**Izveštaj – Prva lab vežba**

**Algoritam: Selection Sort**

**Kod:**

public static void SelectionSort(int [] niz)

{

long memoryBefore = GC.GetTotalMemory(true); //get memory usage before function

DateTime start = DateTime.Now;

//prva petlja je za pomeranje elemenata niza jedan po jedan

for (int i = 0; i<niz.Length - 1; i++)

{

int minElement = i; //pomocni index za nalazenje min elementa u nesortiranom delu niza

for(int j = i + 1; j< niz.Length; j++)

{

if (niz[j] < niz[minElement])

{

minElement = j;

}

}

//zamena mesta

int swapHelp = niz[minElement];

niz[minElement] = niz[i];

niz[i] = swapHelp;

}

DateTime end = DateTime.Now;

long memoryAfter = GC.GetTotalMemory(true); // Get memory usage after function

// Total Duration

long memoryUsed = memoryAfter - memoryBefore;

TimeSpan ts = (end - start);

Console.WriteLine("The execution time of the program is {0} ms", ts.TotalMilliseconds);

Console.WriteLine("The memory used for this function is: {0} bytes", memoryUsed.ToString());

}

**Vremenska složenost:**

**Algoritam Selection Sort** ima vremensku složenost **O(n2).** **Najbolji slučaj** je kada je niz koji se prosleđuje već sortiran (ovo i ne pravi neku razliku kao na primer kod Insertion sort-a i Bubble Sort-a, gde su bolje performanse ako je niz već delimično sortiran). **Najgori slučaj** za ovaj algoritam jeste ukoliko je niz koji se prosleđuje i koji treba da se sortira u obrnutom redosledu od željenog (opadajući, a treba da bude u rastućem poretku). Tada je potreban kvadratni broj upoređivanja i zamena. **Srednji slučaj** bi bio ukoliko su elementi prosleđenog niza u rendom redosledu (ne može se zaključiti ni da su u rastućem ni da su u opadajućem redosledu).

**Vremenska složenost Selection Sort-a će uvek biti O(n2) jer će on uvek uporedjivati, tražiti najmanji element i razmenjivati elemente nezavisno od toga da li je niz soritran, nesortiran ili delimično sortiran.**

Vreme sortiranja niza korišćenjem Selection Sort-a (average vrednost od 5 rendom merenja):

|  |  |
| --- | --- |
| **Broj elemenata prosleđenog niza** | **Vreme izvršenja sortiranja** |
| 100 | 12.52154ms ≈ 12.52ms |
| 1000 | 15.5604ms ≈ 15.56ms |
| 10 K | 224.21222ms ≈ 224.21ms |
| 100 K | 11675.01ms ≈ 11,675s |
| 1 M | 53.04min |
| 10 M | 3+ h |

**Prostorna složenost:**

Prostorna složenost ovog algoritma je **O(1)** jer ne zahteva dodatan memorijski prostor osim promenljive korišćene za zamenu elemenata niza (swapElement, minElement).

**Algoritam: Heap Sort**

**Kod:**

public static void HeapSort(int[] niz)

{

long memoryBefore = GC.GetTotalMemory(true); //get memory usage before function

DateTime start = DateTime.Now;

BuildHeap(niz);

int n = niz.Length;

for (int i = n - 1; i >= 1; i--)

{

int pom = niz[0];

niz[0] = niz[i];

niz[i] = pom;

n--;

Heapify(0, n, niz);

}

DateTime end = DateTime.Now;

long memoryAfter = GC.GetTotalMemory(true); // Get memory usage after function

// Vreme izvrsenja:

long memoryUsed = memoryAfter - memoryBefore;

TimeSpan ts = (end - start);

Console.WriteLine("The execution time of the program is {0} ms", ts.TotalMilliseconds);

Console.WriteLine("The memory used for this function is: {0} bytes", memoryUsed.ToString());

}

**Pomoćne funkcije za Heap Sort:**

public static void BuildHeap(int[] niz) //fja za kreiranje Heap-a

{

int heap\_size = niz.Length;

for (int i = heap\_size / 2 - 1; i >= 0; i--)

{

Heapify(i, heap\_size, niz);

}

}

public static void Heapify(int i, int heap\_size, int[] niz) // za stabilisanje stabla nakon dodavanja novog elementa

{

int l = 2 \* i + 1; //levi el

int r = 2 \* i + 2; //desni el

int largest;

if (l < heap\_size && niz[l] > niz[i])

{

largest = l;

}

else

largest = i;

if (r < heap\_size && niz[r] > niz[largest])

{

largest = r;

}

if (largest != i)

{

int pom = niz[i];

niz[i] = niz[largest];

niz[largest] = pom;

Heapify(largest, heap\_size, niz);

}

}

**Vremenska složenost:**

**Algoritam Heap Sort** ima vremensku složenost **O(n ).** Najbolji slučaj za Heap Sort se javlja kada je ulazni niz već sortiran u rastućem redosledu (može biti i u opadajućem redosledu jer Heap Sort nije osetljiv na početni redosled). Vremenska složenost Heap Sort-a i dalje iznosi **O(n ),** gde je n broj elemenata u nizu. Algoritam će uvek morati da izgradi heap (BuildHeap, Heapify), što zahteva O(n) vremena, i zatim da obavi stvarno sortiranje, što zahteva **O(n )** vremena. **Znači da će vremenska složenost biti ista i u najgorem i u prosečnom slučaju (uvek O(n )).**

Vreme sortiranja niza korišćenjem Heap Sort-a (average vrednost od 5 rendom merenja):

|  |  |
| --- | --- |
| **Broj elemenata prosleđenog niza** | **Vreme izvršenja sortiranja** |
| 100 | 12.62244ms ≈ 12.62ms |
| 1000 | 12.02524ms ≈ 12.02ms |
| 10 K | 13.43586ms ≈ 13.43ms |
| 100 K | 43.90712ms ≈ 43.91ms |
| 1 M | 474.2449ms ≈ 474.24ms |
| 10 M | 6157.3061ms ≈ 6.16s |

**Prostorna složenost:**

Prostorna složenost ovog algoritma je **O(1)** jer je u pitanju in-place algoritam za sortiranje (ne koristi nijednu dodatnu strukturu već se prostor zauzima samo za niz koji se sortira i za pomoćne varijable).

**Algoritam: Bucket Sort**

**Kod:**

public static void BucketSort(int[] array)

{

long memoryBefore = GC.GetTotalMemory(true); //get memory usage before function

DateTime start = DateTime.Now;

if (array == null || array.Length <= 1)

{

return; //nema smisla sortirati ako nema elemenata ili ukoliko postoji samo 1 element u nizu

}

int max = array[0]; //za trazenje najveceg elementa niza

int min = array[0]; //za trazenje najmanjeg elementa niza

for (int i = 1; i < array.Length; i++)

{

if (array[i] > max)

{

max = array[i];

}

if (array[i] < min)

{

min = array[i];

}

}

int brojBucketa = max - min + 1;

//kreiranje liste bucket-a

List<int>[] buckets = new List<int>[brojBucketa];

for (int i = 0; i < brojBucketa; i++)

{

buckets[i] = new List<int>();

}

//dodavanje elementa niza u odgovarajući Bucket

for (int i = 0; i < array.Length; i++)

{

int chosenBucket = (array[i] - min);

buckets[chosenBucket].Add(array[i]);

}

//vracanje elemenata u niz

int k = 0;

for (int i = 0; i < buckets.Length; i++)

{

for (int j = 0; j < buckets[i].Count; j++)

{

array[k] = buckets[i][j];

k++;

}

}

DateTime end = DateTime.Now;

long memoryAfter = GC.GetTotalMemory(true); // Get memory usage after function

// Vreme izvrsenja:

long memoryUsed = memoryAfter - memoryBefore;

TimeSpan ts = (end - start);

Console.WriteLine("The execution time of the program is {0} ms", ts.TotalMilliseconds);

Console.WriteLine("The memory used for this function is: {0} bytes", memoryUsed.ToString());

}

**Vremenska složenost:**

**Algoritam Bucket Sort** ima prosečnu vremensku složenost **O(n + k)**. Najbolji slučaj za Bucket Sort se javlja kada su elementi niza ravnomerno raspoređeni po bucket-ima. U ovom slučaju, svaki bucket sadrži približno isti broj elemenata, i raspodela elemenata je već blizu sortirane unutar svakog bucket-a (**O(n+k)**). Najgori slučaj se javlja kada su svi elementi smešteni u jedan jedini bucket. Onda se algoritam svodi na sortiranje jednog velikog niza, što može biti neefikasno (**O(n2)**).

Vreme sortiranja niza korišćenjem Bucket Sort-a (average vrednost od 5 rendom merenja):

|  |  |
| --- | --- |
| **Broj elemenata prosleđenog niza** | **Vreme izvršenja sortiranja** |
| 100 | 8.2475ms ≈ 8.25ms |
| 1000 | 13.1396ms ≈ 13.14ms |
| 10 K | 15.662ms ≈ 15.66ms |
| 100 K | 17.8427ms ≈ 17.84ms |
| 1 M | 39.194ms ≈ 39.2ms |
| 10 M | 287.6795ms ≈ 287.68ms |

**Prostorna složenost:**

Prostorna složenost je **O(n+k)** gde je n broj elemenata niza, a k je broj bucket-a koji se kreira.

Vremenska složenost poredjenje algoritama:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Broj elemenata prosleđenog niza** | **Selection**  **Sort** | **Heap**  **Sort** | **Bucket**  **Sort** |  |
| 100 | 12.52ms | 12.62ms | 8.25ms |
| 1000 | 15.56ms | 12.02ms | 13.14ms |
| 10 K | 224.21ms | 13.43ms | 15.66ms |
| 100 K | 11,675s | 43.91ms | 17.84ms |
| 1 M | 53.04min | 474.24ms | 39.2ms |
| 10 M | 3+ h | 6.16s | 287.68ms |

**Zaključak:** Selection Sort se nije pokazao kao efikasan što se tiče sortiranja niza koji ima veliki broj elemenata (vreme izvršenja linearno raste sa porastom broja elemenata niza). Najbolje vremenske performanse je pokazao Bucket Sort (i Heap i Bucket Sort brže sortiraju nizove sa velikim brojem elemenata).

Prostorna složenost poređenje algoritama (utrošak memorije):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Broj elemenata prosleđenog niza** | **Selection**  **Sort** | **Heap**  **Sort** | **Bucket**  **Sort** |  |
| 100 | 11.956 KB | 11.956 KB | 293.408 KB |
| 1000 | 11.956 KB | 11.956 KB | 318.832 KB |
| 10 K | 11.956 KB | 11.956 KB | 469.316 KB |
| 100 K | 11.956 KB | 11.956 KB | 962.944 KB |
| 1 M | 11.956 KB | 11.956 KB | 5552.176 KB |
| 10 M | 11.956 KB | 11.956 KB | 50424.124 KB |

**Zaključak:** Kako su i Selection Sort i Heap Sort in-place algoritmi za sortiranje ne zahtevaju dodatnu memoriju osim za neke dodatne promenljive (uvek konstantna dodatna veličina memorije bez obzira na broj elemenata niza). Bucket Sort, sa druge strane, koristi dodatnu memoriju za kreiranje bucket-a i ona se povećava sa povećanjem elemenata niza.

Andjela Mia Dončov 18627