Université de sciences et technologies Houari Boumedian

Faculté d'Informatique

TP N°4 : Les TRI

Algorithmique avancé et complexité
Prof: Mme.Moussaoui

Etudiants:

- **Belmouloud Mustapha Abdellah** : partie Redaction Rapport
- **Merazi Wail :** partie Generation des données
- **Ben chaban Razika :** partie Code
- **Tabti Ikram :** partie Test

1 Partie theorique

1.1 Tri a Bulles

complexité: O(n^2)→ boucle imbriqué n x n

complexité optimisé: O(n^2) boucle imbriqué 1+2+3+....n-1 borné par O(n^2)

complexité spacial: O(1) → taille de vecteur seulemnt, necessite pas d'autre allocation

Le Tri à Bulles fonctionne par itération sur le tableau, comparant et échangeant des paires d'éléments adjacents s'ils sont dans le désordre. . À chaque passage, l'élément le plus grand "remonte" (ou "bulle") à sa position finale.

Tri à Bulles Optimisé : Il ajoute une optimisation par drapeau (flag) pour détecter l'absence d'échange lors d'un passage complet. Si aucun échange n'est effectué, le tableau est trié, et l'algorithme s'arrête

1.2 Tri Gnome

complexité: O(n^2) aux pire cas où il doit permuter avec tout elemnt precedent de i a chaque nouveau indice donc 1+2+3+....n borné par O(n^2)

complexité spacial: O(1) → taille de vecteur seulemnt, necessite pas d'autre allocation

Le Tri Gnome parcourt le tableau séquentiellement. Il compare l'élément courant avec l'élément précédent, si ils sont dans le bon ordre l'indice est incrémenté sinon ils sont permutés, et l'indice est décrémenté pour vérifier à nouveau la nouvelle paire formée.

1.3 Tri Par Destribution

complexité: O(n x k) nombre d'élément x nombre de chifre de plus grand elemenet dans certain cas peut approcher a O(n)

complexité spacial: theoriquement O(1) mais la majorité des implementations necessite des vecteur temporaires donc pratiquement O(n x k) → SEULEMENT SI ON NÉGLIGE LA MEMOIRE NÉCESSAIRE POUR LES APPELLES RECURSIVES LARGEMENT DEPENDANTE SUR LE LANGUAGE DE PROGRAMATION

Le Tri par Destribution compare tout les element et les classe chifre par chifre commançcant par le chifre moins significatif, c'est un algorithm iteratif qui est theoriquement un excelent melange entre la simplicité et l'éfficacité

1.4 Tri Rapid

complexité: O(n.log(n)) vers O(n^2)

complexité spacial: theoriquement O(1) mais la majorité des implementations necessite des vecteur temporaires donc pratiquement O(n.log(n)) où meme o(n) → SEULEMENT SI ON NÉGLIGE LA MEMOIRE NÉCESSAIRE POUR LES APPELLES RECURSIVES LARGEMENT DEPENDANTE SUR LE LANGUAGE DE PROGRAMATION

Le Tri Rapid largement depend sur le concpet du pivot, qui est un element a utilisé pour classer les elements qui sont inferieur a gauche et les elements superieur a droit, il depend sur la recursivité et la decomposition de vecteur a deux a chaque fois, son eficacité largement depend sur le choix du pivot.

1.5 Tri Par Tas

complexité: O(n.log(n)) a cause de le parcour et la transformation de vecteur en tas

complexité spacial: O(1), il necessite pas des allocations de memoire suplementaires → SEULEMENT SI ON NÉGLIGE LA MEMOIRE NÉCESSAIRE POUR LES APPELLES RECURSIVES LARGEMENT DEPENDANTE SUR LE LANGUAGE DE PROGRAMATION

Le Tri par Tas utilise la structure de données Tas qui est une implémentation d'arbre binaire qui respecte la propriété de tas (le parent est toujours plus grand que ses enfants), Le tableau d'entrée est d'abord transformé en un tas, ensuite, la racine est échangé avec le dernier élément non trié du tas. Le tas est ensuite réctifié pour rétablir la propriété de tas. Ce processus est répété N fois, le tri s'effectuant en place à la fin du tableau.

2 les fonctions utilisé pour chaque algorithme

```
        pour le tri rapid le pivots a été choisi d'une façon aléatoire

void bull(int *vec, int n){
    int stop = 0;
    while(stop==0){
        int swap = 0;
        for(int i=0; i<n-1; i++){
            if(vec[i]>vec[i+1]){
                int y = vec[i];
                vec[i]=vec[i+1];
                vec[i+1] = y;
                swap = 1;
            }
        }
        if(swap == 0) stop = 1;
    }
```

```
}

//tri bull optimisé
void optbull(int *vec, int n){
    int stop = 0;
    int m = n-1;
    while(stop==0){
        int swap = 0;
        for(int i=0; i<n-1; i++){
            if(vec[i]>vec[i+1]){
                int y = vec[i];
                vec[i]=vec[i+1];
                vec[i+1] = y;
                swap = 1;
            }
        }
        m=m-1;
        if(swap == 0) stop = 1;
    }
}

//tri gnome
void gnome(int *vec, int n){
    int i = 0;
    int temp;
    while (i<n-1){
        if(vec[i]<=vec[i+1]){
            i++;
        }else{
            temp = vec[i];
            vec[i] = vec[i+1];
            vec[i+1] = temp;
            if(i>0) i--;
        }
    }
}


//tri par distribution
int cle(int x, int i) {
    while (i-- > 0)
    x /= 10;
    return x % 10;
}

void TriAux(int *T, int n, int i) {
    int count[10] = {0};
    int *output = malloc(n * sizeof(int));

    for (int j = 0; j < n; j++) {
        int d = cle(T[j], i);
```

```
        count[d]++;
    }

    for (int d = 1; d < 10; d++)
    count[d] += count[d - 1];

    for (int j = n - 1; j >= 0; j--) {
        int d = cle(T[j], i);
        output[count[d] - 1] = T[j];
        count[d]--;
    }

    for (int j = 0; j < n; j++)
    T[j] = output[j];

    free(output);
}

void TriBase(int *T, int n, int k) {
    for (int i = 0; i < k; i++)
    TriAux(T, n, i);
}

//tri par tas
void heapSort(int *arr, int n) {
    void heapify(int *arr, int n, int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && arr[left] > arr[largest])
        largest = left;
        if (right < n && arr[right] > arr[largest])
        largest = right;

        if (largest != i) {
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;
            heapify(arr, n, largest);
        }
    }

    for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);

    for (int i = n - 1; i >= 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
```

```
    }
}

//tri rapid
void quickSort(int *arr, int n) {
    int partition(int *arr, int n) {
        if (n <= 1) return 0; // Guard against arrays with 0 or 1 elements
        int pivot_index = rand() % n;
        int temp_pivot = arr[pivot_index];
        arr[pivot_index] = arr[n - 1];
        arr[n - 1] = temp_pivot;

        int pivot = arr[n - 1];
        int i = -1;

        for (int j = 0; j < n - 1; j++) {
            if (arr[j] <= pivot) {
                i++;
                int tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
            }
        }

        int tmp = arr[i + 1];
        arr[i + 1] = arr[n - 1];
        arr[n - 1] = tmp;

        return i + 1;
    }
    if (n < 2)
    return;

    int pi = partition(arr, n);
    quickSort(arr, pi);
    quickSort(arr + pi + 1, n - pi - 1);
}
```

3 Fonction pour la géneration des données

```
void reset(int *arr, int order, int n){
    if(order == 0){
        for (int i = 0; i < n; i++){
            arr[i] = rand() % n;
        }
    }else if (order == 1)
    {
        for (int i = 0; i < n; i++){
            arr[i] = i;
        }
```

```
    }else if (order==2)
    {
        for (int i = 0; i < n; i++){
            arr[i] = n-i;
        }
    }
}
```

3.1 La Partie Tests

les tests ont été faits sure un laptop i5 2450M, 2 cores @2.5Ghz et 8GB de ram avec un system d'operation windows10
A cause de constrainte de RAM le test des vecteurs de tailles plus grande que 512 million (a peut prés allocation de 2GB DE RAM) n'etaient possible, le compliateur de c affiche "killed" a chaque fois, aussi les algorithmes de complexité $O(n^2)$ ont prennent plus de 2 heurs chacun pour les tailles plux grands que 400 000 elements, donc on a arreté le test a 400 000 elements pour gnome et bulles et a 512 millions pour le reste

3.1.1 la fonction utilisée pour mesurer le temps d'execution

la fonction utilisé pour mesurer le temps:

```
double now() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec * 1e-9;
}
```

3.1.2 les tableaux resultants

Table 1: Temps d'Exécution des Algorithmes (Données Aléatoires) en secondes

Taille (N)	Tri à Bulles (s)	Tri Opt. (s)	Tri Gnome (s)	Tri Rapide (s)	Tri par Tas (s)	Distribution (s)
1 000	0.0075	0.0066	0.0050	0.0002	0.0004	0.0003
50 000	17.1187	15.9330	6.7160	0.0168	0.0267	0.0145
100 000	66.8822	65.8661	26.7150	0.0352	0.0499	0.0220
200 000	277.0107	274.6002	107.3310	0.0603	0.0915	0.0419
400 000	1127.4734	1084.1380	423.0636	0.1406	0.1878	0.0763
800 000	—	—	—	0.2371	0.4093	0.1535
16 000 000	—	—	—	5.6972	16.6891	5.0405
32 000 000	—	—	—	11.5819	35.9738	10.1005
64 000 000	—	—	—	24.1433	80.9130	20.2348
128 000 000	—	—	—	48.8077	183.5881	48.7726
256 000 000	—	—	—	102.7933	417.1062	97.7832
512 000 000	—	—	—	211.0458	930.9430	195.0532

Table 2: Temps d'Exécution des Algorithmes (Données Triées) en secondes

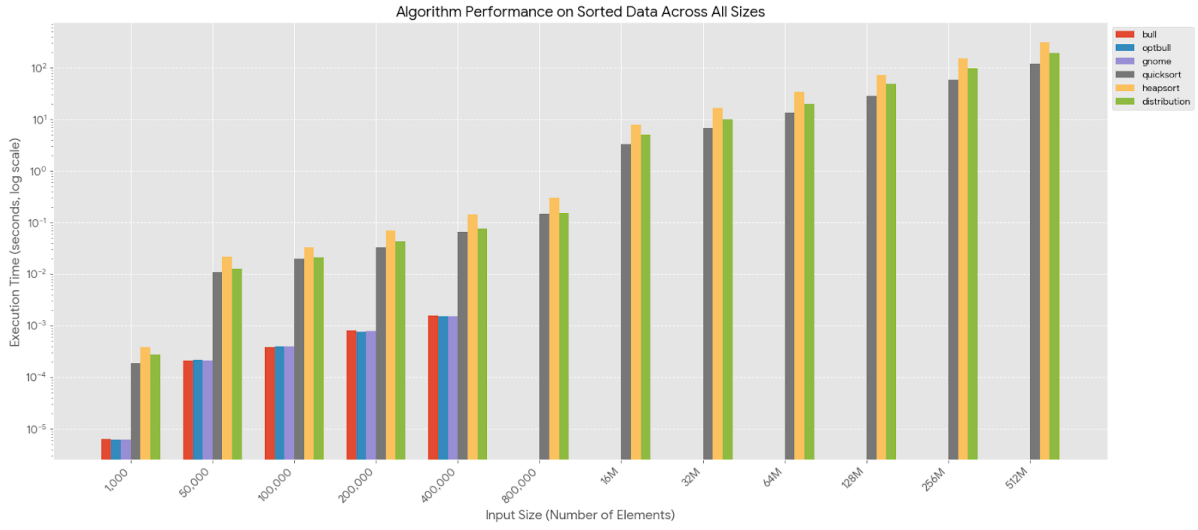
Taille (N)	Tri à Bulles (s)	Tri Opt. (s)	Tri Gnome (s)	Tri Rapide (s)	Tri par Tas (s)	Distribution (s)
1 000	0.0000	0.0000	0.0000	0.0002	0.0004	0.0003
50 000	0.0002	0.0002	0.0002	0.0109	0.0216	0.0126
100 000	0.0004	0.0004	0.0004	0.0200	0.0334	0.0214
200 000	0.0008	0.0008	0.0008	0.0331	0.0702	0.0432
400 000	0.0016	0.0015	0.0015	0.0655	0.1439	0.0777
800 000	—	—	—	0.1480	0.3031	0.1536
16 000 000	—	—	—	3.3034	7.8216	5.0467
32 000 000	—	—	—	6.7303	16.5577	10.0514
64 000 000	—	—	—	13.6136	34.7580	20.2269
128 000 000	—	—	—	28.5567	72.6117	48.6163
256 000 000	—	—	—	58.2663	151.3416	97.3471
512 000 000	—	—	—	119.1080	313.7199	194.3825

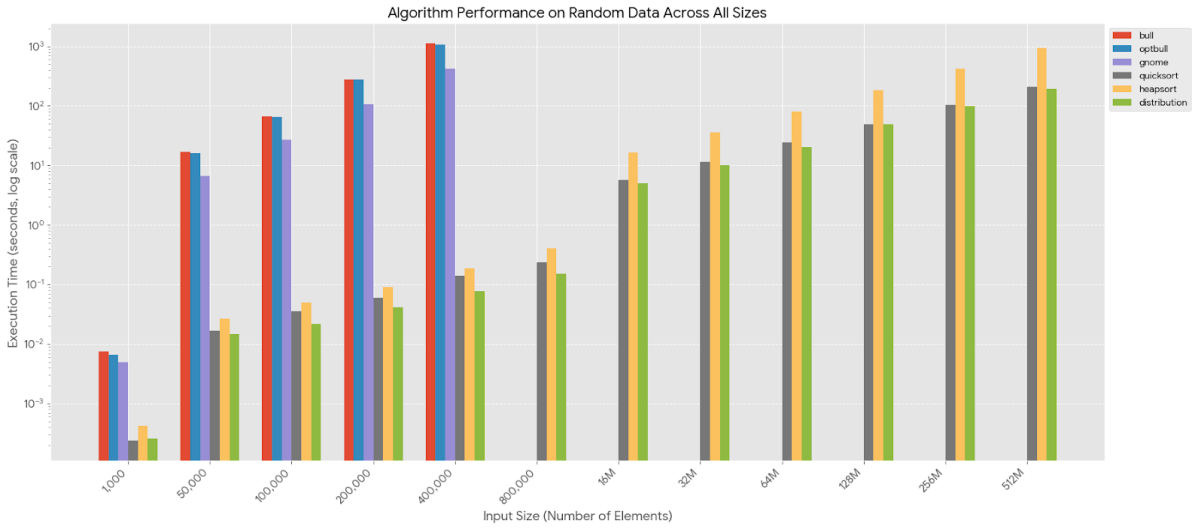
Table 3: Temps d'Exécution des Algorithmes (Données Triées à l'Envers) en secondes

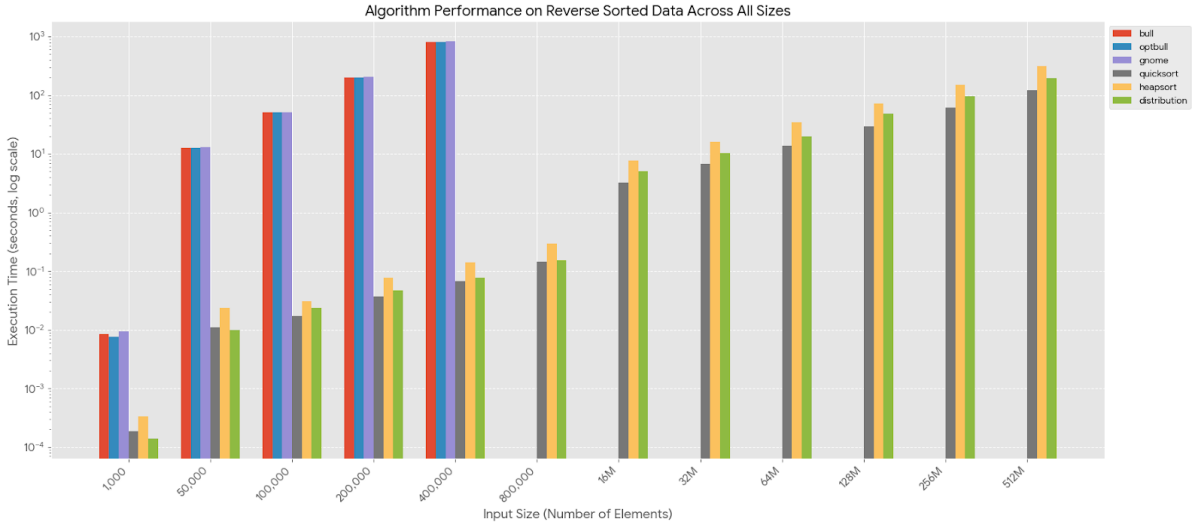
Taille (N)	Tri à Bulles (s)	Tri Opt. (s)	Tri Gnome (s)	Tri Rapide (s)	Tri par Tas (s)	Distribution (s)
1 000	0.0084	0.0076	0.0095	0.0002	0.0003	0.0001
50 000	12.7670	12.7488	12.9788	0.0111	0.0233	0.0099
100 000	50.6943	50.6623	51.4899	0.0170	0.0311	0.0238
200 000	202.8847	203.7637	208.3136	0.0373	0.0773	0.0473
400 000	815.2160	809.3667	833.2695	0.0673	0.1404	0.0775
800 000	—	—	—	0.1456	0.2983	0.1530
16 000 000	—	—	—	3.2664	7.6600	5.0308
32 000 000	—	—	—	6.7328	16.2266	10.1634
64 000 000	—	—	—	13.9132	34.1118	20.1326
128 000 000	—	—	—	29.2840	71.8379	48.6382
256 000 000	—	—	—	60.8722	150.2297	97.3845
512 000 000	—	—	—	123.1465	312.8984	194.5394

3.1.3 les Graphs resultants

les temps d'executiosn ont été normalisé par log pour mieux visialiser les resultats







4 Remarque et Conclusion finale

Les résultats théoriques et pratique sont compatibles, on remarque que les algorithmes de complexité O(N^2) sont boucoup trop lents mais le plus rapid entre eux était gnome, entre les algorithmes plus rapid de complexité plus simple pour les données aleatoir tri par destribution etait le plus rapid ce que reflait l'étude theorique, le tri rapid est le deuxieme plus rapid et le tri par tas est 4 fois plus long ce qui est a cause a la necessité de creation des tas
pour les données deja triées ou triées inversement le tri rapid pérforme mieux que le tri par destribution, car ça facilite le travaille du pivot
on remarque aussi que le tri par destribution était le plus stable avec des temps presque identiques pour les données aléatoires, triées and triées inversement .