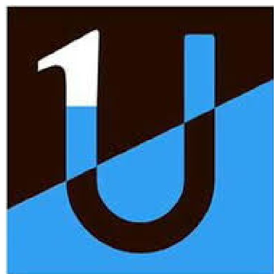
N° D'ANONYMAT:



ANNÉE UNIVERSITAIRE 2022-2023
SESSION 1 DE PRINTEMPS

Parcours/Étape: L3 Informatique **Code UE:** 4TIN602U
Épreuve: Techniques Algorithmiques et Programmation
Date: 13/06/2023 **Heure:** 9h00 **Durée:** 1h00
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

SOLUTION. **[Total de points : 30 pts]**

Question de cours

Question 1. *Parmi les notions suivantes, quelles sont celles qui ont été abordées en CM ou en TD ? [Cochez la ou les cases correspondantes.]*

- ☐ *le Master Theorem*
- ☐ *les algorithmes de Deep Learning*
- ☐ *les algorithmes distribués*
- ☐ *la compilation*
- ☐ *l'indécidabilité*
- ☐ *la théorie de la relativité (restreinte)*
- ☐ *la programmation dynamique*
- ☐ *les heuristiques*
- ☐ *les algorithmes parallèles*
- ☐ *les modèles génératifs de langages*
- ☐ *les algorithmes probabilistes*
- ☐ *les tables de hachage*
- ☐ *la logique temporelle*
- ☐ *la programmation par contraintes*

SOLUTION. **[7 pts]** Master Theorem, l'indécidabilité, la programmation dynamique, les heuristiques, les algorithmes probabilistes, les tables hachage (-1 pts par erreur).

Le nombre de chaînes binaires

On s'intéresse aux chaînes binaires de longueur n ayant exactement k bits à un, pour un certain entier $k \in [0, n]$. Par exemple, `001010111010` est une telle chaîne binaire avec $n = 12$ et $k = 6$. On notera $s(n, k)$ leur nombre. Dit autrement, $s(n, k)$ représente le nombre de chaînes binaires de longueur n avec k bits à un (et donc $n - k$ bits à zéros).

Question 2. *Donnez toutes les chaînes binaires de longueur $n = 4$ avec $k = 2$ bits à un. En déduire $s(4, 2)$.*

SOLUTION. **[3 pts]** $0011, 0101, 0110, 1001, 1010, 1100$. Du coup $s(4, 2) = 6$.

Question 3. *En remarquant qu'il y a autant de chaînes binaires de longueur n avec 2 bits à un que de façons de choisir 2 éléments dans un ensemble de taille n, donnez une formule close pour $s(n, 2)$.*

SOLUTION. **[3 pts]** $s(n, 2) = \binom{n}{2} = n(n-1)/2$.

Afin d'établir une formule de récurrence pour $s(n, k)$, on remarque qu'il y a deux catégories de chaînes binaires de longueur n avec k bits à un, en supposant $0 < k < n$:

- Celles dont le premier bits est à 1 : il y en a exactement $s(n-1,k-1)$, car pour obtenir une telle chaîne il suffit d'en prendre une de longueur $n-1$ avec $k-1$ bits à 1 et de la précéder d'un 1.
- Celles dont le premier bits est à 0 : il y en a exactement $s(n-1,k)$, car pour obtenir une telle chaîne il suffit d'en prendre une de longueur $n-1$ avec k bits à 1 et de la précéder d'un 0.

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

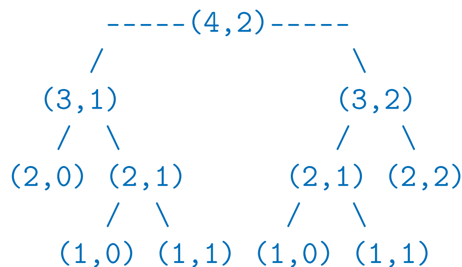
Question 4. *En vous appuyant sur la discussion précédente, écrire en* `C` *une fonction récursive* `long s_rec(int n, int k)` *renvoyant l'entier $s(n,k)$ pour des tous les entiers n,k tels que $n \geqslant 1$ et $0 \leqslant k \leqslant n$.*

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

```
long s_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 5. *Construisez l'arbre des appels pour* `s_rec(4,2)` *en observant qu'il possède 6 feuilles.*

Solution. **[3 pts]**



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

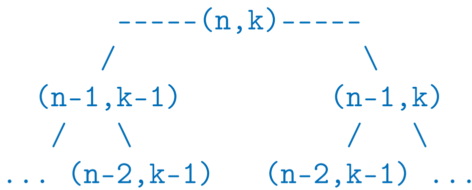
Question 6. *En analysant le nombre de sommets de l'arbre des appels, en particulier son nombre de feuilles, montrez que la complexité de* `s_rec(n,k)` *peut-être exponentielle en n.*

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

Question 7. *Montrez que pour chaque n suffisamment grand, l'arbre des appels de* `s_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 $s(n,k)-1 = \Theta(2^n/\sqrt{n})$ 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 5, tout aussi convainquant, est de regarder ce qu'il se passe pour les deux premiers niveaux de l'arbre des appels pour `s_rec(n,k)` :



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 `s_rec(n,k)`, et être beaucoup plus efficace, on va utiliser la technique de mémorisation (ou de programmation dynamique).

Question 8. *Écrire en* `C` *une nouvelle fonction non-récursive* `s_prog_dyn(n,k)` *renvoyant* $s(n,k)$ *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. **[3 pts]** Principe. On peut utiliser une table $S[1..n][0..k]$ (n lignes de $k+1$ colonnes) pour stocker chaque $s(i,j)$ pour $i = 1..n$ et $j = 0..k$ qu'on remplit ligne par ligne. Cela donne par exemple pour $n = 5$ et $k = 3$:

```
   S            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 $s(n,k) = s(n-1,k-1) + s(n-1,k)$, comme celle du triangle de Pascal (d'ailleurs c'est le triangle de Pascal), ne fait intervenir que la ligne précédente de la table S. 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 $P[j] = s(i-1, j)$ pour tout $j = 0, ..., k$. L'autre, S, contient toutes les valeurs $s(i,j)$ que l'on calcule avec la formule récursive $S[j] = s(i-1, j-1) + s(i-1, j) = P[j-1] + P[j]$. Pour passer à la ligne suivante, on incrémente i puis on échange P et S, ce qui évite la recopie. Il faut faire attention aux bords, notamment lorsque $i \leqslant k$.

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

On pouvait également remarquer que $s(n,k) = s(n,n-k)$ 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 s_prog_dyn2(int n, int k){
      static long S[n+1][k+1]; // variable globale
      for(i=0;i<=n;i++) // initialisation de la table S[][]
        for(j=0;j<=k;j++)
          S[i][j]=-1;
      return s_mem(n,k);
    }

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

Question 9. *Quelles sont les complexités en temps et en espace de votre fonction* `s_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.