Université de Genève
-
Sciences Informatiques

Algorithmique - TP 01

Noah Munz (19-815-489)

<Date> 2022

TP 01

1 Elément majoritaire

1.2 Analyse

(Sauf mention du contraire les log sont en base 2).

1.2.1 Un élément majoritaire dans A est-il nécessairement aussi un éléement majoritaire dans `reduce(A)` ? Pourquoi? Donner un exemple.

Oui un élément majoritaire de A est forcément un élément majoritaire de $A' =$ `reduce(A)` .
En effet, `reduce(A)` transforme A en une liste qui "accentue" la fréquence d'apparation des éléments. C'est à dire que les éléments qui apparaissent fréquemment dans A apparaitront encore plus fréquemment dans A' et inversement (relativement aux tailles respectives de A et A' évidemment.)
Les éléments qui apparaissent peu souvent dans A vont simplement disparaître de A'.

Il est donc logique qui si a est l'élément majoritaire de A, alors il le sera aussi celui de A' (car sa fréquence d'apparition relative sera conservé ou amplifiée). En effet si a est majoritaire, il y aura au moins toujours une paire (a, a) qui sera formé dans `reduce(A)` .
(Par pigeonhole principle $|A|_a \geq \lfloor n/2 \rfloor + 1$ et on forme $n/2$ paires, on aura donc forcément une paire avec 2 a.)

On voit que cette garantie qu'un élément x ne disparaisse pas de A' n'est remplie qu'a partir de $|A|_x > \lfloor n/2 \rfloor$, or cette dernière impliquerait que x soit l'élément majoritaire.

Exemple:

$$A = [0, 0, 3, 3, 3, 2, 0, 0, 0, 0]$$

`reduce(A)` $= A' = [0, 3, 0, 0]$

`reduce(A')` $= A'' = [0]$

Et 0 est bien l'élément majoritaire.

1.2.2 Un élément majoritaire dans reduce(A) est-il nécessairement aussi un élément majoritaire dans A? Pourquoi? Donner un exemple.

Non pas forcément, suivant l'ordre dans lequel les éléments sont placés dans A (si un certain groupe est "rangé en paire"), on peut avoir un élément majoritaire qui se fait générer par `reduce(A)` sans qu'il soit celui de A.
Par exemple si on a deux éléments a, b t.q. $|A|_a = |A|_b$ les 2 proches de $n/2$.

$$A = [0, 0, 1, 0, 0, 0, 1, 1, 1]$$

`reduce(A)` $= A' = [0, 0, 1]$

`reduce(A')` $= A'' = [0]$

Et 0 n'est pas l'élément majoritaire. En effet, il apparait 5 fois et $5 < \lfloor\ |A|/2\ \rfloor + 1$.

1.2.3 Considérant le pire scénario pour les deux cas, en pratique, l'algorithme va-t-il s'exécuter plus rapidement avec 2^n éléments ou avec $2^n - 1$ éléments pour $n > 2$? Pourquoi? Donner un exemple.

Pour déterminer quand il va s'executer plus rapidement, on décompose les différents cas:

Avec les 2^n éléments car lorsque $|A|$ est impaire, l'algorithme doit d'abord enlever le dernier element de A puis vérifier si c'est l'élément majoritaire.
i.e. Dans le pire des cas avec $|A| = N = 2^n - 1$, l'algorithme va:

- prendre le dernier élément a_{N-1}
- appeler `is_majority_element(A,` a_{N-1}`)` qui va, au pire des cas, itérer sur tout A et déterminer qu'il ne l'est pas. ($O(2^n - 1)$)
- Puis appeler `reduce` au maximum $\log(2^{n-1}) = (n-1)$ fois sur `A[:-1]`
- On aura donc au pire $N + R(2^{n-1})$ opérations où $R(X)$ est le nombre d'opération que font récursivement les $(n-1)$ `reduce()` au total pour obtenir une liste de taille 0 ou 1, le tout sur une liste de taille X (dans le pire des cas).

Si $|A| = N = 2^n$, l'algorithme va:

- Appeler `reduce` au maximum $\log(2^n) = n$ fois sur `A`
- On aura donc $R(2^n)$ opérations

Soit $B := A[: -2]$ ($|B| = 2^{n-1}, |A| = 2^n$).
Si le fait d'avoir un 2 éléments de plus dans A va plus pénaliser le runtime total que la boucle en $O(2^n - 1)$ au debut de B. i.e. Si:

$$(2^n - 1 + R(2^{n-1})) - R(2^n) > 0$$

on aura que l'algorithme s'executera plus rapidement pour 2^n éléments.

On calcule $R(N)$:
On a vu dans le cours que `REDUCE` génère au plus $S(N) = \sum_{i=0}^{\log N} 2^i = 2N - 1$ opérations, sauf que l'on considère le pire des cas où on a une liste de taille impaire à chaque étape, ce qui rajoute N opérations à chaque fois. On trouve donc

$$S(N) = \sum_{i=0}^{\log N} (2^i + N) = \sum_{i=0}^{\log N} 2^i + \sum_{i=0}^{\log N} N = 2N - 1 + (N \log N)$$

Pour $\mathrm{N} = 2^n$, on obtient:

$$\begin{aligned} S(2^n) &= 2 \cdot 2^n - 1 + (2^n \cdot \log(2^n)) \\ &= 2^{n+1} - 1 + (2^n \cdot n) \\ &= \boxed{2^n(n+2) - 1} \end{aligned}$$

Pour $\mathrm{N} = 2^{n-1}$, on a donc $S(2^{n-1}) = 2^{n-1}(2 + (n-1)) - 1 \ = \boxed{2^{n-1}(n+1) - 1}$
Ce qui nous donne

$$\begin{aligned} (2^n - 1 + R(2^{n-1})) - R(2^n) &= (2^n - 1 + (2^{n-1}(n+1) - 1)) - (2^n(n+2) - 1) \\ &= 2^{n-1}(2 + n + 1) - 2 \ - \ 2^n(n+2) + 1 \\ &= 2^{n-1}(n+3) - 1 - 2^n(n+2) \\ &= -2^{n-1}(n+1) - 1 \end{aligned}$$

Ce qui est évidemment négatif pour tout $n > 2$.

L'algorithme effectue donc, $2^{n-1}(n+1) + 1$ opérations en plus pour un 2^n éléments.

Ceci en considérant que les `reduce` sur une liste de taille 2^n peut retourner des A' de taille impaire car on traite toujours le pire des cas possible.

1.3 Comparaison d'éxécution

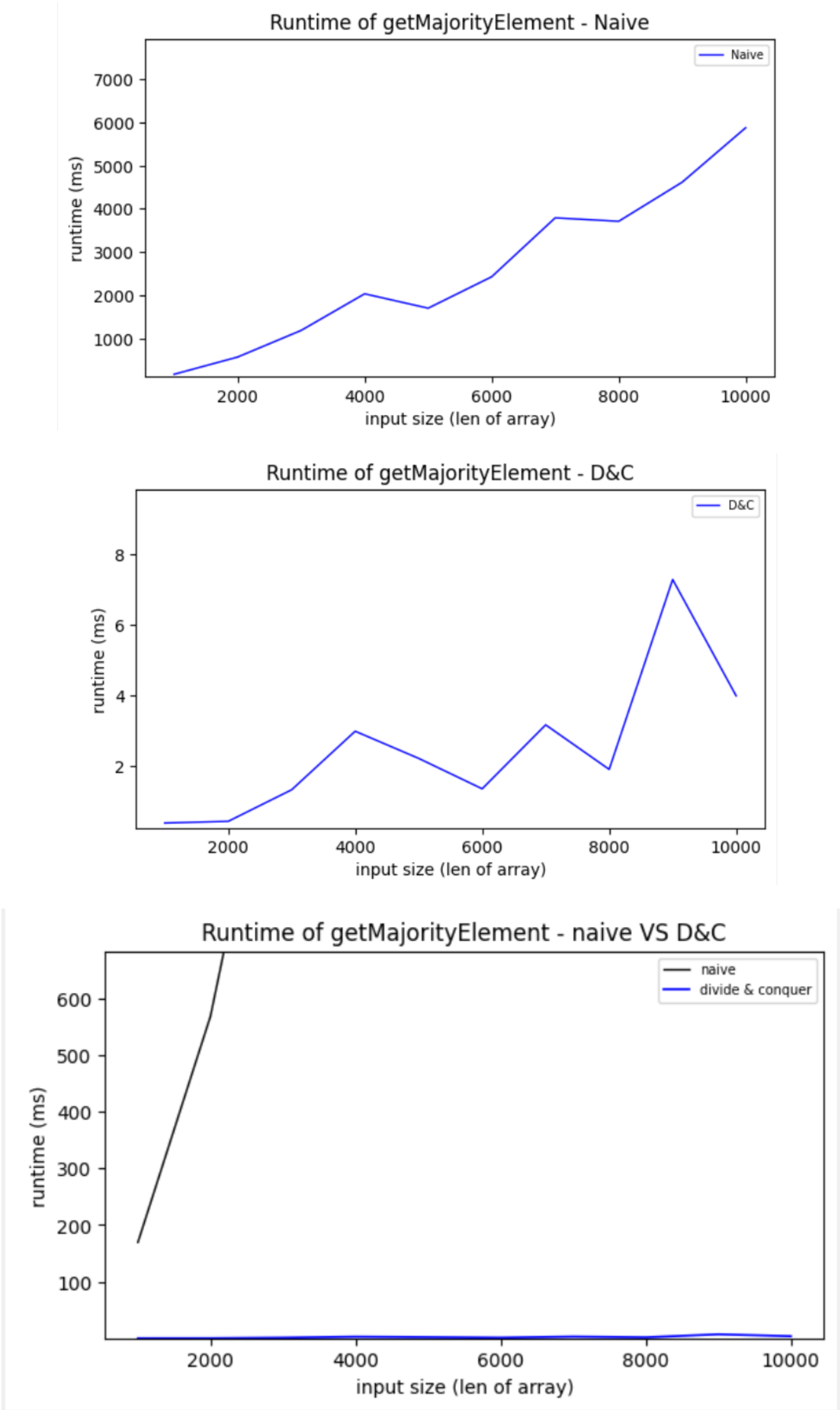

Figure 1: Runtime des fonctions naive (1.1), D&C (1.2) et les 2 ensembles (1.3)

On a comparé les runtimes de la fonction `naive()` ("getMajorityElement") et `dandc()` ("Divide&Conquer")

On voit que le runtime de `dandc()` devient tellement insignifiant devant celui de `naif()` que l'on a du mal à distinguer la courbe de `dandc()` de l'axe des abscisses ($t = 0$ms).

En effet le runtime de `naif()` explose si fort que la fonction met plus de 7 secondes a s'executer pour $n = 10000$, c'est plus de 1000 fois plus élevé que `dandc()` pour un input de la même taille.

2 Exponentiation

2.1 Algorithme naïf

On a l'implémentation naïve ci-dessous, et on cherche sa complexité.

```
def exp_naive(base, p):
    """Naive implementation of exponentiation"""
    pr = 1
    for _ in range(p):
        pr *= base
    return pr
```

Soit $b :=$ `base`, on voit que sa complexité est $O(p \cdot f(b^{p-1},\ b))$ où $f(x, y)$ est la complexité du produit de x par y. (Complexité du produit qui va de $O(n^2)$ pour la multiplication "longue" i.e. basique et jusqu'à $O(n \log n)$ pour les algos plus optimisés)
Il est en tout cas $\Omega(n)$

2.2 Algorithme D & C

Dans l'algorithme divide & conquer, on a une fonction `exp_dandc(b, p)` qui va appeler une autre fonction recursive `exp_rec` qui va être appelé pour chaque puissance de 2 qui composante p (i.e. la notation binaire de p). `exp_rec` se fait donc appeler $\lfloor \log p \rfloor$ fois.

La complexité de chacun de ses appel est constituée de au pire le produit entre $b^{(2^{i-1})}$ et lui même (l.197 de *tp1.py*) plus celui de $b^{(2^i)}$ et des autres $\prod_{k=0}^{i-1} b^{(2^k)}$ avant (l.198 de *tp1.py*).
La complexité de `exp_rec` reste donc logarithmique comme demandé.

2.3 Comparaison

De la même manière que précedemment, on voit sur les graphs ci-dessous, que le runtime de `exp_dandc()` devient insignifiant devant celui de `exp_naif()`, que l'on confondrait l'axe des abcsisses avec la courbe bleu.

Ici l'écart est moins marqué mais il n'en reste pas moindre.
Pour un nombre qui avoisine les 4 chiffres l'algorithme naif met presque 2ms à répondre contre 0.030 - 0.040ms pour la version divide and conquer.
(Voir graphs en page 5.)

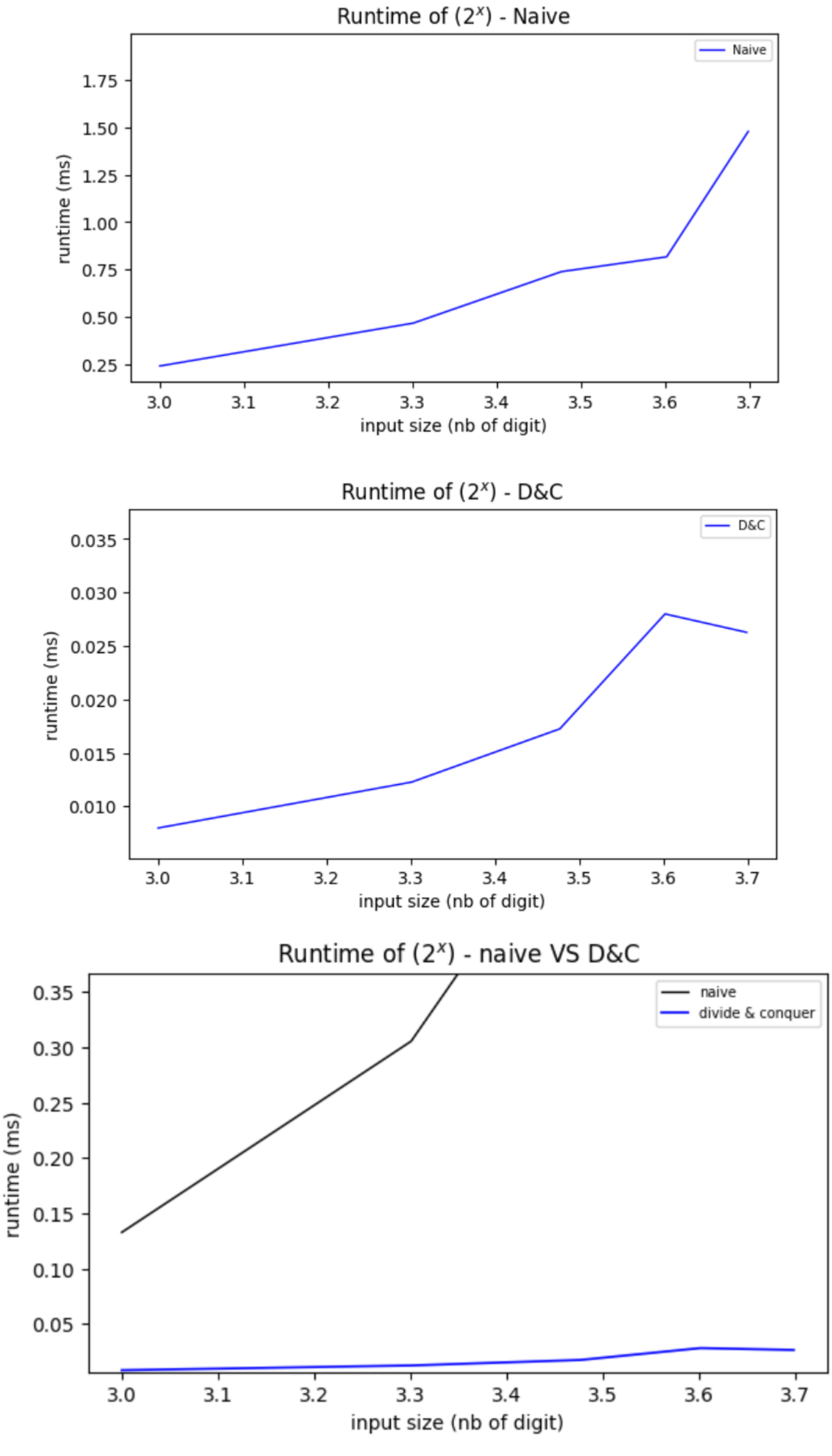

Figure 2: Runtime des fonctions `exp()` pour les solutions naive (2.1), D&C (2.2) et les 2 ensembles (2.3)