Algorytmy i struktury danych

Wykład IV – sortowanie

Paweł Rembelski

PJWSTK

16 października 2009

Dla uproszczenia w poniższym wykładzie rozważamy algorytmy sortowania w zbiorze liczb naturalnych z relacją $\leq$. Należy jednak pamiętać, że opisane rozwiązania są poprawne w dowolnym uniwersum U z relacją porządku liniowego $\preceq$.

Definicja problemu

Problem, struktura i specyfikacja algorytmu

Problem

Niech T będzie niepustym n-elementowym wektorem różnych liczb naturalnych. Podać algorytm $Alg\,(T, n)$ porządkujący elementy wektora T tak, że $\forall\,(0 \le i < n-1)\,(T\,[i] < T\,[i+1])$. Rezultat porządkowania będziemy nazywać *wektorem uporządkowanym* (*posortowanym*) dla wektora T.

Struktura dla algorytmu

Struktura dla algorytmu Alg: standardowa struktura liczb naturalnych.

Specyfikacja algorytmu

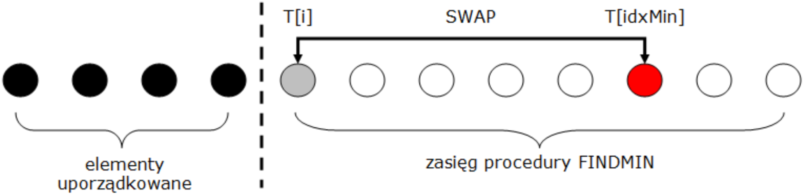
Specyfikację algorytmu Alg stanowi para $\langle WP, WK\rangle$, gdzie warunki początkowy i końcowy są postaci kolejno:

- WP : T jest niepustym wektorem różnych liczb naturalnych, $n \in \mathbb{N}^+$, $|T| = n$,
- WK : $Alg\,(T, n) = T'$, gdzie T' jest wektorem uporządkowanym dla wektora T.

Algorytm sortowania przez selekcję

Pomysł. Niech $i = 0$,

- $n - 1$-krotnie powtórz następujące działanie:
  - stosując algorytm FindMin wyszukaj indeks elementu najmniejszego wśród elementów $T[i], T[i+1], \ldots, T[n-1]$, niech będzie to *idxMin*,
  - zamień element $T[idxMin]$ z elementem $T[i]$,
  - zwiększ i o jeden.



Zadanie. Przedstaw działanie algorytmu sortowania przez selekcję dla następujących danych wejściowych:

$$A = [10, 7, 6, 4, 2, 11, 16, 8, 3, 1, 9].$$

Rozwiązanie problemu – **algorytm SelectionSort**:

```
void SelectionSort(int T[], int n) {
```

WP : T jest niepustym wektorem różnych liczb naturalnych, $n \in \mathbb{N}^+, |T| = n$

```
  int i, idxMin;

  for (i:=0; i<n-1; i:=i+1) {
    idxMin=FindMin(T[i...n-1],n-i);
    if (idxMin≠i) SWAP(T,idxMin,i);
```

procedura wyszukania elementu minimalnego analogiczna do omówionej metody FindMax

```
  }

```

WK : $SelectionSort\,(T, n) = T'$, gdzie T' jest wektorem uporządkowanym dla wektora T

```
}
```

Poprawność algorytmu SelectionSort

- poprawność częściowa: z poprawności częściowej algorytmu FindMin wynika, że po i-tej iteracji pętli zachodzi $\forall (0 \leq j < i) (T[j] < T[j+1])$ i $\forall (i < k < n) (T[i] < T[k])$. Stąd tuż po wykonaniu procedury FindMin i instrukcji SWAP w wierszach 5 i 6 prawdą jest, że $\forall (0 \leq j < i+1) (T[j] < T[j+1])$ i $\forall (i+1 < k < n) (T[i] < T[k])$. Ostatecznie po inkrementacji zmiennej i ponownie zachodzi $\forall (0 \leq j < i) (T[j] < T[j+1])$ i $\forall (i < k < n) (T[i] < T[k])$ – **odtworzenie niezmiennika**. Po zakończeniu pętli iteracyjnej mamy $i = n-1$, stąd $\forall (0 \leq j < n-1) (T[j] < T[j+1])$ i $T[n-2] < T[n-1]$, zatem wektor T jest uporządkowany.
- warunek stopu: algorytm pomocniczy FindMin spełnia własność stopu, stąd każda iteracja pętli algorytmu ma spełnia ową własność. Zmienna i inicjalizowana wartością 0 jest inkrementowana z każdą iteracja pętli o 1, stąd po $n-1$ iteracjach $i = n-1$, co kończy działanie algorytmu.

Złożoność czasowa algorytmu SelectionSort – wariant I

- operacja dominująca: porównanie elementów rozważanego uniwersum,
- złożoność czasowa: $T(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = \Theta\left(n^2\right)$.

Złożoność czasowa algorytmu SelectionSort – wariant II

- operacja dominująca: przestawienie elementów rozważanego uniwersum,
- średnia złożoność czasowa: $A(n) = \frac{n-1}{n} + \frac{n-2}{n-1} + \ldots + \frac{1}{2} = \ldots$
- pesymistyczna złożoność czasowa: $W(n) = n - 1 = \Theta(n)$.

Zadanie. Podaj przykład wektora wejściowego długości n, dla którego algorytm SelectionSort działa w sposób pesymistyczny względem liczby operacji przestawiania elementów.

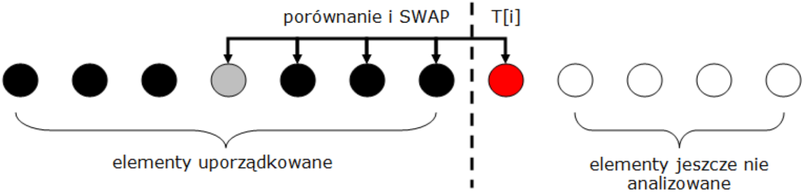
Złożoność pamięciowa algorytmu SelectionSort

Złożoność pamięciową algorytmu SelectionSort można oszacować przez $\Theta(1)$.

Algorytm sortowania przez wstawianie

Pomysł. Niech $i = 1$,

- $n - 1$-krotnie powtórz następujące działanie:
  - wyszukaj sekwencyjnie pozycję dla elementu $T[i]$ w uporządkowanym fragmencie wektora $T[0], T[1], \ldots, T[i-1]$ i jednocześnie przestawiając odpowiednie elementy wstaw rozważany element na właściwą pozycję tak, że powstały wektor $T[0], T[1], \ldots, T[i]$ będzie wektorem uporządkowanym,
  - zwiększ i o jeden.



Zadanie. Przedstaw działanie algorytmu sortowania przez wstawianie dla następujących danych wejściowych:

$$A = [10, 7, 6, 4, 2, 11, 16, 8, 3, 1, 9].$$

Rozwiązanie problemu – **algorytm InsertionSort**:

```
void InsertionSort(int T[], int n) {
  int i, j;

  for (i:=1; i<n; i:=i+1) {
    j:=i;

    while ((j>0) AND (T[j-1]>T[j])) {
      SWAP(T,j-1,j);
      j:=j-1;
    }
  }

}
```

WP : T jest niepustym wektorem różnych liczb naturalnych, $n \in \mathbb{N}^+, |T| = n$

WK : $InsertionSort\,(T, n) = T'$, gdzie T' jest wektorem uporządkowanym dla wektora T

Poprawność algorytmu InsertionSort

- poprawność częściowa: niezmiennikiem pętli zewnętrznej jest formuła $\forall(0 \leq j < i)(T[j] < T[j+1])$. Po wykonaniu pętli wewnętrznej w wierszach 7-10 prawdą jest, że $\forall(0 \leq j \leq i)(T[j] < T[j+1])$, stąd po inkrementacji zmiennej $i := i+1$ ponownie zachodzi $\forall(0 \leq j < i)(T[j] < T[j+1])$ – **odtworzenie niezmiennika**. Po zakończeniu pętli iteracyjnej mamy $i = n$, stąd $\forall(0 \leq j < n)(T[j] < T[j+1])$, zatem wektor T jest uporządkowany.
- warunek stopu pętli wewnętrznej: z warunku początkowego $n \in \mathbb{N}^+$ i własności pętli zewnętrznej $0 < i < n$. Zmienna j inicjalizowana wartością i jest dekrementowana z każdą iteracja pętli o 1, stąd po co najwyżej j iteracjach pętli $j = 0$ i nie jest spełniony pierwszy koniunkt dozoru pętli $j > 0$, co kończy jej działanie.
- warunek stopu: pętla wewnętrzna spełnia własność stopu, stąd każda iteracji pętli zewnętrznej algorytmu spełnia ową własność. Zmienna i inicjalizowana wartością 0 jest inkrementowana z każdą iteracja pętli zewnętrznej o 1, stąd po $n-1$ iteracjach $i = n-1$, co kończy działanie algorytmu.

Złożoność czasowa algorytmu InsertionSort – wariant I

- operacja dominująca: porównanie elementów rozważanego uniwersum,
- średnia złożoność czasowa:

$$A(n) \quad = \quad 1 + \frac{3}{2} + \frac{4}{2} + \ldots + \frac{n}{2} = \frac{n(n+1)}{4} - \frac{1}{2} = \Theta\left(n^2\right),$$

- pesymistyczna złożoność czasowa: $W(n) = 1 + 2 + \ldots + n - 1 = \frac{n(n-1)}{2} = \Theta\left(n^2\right)$.

Złożoność czasowa algorytmu InsertionSort – wariant II

- operacja dominująca: przestawienie elementów rozważanego uniwersum,
- średnia złożoność czasowa:

$$A(n) \quad = \quad \frac{1}{2} + \frac{2}{2} + \frac{3}{2} + \ldots + \frac{n-1}{2} = \frac{n(n-1)}{4} = \Theta\left(n^2\right),$$

- pesymistyczna złożoność czasowa: $W(n) = 1 + 2 + \ldots + n - 1 = \frac{n(n-1)}{2} = \Theta\left(n^2\right)$.

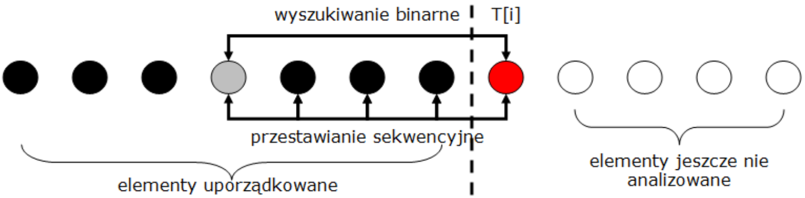
Złożoność pamięciowa algorytmu InsertionSort

Złożoność pamięciową algorytmu InsertionSort można oszacować przez $\Theta(1)$.

Algorytm szybkiego sortowania przez wstawianie – szkic

Pomysł. Niech $i = 1$,

- $n - 1$-krotnie powtórz następujące działanie:
  - wyszukaj stosując rozszerzony algorytm wyszukiwania binarnego* pozycję dla elementu $T[i]$ w uporządkowanym fragmencie wektora $T[0], T[1], \ldots, T[i-1]$, niech będzie to indeks idx,
  - przestaw sekwencyjnie elementy $T[idx], T[idx+1], \ldots, T[i-1]$ o jedną pozycję „w prawo" w rozważanym fragmencie wektora wejściowego,
  - przypisz $T[idx] := T[i]$,
  - zwiększ i o jeden.



---

* standardowo dla algorytmu wyszukiwań binarnych zakładamy, że szukamy indeksu elementu x należącego do rozważanego uniwersum. W wersji rozszerzonej algorytmu to założenie nie musi być spełnione, wtedy interesuje nas indeks elementu $T[i]$ takiego, że $T[i] < x$ i $T[i+1] > x$.

Złożoność czasowa algorytmu QuickInsertionSort – wariant I

- operacja dominująca: porównanie elementów rozważanego uniwersum,
- złożoność czasowa: $T(n) = \sum_{i=1}^{n-1} O(\lg i) = O(\lg(n-1)!) = O(n \lg n)$.

Złożoność czasowa algorytmu QuickInsertionSort – wariant II

- operacja dominująca: przestawienie elementów rozważanego uniwersum,
- średnia złożoność czasowa:

$$A(n) = \frac{1}{2} + \frac{2}{2} + \frac{3}{2} + \ldots + \frac{n-1}{2} = \frac{n(n-1)}{4} = \Theta\left(n^2\right),$$

- pesymistyczna złożoność czasowa: $W(n) = 1 + 2 + \ldots + n - 1 = \frac{n(n-1)}{2} = \Theta\left(n^2\right)$.

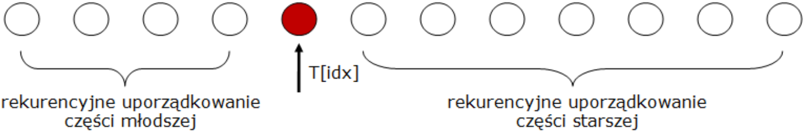
Złożoność pamięciowa algorytmu QuickInsertionSort

Złożoność pamięciową algorytmu QuickInsertionSort można oszacować przez $\Theta(1)$ dla iteracyjnej implementacji rozszerzonego algorytmu wyszukiwania binarnego.

Algorytm sortowania szybkiego

Pomysł. Powtarzaj rekurencyjnie następujący schemat działania z **ustaloną procedurą podziału** względem mediany:

- podziel stosując procedurę podziału elementy aktualnie rozważanego fragmentu wektora względem mediany $T[idx]$ na część młodszą i część starszą,
- powtórz postępowania dla części młodszej,
- powtórz postępowania dla części starszej.



Zadanie. Przedstaw działanie algorytmu sortowania szybkiego dla następujących danych wejściowych:

$$A = [10, 7, 6, 4, 2, 11, 16, 8, 3, 1, 9] .$$

Rozwiązanie problemu – **algorytm QuickSort**:

```
void QuickSort(int T[], int n) {
```

WP : T jest niepustym wektorem różnych liczb naturalnych, $n \in \mathbb{N}^+$, $|T| = n$

```
  int idx;

  idx:=Rozdziel(T,n);
```

procedura Split albo Partition

```
  if (idx>1)
    QuickSort(T[0...idx-1],idx);

  if (n-idx-1>1)
    QuickSort(T[idx+1...n-1],n-idx-1);

```

WK : $QuickSort(T, n) = T'$, gdzie T' jest wektorem uporządkowanym dla wektora T

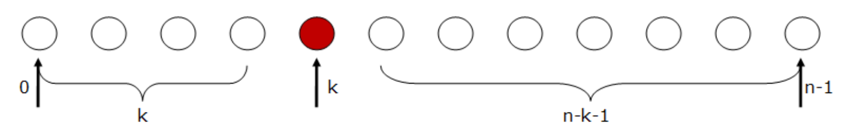
```
}
```

Poprawność algorytmu QuickSort

- poprawność częściowa: dla $n = 1$ poprawność częściowa algorytmu QuickSort wynika z poprawności częściowej algorytmu podziału Rozdziel (np. metoda Partition). Wtedy jednoelementowy wektor T jest zarazem wektorem uporządkowanym. Dla $n > 1$ i po zastosowaniu algorytmu Rozdziel element $T[idx]$ znajduje się na właściwej pozycji, następnie w wierszu 9 rekurencyjnie porządkujemy wektor $T[0 \ldots idx - 1]$ długości idx (czyli w tzw. części młodszej podziału) oraz w wierszu 11 rekurencyjnie porządkujemy wektor $T[idx + 1 \ldots n - 1]$ długości $n - idx - 1$ (czyli w tzw. części starszej podziału).
- warunek stopu: ponieważ $n \in \mathbb{N}^+$ i ciąg kolejnych rozmiarów aktualnie rozważanego fragmentu wektora wejściowego jest ciągiem ściśle malejącym, to po co najwyżej $n - 1$ wywołaniach rekurencyjnych algorytmu QuickSort przestają być prawdziwe warunki $(n - idx - 1 > 1)$ oraz $(idx > 1)$ z wierszy kolejno 9 i 11 , co kończy zejście rekurencyjne rozważanej procedury.

Złożoność algorytmu QuickSort

- operacja dominująca: porównanie elementów rozważanego uniwersum,
- średnia złożoność czasowa: zakładamy, że rozkład elementów n-elementowego wektora T jest losowy, procedura rozdzielania została zaimplementowana zgodnie z metodą Partition albo Split, wtedy $A(n)$ wynosi:



$$A(n) = \begin{cases} 0 & \text{dla } n \leq 1 \\ n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}\left(A(k) + A(n-k-1)\right) & \text{dla } n > 1 \end{cases},$$

czyli* $A(n) = O(n \lg n)$.

* rozwiązanie równania w Dodatku A

Złożoność algorytmu QuickSort c.d.

- pesymistyczna złożoność czasowa: zakładamy, że elementy n-elementowego wektora T są uporządkowane rosnąco, szukamy elementu 1-szego co do wielkości, procedura rozdzielania została zaimplementowana zgodnie z metodą Split*, wtedy:

$$W(n) = \begin{cases} 0 & \text{dla } n \leq 1 \\ n-1+W(n-1) & \text{dla } n > 1 \end{cases},$$

czyli

$$\begin{aligned} W(n) &= n-1+W(n-1) = n-1+n-2+W(n-2) = \ldots = \\ &= \ldots = n-1+n-2+\ldots+0 = \frac{n(n-1)}{2} = \Theta\left(n^2\right). \end{aligned}$$

- złożoność pamięciowa: $O(n)$ z uwzględnieniem kosztów rekursji ($O(1)$ w przeciwnym przypadku).

Zadanie. Podaj możliwie dokładne oszacowanie złożoności oczekiwanej i pesymistycznej algorytmu QuickSort, jeżeli za operację dominującą przyjmiemy przestawianie elementów wektora wejściowego.

---

* jaki jest układ danych wejściowych dla przypadku pesymistycznego wykonania algorytmu QuickSort, jeżeli procedura rozdzielania została zaimplementowana zgodnie z metodą Partition?

Własności algorytmów sortujących

Sortowanie w miejscu

Algorytm sortowania *Alg* danych rozmiaru *n sortuje w miejscu* wtedy i tylko wtedy, gdy $S(Alg, n) = O(1)$.

Pytanie. Który z przedstawiony powyżej algorytmów sortowania sortuje w miejscu?

Sortowanie stabilne

Algorytm sortowania *Alg* danych rozmiaru *n sortuje stabilnie* wtedy i tylko wtedy, gdy porządek występowania danych powtarzających się* przed procesem uporządkowania jest zachowany po owym procesie.

Pytanie. Który z przedstawiony powyżej algorytmów sortowania jest stabilny?

---

* rozważając problem sortowania przyjęliśmy, że wektor wejściowy zawiera różne elementy, wtedy własność stabilności spełniona jest w sposób trywialny. Jej istotę w sposób nietrywialny można rozważać dopiero dla danych pozbawionych założenia niepowtarzalności elementów.

Dodatek A – rozwiązanie równania $A(QuickSort, n)$

Twierdzimy, że $A(n) = O(n \lg n)$, dla

$$A(n) = \begin{cases} 0 & \text{dla } n \leq 1 \\ n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}\left(A(k) + A(n-k-1)\right) & \text{dla } n > 1 \end{cases}$$

Rozważmy dalej równanie dla $n > 1$, wtedy

$$\begin{aligned} A(n) &= n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}\left(A(k) + A(n-k-1)\right) \\ &= n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}A(k) + \frac{1}{n}\sum_{k=0}^{n-1}A(n-k-1) \\ &= n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}A(k) + \frac{1}{n}\sum_{k=n-1}^{0}A(k) = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1}A(k) \\ nA(n) &= n(n-1) + 2\sum_{k=0}^{n-1}A(k), \end{aligned}$$

stąd dla n oraz $n-1$ otrzymujemy

$$\begin{cases} nA\left(n\right) = n\left(n-1\right) + 2\sum_{k=0}^{n-1} A\left(k\right) \\ \left(n-1\right)A\left(n-1\right) = \left(n-1\right)\left(n-2\right) + 2\sum_{k=0}^{n-2} A\left(k\right) \end{cases}$$

i po odjęciu stronami równania dla $\left(n-1\right)A\left(n-1\right)$ od równania dla $nA\left(n\right)$ zachodzi

$$\begin{aligned} nA\left(n\right) - \left(n-1\right)A\left(n-1\right) &= 2\left(n-1\right) + 2A\left(n-1\right) \\ nA\left(n\right) &= 2\left(n-1\right) + \left(n+1\right)A\left(n-1\right) \end{aligned}$$

stąd po podzieleniu obu stron przez $n\left(n+1\right)$

$$\frac{A\left(n\right)}{n+1} = \frac{A\left(n-1\right)}{n} + 2\frac{n-1}{n\left(n+1\right)} = \frac{A\left(n-1\right)}{n} + 2\left(\frac{1}{n+1} - \frac{1}{n\left(n+1\right)}\right).$$

Zatem

$$\begin{aligned}
\frac{A\,(n)}{n+1} &= \frac{A\,(n-2)}{n-1} + 2\left(\frac{1}{n} - \frac{1}{(n-1)\,n}\right) + 2\left(\frac{1}{n+1} - \frac{1}{n\,(n+1)}\right) \\
&= \frac{A\,(1)}{2} + 2\left(\frac{1}{3} - \frac{1}{2\cdot 3}\right) + \ldots + 2\left(\frac{1}{n} - \frac{1}{(n-1)\,n}\right) + 2\left(\frac{1}{n+1} - \frac{1}{n\,(n+1)}\right) \\
&= 2\sum_{k=2}^{n}\left(\frac{1}{k+1} - \frac{1}{k\,(k+1)}\right) = 2\sum_{k=1}^{n}\left(\frac{1}{k+1} - \frac{1}{k\,(k+1)}\right) \\
A\,(n) &= 2\,(n+1)\left(\sum_{k=1}^{n}\frac{1}{k+1} - \sum_{k=1}^{n}\frac{1}{k\,(k+1)}\right).
\end{aligned}$$

Ponieważ

$$\sum_{k=1}^{n}\frac{1}{k\,(k+1)} \quad \leq \quad \sum_{k=1}^{\infty}\frac{1}{k\,(k+1)} = 1,$$

to dla pewnego $\frac{1}{2} \leq c_1 \leq 1$

$$A\,(n) = 2\,(n+1)\left(\sum_{k=1}^{n} \frac{1}{k+1} - c_1\right).$$

Dalej

$$\sum_{k=1}^{n} \frac{1}{k+1} \quad = \quad \sum_{k=1}^{n} \frac{1}{k} - 1 + \frac{1}{n+1} = H_n - \frac{n}{n+1},$$

gdzie H_n jest n-tą liczbą harmoniczną, tj. $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$, dla której zachodzi

$$H_n = \ln n + c_2,$$

gdzie $\gamma \leq c_2 \leq 1$ i γ jest stałą Eulera ($\gamma \approx 0,577$). Ostatecznie

$$A\,(n) = 2\,(n+1)\left(\ln n + c_2 - \frac{n}{n+1} - c_1\right).$$

$$\begin{aligned} A\left(n\right) &= 2\left(n+1\right)\left(\ln n + c_2 - \frac{n}{n+1} - c_1\right) \\ &= \frac{2}{\lg e}\left(n+1\right)\lg n - 2n + 2\left(n+1\right)\left(c_2 - c_1\right). \end{aligned}$$

Pamiętając, że $\gamma - 1 \leq \left(c_2 - c_1\right) \leq \frac{1}{2}$, to

$$\begin{aligned} A\left(n\right) &= \frac{2}{\lg e} n \lg n + O\left(\lg n\right) + O\left(n\right) \\ &= \frac{2}{\lg e} n \lg n + O(n), \end{aligned}$$

dla $\frac{2}{\lg e} \approx 1,386$. Stąd

$$A\left(n\right) = O\left(n \lg n\right).$$

Literatura

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Wprowadzenie do algorytmów*, WNT 2004.
2. L. Banachowski, K. Diks, W. Rytter, *Algorytmy i struktury danych*, WNT 1996.
3. A. V. Aho, J. E. Hopcroft, J. D. Ullman, *Algorytmy i struktury danych*, Helion 2003.
4. A. Dańko, T. L. Lee, G. Mirkowska, P. Rembelski, A. Smyk, M. Sydow, *Algorytmy i struktury danych – zadania*, PJWSTK 2006.
5. R. Sedgewick, *Algorytmy w C++*, RM 1999.
6. N. Wirth, *Algorytmy + struktury danych = programy*, WNT 1999.
7. A. Drozdek, D. L. Simon, *Struktury danych w języku C*, WNT 1996.
8. D. Harel, *Rzecz o istocie informatyki – Algorytmika*, WNT 2001.

Zadania ćwiczeniowe

1. Zaimplementuj algorytm SelectionSort.
2. Zaimplementuj algorytm InsertionSort.
3. Przeprowadź doświadczalnie analizę porównawczą efektywności algorytmów InsertionSort i SelectionSort względem liczby następujących operacji dominujących:
    1. operacja porównania elementów wektora wejściowego,
    2. operacja przestawienia elementów wektora wejściowego..
4. Zaproponuj specyfikację dla rozszerzonego algorytmu wyszukiwania binarnego. Zaimplementuj algorytm, uzasadnij jego poprawność oraz oszacuj złożoność.
5. Zaimplementuj algorytm FastInsertionSort.
6. Przeprowadź doświadczalnie analizę porównawczą efektywności algorytmów InsertionSort i FastInsertionSort względem liczby następujących operacji dominujących:
    1. operacja porównania elementów wektora wejściowego,
    2. operacja przestawienia elementów wektora wejściowego.
7. Zaimplementuj algorytm QuickSort.
8. Przeprowadź doświadczalnie analizę porównawczą efektywności algorytmów FastInsertionSort i QuickSort względem liczby następujących operacji dominujących:
    1. operacja porównania elementów wektora wejściowego,
    2. operacja przestawienia elementów wektora wejściowego.
9. Zaproponuj iteracyjną wersję algorytmu QuickSort. Zaimplementuj algorytm, uzasadnij jego poprawność oraz oszacuj złożoność.
10. Niech X oraz Y będą wektorami odpowiednio n oraz $n+1$ parami różnych liczb naturalnych takimi, że $X \subset Y$. Zaprojektuj możliwie efektywną funkcję
    `int FIND(int X[], int Y[], int n),`
    która wyznaczy indeks elementu wektora Y, który jednocześnie nie należy do wektora X. Uzasadnij poprawność oraz oszacuj złożoność rozwiązania.

11. Oszacuj koszt i uzasadnij poprawność następującego algorytmu sortowania:
    - optymalnym algorytmem dla problemu min-max wyszukuję minimum i maksimum dla danego ciągu i ustawiam je odpowiednio na początku i na końcu tablicy,
    - powtórz rozumowanie dla pozostałych elementów.

12. Dany jest ciąg n par (*indeks*, *kolor*) uporządkowany rosnąco ze względu na wartość pierwszej składowej, gdzie *indeks* jest pewną liczbą naturalną, a *kolor* jest elementem zbioru kolorów *{żółty, czerwony, niebieski}*. Zaproponuj możliwie efektywny algorytm sortowania ciągu par tak, że jego elementy będą ułożone kolorami (w kolejności niebieskie, żółte, czerwone) oraz elementy o tym samym kolorze pozostaną uporządkowane rosnąco ze względu na wartość składowej *indeks*. Oszacuj koszt i krótko uzasadnij poprawność zaproponowanego algorytmu.

13. Algorytm sortowania metodą Shella (metoda malejących przyrostów): niech T będzie wektorem n parami różnych liczb naturalnych, oraz $h_1, h_2, \ldots, h_k$ ściśle malejącym ciągiem k liczb naturalnych, gdzie $h_k = 1$. Dla każdego $1 \leq i \leq k$ wykonaj kolejno:
    1. podziel w miejscu wektor T na h_i podwektorów $T_{i,1}, T_{i,2}, \ldots, T_{i,h_i}$ tak, że:
    $$\begin{aligned} T_{i,1} &= \left[T[0], T[h_i], T[2h_i], \ldots, T\left[l_{i,1} h_i\right]\right], \\ T_{i,2} &= \left[T[1], T[h_i+1], T[2h_i+1], \ldots, T\left[l_{i,2} h_i\right]\right], \\ &\vdots \\ T_{i,h_i} &= \left[T[h_i-1], T[2h_i-1], T[3h_i-1], \ldots, T\left[l_{i,h_i} h_i\right]\right], \end{aligned}$$
    2. posortuj każdy z podwektorów oddzielnie algorytmem sortowania przez wstawianie,
    3. jeżeli $h_i = 1$, to zakończ działanie algorytmu, w p.p. wykonaj podobne postępowanie dla kolejnego współczynnika ciągu h_{i+1}.

Kolejno:

- przeanalizuj działanie prezentowanego algorytmu dla przykładowego wektora T składającego się z 12-stu liczb naturalnych postaci

$$T = [10, 8, 6, 20, 4, 3, 22, 1, 0, 15, 16]$$

  oraz ciągu współczynników $5, 3, 2, 1$,
- zaimplementuj algorytm Shella,
- uzasadnij poprawność algorytmu i oszacuj jego złożoność.