1 Sortowanie przez scalanie

1.1 Sformułowanie zadania

Zad. 2 (8 pkt.) Podaj iteracyjną (a nie rekurencyjną) wersję algorytmu sortowania przez scalanie.

Przyjmij, że dane znajdują się w n elementowej tablicy indeksowanej liczbami naturalnymi. Podczas scalania możesz użyć n elementowej tablicy pomocniczej. Poza tym – skończenie wielu zmiennych prostych.

Idea: Na dany ciąg liczb (zapisany w tablicy) można spojrzeć też w inny sposób: jest to ciąg uporządkowanych ciągów liczb. Na początku jest to ciąg ciągów jednoelementowych, a więc uporządkowanych. Możemy brać uporządkowane ciągi parami i scalać je. Po pierwszym scalaniu wszystkich kolejnych par otrzymamy ciąg uporządkowanych ciągów dwuelementowych. Po drugim – czteroelementowych, itd. Zauważ jednak, że ostatni z uporządkowanych ciągów nie musi być pełnej długości.

1.2 Analiza zadania i krok pierwszy

Daną dla programu jest tablica np. liczb całkowitych, indeksowana liczbami od 0 do $n-1$. Tę tablicę mamy uporządkować (posortować) przez scalanie. Metoda porządkowania bardziej szczegółowo została opisana w treści zadania. Zakłada, że w danej tablicy kolejne fragmenty ustalonej długości są uporządkowane. Potrzebna więc będzie zmienna pamiętająca długość uporządkowanych fragmentów. Powiedzmy, że będzie to zmienna **dl**. Dane powinny więc być takie, że fragmenty

$$a[0]\ldots a[dl-1],\ \ a[dl]\ldots a[2dl-1],\ \ a[2dl]\ldots a[3dl-1],\ \ a[3dl]\ldots a[4dl-1],\ldots$$

są już uporządkowane. Powinniśmy też umieć je przekształcić w takie dane, aby uporządkowane były fragmenty

$$a[0]\ldots a[2dl-1],\ \ a[2dl]\ldots a[4dl-1],\ldots$$

Będziemy nazywać tę czynność słowem **scal_wszystkie_pary**. Główny program może więc mieć postać:

1) **Dane:** tablica **a** indeksowana liczbami od **0** do **n − 1**.
2) **dl = 1**
3) **while (dl < n) {**
4) **scal_wszystkie_pary;**
5) **dl = dl + dl;**
6) **}**

Niezmiennikiem pętli w tym programie powinno być stwierdzenie, że *w tablicy* **a** *kolejne fragmenty długości* **dl** *są uporządkowane*. Jeżeli tak będzie, to program będzie poprawnie sortować. Proszę sprawdzić.

1.3 Procedura scal_wszystkie_pary

Należy więc jeszcze opracować procedurę **scal_wszystkie_pary**. Będą w niej potrzebne dwie zmienne pamiętające, które fragmenty danej tablicy są scalane. początkowo zmienne te będą pamiętać pierwsze elementy scalanych fragmentów. Później będą pamiętać indeksy scalanych elementów interesujących nas fragmentów. Indeksy do pierwszego fragmentu będzie pamiętać zmienna **pw**, a do drugiego – zmienna **dr**. Cała procedura może zostać napisana w następujący sposób i korzystać z pomocniczej procedury **scal_parę** (która robi to, na co wskazuje jej nazwa):

1) **pw = 0; dr = dl;**
2) **while (dr < n) {**
3) **scal_parę;**
4) Komentarz: powyższa linia powinna scalać dwa fragmenty tablicy **a**, a mianowicie $a[pw] \ldots a[dr-1]$ oraz $a[dr] \ldots a[dr+dl-1]$ (z pewnymi wyjątkami) w jeden uporządkowany frgment $a[pw] \ldots a[dr+dl-1]$
5) **pw = dr + dl;**
6) **dr = pw + dl;**
7) **}**

1.4 Procedura scal_parę

Jest to zwykłą procedurą scalającą, ale trochę powinna zostać dostosowana do sytuacji, w której się znaleźliśmy (musimy dostosować się sposobu operowania danymi, nie musimy przepisywać części danych). Można ją tak oto napisać:

1) Komentarz: najpierw pierwszy scalany fragment przepisujemy do pomocniczej tablicy. Robimy w ten sposób miejsce do zapisywania scalanego ciągu.
2) **i = 0;**
3) **for (k = pw; k++; k < dr) {**
4) **pom[i] = a[k]; i++; }**
5) Komentarz: przystępujemy do scalania danych z tablicy pomocniczej i z drugiego fragmentu tablicy **a**. Wynik scalania ma zostać zapisany początkowo na miejscu pierwszego fragmentu, a później – także drugiego. Zmienne **i**, **j** i **k** wskazują aktualne elementy odpowiednio tablicy pomocniczej, drugiego fragmentu tablicy **a** i i fragmentu **a** przeznaczonego do zapisywania ciągu wynikowego. Zaczynamy od wartości początkowych tych zmiennych.

6) **i = 0; j = dr; k = pw;**

7) Komentarz: dalej obliczamy, gdzie jest koniec drugiego fragmentu danych. Pamiętajmy, że ten fragment może być niepełny. W rzeczywistości znajdujemy najmniejszą liczbę, która nie może być indeksem danej z drugiego fragmentu.

8) **ogr = dr + dl;**

9) **if (ogr > n) ogr = n;**

10) Komentarz: w końcu zaczynamy scalać ciągi $pom[0] \ldots pom[dr-1]$ oraz $a[dr] \ldots a[ogr-1]$, wynik scalania powinien znaleźć w tablicy $a[pw] \ldots a[ogr-1]$.

11) **while (i < dr && j < ogr) {**

12) **if (pom[i] < a[j]) { a[k] = pom[i]; i++; }**

13) **else { a[k] = a[j]; j++; };**

14) **k++;**

15) **}**

16) Komentarz: pozostało przepisać nie scalone dotychczas części danych. W istniejącej sytuacji wystarczy przepisać resztę danych z tablicy pomocniczej: niescalone dane z drugiego fragmentu są już na swoim miejscu.

17) **while (i < dl) {**

18) **a[k] = pom[i]; i++; k++; }**

To już cały program sortujący przez scalanie, ale przydałoby się jeszcze zastanowić, czy jest on poprawny. Główne argumenty o tym świadczące są podobne do używanych w uzasadnieniu poprawności zwykłego algorytmu sortującego przez scalanie. Dodatkowo przydałoby się sprawdzić, czy zapisując wyniki do tablicy z danymi nie zniszczymy przedwcześnie jakiejś danej lub kopiując za mało danych doprowadzimy do przypadkowego powielenia jakiejś liczby (efekt w obu przypadkach będzie podobny, coś zniknie i coś się pojawi, ale będzie to spowodowane różnymi przyczynami) .