Proiect sortări

Dobricean Ioan-Dorian
Dănăilă Mihai-Teodor
Gheorghieș Petruț-Rareș

Sortările studiate

1. Radix Sort (baza 2^16 și baza 10)
2. Merge Sort
3. Shell sort
4. Count Sort
5. Tim sort
6. Heap sort
7. std::sort

Radix Sort

- **Radix Sort**: Sortează elementele pe baza unităților (cifre sau caractere).
- **Versatilitate**: Funcționează în orice bază numerică, nu doar în baza 10, noi definim care sunt unitățile.
- **Complexitate**: O(nk), unde n este numărul de elemente, iar k numărul de unități.
- **Utilizare**: Eficient pentru numere întregi sau șiruri de caractere, mai ales când k nu este mare.
- **Avantaj**: Performanță bună pe seturi mari de date, fără comparații directe între elemente.

Radix Sort

```
int getMax(int arr[], int n) {
        int mx = arr[0];
        for (int i = 1; i < n; i++)
        if (arr[i] > mx)
        mx = arr[i];
        return mx;
}
void radixSort(int arr[], int n, int base) {
        int m = getMax(arr, n);
        for (int exp = 1; m / exp > 0; exp *= base)
        countSort(arr, n, exp, base);
}
```

```
void countSort(int arr[], int n, int exp, int base) {
        int* output = new int[n];
        int* count = new int[base](); // Alocare dinamică pentru count
cu inițializare la 0

        for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % base]++;

        for (int i = 1; i < base; i++)
        count[i] += count[i - 1];

        for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % base] - 1] = arr[i];
        count[(arr[i] / exp) % base]--;
        }

        for (int i = 0; i < n; i++)
        arr[i] = output[i];

        delete[] output; // Eliberăm memoria alocată pentru output
        delete[] count;  // Eliberăm memoria alocată pentru count
}
```

Merge Sort

- **Merge Sort:** Sortează elementele prin împărțirea în subsecvente pe care le sortează și după le interclasează, folosind metoda Divide Et Impera
- **Complexitate :** O(nlogn), unde n este numărul de element
- **Utilizare:** eficient pentru structuri de date mari, datorită complexității
- **Avantaj:** complexitate mică și în cel mai rău caz posibil
- **Dezavantaj:** necesită multă memorie datorită faptului că este un algoritm recursiv

Merge Sort

```
void mergeArrays(int arr[],int st,int mij,int dr)
{
        int aux[10005],n=0,i=st,j=mij+1;
        while(i<=mij && j<=dr)
          if(arr[i]<arr[j]) aux[++n]=arr[i++];
          else aux[++n]=arr[j++];
        while(i<=mij) aux[++n]=arr[i++];
        while(j<=dr) aux[++n]=arr[j++];
        for(i=st;i<=dr;i++) arr[i]=aux[i-st+1];
}
```

```
void mergeSort(int arr[],int st,int dr)
{
        if(st<dr)
        {
        int mij=(st+dr)>>1;
        mergeSort(arr,st,mij);
        mergeSort(arr,mij+1,dr);
        mergeArrays(arr,st,mij,dr);
        }
}
```

Tim Sort

- **Tim Sort:** o combinație între Insertion Sort și Merge Sort
- **Complexitate medie:** O(nlogn), unde n este numărul de element
- **Complexitate în cel mai bun caz:** O(n), când vectorul e deja sortat
- **Complexitate în cel mai rău caz:** O(nlogn)
- **Utilizare:** eficient pentru structuri de date mari
- **Avantaj:** în cel mai bun caz, complexitate liniară

Tim Sort

```
const int min_run=32;
int calcMinRun(int n)
{
        int r=0;
        while(n>=min_run)
        {
        r|=n&1;
        n>>=1;
        }
        return n+r;
}
void insertSort(int arr[], int st, int dr)
{
        int i,j;
        for(i=st+1;i<=dr;i++)
for(j=i;j>st&&arr[j]<arr[j-1];j--)
        swap(arr[j],arr[j-1]);
}
```

```
void mergeArrays(int arr[],int st,int mij,int dr)
{
        int aux[10005],n=0,i=st,j=mij+1;
        while(i<=mij && j<=dr)
          if(arr[i]<arr[j]) aux[++n]=arr[i++];
          else aux[++n]=arr[j++];
        while(i<=mij) aux[++n]=arr[i++];
        while(j<=dr) aux[++n]=arr[j++];
        for(i=st;i<=dr;i++) arr[i]=aux[i-st+1];
}
```

```
void timSort(int arr[],int n)
{
        int i,j,st,dr,run,mij;
        run=calcMinRun(n);
        for(i=1;i<=n;i+=run)

insertSort(arr,i,min(i+run-1,n));
        for(i=1;i<=n;i*=2)
        for(j=1;j<=n;j+=2*i)
        {
        st=j;
        mij=j+i-1;
        dr=min(j+2*i-1,n);
        if(mij<dr)
mergeArrays(arr,st,mij,dr);
        }
}
```

Shell Sort

- **Gap Descrescător**: Sortează elementele la distanțe mari, apoi reduce gap-ul.
- **Eficiență**: Mai rapid decât sortarea prin inserție tradițională datorită schimburilor la distanțe mari.
- **Secvența de Gap**: Performanța variază în funcție de alegerea secvenței de gap.
- **Adaptabil**: Performanța se adaptează în funcție de structura datelor de intrare.
- **Complexitate**: General mai eficient decât O(n^2), dar fără a atinge eficiența O(n log n) a celor mai bune algoritmi de sortare.

Shell Sort

```
void shellSort(int* arr, int n) {
   for (int gap = n/2; gap > 0; gap /= 2) {
      for (int i = gap; i < n; i++) {
         int temp = arr[i];
         int j;
         for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
            arr[j] = arr[j - gap];
         }
         arr[j] = temp;
      }
   }
```

Heap Sort

- **Structură de Date**: Folosește un heap binar pentru a sorta elementele.
- **Complexitate**: Oferă o complexitate timp de O(n log n) pentru cazurile mediu și cel mai rău.
- **In-place**: Sortează array-ul în loc, fără a necesita spațiu suplimentar semnificativ.
- **Eficiență**: Eficient pentru seturi mari de date datorită complexității sale temporale predictibile.

Heap Sort

```
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) {
    swap(arr[i], arr[largest]);
    heapify(arr, n, largest);
  }
```

```
void heapSort(int* arr, int n) {
  for (int i = n / 2 - 1; i >= 0; i--) {
    heapify(arr, n, i);
  }

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

Counting Sort

- Nu este adecvat pentru numere în virgulă mobilă: Algoritmul nu poate sorta eficient numere de acest tip.
- Eficiența scade pentru valori mari ale elementelor: Necesită memorie proporțională cu valoarea maximă din șir, făcându-l nepotrivit pentru seturi de date cu valori foarte mari.
- Utilizare practică limitată: În practică, Counting Sort nu este recomandat pentru valori maxime ale elementelor care depășesc 10^6, din cauza cerințelor crescute de memorie și a eficienței reduse.
- Pentru testele noastre am implementat doua variante cu VAL_MAX = 10^6 și VAL_MAX=N

Counting Sort

```
void countingSort(int arr[], int n) {

        int maxVal = arr[0];
        int minVal = arr[0];
        for (int i = 1; i < n; i++) {
                if (arr[i] > maxVal) maxVal
= arr[i];
                if (arr[i] < minVal) minVal =
arr[i];
        }

        int range = maxVal - minVal + 1;


        int* count = new int[range]();


        for (int i = 0; i < n; i++) {
                count[arr[i] - minVal]++;
        }

minVal;
        int index = 0;
        for (int i = 0; i < range; i++) {
                while (count[i] > 0) {
                        arr[index++] = i +
                        count[i]--;
                }
        }


        delete[] count;
}
```

Grilă rezultate timp

Sortarea	N=10^5 (int)	N=10^6 (int)	N=10^7 (int)	N=10^8 (int)	N=5*10^8 (int)	N=10^7 (float)	N=10^8 (float)	N=5*10^8 (float)
Radix sort(10)	5 ms	59 ms	684 ms	6314 ms	97916 ms	3073 ms	39356 ms	298707 ms
Radix sort (2^16)	2 ms	20 ms	207 ms	2272 ms	65384 ms	740 ms	8247 ms	41517 ms
Counting Sort Max_VAL = 10^7	36 ms	60 ms	332 ms	2051 ms	15827 ms	-	-	-
Merge Sort	26 ms	297 ms	3473 ms	40543 ms	212920 ms	3678 ms	42614 ms	227263 ms
Tim sort	28 ms	325 ms	3627 ms	40895 ms	227259 ms	3849 ms	43176 ms	239822 ms

Grilă rezultate timp

Sortarea	N=10^5 (int)	N=10^6 (int)	N=10^7 (int)	N=10^8 (int)	N=5*10^8 (float)	N=10^7 (float)	N=10^8 (float)	N=5*10^8 (float)
std::sort	7 ms	87 ms	1658 ms	11115 ms	54990 ms	1280 ms	14667 ms	73789 ms
Counting Sort VAL_MAX = N	3 ms	26 ms	308 ms	3554 ms	76507 ms	-	-	-
Shell Sort	29 ms	366 ms	4992 ms	58462 ms	329158 ms	6870 ms	98606 ms	299822 ms
Heap Sort	29 ms	360 ms	4500 ms	57999 ms	325354 ms	5475 ms	87024 ms	258822 ms

Concluzii

- **Radix Sort are performanțe impresionante**, în special atunci când este implementat cu o bază de 2^16. Această variantă se descurcă semnificativ mai bine decât versiunea sa cu baza 10, evidențiind eficiența optimizarilor bazate pe baza de calcul. Acesta este chiar mai rapid decat std::sort
- Contrar așteptărilor **Radix Sort este chiar mai rapid decat Counting sort**, singurul caz in care Countig Sort castiga este în cel *particular* când VAL_MAX << N, însă comparand pur teoretic vedem că dacă VAL_MAX = N, Counting Sort este mai lent.
- **Merge Sort, Tim Sort, Heap Sort, Shell Sort arată o scalabilitate redusă pe măsură ce dimensiunea setului de date crește**, cu timpi de execuție crescând semnificativ pentru seturi de date foarte mari. Aceste metode de sortare devin mai puțin eficiente pentru seturi de date foarte mari, comparativ cu alte algoritmi analizați.
- **std::sort, care este o implementare optimizată** și polivalentă disponibilă în biblioteca standard C++, arată performanțe robuste atât pentru numere întregi, cât și pentru numere în virgulă mobilă, menținând timpi rezonabili de execuție chiar și pentru seturi de date de dimensiuni foarte mari. Aceasta subliniază eficiența și adaptabilitatea algoritmilor bine optimizați din bibliotecile standard
- De precizat ca Shell Sort care are o complexitate teoretica de O(N^2) se descurca foarte bine in practică, având timp puțin mai mari decât cei O(nlog2n)