N° d'anonymat:

Année Universitaire 2021-2022
Session 1 de printemps

Parcours/Étape: L3 Informatique **Code UE:** 4TIN602U
Épreuve: Techniques Algorithmiques et Programmation
Date: 14/06/2022 **Heure:** 9h00 **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, il est impératif de justifiez vos réponses par une explication, un exemple ou un contre-exemple. Sans justification de votre réponse, pas de point.

Question 1. *Il peut arriver que la complexité en espace d'un algorithme soit plus grande (en ordre de grandeur) que sa complexité en temps. Vrai ou Faux ? Justifiez.*

Solution. **[3 pts]** Faux. En temps T un algorithme ne peut consommer qu'au plus T mots mémoires. Donc la complexité en espace est toujours au plus celle en temps.

Question 2. *Dans l'algorithme A*, si l'heuristique h est monotone, alors la qualité de la solution produite par A* est garantie. Vrai ou Faux ? Justifiez.*

Solution. **[3 pts]** Vrai. Une proposition du cours affirme que si h est monotone, alors A* trouve nécessairement le plus court chemin entre s et t, s'il existe.

Question 3. *Dans l'algorithme A*, l'heuristique h peut être monotone et surestimer toutes les distances. Vrai ou Faux ? Justifiez.*

Solution. **[3 pts]** Vrai. Si, par exemple, l'on pose $h(x,y) = K$. C'est monotone car $h(x,t) \leqslant \omega(x,y) + h(y,t) \Leftrightarrow 0 \leqslant \omega(x,y)$ et cela peut surestimer toutes les distances en posant par exemple $K = \max_{x,y} \mathrm{dist}_G(x,y)$.

Question 4. *Un programme P implémentant correctement un algorithme A peut boucler à l'infini. Vrai ou Faux ? Justifiez.*

Solution. **[3 pts]** Faux. Aucun algorithme ne peut boucler à l'infini, en particulier A. Donc si P est correct pour A, il ne peut pas boucler.

Question 5. *Un algorithme d'approximation peut boucler à l'infini sur certaines instances. Vrai ou Faux ? Justifiez.*

Solution. **[3 pts]** Faux. La solution produite par de tels algorithmes doivent avoir une garantie sur chaque entrée, ce qui ne serait pas le cas s'il bouclait à l'infini sur l'une d'entres elles.

Question 6. *Soit (V,d) une instance du* Voyageur de commerce *telle que, pour tout $u,v \in V$, $d(u,v) = 0$ si $u = v$, et $d(u,v) = K$ sinon, où K est une constante > 0. Quel est alors le facteur d'approximation de l'algorithme* ApproxMST *appliqué à (V,d) ? On rappelle que* ApproxMST *est basé sur le calcul d'un arbre couvrant de poids minimum et d'un parcours en profondeur d'abord de l'arbre.*

SOLUTION. **[3 pts]** Le facteur d'approximation sera 1 dans ce cas. En effet, les distances étant toutes égales, toute tournée sera de longueur $|V| \cdot K$, ce qui est optimale puisque deux points distincts de V sont toujours à distance au moins K.

Question 7. *Soit d la distance euclidienne entre deux points du plan. Alors, la fonction d^2, définie par $d^2(u,v) = (d(u,v))^2$ pour tout points $u, v \in \mathbb{R}^2$, vérifie elle aussi l'inégalité triangulaire. Vrai ou Faux ? Justifiez.*

SOLUTION. **[3 pts]** Faux. Par exemple, on peut construire un triangle du plan avec $AB = 5$ et $AC = CB = 3$. On a alors $AB = 5 \leqslant AC + CB = 3 + 3 = 6$ et pourtant $AB^2 = 25 > AC^2 + CB^2 = 9 + 9 = 18$.

Question 8. *Soit (V, d) une instance du* VOYAGEUR DE COMMERCE *telle que d vérifie l'inégalité triangulaire, et soit u_0 un point quelconque de V. Alors, les tournées produites par l'algorithme glouton depuis u_0 sur les instances (V, d) et (V, d^2) sont forcément identiques, d^2 étant définie comme la distance d au carré (cf. la question précédente). Vrai ou Faux ? Justifiez.*

SOLUTION. **[3 pts]** Vrai. Car le point le plus proche selon d ou d^2 sera le même. C'est d'ailleurs vrai pour toute fonction $d'(u,v) = f(d(u,v))$ où f est une fonction croissante. Cela n'est pas lié à l'inégalité triangulaire.

Détection de permutation

Dans le problème du VOYAGEUR DE COMMERCE nous avons vu que de trouver une tournée de faible longueur pour chaque instance (V, d) revenait à calculer une permutation P des indices des n points de V telle que la longueur $\sum_{i=0}^{n-1} d(V[P[i]], V[P[(i+1) \bmod n]])$ soit la plus petite possible. Comme vu en TD, on représente la permutation P par un simple tableau de n indices, $P[i]$ étant l'indices du i-ème point visité.

Plus formellement, pour qu'un tableau T à n éléments soit une permutation de l'ensemble $\{0, \ldots, n-1\}$, ensemble qu'on notera $[0, n[$ dans la suite pour faire plus court, il faut et il suffit que chaque élément de $[0, n[$ apparaissent dans T une et une seule fois.

Dans cette partie on s'intéresse au problème de savoir si oui ou non un tableau T de n entiers représente une permutation de $[0, n[$. Attention ! Il n'y a pas a priori de condition particulière sur les éléments de T, sinon que ce sont des `int` et que $n \geqslant 1$.

Dans l'exemple ci-dessous, `T1` est une permutation, alors que `T2` ne l'est pas.

```
int T1[] = { 4, 1, 2, 0, 3, 5};
int T2[] = { 2, 0, 5, -1, 2};
```

Question 9. *Donnez toutes les raisons montrant que `T2` n'est pas une permutation.*

SOLUTION. **[3 pts]** 5 et -1 ne devraient pas être dans T_2, il y manque 1, 3 et 4, et enfin 2 y figure deux fois.

Question 10. *Pour simplifier l'écriture des prochains algorithmes, écrivez le code d'une fonction* `bool equal(int T1,int T2,int n)` *renvoyant* `true` *si les n premiers éléments des tableaux* `T1` *et* `T2` *sont identiques, et renvoyant* `false` *sinon.*

SOLUTION. **[3 pts]**

```
bool equal(int T1,int T2,int n){
      for(int i=0; i<n; i++)
        if(T1[i]!=T2[i]) return false;
      return true;
    }
```

Un stagiaire en informatique propose un premier algorithme pour la détection d'une permutation, appelé `check_v1(T,n)`. Son principe consiste à parcourir toutes les permutations P de $[0,n[$ et de renvoyer `true` si l'on détecte que $P = T$, plus précisément si a un moment donné `equal(P,T,n)` renvoie `true`.

Question 11. *Donnez de code de la fonction* `bool check_v1(int T,int n)`*, devant renvoyer* `true` *si et seulement si* T *est une permutation de* $[0,n[$.

Vous pourrez utiliser la fonction `next_permutation(P,n)`*, de complexité* $O(n)$*, renvoyant dans* P *la prochaine permutation dans l'ordre lexicographique. De plus la fonction renvoie* `true` *si la prochaine permutation a pu être déterminée, c'est-à-dire si avant l'appel* P *n'était pas la dernière permutation. On rappelle que la première permutation dans l'ordre lexicographique est* `P={0,1,2,...,n-1};`

SOLUTION. **[3 pts]**

```
bool check_v1(int T,int n){
      int P[n];
      for(int i=0; i<n; i++) P[i]=i;
      do{ if(equal(P,T,n)) return true; }
      while next_permutation(P,n);
      return false;
    }
```

Question 12. *Donnez la complexité en temps de* `check_v1()`.

SOLUTION. **[3 pts]** D'après le code c'est $O(n \cdot n!) = O((n+1)!)$ car `equal()` et `next_permutation()` prennent $O(n)$ alors que le nombre de répétitions dans la boucle `do{...}while()` est $n!$ puisqu'il y a un total de $n!$ permutations à tester dans le pire des cas.

Un deuxième stagiaire propose une autre fonction, `check_v2(T,n)`, qui teste simplement si chaque indice $j \in [0,n[$ se trouve effectivement bien une et une seule fois dans T. En la programmant, il s'aperçoit même qu'il suffit de tester si chaque indice se trouve au moins une fois dans T.

Question 13. *Donnez le code de* `bool check_v2(int T,int n)`.

SOLUTION. **[3 pts]**

```
bool check_v2(int T,int n){
      for(int j=0; j<n; j++){
        int found=false;
        for(int i=0; i<n; i++){
          if(T[i]==j){ found=true; break; }
        }
        if(!found) return false;
      }
      return true;
    }
```

Question 14. *Donnez la complexité en temps de* `check_v2()`.

SOLUTION. **[3 pts]** Il y a deux boucles de taille n, soit $O(n^2)$.

Un troisième stagiaire, plus expérimenté, propose une approche diviser-pour-régner. Pour sa fonction `check_v3(T,n)` il crée deux tableaux auxiliaires, T_1 et T_2 à peu près de même taille, où il y range respectivement tous les éléments de T inférieurs à $n/2$ et tous ceux supérieurs ou égaux à $n/2$.

Plus précisément, il pose $m = \lfloor n/2 \rfloor$ (en faisant simplement `int m=n/2;` en `C`). Dans T_1, il range tous les éléments de T qui sont $< m$, les uns à la suite des autres sans ordre particulier. Si tout se passe bien T_1 devrait être de taille exactement m. Dans T_2, il range tous les autres (donc ceux $\geqslant m$) mais en retranchant m avant de les stocker dans T_2. De cette façon, il observe que si T est une permutation, alors T_1 de taille m contient tous les indices de $[0, m[$ une et une seule fois, et que T_2 de taille $n-m$ contient tous les indices de $[0, n-m[$ une et une seule fois. Dit autrement, si T est une permutation de $[0, n[$ alors nécessairement T_1 est une permutation de $[0, m[$ et T_2 est une permutation de $[0, n-m[$. Et si T n'est pas une permutation, c'est qu'un incident dans le remplissage de T_1 ou T_2 s'est produit. Par exemple, lorsqu'il y a trop d'éléments dans T_1 et pas assez dans T_2 ou le contraire, ce qui va arriver dès qu'il manque un indice ou qu'un indice apparaît deux fois.

Question 15. *Donnez le code récursif de* `bool check_v3(int T,int n)` *en remarquant que le cas* $n = 1$ *est trivial.*

SOLUTION. **[3 pts]**

```
bool check_v3(int T,int n){
      if(n==1) return T[0]==0;
      int m=n/2;
      int T1[m],   i1=0;
      int T2[n-m], i2=m;
      for(int i=0; i<n; i++){
        if(T[i]<m) T1[i1++]=T[i];
              else T2[i2++]=T[i]-m;
        if( i1==m || i2==n ) return false;
      }
      return check_v3(T1,m) && check_v3(T2,n-m);
    }
```

Question 16. *D'après votre code, donnez une équation de récurrence que vérifie la complexité en temps de* `check_v3()`. *En déduire cette complexité.*

SOLUTION. **[3 pts]** Soit $t(n)$ la complexité en temps de `check_v3(T,n)`. D'après le code $t(n) \leqslant 2t(n/2) + cn$, pour une certaine constante $c > 0$. D'après le Master Théorème, $t(n) = O(n \log n)$.

L'approche diviser-pour-régner suggère à un autre stagiaire l'algorithme du tri-fusion et donc une

approche basée sur le tri. La version `check_v4(T,n)` consiste alors simplement à trier T par ordre croissant, disons avec une routine rapide de `<stdlib.h>` comme `qsort(T,n,sizeof(int),fcmp)`, puis à comparer le résultat avec la permutation `P={0,1,2,...,n-1}`. Notez que dans cette version, on s'autorise à modifier T.

Question 17. *Donnez le code de* `bool check_v4(int T,int n)` *en supposant déjà programmée une fonction* `fcmp()` *de comparaison d'entiers.*

SOLUTION. **[3 pts]**

```
bool check_v4(int T,int n){
      qsort(T,n,sizeof(int),fcmp);
      int P[n]; for(int i=0; i<n; i++) P[i]=i;
      return equal(T,P,n);
    }
```

Question 18. *En notant $q(n)$ la complexité en temps de la routine de tri (fonction* `qsort()`*) appliquée à un tableau de n éléments, donnez la complexité en temps de* `check_v4()`*.*

SOLUTION. **[3 pts]** D'après le code, c'est $O(q(n)+n)$.

Un cinquième stagiaire propose encore une autre approche, plus minimaliste qui répond à la question à l'aide d'un seul balayage de T. Supposons que l'on lise l'entier $j = T[i]$. On se sert d'un tableau auxiliaire, disons A, pour indiquer que l'entier j a été rencontré dans T. Plus précisément, on passe $A[j]$ à vrai si l'on rencontre j dans T pour la première fois. Il faut bien sûr initialiser correctement A et faire attention que j soit un indice possible pour A.

Question 19. *Donnez le code de* `bool check_v5(int T,int n)`*.*

SOLUTION. **[3 pts]**

```
bool check_v5(int T,int n){
      bool A[n];
      for(int i=0; i<n; i++) A[i]=false;
      for(int i=0; i<n; i++){
        int j=T[i];
        if( j<0 || j>=n || A[j] ) return false;
        A[j]=true;
      }
      return true;
    }
```

Question 20. *Donnez la complexité de* `check_v5()`*.*

SOLUTION. **[3 pts]** D'après le code, c'est $O(n)$.

Question 21. *L'encadrant de tous ces stagiaires leur suggère l'existence d'une version toute aussi rapide que* `check_v5()`*, mais sans l'utilisation de tableau auxiliaire, l'idée étant d'essayer de placer les éléments de T à leur bonne place ... Décrivez les principes de cette version ultime* `check_v6()`*.*

SOLUTION. **[3 pts]** Comme indiqué, l'idée est d'essayer de placer les éléments de T à leur bonne place, c'est-à-dire avec $T[i] = i$. Plus précisément, on balaye T en commençant à l'indice $i = 0$. On positionne i sur le premier indice tel que $T[i] \neq i$, c'est-à-dire qui ne soit pas à la bonne place. Posons $j = T[i]$. On échange $T[i]$ et $T[j]$. Maintenant, si tout c'est bien passé, il y a un nouvel élément bien placé : j. On recommence alors avec l'indice i, qu'on incrémente donc que si $T[i] = i$. On aura fini, bien sûr, dès qu'on arrive à $i = n$.

Il faut faire attention au cas où T n'est pas une permutation. Lorsqu'on lit $j = T[i]$ et que $j \neq i$, il faut bien vérifier que $j \in]i, n[$. Sinon, c'est qu'il y a un problème car on ne pourra pas l'inverser avec l'élément $T[j]$ situé après $T[i]$ dans T. Il faut aussi vérifier que $T[j]$ ne soit pas déjà égale à j, sinon on pourrait ne jamais pouvoir incrémenter i.

Cela va prendre évidemment un temps $O(n)$ puisqu'à chaque lecture, un nouvel élément est bien placé.

En terme de code (non demandé) cela pourrait être ceci :

```
bool check_v6(int T,int n){
      int i=0,j,tmp;
      while(i<n){
        j=T[i];
        if(j==i){ i++; continue; }
        if( j<i || j>=n || T[j]==j ) return false
        SWAP(T[i],T[j],tmp);
      }
      return true;
    }
```

FIN.