Feuille de travaux dirigés n^o 2
Structures de données

Exercice 2.1 — Recherche des k plus petits éléments

On considère T un tableau non trié de n entiers.

1. Écrivez un algorithme qui calcule et affiche les k plus petits éléments en procédant par recherches successives.

▽ **Correction**

```
/* on fait comme un tri par sélection */
/* en début de tableau */
i <- 1
tant que (i <= k) {
  min <- T[i]
  jmin <- i
  j <- i + 1
  tant que (j <= n) {
    si (T[j] < min) {
      min <- T[j]
      jmin <- j
    }
    j <- j + 1
  }
  /* on échange la valeur min et la valeur i */
  T[jmin] <- T[i]
  T[i] <- min
  afficher T[i]
  i <- i + 1
}
```

2. Écrivez un algorithme qui calcule et affiche les k plus petits éléments en triant le tableau au préalable (la fonction de tri par ordre croissant est donnée *tri*(T) ; son coût est : $n\log(n)$)

▽ **Correction**

```
tri(T)
i <- 1
tant que (i <= k) {
  afficher T[i]
  i <- i + 1
}
```

3. Comparez le temps mis par chacun des algorithmes pour $k = 1..n$.

▽ **Correction**

La première version dans tous les cas il faut regarder les n éléments pour trouver le plus petit élément, et on le fait k fois. Donc la complexité est $O(kn)$.

La seconde version on fait d'abord un tri en $O(n\log(n))$ puis on récupère les k premiers éléments. Donc la complexité est en $O(n\log(n)+k)$.

Quand k est petit la première est plus efficace que la seconde, quand k rest proche de n, la seconde est plus efficace que la première.

Exercice 2.2 — Rappel sur les logarithmes

La fonction logarithme est simplement l'inverse de la fonction exponentielle. Il y a équivalence entre $b^x = y$ et $x = \log_b(y)$.
En informatique, le logarithme de base 2 est souvent utilisé.
Le logarithme de n reflète combien de fois que l'on doit doubler le nombre 1 pour obtenir le nombre n. De façon équivalente il reflète aussi le nombre de fois où l'on doit diviser n pour obtenir 1.
On vous rappelle que la base du logarithme n'a pas d'importance

$$\log_b(a) = \frac{\log_c(a)}{\log_c(b)}$$

Par la suite, on utilisera uniquement des logarithmes à base 2.

1. Combien de fois doit-on doubler 1 pour obtenir n ?

▽ **Correction**

$\lceil\log(n)\rceil$ fois.

2. Combien de bits a-t-on besoin pour représenter le nombre 2^i, le nombre n et les nombres de 0 à $2^{32}-1$?

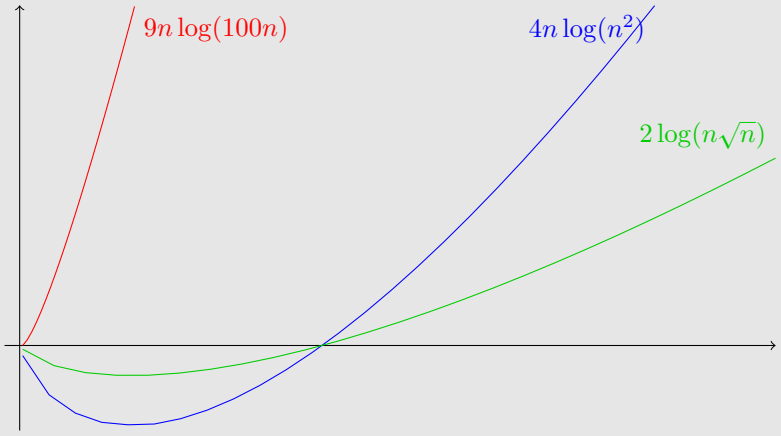
▽ **Correction**

pour 2^i -> $i+1$ bits
pour n -> $\lfloor\log(n)\rfloor$ bits
pour les nombres de 0 à $2^{32}-1$ -> 32 bits

3. Supposez que vous ayez trois algorithmes en face de vous, le premier faisant $9n\log(100n)$ opérations, le second $4n\log(n^2)$ et le troisième $2\log(n\sqrt{n})$. Classez les algorithmes selon leur performance en fonction de la valeur de n.

▽ **Correction**

$9n\log(100n) = 9n(\log(n) + log(100))$
$54n + 9n\log(n) \leqslant 9n(\log(n) + log(100)) \leqslant 63n + 9n\log(n)$
$4n\log(n^2) = 8n\log(n)$
$2\log(n\sqrt{n}) = 3\log(n)$



Exercice 2.3 — Le juste prix

Votre voisin choisit un nombre entre 1 et 1000. Vous devez deviner ce nombre le plus rapidement possible, autrement dit en faisant le moins d'essais possibles. À chaque essai, vous proposez un nombre et votre voisin vous dit si le nombre choisi est supérieur ou inférieur. Si vous avez trouvé le bon nombre, il doit aussi vous le dire. Proposez une stratégie qui minimise le nombre maximal d'essais.
Quelle est la probabilité de trouver le bon nombre en choisissant 10 nombres au hasard ? 20 ? 50 ? Comparez avec le nombre d'essais de votre stratégie.

▽ **Correction**

Utiliser la dichotomie, en $\log(1000) \simeq 10$

Exercice 2.4 — Dynamic Array

Le but de cet exercice est de vous faire comprendre les tableaux dynamiques, autrement dit des tableaux dont la taille augmente selon les besoins. On va simuler leur fonctionnement.
Dans les ordinateurs, la mémoire est gérée comme un immense tableau (on a accès immédiatement à n'importe quelle case) dans lequel on réserve des sous-tableaux, qui représentent les parties mémoire réservées.
On nomme M cette mémoire et $entier$ ALLOUER($entier$ n) la fonction qui réserve n cases consécutives dans la mémoire et qui renvoie l'indice de la première case mémoire réservée.

Principe de l'algorithme :

1. On crée un dynamic array de n cases.
2. On reçoit des valeurs et on les place successivement dans le dynamic array. Le nombre de valeurs à placer n'est pas connu à l'avance. Si le nombre de valeurs est supérieur à n, alors on agrandit le dynamic array de la façon suivante :
   — on crée un autre dynamic array dont la taille est doublée ;
   — on copie les valeurs de l'ancien dynamic array dans le nouveau ;
   — on va à l'étape 2.

Questions :

1. Écrivez le pseudo-code correspondant à cet algorithme. La fonction $entier$ GÉNÉRATEUR() fournit les valeurs attendues. Elle renvoie -1 pour signifier la fin de la génération des valeurs.

▽ **Correction**

```
– version mémoire
  début <– ALLOUER(n)
  i <– 0
  nb <– GÉNÉRATEUR()
  tant que (nb != –1) {
    si (i = n) {
      nouv <– ALLOUER(2 * n)
      recopie(nouv, début, i) // recopie des i éléments
      n <– n * 2
      début <– nouv
    }
    M[début + i] <– nb
    i <– i + 1
    nb <– GÉNÉRATEUR()
  }
  // n cases allouées, i éléments pertinents

– version à la C
  T <– ALLOUER(n)
  i <– 1
  nb <– GÉNÉRATEUR()
  tant que (nb != –1) {
    si (i = n + 1) {
      Tbis <– ALLOUER(2 * n)
      recopie(Tbis, T, n) // recopie des n éléments
      n <– n * 2
      T <– Tbis
    }
    T[i] <– nb
    i <– i + 1
    nb <– GÉNÉRATEUR()
  }
  // n cases allouées, i–1 éléments pertinents
```

2. Supposez qu'à la fin vous ayez m éléments en ayant commencé avec $n = 1$. Combien de fois la fonction ALLOUER a-t-elle été appelée ? Calculez le nombre total de cases consommées. Combien de copies de valeurs ont été faites ?

▽ Correction

Pour m éléments en ayant commencé avec $n = 1$, on a fait appel à $p = \lceil \log(m) \rceil$ fois la fonction ALLOUER. On a donc consommé $\sum_{i=1}^{p} 2^i = 2^{p+1} - 1$ cases mémoire et effectué $\sum_{i=1}^{p-1} 2^i = 2^p - 1$ copies.

3. Que pensez-vous des dynamic arrays ?

▽ Correction

Pratique car cela permet d'avoir les avantages d'un tableau sans forcément connaître la taille à priori. Malheureusement dés qu'il faut doubler la taille c'est coûteux à la fois en mémoire et en temps.

4. On reçoit un fichier de notes du bac entre 0 et 20. Pour commencer, on doit le mettre en mémoire. Comme on ne connaît pas le nombre de notes à l'avance, on décide d'utiliser le principe des dynamic arrays. Puis, on veut faire un histogramme des notes, autrement dit compter le nombre de fois où chaque note apparaît. On supposera que les notes ne sont qu'entières. Écrivez l'algorithme réalisant ces opérations.

▽ **Correction**

```
début_compteur <- ALLOUER(21) // tableau pour compter les notes de 0 à 20
i <- 0
tant que (i <= 20) {
  M[début_compteur + i] <- 0 // 0 notes au début
}

début_notes <- ALLOUER(n)
i <- 0
note <- GÉNÉRATEUR()
tant que (note != -1) {
  si (i = n) {
    nouv <- ALLOUER(2 * n)
    recopie(nouv, début_notes, i) // recopie des i éléments
    n <- n * 2
    début_notes <- nouv
  }
  M[début_notes + i] <- note
  M[début_compteur + note] <- M[début_compteur + note] + 1
  i <- i + 1
  note <- GÉNÉRATEUR()
}
```