Composition d'informatique n°4

Corrigé

⋆ ⋆ ⋆

1 Cas à deux machines

Question 1 L'ordonnancement $(2, 3, 1, 0)$ est optimal. En effet, le temps total de cet ordonnancement est 11. Comme le temps minimal sur la machine 0 est $t_{2,0} = 1$, l'exécution sur la machine 1 ne peut pas commencer avant le temps 1. Comme la somme des temps sur la machine 1 est 10, aucun ordonnancement ne peut avoir un temps total strictement inférieur à $1 + 10 = 11$.

Question 2 On peut écrire une fonction utilitaire :

```
int max(int x, int y){
    return (x < y)?y:x;
}
```

Ensuite, on utilise deux variable qui garde en mémoire le temps de fin de la dernière tâche exécutée sur chaque machine, qu'on met à jour selon les formules données par l'énoncé.

```
int temps_total(int n, int* temps0, int* temps1, int* sigma){
    int f0 = 0;
    int f1 = 0;
    for (int i=0; i<n; i++){
        f0 += temps0[sigma[i]];
        int d1 = max(f0, f1);
        f1 = d1 + temps1[sigma[i]];
    }
    return f1;
}
```

Question 3 On remarque que $\min\limits_{i=0}^{3} t_{i,0} \geqslant \max\limits_{i=0}^{3} t_{i,1}$. Cela signifie que pour tous i et j, $\min(t_{\sigma(i),0} t_{\sigma(j),1}) = t_{\sigma(j),1}$. Il suffit donc de vérifier que les temps sur la deuxième machine sont décroissants.

On constate donc que l'ordonnancement σ_1 ne vérifie pas la condition de Johnson, car $t_{0,1} < t_{1,1}$, et que l'ordonnancement σ_2 vérifie bien la condition de Johnson, car les temps sont décroissants sur la deuxième machine.

Question 4 Soit σ un ordonnancement vérifiant $\tau_{\sigma(0)} \preccurlyeq \tau_{\sigma(1)} \preccurlyeq \ldots \preccurlyeq \tau_{\sigma(n-1)}$. Soit $0 \leqslant i < j \leqslant n$. Montrons que si $\tau_i \preccurlyeq \tau_j$, alors $\min(t_{i,0}, t_{j,1}) \leqslant \min(t_{i,1}, t_{j,0})$ (ce qui correspond à ce qu'on veut montrer, quitte à mettre des σ devant i et j).

- si $\tau_i \in I_0$ et $\tau_j \in I_1$, alors $t_{i,0} < t_{i,1}$ et $t_{j,1} \leqslant t_{j,0}$, donc le minimum des membres gauches sera bien inférieur au minimum des membres droits ;
- si $\tau_i, \tau_j \in I_0$ et $t_{i,0} \leqslant t_{j,0}$, alors $t_{i,0} \leqslant t_{j,0} < t_{j,1}$, donc $\min(t_{i,0}, t_{j,1}) = t_{i,0} \leqslant \min(t_{i,1}, t_{j,0})$ (car l'inégalité est pour chaque terme par hypothèse) ;
- de même si $\tau_i, \tau_j \in I_0$ et $t_{i,1} \geqslant t_{j,1}$.

Question 5 On écrit d'abord une fonction pour la relation stricte :

```
bool cmp_strict(tache taui, tache tauj){
    if (taui.tm0 < taui.tm1 && tauj.tm0 >= tauj.tm1) return true;
    if (taui.tm0 < taui.tm1 && taui.tm0 < tauj.tm0) return true;
    if (tauj.tm0 >= tauj.tm1 && taui.tm1 > tauj.tm1) return true;
    return false;
}
```

On note que dans le deuxième cas, on n'a pas besoin de rajouter la condition `tauj.tm0 < tauj.tm1`, car si on arrive au deuxième test, c'est que le premier n'a pas été vérifié. De même pour la condition `taui.tm0 >= taui.tm1` dans le troisième cas.

On peut alors écrire la fonction demandée :

```
bool cmp_johnson(tache taui, tache tauj){
    return cmp_strict(taui, tauj) || (!cmp_strict(tauj, taui) && taui.ind <= tauj.ind);
}
```

Question 6 Comme la complexité attendue est $\mathcal{O}(n^2)$ on peut se contenter de n'importe quel tri naïf. On choisit ici le tri bulles pour sa simplicité d'écriture (un tri insertion ou sélection auraient convenu).

On commence par une fonction qui échange deux valeurs dans un tableau de tâches.

```
void swap(tache* tab, int i, int j){
    tache tmp = tab[i];
    tab[i] = tab[j];
    tab[j] = tmp;
}
```

Ensuite, on implémente le tri à bulles.

```
void tri_taches(int n, tache* tab, bool (*compare)(tache, tache)){
    for (int i=1; i<n; i++){
        for (int j=0; j<n-i; j++){
            if (!compare(tab[j], tab[j+1]))
                swap(tab, j, j+1);
        }
    }
}
```

De part la taille des boucles, le tri est bien en $\mathcal{O}(n^2)$.

Question 7 On se contente de créer un tableau et de recopier les indices.

```
int* taches_vers_ordo(int n, tache* tab){
    int* sigma = malloc(n * sizeof(int));
    for (int i=0; i<n; i++){
        sigma[i] = tab[i].ind;
    }
    return sigma;
}
```

Question 8 On commence par créer le tableau de tâches.

```c
tache* creer_taches(int n, int* temps0, int* temps1){
    tache* tab = malloc(n * sizeof(tache));
    for (int i=0; i<n; i++){
        tache tch = {.tm0 = temps0[i], .tm1 = temps1[i], .ind = i};
        tab[i] = tch;
    }
    return tab;
}
```

Ensuite on le trie selon la relation de Johnson, avant de le convertir en ordonnancement.

```c
int* johnson(int n, int* temps0, int* temps1){
    tache* tab = creer_taches(n, temps0, temps1);
    tri_taches(n, tab, cmp_johnson);
    int* sigma = taches_vers_ordo(n, tab);
    free(tab);
    return sigma;
}
```

Question 9 L'opération la plus longue est le tri. Les autres opérations se font en temps linéaire en n. On obtient donc un temps de calcul en $\mathcal{O}(n^2)$ (qu'on aurait pu améliorer à $\mathcal{O}(n \log n)$ avec un tri efficace).

2 Maintenance sur l'une des deux machines

Question 10 Une permutation $\sigma \in \mathfrak{S}_n$ est un certificat valide. On peut calculer le temps total en temps linéaire (donc polynomial) en n, puis vérifier qu'il est $\leqslant B$. On en déduit que $\texttt{OFT} \in \mathsf{NP}$.

Question 11 On pose $S = \sum_{i=0}^{k-1} x_k$. On montre l'équivalence :

- ($\Rightarrow$) si $(x_0, \dots, x_k, C)$ est une instance positive de `SOMME PARTIELLE`, soit $I \subseteq [\![0, k-1]\!]$ tel que $\sum_{i\in I} x_i = C$. Alors pour $I' = I \cup \{k+1\}$, on a $\sum_{i\in I'} x_i = C + 2S - C = 2S = S - C + S + C = \sum_{i\notin I} x_i + x_k = \sum_{i\notin I'} x_i$, donc $(x_0, \dots, x_{k+1})$ est bien une instance positive de `PARTITION` ;
- ($\Leftarrow$) si $(x_0, \dots, x_{k+1})$ est une instance positive de `PARTITION`, alors il existe $I' \subseteq [\![0, k+1]\!]$ tel que $\sum_{i\in I'} x_i = \sum_{i\notin I'} x_i = 2S$.

  La somme vaut $2S$ car la somme totale des éléments vaut $4S$. Ainsi, sachant que $x_k > S$ et $x_{k+1} \geqslant S$, on en déduit que $k \notin I'$ et $k+1 \in I'$ (quitte à considérer le complémentaire de I'). Finalement, on pose $I = I' \setminus \{k+1\} \subseteq [\![0, k-1]\!]$ et on a $\sum_{i\in I'} x_i = x_{k+1} + \sum_{i\in I} x_i$, soit $\sum_{i\in I} x_i 2S - x_{k+1} = C$. Donc $(x_0, \dots, x_{k-1})$ est une instance positive de `SOMME PARTIELLE`.

Question 12 Si on n'a pas l'hypothèse $\sum_{i=0}^{k-1} x_i \geqslant C > 0$, alors :

- si $C = 0$, l'instance est positive. Il suffit de renvoyer une instance positive quelconque, comme $(1, 1)$;
- si $\sum_{i=0}^{k-1} x_i < C$, l'instance est négative. Il suffit de renvoyer une instance négative quelconque, comme $(1, 2)$.

La construction de $(x_0, \dots, x_{k+1})$ se fait en temps polynomial, y compris dans les deux cas particuliers précédents, et l'équivalence précédente nous donne bien la réduction voulue.

Question 13 Comme pour la question précédente, si la somme n'est pas paire, l'instance est négative pour PARTITION, et on peut renvoyer une instance négative de OFT quelconque, comme $((1),(1),0,1,1)$.

Si l'un des x_i vaut zéro, alors la positivité de l'instance est la même que celle de l'instance sans x_i. On peut donc retirer tous les x_i nuls sans perte de généralité.

Si l'un des $x_i > S$, alors l'instance est nécessairement négative, car quelle que soit la partition, la partie qui contient x_i sera $> S$. On peut donc renvoyer une instance négative de OFT quelconque.

Question 14 Supposons que $(x_0, \ldots, x_{k-1})$ est une instance positive de PARTITION et soit $I \subseteq [\![0, k-1]\!]$ tel que $\sum_{i\in I} x_i = S = \sum_{i\notin I} x_i$. Sans perte de généralité, supposons $I = \{1, \ldots, |I|\}$.

On pose σ un ordonnancement tel que $\sigma(0) = k\}$ et $\{\sigma(1), \ldots, \sigma(|I|)\} = I$: on ordonne d'abord la tâche k, puis toutes les tâches d'indices dans I.

Alors dans une exécution des tâches selon σ, il n'y a aucun temps d'attente sur la machine 1. En effet, il n'y a pas de temps d'attente :

- pour la tâche k car $t_{k,0} = 0$;
- pour la tâche 1 car $t_{k,1} \geqslant t_{1,0}$;
- pour la tâche $1 < i \leqslant |I|$, car la somme des temps sur la machine 1 pour les tâches $k, 1, \ldots, i-1$ est supérieure à la somme des temps sur la machine 0 pour les tâches $k, 1, \ldots, i$;
- pour la tâche $|I| < i$ et la tâche 0, car la somme des temps sur la machine 1 pour les tâches $k, 1, \ldots, |I|$ est $x_{\max} + \sum_{i\in I} x_i(S+1) = x_{\max} + S(S+1) \geqslant \sum_{i\in I} x_i + y - x = S + S^2 + S - S = S(S+1)$ et par les mêmes arguments que précédemment.

On en déduit que le temps total est $\sum_{i=0}^{k} t_{i,1} = 2S(S+1) + x_{\max} = B$. Cela forme bien une instance positive de OFT.

Question 15 Réciproquement, si (t_0, t_1, x, y, B) est une instance positive de OFT, alors il ne doit y avoir aucun temps d'attente sur la machine 1. Comme $x_i > 0$ pour tout i, si σ est un ordonnancement optimal, alors $\sigma(0) = k$ (car la tâche k est la seule de temps nul sur la machine 0).

Supposons que pour cet ordonnancement, il existe un indice i_0 tel que $d_{\sigma(i_0),0} < x = S$ et $f_{\sigma(i_0),0} \geqslant y + 1 = S(S+1) + 1$ (une tâche s'exécute de part et d'autre de la pause). Alors sur la machine 1, la tâche $\sigma(i_0 - 1)$ termine au temps :

$$x_{\max} + \sum_{i=1}^{i_0-1} x_i(S+1) \leqslant x_{\max} + (S-1)(S+1)$$

En effet, $\sum_{i=1}^{i_0-1} x_i = d_{\sigma(i_0),0} \leqslant x - 1 = S - 1$.

Sachant que $x_{\max} \leqslant S$, on en déduit que le temps de fin de la tâche $\sigma(i_0 - 1)$ est $x_{\max} + S^2 - 1 \leqslant S^2 + S - 1 < f_{\sigma(i_0),0}$. Il y aura donc un temps d'attente sur la machine 1 entre la fin de la tâche $\sigma(i_0)$ sur la machine 0 et le début de cette tâche sur la machine 1.

Par l'absurde, on en déduit qu'il n'y a pas de tâche s'exécutant de part et d'autre de la pause. On pose I l'ensemble des tâches (sauf k) s'exécutant sur la machine 0 avant x. On a bien $\sum_{i\in I} x_i = x = S$, donc $(x_0, \ldots, x_{k-1})$ est bien une instance positive de PARTITION.

Question 16 Les trois questions précédentes permettent de montrer que PARTITION $\leqslant_{\mathsf{m}}^{\mathsf{p}}$ OFT, car la construction se fait en temps polynomial.

On a montré que OFT $\in$ NP. De plus, SOMME PARTIELLE $\leqslant_{\mathsf{m}}^{\mathsf{p}}$ PARTITION $\leqslant_{\mathsf{m}}^{\mathsf{p}}$ OFT et SOMME PARTIELLE est NP-difficile par hypothèse, donc OFT est NP-difficile, donc NP-complet.

Question 17 C'est la même idée, sauf qu'on rajoute un test pour ajouter le temps $y - x$ lorsqu'une tâche s'effectue de part et d'autre de la pause.

```
int temps_maintenance(int n, int* temps0, int* temps1, int x, int y, int* sigma){
    int f0 = 0, f1 = 0;
    for (int i=0; i<n; i++){
        if (f0 <= x && f0 + temps0[sigma[i]] > x)
            f0 += temps0[sigma[i]] + y - x;
        else
            f0 += temps0[sigma[i]];
        int d2 = max(f0, f1);
        f1 = d2 + temps1[sigma[i]];
    }
    return f1;
}
```

Question 18 On distingue :

- si la somme des temps sur la machine 0 est inférieure à x, alors on se ramène au problème précédent pour lequel on a admis que l'algorithme de Johnson était optimal ;
- sinon, on peut montrer qu'en fait tout ordonnancement fournit une 2-approximation. Notons F et F^* les temps d'exécution par un ordonnancement quelconque et un ordonnancement optimal respectivement. Alors :
  - sachant qu'il faut exécuter toutes les tâches sur la machine 0 et déborder après la pause, $\sum_i t_{i,0} + y - x \leqslant F^*$;
  - sachant qu'il n'y aucun temps d'attente sur la machine 1 après l'exécution de toutes les tâches sur la machine 0, $F - (\sum_i t_{i,0} + y - x) \leqslant \sum_i t_{i,1} \leqslant F^*$.

  En combinant les deux, on obtient : $F \leqslant 2F^*$.

Question 19 Supposons que $f_{\sigma(i),0} < d_{\sigma(i),1}$. Alors $d_{\sigma(i),1} = f_{\sigma(i-1),1}$. On en déduit que l'indice critique est inférieur ou égal à $i-1$, ce qui est absurde par hypothèse.

Question 20 On distingue :

- si 0 est critique pour σ_1, alors le temps de fin de la tâche $\sigma_1(0)$ sur la machine 1 est une borne inférieure de F^*. On en déduit que
$$F_1 = f_{\sigma_1(0),1} + \sum_{i=1}^{n-1} t_{\sigma_1(i),1} \leqslant F^* + \sum_{i=1}^{n-1} t_{\sigma_1(i),1}$$
- sinon, soit i_c l'indice critique pour σ_1. Alors le temps de fin de la tâche $\sigma_1(i_c)$ sur la machine 0 est une borne inférieure de F^*. On en déduit que :
$$F_1 = f_{\sigma_1(i_c),0} + \sum_{i=i_c}^{n-1} t_{\sigma_1(i),1} \leqslant F^* + \sum_{i=1}^{n-1} t_{\sigma_1(i),1}$$

Question 21

1. Sur un ordonnancement optimal, la première tâche à terminer après $f_{\sigma_2(i_c),0}$ a pu commencer avant ce temps. Ce n'est pas le cas pour la tâche $\sigma_2(i_c)$. Comme il n'y a pas d'attente sur la première machine, on en déduit l'inégalité voulue.
2. Notons $J = \{\sigma_2(i) \mid i > i_c\}$ et $K = I \cap J$. L'inégalité précédente se traduit par $\sum_{i \in I\setminus K} t_{i,0} \geqslant \sum_{i \in J\setminus K} t_{i,0}$. De plus, par décroissance des $\dfrac{t_{i,1}}{t_{i,0}}$ dans l'ordonnancement σ_2, on a :
$$\sum_{i\in I\setminus K} t_{i,1} = \sum_{i\in I\setminus K} t_{i,0}\frac{t_{i,1}}{t_{i,0}} \geqslant \sum_{i\in I\setminus K} t_{i,0}\frac{t_{i_c,1}}{t_{i_c,0}} \geqslant \sum_{i\in J\setminus K} t_{i,0}\frac{t_{i_c,1}}{t_{i_c,0}} \geqslant \sum_{i\in J\setminus K} t_{i,0}\frac{t_{i,1}}{t_{i,0}} = \sum_{i\in J\setminus K} t_{i,1}$$

Soit finalement $\sum_{i\in I} t_{i,1} \geqslant \sum_{i\in J} t_{i,1}$. En ajoutant $f_{\sigma_2(i_c),0}$ aux deux membres, on obtient :

$$F^* \geqslant f_{\sigma_2(i_c),0} + \sum_{i\in I} t_{i,1} \geqslant f_{\sigma_2(i_c),0} + \sum_{i\in J} t_{i,1} = F_2 - t_{\sigma_2(i_c),1}$$

Question 22 On distingue :

- s'il existe τ_i telle que $t_{i,1} > \dfrac{F^*}{2}$, alors $\sum_{j\neq i} t_{j,1} < \dfrac{F^*}{2}$. Par la question 20, comme la plus longue tâche sur la machine 1 est ordonnancée en premier, on en déduit que $F_1 < F^* + \dfrac{F^*}{2} = \frac{3}{2}F^*$;
- sinon, par la question 21, $F_2 \leqslant F^* + t_{\sigma(i),1} \leqslant F^* + \dfrac{F^*}{2} = \frac{3}{2}F^*$.

Étant donné qu'on renvoie l'ordonnancement qui minimise le temps total, l'algorithme est bien une $\frac{3}{2}$-approximation.

Question 23 Pour simplifier l'écriture de σ_1, on va trier les tâches par temps décroissant sur la machine 1 (ce qui donne bien le résultat attendu).

On écrit deux fonctions de comparaisons pour chacune des permutations. On fait attention à éviter les divisions pour σ_2, car on travaille avec des entiers.

```
bool cmp1(tache taui, tache tauj){
    return taui.tm1 >= tauj.tm1;
}

bool cmp2(tache taui, tache tauj){
    return taui.tm1 * tauj.tm0 >= taui.tm0 * tauj.tm1;
}
```

Ensuite, on réutilise les fonctions précédentes.

```
int* approximation(int n, int* temps0, int* temps1, int x, int y){
    tache* tab = creer_taches(n, temps0, temps1);
    tri_taches(n, tab, cmp1);
    int* sigma1 = taches_vers_ordo(n, tab);
    tri_taches(n, tab, cmp2);
    int* sigma2 = taches_vers_ordo(n, tab);
    free(tab);
    int tt1 = temps_maintenance(n, temps0, temps1, x, y, sigma1);
    int tt2 = temps_maintenance(n, temps0, temps1, x, y, sigma2);
    if (tt1 < tt2){
        free(sigma2);
        return sigma1;
    }
    free(sigma1);
    return sigma2;
}
```

3 Cas général

Question 24 On pourrait ici créer une matrice pour calculer tous les $f_{i,j}$ et renvoyer $f_{\sigma(i-1),m-1}$, mais on peut le faire en utilisant moins d'espace. On utilise ici un tableau de taille $m+1$, où `temps_fin.(j + 1)` correspond au temps de fin de la i'-ème tâche à être effectuée sur la machine j (avec la machine -1 qui a des temps nuls, pour faciliter les calculs).

```
let temps_partiel temps sigma i =
    let m = Array.length temps.(0) in
    let temps_fin = Array.make (m + 1) 0 in
    for i' = 0 to i - 1 do
        for j = 0 to m - 1 do
            let dij = max temps_fin.(j) temps_fin.(j + 1) in
            temps_fin.(j + 1) <- temps.(sigma.(i')).(j) + dij
        done
    done;
    temps_fin.(m)
```

Question 25 L'algorithme glouton choisirait dans un premier temps de placer la tâche 1 avant 0, car cela donne un temps de 13 au lieu de 14. Ensuite, on obtiendrait $(1, 0, 2)$, de temps total 17, contre 18 et 19 pour $(1, 2, 0)$ et $(2, 1, 0)$.

L'ordonnancement $(0, 1, 2)$ donne pourtant un temps total de 16, ce qui est strictement mieux que le résultat donné par l'algorithme glouton, qui n'est donc pas optimal.

Question 26 On garde en mémoire deux tableaux : un tableau `sigma` qui correspond à l'ordonnancement en cours d'étude, et un ordonnancement correspondant à celui conservé par les différentes itérations de l'algorithme glouton. Pour chaque valeur de i entre 1 et $n-1$, on essaie d'insérer l'élément i dans l'ordonnancement partiel obtenu jusqu'ici. Pour cela, on commence par l'envisager en dernière position (à laquelle il se trouve naturellement de par la création de `sigma`), puis on le permute à chaque étape et on compare avec la meilleure solution trouvée jusqu'ici, et on remplace si besoin (en faisant une copie pour éviter les modifications). On pense bien à repartir de la meilleure solution après chaque itération.

```
let glouton temps =
    let n = Array.length temps in
    let sigma = ref (Array.init n Fun.id) and
        sigma_best = ref (Array.init n Fun.id) in
    for i = 1 to n - 1 do
        sigma := Array.copy !sigma_best;
        for k = i downto 1 do
            swap !sigma k (k - 1);
            if temps_partiel temps !sigma (i + 1) <
                temps_partiel temps !sigma_best (i + 1) then
                sigma_best := Array.copy !sigma;
        done
    done;
    !sigma_best
```

Question 27 La fonction `temps_partiel` se calcule en temps $\mathcal{O}(i \times m)$. Ainsi, la complexité totale de l'algorithme est $\mathcal{O}\left(\sum_{i=2}^{n} i^2 m\right) = \mathcal{O}(n^3 m)$. En effet, les copies et créations de tableaux sont de complexité négligeable par rapport à cette complexité.

Question 28 On garde en mémoire une variable contenant le meilleur ordonnancement et le meilleur temps (initialisé par exemple avec la permutation identité). Ensuite, en commençant avec un ordonnancement vide, étant donné un ordonnancement partiel $\tilde{\sigma}$:

- si $\tilde{\sigma}$ est total (il contient toutes les tâches), on calcule son temps et on met à jour le meilleur ordonnancement et meilleur temps si l'actuel est mieux ;
- sinon, pour chaque valeur qui n'est pas dans $\tilde{\sigma}$, on la rajoute en fin de $\tilde{\sigma}$ et on relance un appel récursif ;
- on peut arrêter l'exploration si le temps partiel de $\tilde{\sigma}$ dépasse déjà la meilleure borne trouvée (le temps total sera nécessairement plus grand).

À la fin de ces calculs, on aura exploré toutes les solutions possibles auront été explorée, et on renvoie la meilleure trouvée.

Question 29 Si $\tilde{\sigma}$ est un ordonnancement partiel, parmi toutes les manières de le compléter, on ne pourra pas avoir un temps total inférieur au temps partiel auquel on rajoute, sans pause, les temps des tâches manquantes sur la dernière machine. Ainsi, $h(\tilde{\sigma}) \leqslant \Phi(t, \sigma)$, pour tout σ qui complète $\tilde{\sigma}$. L'heuristique est bien une heuristique admissible, car il s'agit d'un problème de minimisation.

Question 30 On peut par exemple explorer les enfants $\tilde{\sigma}_i$ par ordre croissant de $h(\tilde{\sigma}_i)$, en supposant que $h(\tilde{\sigma})$ est une estimation du temps nécessaire pour terminer l'ordonnancement au mieux.

⋆ ⋆ ⋆