TP #5 — Trois problèmes sur les tableaux

Le but de ce TP est d'étudier trois problèmes sur les tableaux (d'entiers), pas tout à fait indépendants.
Pour tout ce TP, il est vivement conseillé de compiler votre programme avec l'option `-fsanitize=address`. (en plus des options habituelles).
Les seules fonctions C à écrire sont celles qui sont explicitement demandées, ainsi que les fonctions nécessaires pour les tests. Pour toutes les autres questions, il est attendu que vous **rédigiez** *a minima* une ébauche de réponse (sur feuille ou comme un commentaire dans votre programme, par exemple).

Sélection du k-ième plus grand élément

Soit un tableau a de n entiers de type `int` pas nécessairement distincts, on souhaite trouver et renvoyer le k-ième plus grand élément du tableau (où l'on commence à numéroter les éléments par zéro : $k \in [\![0, n-1]\!]$).

1. Expliquez comment résoudre ce problème si le tableau a est supposé trié par ordre croissant.
2. Déduisez-en un algorithme pour le cas général.
3. Écrivez une fonction C de signature :
   ```
   int select_k(size_t n, int a[n], size_t k)
   ```
   qui résout ce problème. Son coût (temporel, dans le pire cas) doit être au plus quadratique en la longueur n du tableau.
   Cette fonction peut modifier les éléments de son argument a, à condition que le nombre d'occurrences de chaque élément ne change pas (autrement dit, les modifications se limitent à leur appliquer une permutation arbitraire).
   Pour répondre à cette question, vous êtes libres d'utiliser n'importe quelle fonction déjà implémentée lors d'un précédent TP (par exemple à tout hasard, *select_sort*).
4. Quelle assertion « `assert` » pourrait-il être judicieux d'inclure dans cette fonction ? (N'hésitez pas à en inclure autant que possible & pertinent dans tout le reste du sujet.)
5. Testez *via* une fonction de test dédiée.

N.B. Il est possible de résoudre ce problème en temps seulement *linéaire* en n (pour cela, *on ne trie pas* le tableau a).

Recherche des k plus grands éléments

Soit a comme précédemment, on souhaite maintenant trouver tous les $k > 0$ plus grands éléments de a, avec multiplicité (c'est à dire que si par exemple la valeur

maximale présente dans le tableau y apparaît plusieurs fois, elle doit apparaître dans le résultat autant de fois (dans la limite de k)).

6. Expliquez comment résoudre ce problème si le tableau a est supposé trié par ordre croissant.

On pourrait (comme au problème précédent) résoudre le cas général en commençant par trier le tableau a. On souhaite cependant éviter de procéder ainsi, à la fois pour varier les approches et parce qu'il est possible de faire mieux (en supposant un algorithme de coût linéaire pour le problème précédent)
On commence par considérer une approche simple, mais qui n'est pas la meilleure quand k peut prendre des valeurs arbitraires.

7. Proposez un algorithme qui résout ce problème pour $k = 1$.
8. Déduisez-en un algorithme qui résout ce problème avec un coût (temporel, pire cas, à constantes près) majoré par celui de kn opérations élémentaires. Cet algorithme ne doit utiliser aucun autre algorithme comme sous-routine, à l'exception éventuelle d'une recherche de maximum.
9. Écrivez une fonction C de signature :
   ```
   void k_largest1(size_t n, int a[n],
                   size_t k, int r[k])
   ```
   qui implémente votre algorithme ci-dessus, et modifie son argument r pour qu'il contienne les k plus grands éléments de a (avec multiplicité), dans un ordre quelconque.
   Vous êtez libres d'introduire toute fonction supplémentaire vous paraissant utile, et l'on rappelle que si t est un tableau de tn éléments, vous pouvez le fournir comme argument d'une fonction prenant un paramètre tableau en indiquant n'importe quelle taille $0 < tn' \leq tn$, afin que cette fonction n'en considère que les tn' premiers éléments.
10. Testez *via* une fonction de test dédiée.

On suppose maintenant (pour l'instant) que les éléments de a sont tous distincts.

11. Proposez un algorithme qui résout ce problème en effectuant *un* appel à un algorithme de recherche du k-ième plus grand élément. En dehors de cet appel, votre algorithme devra avoir un coût (temporel, pire cas, à constantes près) majoré par celui de n opération élémentaires. (Ainsi, si l'on utilise un algorithme de coût linéaire pour la recherche du k-ième plus grand élément, celui-ci sera *aussi* de coût linéaire.)
12. Écrivez une fonction C de signature :
    ```
    void k_largest2(size_t n, int a[n],
                    size_t k, int r[k])
    ```
    qui implémente votre algorithme ci-dessus, et modifie son argument r pour qu'il contienne les k plus grands éléments de a (avec multiplicité).
    Celle-ci devra utiliser votre précédente fonction *select_k* pour la recherche du k-ième plus grand élément.

13. Testez *via* une fonction de test dédiée.

14. Que faut-il éventuellement modifier dans votre algorithme précédent afin qu'il reste correct quand les éléments de a ne sont plus nécessairement deux à deux distincts ?

15. Écrivez (si besoin) une fonction C de signature :

```
void k_largest3(size_t n, int a[n],
                size_t k, int r[k])
```

qui implémente votre algorithme modifié.

16. Testez *via* une fonction de test dédiée.

Recherche d'un élément majoritaire

Soit a comme précédemment, on souhaite maintenant trouver un élément qui y soit *majoritaire* ; c'est à dire, un élément dont le nombre d'occurrences est le plus élevé (à éventuelle égalité près) parmi tous les éléments de a.
On commence par s'intéresser au cas général, où il n'y a pas forcément de majorité absolue.

17. Expliquez comment résoudre ce problème si le tableau a est supposé trié.

18. Déduisez en un algorithme pour le cas général.

19. Écrivez une fonction C de signature :

```
int maj(size_t n, int a[n])
```

qui renvoie une valeur e qui apparaît une majorité (éventuellement non absolue) de fois dans a.

20. Testez *via* une fonction de test dédiée.

On suppose maintenant que a possède un élément strictement majoritaire. Autrement dit, si $n = 2n'$ est pair (resp. $n = 2n' + 1$ est impair), cet élément apparaît au moins $n' + 1$ fois, et donc au moins deux (resp. une) fois de plus que n'importe quel autre élément.

21. Expliquez comment résoudre le problème dans ce cas là en faisant un appel à un algorithme renvoyant le k-ième plus grand élément de a, *et aucune autre opération*.

On souhaite maintenant étudier & implémenter un algorithme dû à Boyer & Moore, qui résout ce problème efficacement et avec élégance (et sans se baser sur un algorithme de tri ou de sélection). Il utilise le principe de « vote » suivant :

- on initialise deux variables x et c à la valeur du premier élément du tableau et à *1* respectivement
- on parcourt un à un les éléments restant du tableau, et pour chacun de ces éléments e :
  - si le compteur c vaut 0, on le modifie en 1 et l'on affecte e dans x
  - sinon, si e est égal à la valeur courante de x, on incrémente c de 1

— sinon on décrémente c de 1

— on renvoie la valeur courante de x

22. Écrivez une fonction C de signature :
```
int amaj(size_t n, int a[n])
```
qui implémente l'algorithme ci-dessus.

23. Testez *via* une fonction de test dédiée.

Pour finir, on désire montrer que l'algorithme ci-dessus est correct (mais l'on n'utilisera pas d'invariant pour cela…).

24. Expliquez pourquoi celui-ci semble intuitivement l'être.

On introduit un ensemble de variables (au sens mathématique) ξ_i et γ_i, qui dénotent respectivement les valeurs des variables (au sens informatique) x et c après que le i-ème élément $a[i]$ a été considéré par l'algorithme ci-dessus. Ainsi, on a par exemple $\xi_0 = a[0]$ et $\gamma_0 = 1$, ce qui correspond aux valeurs initiales des variables x et c. On introduit également deux variables (mathématiques) supplémentaires $\xi_{-1} = \bot$, $\gamma_{-1} = 0$ (où « $\bot$ » se lit « *bottom* » et représente une absence de valeur), qui représentent les valeurs « virtuelles » des variables (informatiques) avant le début de l'exécution de l'algorithme proprement dit. Enfin, on note e_M l'élément strictement majoritaire recherché (que l'on sait exister).

On suppose l'existence de $i \geq -1$, $j < i$ tels que $\gamma_i = \gamma_j = 0$, et il n'existe aucun $i < k < j$ tel que $\gamma_k = 0$ (autrement dit, i et j sont deux points successifs de l'exécution de l'algorithme où le compteur c vaut zéro).

25. Montrez que $j - i$ est un nombre impair (que l'on note $2k+1$), et que l'élément $a[i+1]$ apparaît exactement k fois parmi les éléments $a[i+1 : j]$ de a d'indice compris entre $i+1$ et j (tous deux inclus).

26. Montrez que e_M apparaît *au plus* k fois parmi ces mêmes éléments $a[i+1 : j]$.

27. En déduire que la valeur finale γ_{n-1} de la variable (informatique) c est non nulle.

28. En déduire également que $\xi_{n-1} \neq e_M$ est absurde, où ξ_{n-1} est la valeur finale (renvoyée par l'algorithme) de la variable (informatique) x.

29. Conclure.

On peut également prouver la correction d'une autre façon : on introduit un nouvel ensemble de variables σ_i qui valent γ_i si $\xi_i = e_M$, et $-\gamma_i$ sinon (on a donc notamment $\sigma_{-1} = 0$).

30. Montrez que si $a[i] = e_M$, alors $\sigma_i = \sigma_{i-1} + 1$.

31. Montrez que dans les autres cas, $|\sigma_i - \sigma_{i-1}| \leq 1$.

32. En déduire que $\sigma_{n-1} > 0$.

33. Conclure.

Et si l'on suppose maintenant que l'on souhaite trouver l'élément strictement majoritaire *s'il existe*, et indiquer qu'il n'en existe pas dans le cas contraire ?

34. Montrez que ce problème se résout facilement si l'on dispose d'un algorithme qui renvoie l'élément strictement majoritaire s'il existe, et est de comportement indéfini sinon.