Lekcja D1. (pp)

Temat: Sortowanie metodą scalania i sortowanie kubełkowe.

Cele lekcji:

Poznanie sortowania poprzez scalanie.
Poznanie algorytmu rekurencyjnego.
Poznanie algorytmu sortowania kubełkowego.
Poznanie metody scalania z wykorzystaniem algorytmu sortowania kubełkowego.

Uczeń:

– wie, co to są algorytmy proste
– umie tworzyć i stosować algorytmy proste wie, co to jest sortowanie metodą scalania
– umie tworzyć algorytmy wykorzystujące sortowanie metodą scalania
– umie stosować algorytmy wykorzystujące sortowanie metodą scalania
– wie, co to jest rekurencja
– umie tworzyć i stosować algorytmy wykorzystujące sortowanie rekurencyjne
– wie, co to jest scalanie zbiorów uporządkowanych
– umie stworzyć algorytm scalania dwóch zbiorów uporządkowanych

– wie, czym jest sortowanie kubełkowe
– zna cechy algorytmu sortowania
– umie opracować algorytm sortowania kubełkowego
– umie opracować algorytm sortowania metodą scalania

Przebieg lekcji:

1. Zapoznanie z celami lekcji.
2. Sortowanie metodą scalania – definicja.
3. Scalanie zbiorów uporządkowanych.
4. Algorytm – lista kroków.
5. Tworzenie programu w C++ realizujący algorytm sortowania metodą scalania.
6. Sortowanie kubełkowe – definicja.
7. Sortowanie kubełkowe - opis.
8. Algorytm – lista kroków.
9. Tworzenie programu w C++ realizujący algorytm sortowania metodą scalania.

Film:

Sortowanie przez scalanie

https://www.youtube.com/watch?v=iJyUFvvdfUg

Sortowanie kubełkowe

https://www.youtube.com/watch?v=TS6IiMOfxFE

https://binarnie.pl/sortowanie-kubelkowe/

$O(n \log_2 n)$ – złożoność algorytmu

0(n) – złożoność pamięciowa

Opis:

Sortowanie przez scalanie

Sortowanie przez scalanie należy do algorytmów, które wykorzystują metodę "**dziel i zwyciężaj**". Złożoność algorytmu wynosi

$$n \cdot log\, n$$

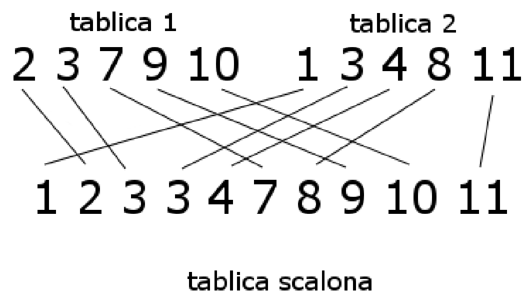
, a więc jest on znacznie wydajniejszy niż sortowanie bąbelkowe, przez wstawianie czy przez selekcję, gdzie złożoność jest kwadratowa. Żeby zrozumieć zasadę działania przyjrzyjmy się najpierw dwóm posortowanym tablicom:

tablica 1: **2 3 7 9 10**

tablica 2: **1 3 4 8 11**

Zauważmy, że możemy **liniowo** scalić te dwa ciągi liczb i uzyskać jedną posortowaną tablicę postępując ze schematem:

- ustawiamy liczniki na początki tablic posortowanych,
- następnie porównujemy elementy i mniejszy lub równy element wskakuje jako pierwszy w scalonej tablicy,
- zwiększamy licznik w tej tablicy, z której "zabraliśmy element",
- czynność powtarzamy aż do wyczerpania danych z obu tablic.



Zalety algorytmu

- prostota implementacji
- wydajność
- stabilność
- algorytm sortuje zbiór **n-elementowy** w czasie proporcjonalnym do liczby

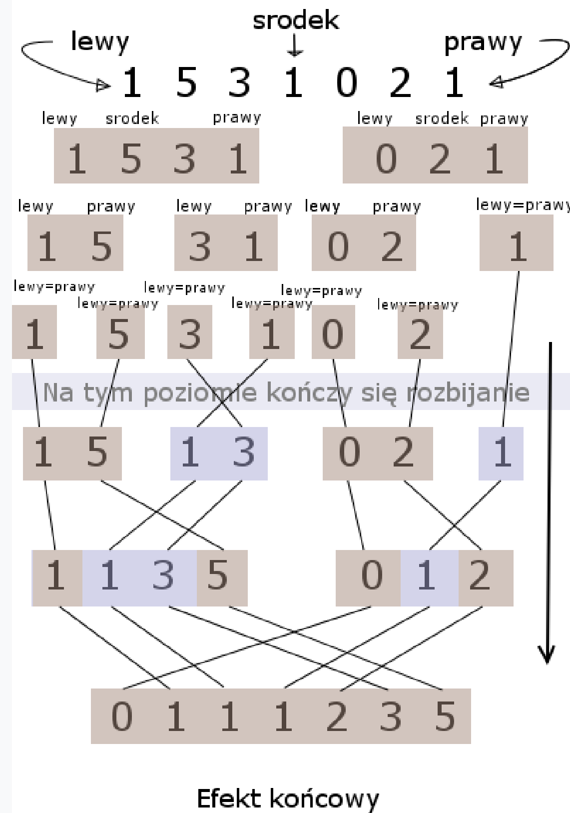
$$nlogn$$

bez względu na rodzaj danych wejściowych

Wady algorytmu

- podczas scalania potrzebny jest dodatkowy obszar pamięci przechowujący kopie podtablic do scalenia

Schemat:




```
#include<iostream>
using namespace std;

int *pom; //tablica pomocnicza, potrzebna przy scalaniu

//scalenie posortowanych podtablic
void scal(int tab[], int lewy, int srodek, int prawy)
{
int i = lewy, j = srodek + 1;

  //kopiujemy lewą i prawą część tablicy do tablicy pomocniczej
  for(int i = lewy;i<=prawy; i++)
    pom[i] = tab[i];

  //scalenie dwóch podtablic pomocniczych i zapisanie ich
  //we własciwej tablicy
  for(int k=lewy;k<=prawy;k++)
  if(i<=srodek)
    if(j <= prawy)
         if(pom[j]<pom[i])
             tab[k] = pom[j++];
         else
             tab[k] = pom[i++];
    else
        tab[k] = pom[i++];
  else
      tab[k] = pom[j++];
}

void sortowanie_przez_scalanie(int tab[],int lewy, int prawy)
{
//gdy mamy jeden element, to jest on już posortowany
if(prawy<=lewy) return;

//znajdujemy srodek podtablicy
int srodek = (prawy+lewy)/2;
```

```
//dzielimy tablice na część lewą i prawa
sortowanie_przez_scalanie(tab, lewy, srodek);
sortowanie_przez_scalanie(tab, srodek+1, prawy);

//scalamy dwie już posortowane tablice
scal(tab, lewy, srodek, prawy);
}

int main()
{
int *tab,
n; //liczba elementów tablicy
cout << "Podaj ilośc elementów tablicy: ";
cin>>n;
tab = new int[n]; //przydzielenie pamięci na tablicę liczb
pom = new int[n]; //przydzielenie pamięci na tablicę pomocniczą

//wczytanie elementów tablicy
for(int i=0;i<n;i++)
{
	cout <<"podaj element ["<< i+1 <<"]= ";
	cin>>tab[i];
}
//sortowanie wczytanej tablicy
sortowanie_przez_scalanie(tab,0,n-1);

//wypisanie wyników
for(int i=0;i<n;i++)
	cout<<tab[i]<<" ";
return 0;
}
/* wersja ulepszona
//scalenie posortowanych podtablic
void scal(int tab[], int lewy, int srodek, int prawy)
{
  int i, j;
  //zapisujemy lewą część podtablicy w tablicy pomocniczej
  for(i = srodek + 1; i>lewy; i--)
    pom[i-1] = tab[i-1];
  //zapisujemy prawą część podtablicy w tablicy pomocniczej
  for(j = srodek; j<prawy; j++)
    pom[prawy+srodek-j] = tab[j+1];
  //scalenie dwóch podtablic pomocniczych i zapisanie ich
  //we własciwej tablicy
  for(int k=lewy;k<=prawy;k++)
    if(pom[j]<pom[i])
      tab[k] = pom[j--];
    else
      tab[k] = pom[i++];
}
*/
```

Sortowanie kubełkowe

powró

Sortowanie kubełkowe należy do algorytmów, które porządkują dane w czasie liniowym. Dane, które będziemy sortować muszą jednak spełniać pewne założenia - **muszą być równo rozłożone** i **musimy znać ich zakres**. Nie może być takiej sytuacji, że jeśli chcemy posortować, powiedzmy milion liczb całkowitych, należących do przedziału [0, 1000 000 000], gdzie 900 tysięcy jest nie większych niż np. 10 milionów.

Zalety algorytmu:

- złożoność czasowa jest rzędu

$$O(n)$$

- algorytm nie potrzebuje dodatkowej tablicy (sortowanie odbywa się w miejscu)
- jest łatwy w implementacji

Algorytm posiada także wady:

- dane muszą być równomiernie rozłożone, aby złożoność była liniowa
- trzeba znać dokładny rozstęp zbioru (różnicę między największą i najmniejszą wartością do posortowania)

Działanie algorytmu

Na początku założenia:

- liczby do posortowania znajdują się w przedziale [0, 2000 000 000)
- liczby są równomiernie rozłożone
- dla **n** liczb tworzymy **n** kubełków

W pierwszej kolejności wyznaczamy długość przedziału pojedynczego kubełka:

$$p = 2000000000/n$$

Dla ułatwienia, sortować będziemy 10 liczb, zatem

$$p = 2000000000/10 = 200000000$$

Poszczególne kubełki będą reprezentować następujące przedziały:

1. [0, 200 000 000)
2. [200 000 000, 400 000 000)
3. [400 000 000, 600 000 000)
4. [600 000 000, 800 000 000)
5. [800 000 000, 1000 000 000)
6. [1000 000 000, 1200 000 000)
7. [1200 000 000, 1400 000 000)
8. [1400 000 000, 1600 000 000)
9. [1600 000 000, 1800 000 000)
10. [1800 000 000, 2000 000 000)

Sortować będziemy zbiór dziesięciu liczb całkowitych:

300 000 000, 2399, 900 000 000, 1899 999 999, 500 000 001, 1400 200 000, 1799 999 999, 123 456 678, 1999 999 999, 1210 000 000

Każdą z tych liczb wkładamy do odpowiedniego kubełka:

1. [0, 200 000 000): **2399, 123 456 678,**
2. [200 000 000, 400 000 000): **300 000 000,**
3. [400 000 000, 600 000 000): **500 000 001,**
4. [600 000 000, 800 000 000)
5. [800 000 000, 1000 000 000): **900 000 000**
6. [1000 000 000, 1200 000 000)
7. [1200 000 000, 1400 000 000): **1210 000 000**
8. [1400 000 000, 1600 000 000): **1400 200 000,**
9. [1600 000 000, 1800 000 000): **1799 999 999,**
10. [1800 000 000, 2000 000 000): **1899 999 999, 1999 999 999**

Następnie przeglądamy każdy kubełek i jeśli jest w nim więcej niż jeden element, wykonujemy sortowanie dowolnym algorytmem. Zauważmy, że liczby są równomiernie rozłożone, co za tym idzie, każdy kubełek będzie miał co najwyżej kilka elementów. Do posortowania takiego kubełka

```
//program do sortowania n-elementowej tablicy
//przy pomocy algorytmu sortowania kubełkowego
//dla liczb całkowitych dodatnich
#include <iostream>
using namespace std;
void printTab(int *tab, int n);
int*  findMinAndMax(int* tab, int n);

void bucketSort(int *tab, int n, int yMin, int yMax) {

	int* buckets = new int[yMax - yMin + 1];

	//Wstawiamy początkowe wartości liczników
	for (int x = 0; x < (yMax - yMin + 1); x++)
	{
		buckets[x] = 0;
	}

	cout << endl << "Stworzono kubelki: " << endl;
	cout << "\t| ";
	for (int x = 0; x < (yMax - yMin + 1); x++) {
		cout << "B" << x+yMin << " | ";
	}
	cout << endl;

	for (int x = 0; x < n; x++)
```

```
	{
		//Zliczamy ilość wystąpień poszczególnych elementów
		buckets[tab[x] - yMin]++;
	}

	cout << endl << "Zliczono elementy: " << endl;
	cout << "\t| ";
	for (int x = 0; x < (yMax - yMin + 1); x++) {
		cout << "B" << x + yMin << "=" << buckets[x] <<" | ";
	}
	cout << endl;

	//Zapisujemy elementy w ilości podyktowanej przez kubełki
	//w tablicy
	int lastIndex = 0;
	for (int x = 0; x < (yMax - yMin + 1); x++)
	{
		int y;
		for (y = lastIndex; y < buckets[x] + lastIndex; y++)
		{
			tab[y] = x + yMin;
		}
		lastIndex = y;
	}
}

int main() {
	int n;

	cout << "Wprowadz liczbe elementow tablicy: ";
	cin >> n;

	//Dynamiczne tworzenie tablicy
	int *tab = new int[n];

	cout << "\nWprowadz " << n << " liczb do posortowania" << endl;
	cout << "Zatwierdz kazda z nich klawiszem Enter:" << endl;

	for (int x = 0; x < n; x++) {
		cin >> tab[x];
	}

	cout << endl << "Tablica przed posortowaniem:" << endl;
	printTab(tab, n);

	int* minAndMax = findMinAndMax(tab, n);
	cout << endl << "Minimalny element tablicy: " << minAndMax[0] << endl;
	cout << "Maksymalny element tablicy: " << minAndMax[1] << endl;
	cout << endl << "Rozpoczecie sortowania" << endl;
	bucketSort(tab, n, minAndMax[0], minAndMax[1]);

	cout << endl << "Oto tablica po sortowaniu:" << endl;
```

```
	printTab(tab, n);

	delete[] tab;
	delete[] minAndMax;
	system("pause");
	return 0;

}

void printTab(int *tab, int n) {
	cout << endl << "\t| ";
	for (int x = 0; x < n; x++) {
		cout << tab[x] << " | ";
	}
	cout << endl << endl;
}

//Funkcja do wyszukiwania największej i
//najmniejszej wartości w tablicy
int*  findMinAndMax(int* tab, int n)
{
	int* minAndMax = new int[2];
	minAndMax[0] = tab[0];
	minAndMax[1] = tab[0];

	for (int x = 0; x < n; x++)
	{
		if (tab[x] < minAndMax[0])
		{
			minAndMax[0] = tab[x];
		}
		if (tab[x] > minAndMax[1])
		{
			minAndMax[1] = tab[x];
		}
	}

	return minAndMax;
}
```