Devoir maison n°2

Corrigé

⋆ ⋆ ⋆

1 Castors affairés

Question 1 Le nombre de programmes contenant n caractères est 128^n (qui est bien fini). Malheureusement, on ne peut pas se contenter de tous les exécuter et ne garder que celui qui s'exécute sans erreur, termine son calcul en temps fini et renvoie un entier, le plus grand possible, car certains de ces programmes peuvent avoir une exécution qui ne termine pas.

Question 2 La fonction C est correctement définie, car chaque l'exécution de chaque programme peut soit terminer en temps fini, soit ne pas terminer (il n'y a pas d'entre deux, les programmes contenant des erreurs terminent en temps fini). De plus, le programme contenant n répétitions du caractère `1` est correct, termine en temps fini et renvoie un entier. Ainsi, l'ensemble des entiers renvoyés par des programmes de taille n qui terminent est un ensemble fini, non vide, inclus dans $\mathbb{Z}$, il possède donc un maximum.

Question 3 On peut rajouter une espace à la fin d'un programme sans changer le déroulé de son exécution (l'espace sera ignorée). On en déduit que pour $n \in \mathbb{N}^*$, $C(n+1) \geqslant C(n)$: si on dispose d'un castor affairé de taille n, alors il existe un programme de taille $n+1$ qui renvoie le même entier, qui est donc inférieur ou égal à l'entier renvoyé par un castor affairé de taille $n+1$.

Par ailleurs, on peut parenthéser un castor affairé de taille n et rajouter `+1`. Un tel programme modifié terminera toujours son exécution en temps fini et renverra un entier strictement plus grand (on suppose une représentation des entiers sur une mémoire infinie). Sa taille sera exactement $n+4$. On en déduit bien que $C(n+4) > C(n)$.

Question 4 Les seuls programmes à un seul caractère qui renvoient un entier sont des programmes constitués d'un des 10 chiffres. Ainsi, $C(1) = 9$. Cela ne contredit pas la non-calculabilité, car cela ne donne pas de méthode générale permettant de calculer $C(n)$ pour tout entier n.

Question 5 Notons $m = \lfloor \log_{10} n \rfloor$, et $n = (c_m c_{m-1} \ldots c_1 c_0)_{10}$ l'écriture décimale de n. f étant calculable, il existe un programme OCaml commençant par `let f x =` permettant de définir une fonction calculant $f(x)$, dont l'exécution termine toujours. Notons k_0 la taille de ce programme. En le complétant par `f`$c_m c_{m-1} \ldots c_1 c_0$, on obtient un programme de taille $k_0 + 2 + m + 1$, dont l'exécution termine toujours et qui calcule $f(n)$. En posant $k = k_0 + 3$, on obtient un programme de taille $m + k$ qui renvoie un entier, donc cet entier est inférieur ou égal à l'entier renvoyé par un castor affairé de taille $m + k$, soit $f(n) \leqslant C(m+k)$.

Question 6 On a $\dfrac{n}{\lfloor \log_{10} n \rfloor + k + 4} \underset{n \to +\infty}{\longrightarrow} +\infty$, donc pour n_0 assez grand, $n_0 \geqslant \lfloor \log_{10} n_0 \rfloor + k + 4$. Pour un tel n_0, par la question 3, on a :

$$f(n_0) \leqslant C(\lfloor \log_{10} n_0 \rfloor + k) < C(\lfloor \log_{10} n_0 \rfloor + k + 4) \leqslant C(n_0)$$

Soit $f(n_0) < C(n_0)$.

Question 7 La question précédente montre que $f \neq C$. Comme ce résultat est valable pour toute fonction calculable, on en déduit que C n'est égale à aucune fonction calculable, donc n'est pas calculable elle-même.

Question 8 Supposons que le problème `Arrêt` est décidable. Alors le programme suivant pourrait calculer $C(n)$, pour tout n :

- on énumère tous les programmes OCaml de taille n ;

– on décide, à l'aide d'un programme qui résout `Arrêt`, lesquels ont une exécution en temps fini ;
– on exécute les programmes qui terminent à l'aide d'une machine universelle, en gardant en mémoire les valeurs renvoyées, si ce sont des entiers ;
– on renvoie le maximum de tous ces entiers.

Ainsi, le problème du castor affairé serait calculable. Par la question précédente, on en déduit par l'absurde que `Arrêt` n'est pas décidable.

2 Algorithme de Prim

Question 9 On fait une allocation mémoire et on remplit les différents champs. Notons qu'il n'est pas nécessaire ici d'initialiser le tableau de données (tant qu'on ne lit pas des valeurs non initialisées).

```
tasmin* creer_tas(int capa){
    tasmin* tas = malloc(sizeof(tas));
    tas->taille = 0;
    tas->capa = capa;
    tas->data = malloc(capa * sizeof(arete));
    return tas;
}
```

Question 10 Il suffit d'avoir en tête que chaque `malloc` doit donner lieu à un `free`.

```
void liberer_tas(tasmin* tas){
    free(tas->data);
    free(tas);
}
```

Question 11 En ayant en tête la représentation arborescente des tas binaires, l'algorithme d'insertion consiste à insérer le nouvel élément à la première feuille disponible, puis à le faire remonter dans l'arbre tant que la priorité de son parent est supérieure.

L'appel à `assert` permet de vérifier que le tas n'est pas plein.

```
void inserer(tasmin* tas, arete a){
    assert(tas->taille < tas->capa);
    int i = tas->taille;
    tas->taille++;
    tas->data[i] = a;
    while (i > 0 && tas->data[(i - 1) / 2].pds > a.pds){
        tas->data[i] = tas->data[(i - 1) / 2];
        i = (i - 1) / 2;
        tas->data[i] = a;
    }
}
```

Question 12 Il suffit d'envisager les 4 possibles : le nœud n'a pas d'enfant, il n'a qu'un seul enfant (qui est le gauche), ou il a deux enfants (auquel cas il faut comparer les priorités).

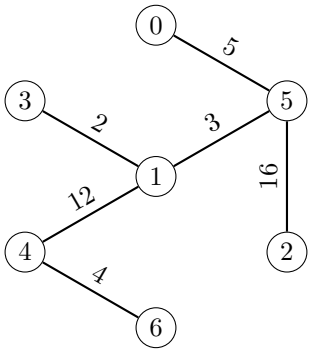
```
int enfant_min(tasmin tas, int i){
    int eg = 2 * i + 1;
    int ed = 2 * i + 2;
    if (eg >= tas.taille)
        return i;
    if (eg == tas.taille - 1 || tas.data[eg].pds < tas.data[ed].pds)
        return eg;
    return ed;
}
```

Question 13 L'algorithme d'extraction consiste à échanger la racine et la dernière feuille, puis à faire descendre l'élément qui s'est retrouvé à la racine tant qu'il possède un enfant de priorité inférieure.

```
arete extraire(tasmin* tas){
    assert(tas->taille > 0);
    tas->taille--;
    arete a = tas->data[tas->taille];
    tas->data[tas->taille] = tas->data[0];
    tas->data[0] = a;
    int i = 0;
    int emin = enfant_min(*tas, 0);
    while (a.pds > tas->data[emin].pds){
        tas->data[i] = tas->data[emin];
        tas->data[emin] = a;
        i = emin;
        emin = enfant_min(*tas, i);
    }
    return tas->data[tas->taille];
}
```

Question 14 On remarque que la profondeur du nœud qui se déplace dans l'arbre diminue (resp. augmente) d'exactement 1 à chaque passage dans la boucle, dans `inserer` (resp. extraire). On en déduit que les complexités sont linéaires en la hauteur de l'arbre, qui est logarithmique en la taille du tas.

Question 15 On obtient l'arbre couvrant :



Question 16 Pas de difficulté pour cette fonction.

```
liste* cons(liste* lst, int t, int pds){
    liste* lst_ret = malloc(sizeof(liste));
    lst_ret->suiv = lst;
    lst_ret->t = t;
    lst_ret->pds = pds;
    return lst_ret;
}
```

Question 17 On compte ici la somme des degrés, donc la somme des longueurs des listes d'adjacence, qu'on divise par 2 pour avoir le nombre d'arêtes.

```
int nombre_aretes(graphe g){
    int na = 0;
    for (int s=0; s<g.n; s++){
        liste* lst = g.adj[s];
        while (lst != NULL){
            na++;
            lst = lst->suiv;
        }
    }
    return na / 2;
}
```

Question 18 On se contente de parcourir la liste d'adjacence, de créer les arêtes et de les insérer dans le tas.

```
void ajout_voisins(graphe g, tasmin* tas, int s){
    liste* lst = g.adj[s];
    while (lst != NULL){
        arete a = {.s = s, .t = lst->t, .pds = lst->pds};
        inserer(tas, a);
        lst = lst->suiv;
    }
}
```

Question 19 On commence par écrire une petite fonction utilitaire qui ajoute une arête dans un graphe :

```
void ajout_arete(graphe g, arete a){
    g.adj[a.s] = cons(g.adj[a.s], a.t, a.pds);
    g.adj[a.t] = cons(g.adj[a.t], a.s, a.pds);
}
```

Dès lors, on peut appliquer l'algorithme de Prim. Une légère différence avec l'algorithme donné en pseudo-code : lorsqu'on extrait une arête du tas, on commence par vérifier que $t \notin R$ (car t a pu être ajouté à R), où R est représenté par un tableau de booléens (ce qui permet de faire ces tests et ajouts à R en temps constant).

```
graphe prim(graphe g){
    graphe acm;
    acm.n = g.n;
    acm.adj = malloc(g.n * sizeof(liste*));
    bool* R = malloc(g.n * sizeof(bool));
    for (int s=0; s<g.n; s++){
        acm.adj[s] = NULL;
        R[s] = false;
    }
    tasmin* tas = creer_tas(nombre_aretes(g));
    R[0] = true;
    ajout_voisins(g, tas, 0);
    int nb_sommets = 1;
    while (nb_sommets < g.n){
        arete a = extraire(tas);
        if (!R[a.t]){
            R[a.t] = true;
            ajout_voisins(g, tas, a.t);
            ajout_arete(acm, a);
            nb_sommets++;
        }
    }
    free(R);
    liberer_tas(tas);
    return acm;
}
```

Question 20 Les appels à `ajout_arete` sont en temps constant. La création du tas, du graphe vide et de R se font en $\mathcal{O}(|S|+|A|)$. Il faut donc compter la complexité des appels à `extraire` et `ajout_voisins` :

- il y a au plus $n-1=|S|-1$ appels à `extraire`, dans un tas qui contient au plus de l'ordre de $|A|$ arêtes (une arête n'est jamais ajoutée qu'au plus deux fois). La complexité totale est donc en $\mathcal{O}(|S|\log|A|)$;
- chaque arête est ajoutée au plus deux fois au tas, soit une complexité totale en $\mathcal{O}(|A|\log|A|)$.

Sachant que le graphe est connexe, on a $|S|=\mathcal{O}(|A|)$, soit une complexité totale en $\mathcal{O}(|A|\log|A|)=\mathcal{O}(|A|\log|S|)$, comme l'algorithme de Kruskal.

Question 21 Au cours de la boucle Tant que de l'algorithme, la propriété « (R,B) est connexe » est un invariant de boucle. C'est clair, car à chaque passage dans la boucle, on ajoute une arête reliant un sommet de R à un sommet de $\overline{R}$ qu'on ajoute à R. Finalement, à la fin de l'algorithme, $R=S$ et on renvoie (S,B).

L'algorithme termine bien, car le graphe G étant connexe, on pourra toujours trouver une arête entre un sommet de R et de $S\setminus R$, tant que $R\neq S$.

Sachant qu'il y a eu $|S|-1$ passage dans la boucle, $|B|=|S|-1$. (S,B) est donc un graphe connexe à $|S|-1$ arêtes. Il s'agit donc d'un arbre. Comme son ensemble de sommets est S, c'est un arbre couvrant.

Question 22 Notons $a_i=\{s,t\}$. Il existe dans T^* un chemin de s à t. Sachant que $s\in R$ et $t\notin R$, il existe une arête de ce chemin reliant un sommet de R à un sommet en dehors de R. En effet, si ce chemin est $(s_0,s_1,\ldots,s_k)$, alors il existe un indice $i\in[\![1,k]\!]$ minimal tel que $s_i\in S\setminus R$ (car $s_k\in S\setminus R$). Par minimalité, l'arête $a^*=\{s_{i-1},s_i\}$ est bien l'arête cherchée.

Question 23 On a par ailleurs $f(a_i)\leqslant f(a^*)$. En effet, c'est l'arête a_i qui a été choisie lors de l'algorithme de Prim.

Dès lors, considérons $T'=(S,B^*\cup\{a_i\}\setminus\{a^*\})$ et remarquons :

- T' est connexe : en effet, il existe toujours un chemin entre u et v, de la forme $u\rightsquigarrow s\xrightarrow{a_i}t\rightsquigarrow v$;
- T' possède $|S|-1$ arêtes car $a_i\notin B^*$ et $a^*\in B^*$. On en déduit que T' est un arbre couvrant ;
- $f(a^*)\leqslant f(a_i)$. En effet, $f(T')=f(T^*)+f(a_i)-f(a^*)\leqslant f(T^*)$ car T^* est un arbre couvrant minimal.

On en déduit finalement que $f(T')=f(T^*)$, donc T' est un arbre couvrant minimal de G. Par ailleurs, la

première arête choisie par l'algorithme de Prim qui n'apparaît pas dans T' est une arête a_j avec $j > i$. C'est contradictoire avec la construction de T^*. On en déduit que T est bien un arbre couvrant minimal.

$\star \star \star$