TP : Algorithmes de tri

Début novembre 2012, on estimait à 40 milliards le nombre de pages Internet indexées par les principaux moteurs de recherches. Avec un tel volume d'information, il est essentiel de pouvoir accéder rapidement aux résultats les plus pertinents, et pour cela d'être capable de trier efficacement un grand nombre d'éléments.

L'objectif de ce TP est de vous faire découvrir quelques algorithmes de tri et de vous initier à cette notion d' "efficacité" d'un programme, ce qu'on appelle formellement la complexité.

1) Support en ligne

1.a) Récupérer les fichiers

Pour rendre les choses plus faciles à appréhender et pour vous faire gagner du temps, un certain nombre d'outils sont déjà implémentés.

Commencez donc par récupérer le fichier disponible à l'adresse suivante (il vous suffit de cliquer sur la séance correspondant à ce TP puis sur *Fonctions pré-implémentées*) :

http://andre.lovichi.free.fr/teaching/ea/2012-2013

Enregistrez ce fichier dans votre répertoire de travail habituel, et ouvrez-le avec Emacs.

Vous pouvez jeter un œil aux différentes fonctions, mais ne vous attardez pas trop et ne vous inquiétez pas s'il y a des lignes de code que vous ne comprenez pas. L'essentiel est en outre décrit dans les paragraphes suivants.

1.b) Les outils à votre disposition

Génération aléatoire

Pour étudier les tris, il est pratique de disposer d'un grand nombre de vecteurs pour faire des tests. Plutôt que de les créer manuellement élément par élément, vous allez pouvoir générer automatiquement des vecteurs dont le contenu sera tiré au hasard.

Pour commencer, on spécifie quelques variables globales (vous pouvez modifier ces valeurs mais conservez les mêmes ordres de grandeur pour éviter des temps de calcul trop longs) :

- `nb_elements` : le nombre d'éléments, c'est-à-dire la taille des vecteurs
- `vmin` et `vmax` : les valeurs minimale et maximales (la valeur de chaque élément sera tirée au hasard entre ces deux bornes)

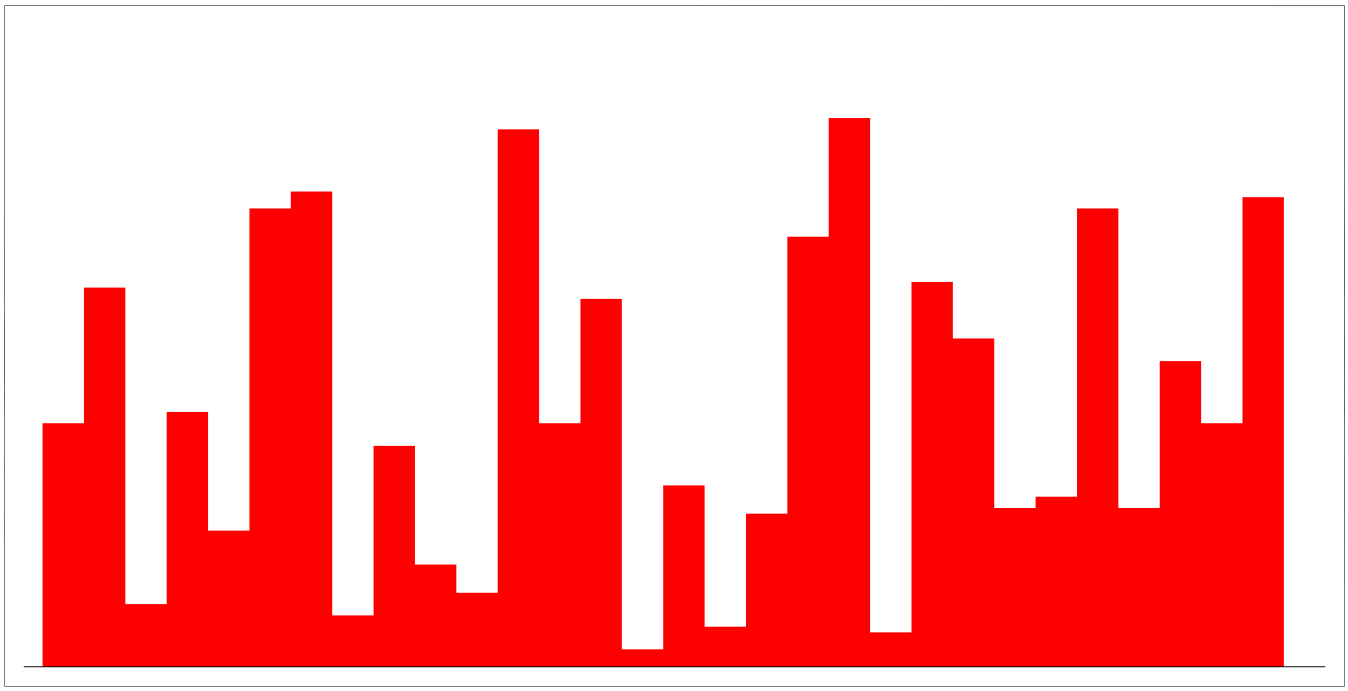
Une fois ces variables choisies, il vous suffit d'utiliser la fonction `random_vect : unit -> int array` pour obtenir un nouveau vecteur non trié.

Lorsqu'on parlera de complexité à la fin du TP, on travaillera avec des vecteurs plus grand. On utilisera pour cela la fonction `random_big_vect : int -> int array` qui prend comme argument la taille du vecteur souhaité.

Affichage

Pour visualiser les effets des différents algorithmes de tri, on va utiliser le module `Graphics` pour réaliser un affichage de nos vecteurs sous forme d'histogrammes.

Comme on peut le voir sur l'exemple ci-dessous, la $i^{ème}$ case du vecteur correspond à la $i^{ème}$ barre de l'histogramme, qui est d'autant plus haute que la valeur de cette case est importante (ici on a un vecteur de taille 30 avec des valeurs comprises entre 0 et 100) :



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
43	67	11	45	24	81	84	9	39	18	13	95	43	65	3	32	7	27	76	97	6	68	58	28	30	81	28	54	43	83

En pratique, vous avez deux fonctions à votre disposition pour afficher un vecteur :

- `tracer_vecteur : int array -> unit` : qui trace le vecteur dans la fenêtre graphique
- `tracer_vecteur_et_attendre : int array -> int -> unit` qui trace le vecteur puis attend `t` millisecondes, `t` étant le second argument de la fonction

On utilisera cette seconde fonction avec des durées de 50 ou 200 millisecondes pour avoir le temps d'observer ce qui se passe pendant qu'on effectue un tri.

N'hésitez pas à tester ces fonctions sur quelques exemples avant d'aller plus loin !

Attention

N'oubliez pas réévaluer l'ensemble de votre code si vous changez les variables globales (sinon certaines fonctions risquent de travailler avec les valeurs précédentes et vous vous exposez à des risques de plantages).

1.c) Votre première fonction

Avant de vous attaquer aux algorithmes de tri, commencez par implémenter (et tester) une petite fonction `echanger : int array -> int -> int -> unit` telle que `echanger v i1 i2` échange les contenus des cases d'indice `i1` et `i2` dans le vecteur `v`.

Testez cette fonction sur des exemples, et essayez de visualiser la différence grâce aux fonctions d'affichage.

2) Tri par sélection

2.a) Principe

Le tri par sélection repose sur une idée assez simple :

- On trouve le plus petit élément du vecteur
- On le place au début du vecteur
- On cherche le deuxième plus petit élément
- On le place dans la deuxième case du vecteur
- etc.

2.b) Implémentation

Implémentez une fonction `tri_par_selection : int array -> unit` qui trie un vecteur selon cette méthode.

Indice : une fois qu'on a placé le plus petit élément dans la première case, le deuxième plus petit élément est le minimum du sous-vecteur `v[1..nb_elements-1]`, composé des indices `1` à `nb_elements-1`.

Affichage : Vous pouvez par exemple utiliser la fonction `tracer_vecteur_et_attendre` à chaque fois que vous déplacez un élément, avec un délai de 200 millisecondes.

3) Tri à bulles

3.a) Principe

L'idée de ce tri est de faire "remonter" les éléments les plus grands vers la fin du vecteur, un peu comme des bulles vers la surface d'un liquide.

Pour cela :

- On parcourt le vecteur du début à la fin
- Si on a `v.(i) > v.(i+1)`, alors on inverse ces deux éléments (`v.(i)` "remonte")
- Si on arrive à la fin du vecteur sans qu'il y ait d'échange, c'est que le vecteur est trié
- Sinon on refait un passage
- etc.

3.b) Implémentation

Implémentez une fonction `tri_a_bulles : int array -> unit` qui trie un vecteur selon cette méthode.

Indice : on a déjà vu deux approches depuis le début de l'année pour faire plusieurs fois la même chose (ici parcourir le vecteur et faire des échanges si besoin est), mais sans savoir à l'avance le nombre de répétitions nécessaires...

Affichage : Vous pouvez par exemple utiliser la fonction `tracer_vecteur_et_attendre` à chaque échange, avec un délai de 50 millisecondes.

4) Tri par insertion

4.a) Principe

Ce tri est aussi appelé "tri du joueur de carte", car il correspond à ce que fait naturellement un joueur de cartes à qui on distribue progressivement des cartes :

- Le joueur a dans sa main des cartes déjà triées
- Il reçoit une nouvelle carte
- Il l'insère au bon endroit dans sa main
- Il reçoit une nouvelle carte
- etc.

4.b) Implémentation

Implémentez une fonction `tri_par_insertion : int array -> unit` qui trie un vecteur selon cette méthode.

Indice : L'idée est de considérer au début du vecteur une main triée (en bleu clair) et d'insérer les éléments un par un (en bleu foncé) en les faisant glisser de la droite vers la gauche jusqu'à la position souhaitée :

1	4	9	3	7	5
1	4	3	9	7	5
1	3	4	9	7	5
1	3	4	9	7	5

Affichage : Vous pouvez utiliser `tracer_vecteur_et_attendre` à chaque échange, avec un délai de 50 millisecondes.

5) Premières notions de complexité

Dans cette dernière partie, l'idée va être de comparer l'efficacité des différents algorithmes implémentés jusque-là.

Pour cela, on utilise la fonction `random_big_vect`, qui prend comme argument la taille du vecteur que l'on souhaite générer (10000 éléments est très bien pour commencer).

Comparez l'efficacité de vos différentes méthodes en mesurant le temps requis pour chacune d'entre elles. Pour plus de précision, vous pouvez ajouter les lignes de codes suivantes à vos fonctions pour avoir une estimation (en secondes) du temps de calcul nécessaire :

- Au début de votre fonction : `let t1 = Unix.gettimeofday() in`
- A la fin de votre fonction : `let duree = Unix.gettimeofday() -. t1 in print_string "Temps de calcul "; print_float duree;;`