2023

Algoritmy třízení

SEMINÁRNÍ PRÁCE

ADAM PEČENKA 3.A

I. Obsah

Obsah

II. Pomocné metody

Jako pomocné metody jsem používal FillList a FillListRandom. Tyto metody byly téměř kompletně identické. Jako parametry přijímaly List a délku, ve zbylých parametrech se lišily. FillList dále přijímala pouze hodnotu, FillListRandom místo hodnoty přijímala horní a dolní limit pro náhodná čísla. Jak název napovídá obě metody naplnily hodnotami List.

```
1.  //Min - Spodní limit
2.  //Max - Horní limit
3.  //Length - Požadovaná délka kolekce
4.  public static void FillListRandom(this List<int> list, int min, int max, int
    length)
5.  {
6.      Random rnd = new();
7.      for (int i = 0; i < length; i++)
8.      {
9.          //Přidá do kolekce novou náhodně vygenerovanou hodnotu
10.         list.Add(rnd.Next(min, max));
11.     }
12. }
13. //Value - Hodnota, kterou chceme kolekci naplnit
14. //Length - Požadovaná délka kolekce
15. public static void FillList(this List<int> list, int value, int length)
16. {
17.     for (int i = 0; i < length; i++)
18.     {
19.         //Přidá do kolekce novou hodnotu
20.         list.Add(value);
21.     }
22. }
```

III. Selection Sort

A. Popis

Selection Sort je jednoduchý třídící algoritmus, který postupně prochází nesetříděnou část Kolekce a umisťuje nejnižší prvky co najde do setříděné části. Většinou se implementuje, způsobem, že začíná od 1. (index 0) prvku a snaží se najít co nejnižší číslo, jakmile najde nižší číslo než číslo na 1. pozici prohodí je. Poté algoritmus začne znovu od 2. (index 1) prvku a tak dále dokud neprojde celé pole. Jeho výhodou je nízká náročnost na paměť. Nevýhodou je časová náročnost.

B. Časová náročnost

Časová náročnost algoritmu, je $O(n^2)$, protože se v algoritmu nachází smyčka uvnitř smyčky, s tím, že obě prochází celé pole.

C. Implementace

```
1.  public static void SelectionSort(List<int> numbers)
2.          {
3.              int lowestNumber;
4.              for (int i = 0; i < numbers.Count; i++)
5.              {
6.                  //Považuji první prvek za nejnižšší
7.                  lowestNumber = numbers[i];
8.                  for (int j = i + 1; j < numbers.Count; j++)
9.                  {
10.                     //Procházím celou kolekci od prvku, který následuje po
    našem vybraném prvku
11.                     if (numbers[j] < lowestNumber)
12.                     {
13.                         //Pokud najdu nový nejnižší prvek prohodím je
    a uložím si nové nejnižší číslo
14.                         lowestNumber = numbers[j];
15.                         (numbers[i], numbers[j]) = (numbers[j], numbers[i]);

16.                     }
17.                 }
18.             }
19.         }
```

IV. Bubble Sort

A. Popis

Bubble Sort je, troufám si říct, nejprostší algoritmus v této seminární práci. Funguje na prostém principu porovnávání sousedících prvků. Pokud je prvek i + 1 větší než prvek i prohodí se. Jeho výhoda je jednoduchost a prostost. V dnešní době je tedy dobrý například jako algoritmus pro začátečnické programátory. Na praktické využití není příliš dobrý kvůli své časové náročnosti. V této seminární práci je brána v potaz upravená verze Bubble sortu, která se ukončí předčasně, pokud je Kolekce již setříděná, ve své původní podobě se vždy provede Bubble sort celý.

B. Časová náročnost

Časová náročnost Bubble Sortu je, stejně jako u Selection Sortu $O(n^2)$, i ze stejného důvodu jako u Selection Sort; dvě vnořené smyčky, které prochází celou Kolekci.

C. Implementace

```
1.  public static void BubbleSort(List<int> numbers)
2.          {
3.              //Projdu Xkrát kolekci; X je počet poležek - 1, zamezí se tím
    šahání na neexistující položky, které by byly mimo rozsah
4.              for (int i = 0; i < numbers.Count - 1; i++)
5.              {
6.                  //Založím boolean hasSorted (třídil), tato proměnná bude
    hlídat, jestli se prohodila čísla a je použita později pro předčasné ukončení
     algoritmu
7.                  bool hasSorted = false;
8.                  //Projdu Xkrát kolekci; X je počet poležek - 1, opět se tím
    zamezí šahání mimo rozsah
9.                  for (int j = 0; j < numbers.Count - 1; j++)
10.                 {
11.                     //Pokud je aktuální položka na indexu J větší, než
    následující vzájemně se prohodí a hasSorted se nastaví na true
12.                     if (numbers[j] > numbers[j + 1])
13.                     {
14.                         (numbers[j + 1], numbers[j]) = (numbers[j], numbers[j
     + 1]);
15.                         hasSorted = true;
16.                     }
17.                 }
18.                 //Vypíše aktuální podobu kolekce ve formátu
    [Počítadlo průchodů]. [{Položka} {Položka} {Položka}...]
19.                 Console.WriteLine($"{i}. {String.Join(" ", numbers)}\n");
20.                 //Pokud se neprohodily čísla [hasSorted == false] ukončí před
    časně algoritmus
21.                 if (!hasSorted) return;
22.             }
23.         }
```

V. Insertion Sort

A. Popis

Insertion Sort je velmi užitečný pro téměř setříděná či malá pole. Je jednoduchý na implementování. Funguje na pricipu rozdělení Kolekce na setříděnou a nesetříděnou část. Jeho nevýhodou je opět časová náročnost, ta je, ale problém pouze s velkými kolekcemi.

B. Časová náročnost

Časová náročnost Insertion Sortu, je $O(n^2)$.

C. Implementace

```
1.  public static void InsertionSort(List<int> numbers)
2.          {
3.              //Projdu Xkrát kolekci; X je počet položek v kolekci
4.              //První položku automaticky bereme jako seřazenou a tak začíná
    druhou položkou (index 1)
5.              for (int i = 1; i < numbers.Count; i++)
6.              {
7.                  //Uložíme si index do proměnné hole a hodnotu v kolekci na
    pozici indexu
8.                  int value = numbers[i];
9.                  int hole = i;
10.                 //Provádíme smyčku dokud je index větší než 0 a zároveň je
    hodnota na pozici [index - 1] větší než hodnota na pozici index
11.                 while (hole > 0 && numbers[hole - 1] > value)
12.                 {
13.                     //Hodnota na pozici [index - 1] se přesune na index
14.                     numbers[hole] = numbers[hole - 1];
15.                     //Snížíme index o -1
16.                     hole--;
17.                 }
18.                 //Uložíme hodnotu, kterou jsme si uložili na začátku průchodu
     do kolekce na index
19.                 numbers[hole] = value;
20.                 //Vypíše průběžný stav kolekce
21.                 Console.WriteLine($"{String.Join(" ", numbers)}\n");
22.             }
23.         }
```

VI. Heap Sort

A. Co je to halda a její vlastnosti

Halda je datová struktura, je úplný binární strom. Halda se postupně větví na Otce a Syny. Každý Otec může mít maximálně 2 syny. Jelikož je Halda úplný binární strom mají všichni Otcové kromě těch v poslední vrstvě vždy 2 syny.

B. Popis

Celý Heap sort pracuje na principech Hald, konkrétně Max/Min heap. Max heap znamená, že otec musí mít vždy vyšší hodnotu než jeho synové. Min znamená, že musí mít otec menší hodnotu než jeho synové. Pokud je heap platný, to znamená, že splňuje předchozí podmínku, je Kolekce seřazená. Výhodou je bezpochyby prakticky konstantní časová náročnost, nevýhodou je, že není příliš vhodný pro menší Kolekce.

C. Časová náročnost

Časová náročnost Heap Sortu je v průměrném i v nejhorším případě O(n*log n)

D. Implementace

```
1.  public static void HeapSort(List<int> numbers)
2.          {
3.
4.              // Sestavíme heap
5.              for (int i = numbers.Count / 2 - 1; i >= 0; i--)
6.              {
7.                  CreateHeap(numbers, numbers.Count, i);
8.              }
9.              // Vytaháme prvek po prvku z Kolekce
10.             for (int i = numbers.Count - 1; i > 0; i--)
11.             {
12.                 // Přesuneme první prvek na konec
13.                 (numbers[0], numbers[i]) = (numbers[i], numbers[0]);
14.
15.                 // Znovu sestavíme heap
16.                 CreateHeap(numbers, i, 0);
17.             }
18.         }
19.         static void CreateHeap(List<int> numbers, int size, int i)
20.         {
21.             int father = i; // otec
22.             int leftSon = 2 * i + 1; // levý syn
23.             int rightSon = 2 * i + 2; // pravý syn
24.
25.             // Pokud jsou synové větší než otec
26.             if (leftSon < size && numbers[leftSon] > numbers[father])
27.                 father = leftSon;
28.
29.             if (rightSon < size && numbers[rightSon] > numbers[father])
30.                 father = rightSon;
31.
32.             //Pokud není otec první prvek přesunu otce na první prvek
33.             if (father != i)
34.             {
35.                 (numbers[i], numbers[father]) = (numbers[father], numbers[i]);
36.
37.                 // Znovu sestavíme heap
38.                 CreateHeap(numbers, size, father);
39.             }
40.         }
```

VII. Merge Sort

A. Princip slévání

Slévání funguje na principu porovnávání hodnot ve 2 Kolekcích. Začneme s indexy 0 na levém i pravém indexu. Porovnáme hodnoty, pokud je hodnota v levé kolekci na pozici levého indexu vyšší přidáme hodnotu z levé kolekce a zvýšíme levý index o 1. Pokud je hodnota v pravé kolekci vyšší provede to stejné pro pravý index. Na kterou pozici máme umístit do konečné kolekce hodnotu zjistíme sečtením pravého i levého indexu. Například pokud jsme z levé kolekce vložili 4 hodnoty a z levé 3 tak musíme novou hodnotu uložit na index 7.

B. Popis

Merge sort funguje na takzvaném principu „Divide and conquer“ („Rozděl a panuj“). Kolekce se rozděluje tak dlouho dokud není rozdělená na jednotlivé hodnoty, které se poté slévají zpět dohromady. Výhodou je časová náročnost, nevýhodou je to, že Merge sort vyžaduje dodatečné pole, toto, ale s dnešní RAM pamětí počítačů není ve valné většině případů problém.

C. Časová náročnost

Časová náročnost Merge sortu je O(n * log n).

D. Implementace

```
1.  public static void MergeSort(List<int> numbers)
2.          {
3.              //Zastavení rekurze
4.              if (numbers.Count < 2) return;
5.              //Uložím si index prostředního čísla
6.              int middle = numbers.Count / 2;
7.              //Založím listy do kterých sleju levou a pravou část původní kole
    kce
8.              List<int> leftList = new List<int>();
9.
10.             for (int i = 0; i < middle; i++)
11.             {
12.                 leftList.Add(numbers[i]);
13.             }
14.
15.             List<int> rightList = new List<int>();
16.
17.             for (int i = middle; i < numbers.Count; i++)
18.             {
19.                 rightList.Add(numbers[i]);
20.             }
21.             //Zavolám MergeSort na oddělenou levou a pravou část původní kole
    kce
22.             MergeSort(leftList);
23.             MergeSort(rightList);
24.             //Sloučím původní kolekci, respektive přepíšu, levou a pravou čás
    tí
25.             Merge(numbers,leftList,rightList);
26.         }
27.         static void Merge(List<int> numbers, List<int> leftList, List<int> ri
    ghtList)
28.         {
29.             //Indexování
30.             int leftIndex = 0;
31.             int rightIndex = 0;
```

```
32.             //Dokud nepřetečou indexy mimo rozměry kolekcí
33.             while (leftIndex < leftList.Count && rightIndex < rightList.Count
    )
34.             {
35.                 //Porovnám hodnoty na indexech v obou kolekcích, pokud je hod
    nota v pravé kolekci větší nebo rovna, uloží se hodnota z levé kolekce, proto
    že je menší a naopak
36.                 if (rightList[rightIndex] >= leftList[leftIndex])
37.                 {
38.                     numbers[leftIndex + rightIndex] = leftList[leftIndex];
39.                     leftIndex++;
40.                 }
41.                 else
42.                 {
43.                     numbers[leftIndex + rightIndex] = rightList[rightIndex];

44.                     rightIndex++;
45.                 }
46.
47.             }
48.             //Protože rozdíl mezi pravou a levou kolekcí by měl být maximálně
     jeden prvek měl by stačit zápis IF ELSE, ale pro klid duše jsem se rozhodl
    pro zápis 2 IFy
49.             if (leftIndex < leftList.Count)
50.             {
51.                 //Slévání nedotřízeného čísla
52.                 while (leftIndex < leftList.Count)
53.                 {
54.                     numbers[leftIndex + rightIndex] = leftList[leftIndex];
55.                     leftIndex++;
56.                 }
57.             }
58.             if(rightIndex < rightList.Count)
59.             {
60.                 //Slévání nedotřízeného čísla
61.                 while (rightIndex < rightList.Count)
62.                 {
63.                     numbers[leftIndex + rightIndex] = rightList[rightIndex];

64.                     rightIndex++;
65.                 }
66.             }
67.         }
```

VIII. Quick Sort

A. Popis

Quick sort, stejně jako Merge sort, je postavený na principu „Divide and conquer“. Na rozdělení pole používá takzvaný pivot. Před pivot umístíme prvky menší než pivot a za něj prvky větší než pivot. Rekurzivně poté rozdělujeme poloviny na menší a menší části, poté spojíme část s prvky menšími prvky, pivot a většími prvky a takto dál pospojujeme všechny části, dokud nejsme u plně seřazeného pole. Dobré možnosti na umístění pivotu je konec Kolekce nebo například medián prvního, prostředního a posledního prvku nebo náhodné umístění pivotu.

B. Časová náročnost

Časová náročnost záleží na umístění pivotu, špatný pivot může vyústit v časovou náročnost $O(n^2)$, průměrná časová náročnost, je O(n* log n). Špatné umístění pivotu je veliká nevýhoda.

C. Implementace

```
1.  static void QuickSort(List<int> numbers, int leftBound, int rightBound)
2.          {
3.              //Ošetření neintuitivních limitů a ukončení rekurze
4.              if (rightBound <= leftBound || rightBound > numbers.Count - 1)
5.                  return;
6.              //Najdeme optimální umístění pivotu
7.              int pivot = QuickSortPartition(numbers, leftBound, rightBound);
8.
9.              QuickSort(numbers, leftBound, pivot - 1);
10.             QuickSort(numbers, pivot + 1, rightBound);
11.         }
12.         static int QuickSortPartition(List<int> numbers, int leftBound, int r
    ightBound)
13.         {
14.
15.             // Pivot se umístí na konec, optimálnější umístění by bylo buďto
    náhodné nebo medián první, poslední a prostředí hodnoty je také dobrá možnost

16.             int pivot = numbers[rightBound];
17.
18.             //Uložím si index, jako index nám poslouží spodní hranice - 1
19.             int index = (leftBound - 1);
20.
21.             for (int i = leftBound; i <= rightBound - 1; i++)
22.             {
23.                 if (numbers[i] < pivot)
24.                 {
25.
26.                     //Pokud je aktuální položka menší než pivot prohodím
    aktuální položku a položku na indexu
27.                     index++;
28.                     (numbers[index], numbers[i]) = (numbers[i], numbers[index
    ]);
29.                 }
30.             }
31.             //Prohodím položku za indexem a položku na pravé horní hranici
32.             (numbers[index + 1], numbers[rightBound]) = (numbers[rightBound],
     numbers[index + 1]);
33.
34.             return (index + 1);
35.         }
```

IX. Counting Sort

A. Popis

Counting sort funguje na principu sčítání výskytu prvků. Je vhodný pro řazení prvků s malým rozsahem. Existují „2 verze“ tohoto algoritmu, jedna s kumulativním součtem a druhá bez. Verze bez kumulativního součtu je použitelná pouze na řazení čísel. Vzhledem k tomu, že pracuje pouze s čísly rozhodl jsem se právě pro verzi bez kumulativního součtu. Nevýhodou Counting sortu je jeho náročnost na paměť, pokud je veliký rozsah mezi hodnotami.
V Counting sortu si jako první najdeme nejmenší a nejvyšší hodnoty, z těch poté dostaneme rozsah hodnot. Poté projdeme celé pole a do „počítacího pole“, které má délku maximální hodnota – minimální hodnota + 1, přičteme jedničku na index [hodnota – minimální hodnota], například pokud je minimální hodnota 13 a nalezneme hodnotu 23 přičteme na index 10 1. Pokud nalezneme 13 a minimální hodnota je stále 13 přičteme na index 0 1. Poté projdeme jednotlivé „součty“ v počítacím poli, na každé hodnotě v počítacím poli, která je vyšší než 0 přidáme do návratového pole hodnotu index + minimální hodnota, tím se dostaneme na původní hodnotu.

B. Časová náročnost

Časová náročnost není na rozdíl od všech předchozích algoritmů závislá pouze na „n“, čili počtu prvků, ale i „k“, maximální hodnotě v kolekci. Jeho časová náročnost je O(n + k).

C. Implementace

```
1.  public static void CountingSort(List<int> numbers)
2.          {
3.              int min = int.MaxValue;
4.              int max = int.MinValue;
5.              //Založím si nový list a naplním ho hodnotami, kterými poté
    přepíšu původní list, tento přístup ulehčuje psaní kódu
6.              List<int> returnList = new List<int>();
7.              returnList.FillList(0, numbers.Count);
8.              //Najdu nejvyšší a nejnižší hodnotu
9.              foreach (var number in numbers)
10.             {
11.                 if (min > number)
12.                     min = number;
13.                 if (max < number)
14.                     max = number;
15.             }
16.             //Založím si a naplním počítací kolekci
17.             List<int> countingList = new List<int>();
18.
19.             countingList.FillList(0, max - min + 1);
20.             for (int i = 0; i < numbers.Count; i++)
21.             {
22.                 //Zvýším hodnotu na indexu [Aktuální hodnota v původní
    kolekci na indexu i - minimální hodnota] o 1
23.                 countingList[numbers[i] - min]++;
24.             }
25.             int j = 0;
26.
27.             for (int i = 0; i < countingList.Count; i++)
28.             {
29.                 //Provedu pro celou délku počítací kolekce
30.                 //Uložím aktuální hodnotu z počítací kolekce
31.                 int value = countingList[i];
```

```
32.                 while (value > 0)
33.                 {
34.                     //Dokud je hodnota vyšší než 0 budu ji snižovat a přepiso
    vat hodnotu na indexu j v návraté kolekci o [index i + mininámlní hodnota]
35.                     returnList[j] = i + min;
36.                     j++;
37.                     value--;
38.                 }
39.             }
40.             //Přepíšu původní kolekci novou kolekcí, také lze provést
    například za pomocí RETURNu
41.             for (int i = 0; i < numbers.Count; i++)
42.             {
43.                 numbers[i] = returnList[i];
44.             }
45.         }
```