (8 + 2 \cdot 3/2)/7 = 11/7$.

Le nombre de sous-ensembles

On s'intéresse à certains sous-ensembles de $\{1, \ldots, n\}$ où $n \geqslant 1$ est un entier. Plus précisément, on s'intéresse à tous ceux ayant exactement k éléments, avec bien sûr $0 \leqslant k \leqslant n$. On notera $b(n, k)$ leur nombre. Dit autrement, $b(n, k)$ représente le nombre de sous-ensembles de $\{1, \ldots, n\}$ à k éléments.

Question 15. *Donnez tous les sous-ensembles de $\{1, 2, 3, 4\}$ à 2 éléments. En déduire $b(4, 2)$.*

SOLUTION. **[3 pts]** Les sous-ensembles sont : $\{1,2\}$, $\{1,3\}$, $\{1,4\}$, $\{2,3\}$, $\{2,4\}$, $\{3,4\}$. Du coup $b(4,2) = 6$.

Question 16. *En remarquant que dans un graphe complet sur n points (ou clique à n sommets), il y a autant d'arêtes que de paires de points, donnez une formule close pour $b(n,2)$.*

SOLUTION. **[3 pts]** Comme vu dans le cours, le graphe complet possède $n(n-1)/2$ arêtes. Par conséquent $b(n,2) = n(n-1)/2$.

Afin d'établir une formule de récurrence pour $b(n,k)$, on remarque qu'il y a deux catégories de sous-ensembles de $\{1,\ldots,n\}$ à k éléments, en supposant $0 < k < n$:

- Ceux qui possèdent n : il y en a exactement $b(n-1,k-1)$, car pour obtenir un sous-ensemble de cette catégorie il suffit de prendre un sous-ensemble de $\{1,\ldots,n-1\}$ à $k-1$ éléments et d'y ajouter n.
- Ceux qui ne possèdent pas n : il y en a exactement $b(n-1,k)$, car tout sous-ensemble de $\{1,\ldots,n-1\}$ à k éléments appartient à cette catégorie.

Pour tous les autres cas, c'est-à-dire lorsque $k = 0$, $n = 1$ ou $n = k$, on peut vérifier facilement que $b(n,k) = 1$.

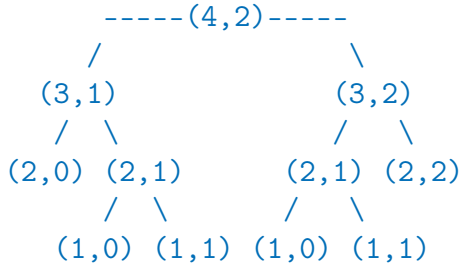
Question 17. *En vous appuyant sur la discussion précédente, écrire en* `C` *une fonction récursive* `long b_rec(int n, int k)` *renvoyant l'entier $b(n,k)$ pour des entiers n,k tels que $n \geqslant 1$ et $0 \leqslant k \leqslant n$.*

SOLUTION. **[3 pts]** Il faut noter que si $n = 1$, alors $k = 0$ ou $k = n$.

```
long b_rec(int n, int k){
      if( k==0 || k==n ) return 1; // formule si k = 0 ou k = n
      return s_rec(n-1,k-1) + s_rec(n-1,k); // formule si 0 < k < n
    }
```

Question 18. *Construisez l'arbre des appels pour* `b_rec(4,2)` *en observant qu'il possède 6 feuilles.*

SOLUTION. **[3 pts]**



On admettra sans preuve que $b(n,k) = \Theta(2^n/\sqrt{n}\,)$, lorsque $k = \lfloor n/2 \rfloor$.

Question 19. *En analysant le nombre de sommets de l'arbre des appels, en particulier son nombre de feuilles, montrez que la complexité de* `b_rec(n,k)` *peut-être exponentielle en n.*

SOLUTION. **[3 pts]** D'après le code, la complexité de `b_rec(n,k)` est en $\Theta(N)$ où N est le nombre de sommets de l'arbre des appels. Le nombre de feuilles est précisément $b(n,k)$, car la seule valeur de retour est 1 et la valeur renvoyée, $b(n,k)$, est la somme des valeurs de retour. L'arbre étant binaire, le nombre de sommets vaut donc $N = 2b(n,k) - 1$. La complexité, pour $k = \lfloor n/2 \rfloor$, vaut donc $\Theta(2^n/\sqrt{n}\,)$ ce qui est exponentielle en n.

Question 20. *Montrez que pour chaque n suffisamment grand, l'arbre des appels de* `b_rec(n,n/2)` *contient plusieurs fois les mêmes nœuds internes, c'est-à-dire des sous-appels qui ne sont pas des feuilles et avec les mêmes paramètres.*

SOLUTION. **[3 pts]** Chaque nœuds de l'arbre est un appel de la forme (i, j) avec $i \in \{1, \ldots, n\}$ et $j \in \{0, \ldots, k\}$. Cela fait donc au plus $n \times (k+1) < (n+1)^2$ nœuds différents. Pour $k = \lfloor n/2 \rfloor$, l'arbre possède $b(n, k) - 1 = \Theta(2^n/\sqrt{n})$ sommets internes ce qui est bien plus grand que $(n+1)^2$, lorsque n est assez grand. Il y a donc des nœuds internes qui apparaissent plusieurs fois.

Un autre argument inspiré de la question 18, tout aussi convainquant, est de regarder ce qu'il se passe pour les deux premiers niveaux de l'arbre des appels pour `b_rec(n,k)` :

```
        -----(n,k)-----
       /               \
   (n-1,k-1)          (n-1,k)
    /   \              /   \
... (n-2,k-1)    (n-2,k-1) ...
```

puis de remarquer que les nœuds $(n-2, k-1)$ se répètent, pour chaque n, k suffisament grands (il faut précisément $0 \leqslant k-1 \leqslant n$).

Pour supprimer les calculs inutiles de `b_rec(n,k)`, et être beaucoup plus efficace, on va utiliser la technique de mémorisation (ou de programmation dynamique).

Question 21. *Écrire en* `C` *une nouvelle fonction non-récursive* `b_prog_dyn(n,k)` *renvoyant* $b(n, k)$ *à l'aide d'un algorithme basé sur la programmation dynamique, c'est-à-dire utilisant une table de mémorisation. (Vous pouvez aussi vous contenter de présenter, avec suffisamment de détails, le principe de votre algorithme.)*

SOLUTION. **[4 pts]** Principe. On peut utiliser une table $B[1..n][0..k]$ (n lignes de $k+1$ colonnes) pour stocker chaque $b(i, j)$ pour $i = 1..n$ et $j = 0..k$ qu'on remplit ligne par ligne. Cela donne par exemple pour $n = 5$ et $k = 3$:

```
   B            k
    \j=0  1  2  3 | 4  5
   --+------------+-----
i=0 | 1           |
  1 | 1  1        |
  2 | 1  2  1     |
  3 | 1  3  3  1  |
  4 | 1  4  6  4  | 1
n=5 | 1  5 10 10  | 5  1
   --+------------+-----
```

Mais on remarque que la récurrence $b(n, k) = b(n-1, k-1) + b(n-1, k)$, comme celle du triangle de Pascal (d'ailleurs c'est le triangle de Pascal), ne fait intervenir que la ligne précédente de la table B. On peut donc ne mémoriser que la ligne courante et la ligne précédente. De plus, il est inutile d'aller au-delà de la colonne k.

Plus précisément, on utilise deux tableaux de $k+1$ cases. On fait varier un indice $i = 1, ..., n$ (les lignes). Un des tableaux, P, contiennent toutes les valeurs $P[j] = b(i-1, j)$ pour tout $j = 0, ..., k$. L'autre, B, contient toutes les valeurs $b(i, j)$ que l'on calcule avec la formule récursive $B[j] = b(i-1, j-1) + b(i-1, j) = P[j-1] + P[j]$. Pour passer à la ligne suivante, on incrémente i puis on échange P et B, ce qui évite la recopie. Il faut faire attention aux bords, notamment lorsque $i \leqslant k$.

```
long b_prog_dyn(int n, int k){
      long T1[k+1], T2[k+1]; // deux lignes d'au plus k+1 cases
      long *P=T1, *B=T2, *t; // P=ligne précédente, B=ligne courante
      int jmax; // calcule jusqu'à la colonne jmax
      B[0]=P[0]=1; // b(i,0)=1, cases seulement lues
      for(int i=1; i<=n; i++){ // pour chaque ligne i
        if(i>k) jmax=k; else jmax=i-1, B[i]=1; // b(i,i)=1
        for(int j=1; j<=jmax; j++) B[j] = P[j-1] + P[j]; // pour chaque colonne j
        t=B, B=P, P=t; // échange P et B
      }
      return P[k];
    }
```

On pouvait également remarquer que $b(n,k) = b(n,n-k)$ et donc optimiser un peu, ce qui n'était pas demandé, avec une ligne placée en tête : `if(k<n-k) k=n-k;`

On pouvait également utiliser une version plus proche de la fonction originale récursive et utilisant la mémorisation. Si elle est plus simple, elle est cependant plus gourmande en mémoire, $O(nk)$ au lieu de $O(k)$.

```
long b_prog_dyn2(int n, int k){
      static long B[n+1][k+1]; // variable globale
      for(i=0;i<=n;i++) // initialisation de la table B[][]
        for(j=0;j<=k;j++)
          B[i][j]=-1;
      return b_mem(n,k);
    }

    long b_mem(int n, int k){ // comme b_rec() ou presque
      if( k==0 || k==n ) return 1;
      if(B[n][k]<0) B[n][k] = b_mem(n-1,k-1) + b_mem(n-1,k);
      return B[n][k];
    }
```

Question 22. *Quelles sont les complexités en temps et en espace de votre fonction* `b_prog_dyn(n,k)` *exprimée en fonction de n et k ?*

SOLUTION. **[2 pts]** Elle utilise $O(k)$ cases mémoires et a une complexité de $O(nk)$ opérations arithmétiques.

FIN.