Przykładowe zadania

z informatyki

na poziomie rozszerzonym

wraz z rozwiązaniami

Zadania 1–12 zawierają po cztery odpowiedzi, z których każda jest albo prawdziwa, albo fałszywa. Zdecyduj, które z podanych odpowiedzi są prawdziwe (P), a które fałszywe (F), a następnie zapisz numer każdej odpowiedzi oraz odpowiednią literę F lub P.

Zadanie 1. (0–1)

Dana jest tabela: Sprawdzian

|  |  |  |
| --- | --- | --- |
| uczen | klasowka | egzamin |
| Abacki | 45 | 0 |
| Babacki | 50 | 80 |
| Cabacki | 100 | 90 |
| Dabacki | 80 | 70 |

Dla powyższej tabeli utworzono następujące zapytanie w SQL:

SELECT uczen

FROM Sprawdzian

WHERE (klasowka > egzamin AND egzamin > 75) OR klasowka < 50

Wynikiem tego zapytania jest

1. Abacki, Babacki.

2. Babacki, Cabacki.

3. Abacki, Cabacki.

4. Abacki, Dabacki.

Rozwiązanie:

1F; 2F; 3P; 4F

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 2. (0–1)

Liczba CB(16) jest równa liczbie

1. 1010101111(2).

2. 313(8).

3. 112011120(3).

4. 203(10).

Rozwiązanie:

1F; 2P; 3F; 4P

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 3. (0–1)

W grafice rastrowej

1. każdy piksel ma jednoznacznie określony kolor.

2. obraz pamiętany jest w postaci obiektów geometrycznych.

3. zaletą jest skalowalność obrazu.

4. mogą być zapisywane zdjęcia z aparatu cyfrowego.

Rozwiązanie:

1P; 2F; 3F; 4P

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 4. (0–1)

Do szyfrowania informacji służy

1. algorytm RSA.

2. metoda bisekcji.

3. PGP.

4. algorytm Huffmana.

Rozwiązanie:

1P; 2F; 3P; 4F

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 5. (0–1)

Dynamicznym przydzielaniem numerów IP w sieci zajmuje się serwer

1. DNS.

2. DHCP.

3. SMTP.

4. FTP.

Rozwiązanie:

1F; 2P; 3F; 4F

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 6. (0–1)

Dane: n– liczba naturalna większa od zera

Funkcja K(n)

1. dla n < 4 wynikiem jest 1
2. dla n >= 4 wynikiem jest K(n–1) – K(n–3)

Dla funkcji K zachodzi

1. dla każdego n>4 zachodzi K(n)<0.

2. K(2) > K(5).

3. K(10) = 3.

4. funkcja jest niemalejąca.

Rozwiązanie:

1F; 2P; 3P; 4F

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 7. (0–1)

Rozważ poniższy algorytm, gdzie n jest liczbą całkowitą nieujemną.

(1) wynik przypisz 0;

(2) dopóki n ≠ 0 wykonuj

(3) wynik przypisz wynik + (n mod 10)

(4) n przypisz n div 10

gdzie: mod to operator reszty z dzielenia,

div to operator dzielenia całkowitego.

Dla podanego algorytmu zachodzi

1. dla n = 36789 wynik = 30.

2. dla n = 11111111 wynik = 8.

3. wynik jest równy sumie cyfr w zapisie dziesiętnym liczby n.

4. dla n = 1234 zmienna wynik w kolejnych iteracjach przyjmuje wartości 1,3,6,10.

Rozwiązanie:

1F; 2P; 3P; 4F

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 8. (0–1)

Rozważ poniższy algorytm, gdzie n jest liczbą całkowitą nieujemną, a[0..n] jest tablicą liczb całkowitych, z – liczbą rzeczywistą.

(1) i przypisz n; y przypisz a[n];

(2) dopóki i≠ 0 wykonuj

(3) i przypisz i–1

(4) y przypisz y\*z + a[i]

Algorytm ten przedstawia realizację

1. obliczania wartości wielomianu dla danej wartości z.

2. obliczenia NWW dla n liczb naturalnych.

3. obliczenia NWD dla n liczb naturalnych.

4. schematu Hornera.

Rozwiązanie:

1P; 2F; 3F; 4P

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 9. (0–1)

Program komputerowy na licencji freeware można

1. rozpowszechniać, jednak z zachowaniem informacji o autorze.

2. wykorzystać do tworzenia nowych programów przez wprowadzanie w nim zmian.

3. stosować do obliczeń.

4. sprzedawać.

Rozwiązanie:

1P; 2F; 3P; 4F

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 10. (0–1)

Zgodnie z przepisami prawa autorskiego dozwolone jest

1. publikowanie pod własnym nazwiskiem, na swojej stronie WWW, skopiowanych zasobów internetowych (zdjęć i artykułów).

2. zamieszczanie na własnej stronie linków do innych stron WWW.

3. zamieszczanie na własnej stronie cudzych programów na licencji freeware.

4. zamieszczenie na stronie internetowej treści utworów, do których wygasły majątkowe prawa autorskie.

Rozwiązanie:

1F; 2P; 3P; 4P

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 11. (0–1)

Do jednoznacznej identyfikacji osoby podpisującej cyfrowy dokument służy

1. podpis elektroniczny.

2. wpisanie imienia i nazwiska.

3. zaszyfrowanie dokumentu.

4. wstawienie zeskanowanego podpisu autora.

Rozwiązanie:

1P; 2F; 3F; 4F.

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 12. (0–1)

Algorytmem sortowania przez porównania jest

1. sortowanie przez wybór.

2. sortowanie kubełkowe.

3. sortowanie przez wstawianie.

4. sortowanie szybkie.

Rozwiązanie:

1P; 2F; 3P; 4P.

Schemat punktowania:

1 pkt – poprawne zapisanie wszystkich odpowiedzi.

0 pkt – błędne zapisanie lub ich brak.

Zadanie 13. Zapisy binarne (0–7)

W tym zadaniu badamy zapisy binarne dodatnich liczb całkowitych. Rozważamy następujący algorytm:

Specyfikacja

Dane: dodatnia liczba całkowita n

Wynik: dodatnia liczba całkowita j równa ...............................

(1) j przypisz 0;

(2) powtarzaj

(3) jeśli n mod 2 = 1, to

(4) j przypisz j+1;

(5) n przypisz n div 2;

(6) aż n = 0;

Uwaga: użyte operatory mod i div oznaczają odpowiednio resztę z dzielenia i dzielenie całkowite. Na przykład 5 mod 2 = 1, 5 div 2 = 2, 6 mod 2 = 0, 6 div 2 = 3.

1. Dokończ zdanie: "Wynik: dodatnia liczba całkowita j równa ............." i zapisz je. Przeanalizuj powyższy algorytm i zapisz wartości zmiennej j po zakończeniu jego działania dla n = 183 oraz dla n = 1022.
2. Ułóż algorytm i zapisz go w wybranej przez siebie notacji (lista kroków lub język programowania, który wybrałeś na egzamin), który dla danej dodatniej liczby całkowitej n oblicza maksymalną liczbę kolejnych jedynek pojawiających się w zapisie binarnym tej liczby.

Specyfikacja

Dane: dodatnia liczba całkowita n

Wynik: dodatnia liczba całkowita m – maksymalna liczba kolejnych jedynek w zapisie binarnym n

Przykład: dla n = 187 wynikiem jest m = 3, ponieważ 187 = (10111011)2

Schemat punktowania:

Liczba punktów za zadanie - 7 pkt

W tym:

za podpunkt a - za każdą poprawną odpowiedź po 1 punkcie, razem 3 punkty

za podpunkt b - za poprawny algorytm – 4 punkty.

W przypadku niepoprawnego algorytmu:

za prawidłowe wyznaczanie maksimum długości bloków jedynek – 1 punkt.

za prawidłowe wyznaczanie długości bloków jedynek – 1 punkt

Zadanie 13. Zapisy binarne (0–7) – rozwiązanie

W tym zadaniu badamy zapisy binarne dodatnich liczb całkowitych. Rozważamy następujący algorytm:

Specyfikacja

Dane: dodatnia liczba całkowita n

Wynik: dodatnia liczba całkowita j równa liczbie jedynek w zapisie binarnym liczby n

(1) j przypisz 0;

(2) powtarzaj

(3) jeśli n mod 2 = 1, to

(4) j przypisz j+1;

(5) n przypisz n div 2;

(6) aż n = 0;

Uwaga: użyte operatory mod i div oznaczają odpowiednio resztę z dzielenia i dzielenie całkowite, np. 5 mod 2 = 1, 5 div 2 = 2, 6 mod 2 = 0, 6 div 2 = 3.

a) Przeanalizuj powyższy algorytm i podaj wartości zmiennej j po zakończeniu jego działania dla n = 183 oraz dla n = 1022.

Dla n=183 dla j=6.

Dla n=1022 dla j=9.

Obliczenia:

183 = (10110111)2

1022 = (1111111110)2

b) Ułóż algorytm i zapisz go w wybranej przez siebie notacji (lista kroków lub język programowania, który wybrałeś na egzamin), który dla danej dodatniej liczby całkowitej n oblicza maksymalną liczbę kolejnych jedynek pojawiających się w zapisie binarnym tej liczby.

Specyfikacja

Dane: dodatnia liczba całkowita n

Wynik: dodatnia liczba całkowita m – maksymalna liczba kolejnych jedynek w zapisie binarnym n

Przykład: dla n = 187 wynikiem jest m = 3, ponieważ 187 = (10111011)2

Algorytm

Każdy ciąg kolejnych jedynek w zapisie binarnym liczby, który nie można już wydłużyć, nazywamy blokiem. W zapisie binarnym liczby 187 mamy trzy bloki jedynek o długościach jeden, trzy i dwa: (10111011)2. Naszym celem jest policzenie długości najdłuższego bloku.

Modyfikujemy algorytm z podpunktu a), wyznaczając kolejne cyfry liczby n, od cyfr najmniej znaczących do cyfr najbardziej znaczących. W momencie wykrycia bloku (pierwszej jedynki w tym bloku) rozpoczynamy zliczanie jedynek w nim zawartych, aż w zapisie binarnym napotkamy zero lub wyznaczymy już wszystkie cyfry zapisu. Po przetworzeniu bloku porównujemy jego długość z długością dotychczas najdłuższego bloku i jeśli policzona długość jest większa od dotychczas największej, aktualizujemy informację o długości najdłuższego bloku.

Poniżej zapis opisanego słowami algorytmu:

(1) m przypisz 0;

(2) powtarzaj

 /\* m – długość dotychczas najdłuższego bloku \*/

(3) jeśli n mod 2 = 1, to

 /\* nowy blok \*/

(4) dl\_bloku przypisz 0; /\* tu zliczamy liczbę jedynek w bloku \*/

(5) powtarzaj

(6) dl\_bloku przypisz dl\_bloku + 1;

(7) n przypisz n div 2;

(8) aż n mod 2 = 0;

(9) jeśli dl\_bloku >m, to

(10) m przypisz dl\_bloku;

(11) n przypisz n div 2;

(12) aż n = 0;

Komentarz

Podpunkt a) w tym zadaniu nie powinien sprawić żadnych trudności. Przedstawiony w nim algorytm jest typowym szkolnym algorytmem wyznaczania kolejnych cyfr dodatniej liczby całkowitej w jej zapisie binarnym, poczynając od cyfry najmniej znaczącej, a kończąc na cyfrze najbardziej znaczącej. Warto zauważyć, że w ten sam sposób można wyznaczyć cyfry w zapisie pozycyjnym przy dowolnej podstawie p, 2 ≤ p ≤ 10. Wystarczy wykonywać operacje dzielenia całkowitego i brania reszty z dzielenia z parametrem p zamiast 2. Dla liczby naturalnej n, n mod p jest najmniej znaczącą cyfrą w zapisie pozycyjnym liczby n przy podstawie p. Dla przykładu 187 mod 10 = 7, 187 mod 2 = 1. Jeśli najmniej znaczącą cyfrą w zapisie przy podstawie p liczby n jest cyfra c, to n = n’•p + c, dla pewnej liczby naturalnej n’. Wówczas n div p = n’ i kolejna cyfrą w zapisie n jest najmniej znacząca cyfra w zapisie n’. Te własności właśnie wykorzystano w algorytmie z podpunktu a).

W punkcie b), oprócz wyznaczania cyfr liczby n, należy zliczać jedynki w blokach kolejnych jedynek. Tutaj najpierw trzeba wykryć blok. To jest proste – blok rozpoczyna się od jedynki. Następnie należy zliczać w pętli kolejne napotkane jedynki, aż pojawi się zero. Tak naprawdę w tym celu wykorzystujemy pętlę z algorytmu w punkcie a). Tak więc cały algorytm składa się z dwóch zagnieżdżonych pętli, których struktury są podobne do pętli z punktu a).

Można sobie wyobrazić inne rozwiązanie. W tym nowym rozwiązaniu zliczamy jedynki za każdym razem od momentu pojawienia się pierwszej jedynki w bloku. Pojawienie się zera powoduje wyzerowanie licznika jedynek. Oto formalny zapis tego algorytmu:

(1) m przypisz 0;

(2) dl\_bloku 0;

(3) dopóki n ≠ 0 wykonuj

(4) jeśli n mod 2 = 1 to

(5) dl\_bloku przypisz dl\_bloku + 1;

(6) w przeciwnym wypadku

(7) jeśli dl\_bloku >m to

(8) m := dl\_bloku;

(9) dl\_bloku przypisz 0;

(10) n przypisz n div 2;

Zadanie 14. Sortowanie (0–6)

W tym zadaniu rozważamy algorytmy sortujące niemalejąco n-elementową tablicę liczb całkowitych a[1..n], gdzie n jest dodatnią liczbą całkowitą. Algorytm sortowania nazywamy lokalnym, gdy podczas sortowania można porównywać i zamieniać ze sobą tylko sąsiednie elementy tablicy. Na przykład dopuszczalne jest porównanie i zamiana elementów a[5] i a[6], natomiast nie można bezpośrednio porównywać i zamieniać ze sobą elementów a[5] i a[7].

1. Które z następujących algorytmów sortowania są algorytmami lokalnymi: bąbelkowy, przez wstawianie liniowe, szybki? Zapisz numer algorytmu a następnie słowo TAK lub NIE, które jest odpowiedzią na pytanie "Czy jest lokalny?".

Algorytm:

1. Bąbelkowy

2. Przez wstawianie liniowe

3. Szybki

1. Dla tablicy a[1..4] = [3,2,4,1] algorytm sortowania przez wstawianie liniowe wykona dokładnie 4 zamiany sąsiednich elementów: (3 z 2), (4 z 1), (3 z 1), (2 z 1).

W podanych poniżej tablicach dobierz odpowiednie wartości "Zawartości" i przepisz te tablice z wymyślonymi wartościami. Do każdej pustej pozycji powinna być znaleziona dokładnie jedna wartość "Zawartości". Wartości "Zawartości" muszą być różnymi liczbami całkowitymi takimi, aby algorytm sortowania przez wstawianie liniowe wykonał na każdej z nich dokładnie 11 zamian sąsiednich elementów.

Tablica 1.

Pozycja Zawartość

1 10

2 1

3 2

4

5 4

6 5

7 6

8

9 7

10 8

Tablica 2.

Pozycja Zawartość

1 1

2 2

3 3

4 5

5 4

6

7

8

9

10

1. Załóżmy teraz, że w jednym kroku możemy posortować blok kolejnych elementów tablicy dłuższy niż 2. Na przykład gdybyśmy mogli sortować bloki o długościach do 9 elementów, wówczas tablicę 10-elementową można by posortować w trzech krokach: najpierw w jednym kroku sortujemy ostatnie 9 elementów. W następnym kroku sortujemy pierwsze 9 elementów. Teraz wiemy, że element najmniejszy jest już na swojej, czyli pierwszej, pozycji w tablicy. Jeszcze jedno sortowanie ostatnich 9 elementów kończy sortowanie całego ciągu. Oznaczmy przez Sort(i,j) sortowanie w jednym kroku bloku kolejnych elementów z pozycji od i do j. Wówczas powyższe sortowanie można zapisać w następujący sposób:
2. Sort(2,10);
3. Sort(1,9);
4. Sort(2,10);

**Przykład:**

Początkowa zawartość tablicy:

 a = [10, 2, 8, 4, 6, 5, 7, 9, 3, 1]

Sortowanie ostatnich 9 elementów (Sort(2,10)):

 a = [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Sortowanie pierwszych 9 elementów (Sort(1,9)):

 a = [1, 2, 3, 4, 5, 6, 7, 8, 10, 9]

Sortowanie ostatnich 9 elementów (Sort(2,10)):

 a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

Zapisz algorytm sortowania tablicy 1000-elementowej w co najwyżej 6 krokach, przy założeniu, że w jednym kroku można posortować blok złożony z co najwyżej 500 elementów.

Schemat punktowania:

Liczba punktów za zadanie - 6 pkt

W tym:

za podpunkt a - za poprawne zapisanie wszystkich odpowiedzi – 1 punkt.

za podpunkt b - za poprawne uzupełnienie pierwszej tablicy (pozycja 4 – 3, pozycja 8 – 9) – 1 punkt. Za poprawne uzupełnienie drugiej tablicy (10,9,8,7,6)
– 1 punkt. Łączna ilość punktów za podpunkt b - 2 punkty.

za podpunkt c - Za poprawny algorytm – 3 punkty. W przypadku błędnego algorytmu:

ustawienie minimum na pierwszej pozycji – 1 punkt.

ustawienie maksimum na ostatniej pozycji – 1 punkt.

Zadanie 14. Sortowanie (0–6) – rozwiązanie

W tym zadaniu rozważamy algorytmy sortujące niemalejąco n-elementową tablicę liczb całkowitych a[1..n], gdzie n jest dodatnią liczbą całkowitą. Algorytm sortowania nazywamy lokalnym, gdy podczas sortowania można porównywać i zamieniać ze sobą tylko sąsiednie elementy tablicy. Na przykład dopuszczalne jest porównanie i zamiana elementów a[5] i a[6], natomiast nie można bezpośrednio porównywać i zamieniać ze sobą elementów a[5] i a[7].

1. Które z następujących algorytmów sortowania są algorytmami lokalnymi: bąbelkowy, przez wstawianie liniowe, szybki? Zapisz numer algorytmu a następnie słowo TAK lub NIE, które jest odpowiedzią na pytanie "Czy jest lokalny?".

Algorytm:

1. Bąbelkowy

2. Przez wstawianie liniowe

3. Szybki

Rozwiązanie: 1. TAK, 2. TAK, 3. Nie

1. Dla tablicy a[1..4] = [3,2,4,1] algorytm sortowania przez wstawianie liniowe wykona dokładnie 4 zamiany sąsiednich elementów: (3 z 2), (4 z 1), (3 z 1), (2 z 1).

W podanych poniżej tablicach dobierz odpowiednie wartości "Zawartości" i przepisz te tablice z wymyślonymi wartościami. Do każdej pustej pozycji powinna być znaleziona dokładnie jedna wartość "Zawartości". Wartości "Zawartości" muszą być różnymi liczbami całkowitymi takimi, aby algorytm sortowania przez wstawianie liniowe wykonał na każdej z nich dokładnie 11 zamian sąsiednich elementów.

Tablica 1.

Pozycja Zawartość

1 10

2 1

3 2

4 3

5 4

6 5

7 6

8 9

9 7

10 8

Tablica 2.

Pozycja Zawartość

1 1

2 2

3 3

4 5

5 4

6 10

7 9

8 8

9 7

10 6

1. Załóżmy teraz, że w jednym kroku możemy posortować blok kolejnych elementów tablicy dłuższy niż 2. Na przykład gdybyśmy mogli sortować bloki o długościach do 9 elementów, wówczas tablicę 10-elementową można by posortować w trzech krokach: najpierw w jednym kroku sortujemy ostatnie 9 elementów. W następnym kroku sortujemy pierwsze 9 elementów. Teraz wiemy, że element najmniejszy jest już na swojej, czyli pierwszej, pozycji w tablicy. Jeszcze jedno sortowanie ostatnich 9 elementów kończy sortowanie całego ciągu. Oznaczmy przez Sort(i,j) sortowanie w jednym kroku bloku kolejnych elementów z pozycji od i do j. Wówczas powyższe sortowanie można zapisać w następujący sposób:
2. Sort(2,10);
3. Sort(1,9);
4. Sort(2,10);

**Przykład:**

Początkowa zawartość tablicy:

 a = [10, 2, 8, 4, 6, 5, 7, 9, 3, 1]

Sortowanie ostatnich 9 elementów (Sort(2,10)):

 a = [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Sortowanie pierwszych 9 elementów (Sort(1,9)):

 a = [1, 2, 3, 4, 5, 6, 7, 8, 10, 9]

Sortowanie ostatnich 9 elementów (Sort(2,10)):

 a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

Zapisz algorytm sortowania tablicy 1000-elementowej w co najwyżej 6 krokach, przy założeniu, że w jednym kroku można posortować blok złożony z co najwyżej 500 elementów

Rozwiązanie:

Kolejne kroki algorytmu:

1. Sort(1,500)

2. Sort(251,750)

3. Sort(501,1000)

4. Sort(1,500)

5. Sort(251,750)

6. Sort(1,500)

Komentarz

To zadanie sprawdza rozumienie zasad działania podstawowych algorytmów sortowania. Wszystkie trzy algorytmy wymienione w punkcie a), to algorytmy sortujące polegające na porównywaniu wartości elementów znajdujących się na różnych pozycjach w tablicy

a i zamianie ich miejscami, jeśli element o większej wartości poprzedza w tablicy element o mniejszej wartości. W algorytmach sortowania, bąbelkowym i przez wstawianie liniowe, porównywane i zamieniane są tylko pary sąsiednich elementów w tablicy. Tak nie jest w algorytmie szybkim, ponieważ w przeciwnym razie stracilibyśmy walor „szybkości”.

Żeby rozwiązać punkt b) zastanówmy się, ile zamian wykonamy sortując algorytmem przez wstawianie tablicę a[1..n] o zadanej zawartości. Oznaczmy przez b[i] liczbę elementów

w tablicy a znajdujących się na pozycjach o indeksach mniejszych od i, ale o wartościach większych od wartości elementu a[i]. Nietrudno zauważyć, że podczas wstawiania elementu a[i] do uporządkowanego już fragmentu tablicy a[1..i-1], zostanie on zamieniony ze wszystkimi elementami o wartościach większych, a jest ich dokładnie b[i]. Zatem łączna liczba zamian wykonywanych w algorytmie sortowania przez wstawianie tablicy a wyniesie b[1]+b[2]+ … +b[n]. Poniżej pokazujemy zawartość tablicy b dla przykładów z punktu b).

Tablica 1.

pozycja Zawartość tablicab

1 10 0

2 1 1

3 2 1

4 3 1

5 4 1

6 5 1

7 6 1

8 9 1

9 7 2

10 8 2

Tablica 2.

pozycja Zawartość tablicab

1 1 0

2 2 0

3 3 0

4 5 0

5 4 1

6 10 0

7 9 1

8 8 2

9 7 3

10 6 4

Do rozwiązania punktu c) możemy zaadoptować algorytm sortowania bąbelkowego. Gdyby tablica a liczyła tylko cztery elementy, to do jej posortowania wystarczy (i potrzeba) 6 wywołań procedury Sort:

Sort(1,2), Sort(2,3), Sort(3,4), Sort(1,2), Sort(2,3), Sort(1,2).

Po trzech pierwszych wywołaniach Sort element o największej wartości znajdzie się już na swojej docelowej pozycji nr 4. Dwa następne wywołania zagwarantują, że na pozycji nr 3 w a znajdzie się element drugi, licząc od największego. Ostatnie wywołanie Sort porządkuje dwa najmniejsze elementy i ustawia je na właściwych, dwóch pierwszych pozycjach w tablicy a. Podobnie dzieje się w zaproponowanym rozwiązaniu sortowania tablicy 1000 elementowej. Pierwsze trzy wywołania Sort gwarantują umieszczenie 250 największych elementów na ich docelowych pozycjach. Kolejne dwa wywołania Sort umieszczają w dobrym porządku, na docelowych pozycjach od 501 do 750, kolejne 250 elementów. Ostatnie sortowanie porządkuje pierwszych 500 najmniejszych elementów. Alternatywne rozwiązanie mogłoby polegać na zaadaptowaniu sortowania przez wstawianie, kiedy to wstawiamy bloki po 250 elementów:

Sort(1,500), Sort(251,750), Sort(1,500), Sort(501,1000), Sort(251,750), Sort(1,500).

Zadanie 15. Miejsce zerowe (0–6)

Przedstawiona poniżej rekurencyjna funkcja Mzer znajduje metodą bisekcji miejsce zerowe funkcji f ciągłej w przedziale <a,b>, z dokładnością do  [epsilon].

Specyfikacja

Dane: liczby a i b takie, że a<b oraz f(a)\*f(b)<0; liczba ɛ>0

Wynik: liczba rzeczywista x z przedziału <a,b> taka, że f(x+á)=0 dla pewnego á takiego, że - ɛ/2<=á<=ɛ/2

Funkcja Mzer(a,b,ɛ)

1. jeżeli b-a>ɛ, to wykonaj:
	1. s przypisz (a+b)/2
	2. jeżeli f(s)=0, to podaj s jako wynik
	3. jeżeli f(a)\*f(s)<0, to podaj Mzer(a,s,ɛ) jako wynik

w przeciwnym przypadku podaj Mzer(s,b,ɛ) jako wynik

1. w przeciwnym przypadku podaj (a+b)/2 jako wynik
2. Na rysunku prezentujemy fragment wykresu funkcji f, dla której wywołujemy funkcję Mzer:

Wykres przecina oś OX w punkcie 72,7. Załóżmy, że funkcja Mzer została wywołana dla a=0 i b=128. Zapisz liczbę kolejnych rekurencyjnych wywołań funkcji Mzer przy podanych początkowych wartościach ɛ równych odpowiednio: 10; 32; 25; 5; 1/5, wiedząc, że liczba wywołań Mzer dla ɛ=10 wynosi 4.

1. Poniżej prezentujemy zapis algorytmu opisanego funkcją Mzer w postaci nierekurencyjnej. Zapis poniższego algorytmu jest niepełny, zapisz poniższy algorytm uzupełniając brakujące elementy tak, aby realizował tę samą metodę poszukiwania miejsca zerowego, którą opisuje funkcja Mzer.

Algorytm:

1. dopóki b-a>ɛ wykonuj
2. s przypisz ..............................
3. jeżeli f(s)=0, to podaj s jako wynik i zakończ wykonywanie algorytmu
4. jeżeli f(a)\*f(s)<0, to b przypisz................................

w przeciwnym przypadku ....................................

1. podaj (a+b)/2 jako wynik

Schemat punktowania:

Łączna liczba punktów za zadanie - 6 punktów. W tym:

za podpunkt a - za wszystkie poprawne odpowiedzi – 3 punkty, za trzy poprawne odpowiedzi – 2 punkty, za dwie poprawne odpowiedzi – 1 punkt.

za podpunkt b - za każde poprawne zapisanie brakującego elementu – 1 punkt, łączna liczba punktów - 3 punkty.

Zadanie 15. Miejsce zerowe (0–6) – rozwiązanie

Przedstawiona poniżej rekurencyjna funkcja Mzer znajduje metodą bisekcji miejsce zerowe funkcji f ciągłej w przedziale <a,b>, z dokładnością do  [epsilon].

Specyfikacja

Dane: liczby a i b takie, że a<b oraz f(a)\*f(b)<0; liczba ɛ>0

Wynik: liczba rzeczywista x z przedziału <a,b> taka, że f(x+á)=0 dla pewnego á takiego, że -ɛ/2<=á<=ɛ/2

Funkcja Mzer(a,b,ɛ)

1. jeżeli b-a>ɛ, to wykonaj:
	1. s przypisz (a+b)/2
	2. jeżeli f(s)=0, to podaj s jako wynik
	3. jeżeli f(a)\*f(s)<0, to podaj Mzer(a,s,ɛ) jako wynik

w przeciwnym przypadku podaj Mzer(s,b,ɛ) jako wynik

1. w przeciwnym przypadku podaj (a+b)/2 jako wynik
2. Na rysunku prezentujemy fragment wykresu funkcji f, dla której wywołujemy funkcję Mzer:

Wykres przecina oś OX w punkcie 72,7. Załóżmy, że funkcja Mzer została wywołana dla a=0 i b=128. Zapisz liczbę kolejnych rekurencyjnych wywołań funkcji Mzer przy podanych początkowych wartościach ɛ równych odpowiednio: 10; 32; 25; 5; 1/5, wiedząc, że liczba wywołań Mzer dla ɛ=10 wynosi 4.

Odpowiedź - podpunkt a:

Liczba wywołań Mzer dla ɛ=32 wynosi 2, dla ɛ=25 wynosi 3, dla ɛ=5 wynosi 5, dla ɛ=1/5 wynosi 10.

1. Poniżej prezentujemy zapis algorytmu opisanego funkcją Mzer w postaci nierekurencyjnej. Zapis poniższego algorytmu jest niepełny, zapisz poniższy algorytm uzupełniając brakujące elementy tak, aby realizował tę samą metodę poszukiwania miejsca zerowego, którą opisuje funkcja Mzer.

Algorytm:

1. dopóki b-a>ɛ wykonuj
2. s przypisz (a+b)/2
3. jeżeli f(s)=0, to podaj s jako wynik i zakończ wykonywanie algorytmu
4. jeżeli f(a)\*f(s)<0, to b przypisz s

w przeciwnym przypadku a przypisz s

1. podaj (a+b)/2 jako wynik

Komentarz

Podpunkt a

Aby obliczyć liczbę kolejnych rekurencyjnych wywołań funkcji Mzer dla każdej podanej wartości ɛ, należy ustalić wartości zmiennych z jakimi będzie ona wywoływana w kolejnych krokach. Funkcja zakończy działanie w momencie, gdy b-a<ɛ .

Dla każdej dokładności startujemy od wywołania funkcji Mzer(0,128,ɛ), zaś kolejne wywołania to:

Dla ɛ=32 mamy dwa wywołania: Mzer(64,128,32), Mzer(64,96,32)

Dla ɛ=25 mamy trzy wywołania: Mzer(64,128,25), Mzer(64,96,25), Mzer(64,80,25)

Dla ɛ=5 mamy pięć wywołań: Mzer(64,128,5), Mzer(64,96,5), Mzer(64,80,5), Mzer(72,80,5), Mzer(72,76,5).

Dla ɛ=5 mamy dziesięć wywołań: Mzer(64,128,1/5), Mzer(64,96,1/5), Mzer(64,80,1/5), Mzer(72,80,1/5), Mzer(72,76,1/5), Mzer(72,74,1/5), Mzer(72,73,1/5), Mzer(72,5;73;1/5), Mzer(72,5;72,75;1/5), Mzer(72,625;72,75;1/5).

Podpunkt b

Aby zapisać nierekurencyjną wersję funkcji Mzer, należy zauważyć prostą własność: w kolejnych krokach zawsze ustalamy środek aktualnego przedziału, tzn. s przypisz (a+b)/2, a następnie sprawdzamy czy miejsce zerowe funkcji leży na lewo od punktu s (wtedy b przypisz s), czy na prawo od punktu s(a przypisz s).

Podpunkt a) sprowadza się do przeanalizowania algorytmu zaprezentowanego w treści zadania na konkretnych danych i wyznaczenia liczby wywołań funkcji rekurencyjnej. Funkcja Mzer opisuje podręcznikowy algorytm znajdowania miejsca zerowego funkcji ciągłej f metodą bisekcji. Zadaniem zaprezentowanej implementacji tego algorytmu jest podanie miejsca zerowego funkcji w przedziale [a,b], z dokładnością do ɛ/2 przy założeniu, że f(a)\*f(b)<0 (zauważmy, że ciągłość funkcji f w powiązaniu z warunkiem f(a)\*f(b)<0 gwarantuje, że f ma miejsce zerowe w przedziale [a,b]).

W treści zadania przedstawiono pseudokod rekurencyjnej wersji algorytmu, która rozpoczyna się od sprawdzenia czy odległość między krańcami przedziału [a,b] jest większa od ε (czyli podwojonej dokładności wyniku):

1. Gdy b – a ≤ε, wówczas mamy gwarancję że środek przedziału [a,b] znajduje się nie dalej niż ɛ/2 od miejsca zerowego funkcji f. Dlatego też jako wynik zwracany jest właśnie środek przedziału, czyli wartość (a+b)/2.
2. Gdy b – a > ε, mamy dwie możliwości. Jeśli środek przedziału [a,b] jest miejscem zerowym funkcji f, zwracamy go oczywiście jako wartość. W przeciwnym razie redukujemy zadanie wyszukania miejsca zerowego w [a,b] do zadania poszukiwania miejsca zerowego w [a,s] lub [s,b], gdzie s to środek przedziału [a,b]. Efektem redukcji jest więc dwukrotne zmniejszenie długości przedziału, w którym poszukujemy miejsca zerowego.

Zauważmy, że gdyby wywołania funkcji Mzer dla a=0 i b=128 oraz podanych w punkcie a) wartości ε nigdy nie kończyły się znalezieniem dokładnej wartości miejsca zerowego (czyli spełnieniem warunku f(s)=0), odpowiedzi w punkcie a) sprowadzałyby się do wskazania ilokrotnie trzeba „połowić” przedział o długości 128 aby uzyskać przedział o długości nie większej niż ε. W rozwiązaniu można by więc ograniczyć się do policzenia, ile takich „połowień” należy wykonać. Aby jednak mieć pewność poprawności rozwiązania, trzeba sprawdzić że wartość środka przedziału s rzeczywiście w naszym przykładzie nie „trafi” idealnie w miejsce zerowe (równe 72,7) w kolejnych wywołaniach rekurencyjnych. Taką skrupulatną analizę przedstawiliśmy omawiając rozwiązanie punktu a).

Punkt b) zadania wymaga zastosowania standardowej techniki zamiany rekurencji na iterację. Zamiast wywoływać funkcję z nowymi wartościami krańców przedziałów, zmieniamy w pętli wartości zmiennych a i b tak, aby odpowiadały one końcom coraz mniejszych przedziałów dla których wywoływana jest funkcja rekurencyjna Mzer. Przedstawiony szkielet algorytmu z pozostawionymi miejscami do uzupełnienia sugeruje sposób, w jaki w tym przypadku należy dokonać zamiany rekurencji na iterację*.*

Zadanie 16. Podzielność (0–10)

W trzech plikach tekstowych liczby1.txt, liczby2.txt i liczby3.txt zapisano po 1000 dodatnich liczb binarnych. W każdym pliku liczby zapisano w kolejnych wierszach po jednej liczbie w wierszu. W pliku liczby1.txt długość zapisu każdej z liczb jest nie większa od 12. W pliku liczby2.txt długość zapisu każdej z liczb jest nie większa od 30, zaś w pliku liczby3.txt długość zapisu każdej liczby nie przekracza 200.

Dla każdego z plików z danymi wyznacz, ile zawiera on

– liczb podzielnych przez 2,

– liczb podzielnych przez 3,

– liczb podzielnych przez 5.

Przykład:

W pliku z 3 liczbami binarnymi:

10101

1100

1110

są 2 liczby podzielne przez 2, 1 liczba podzielna przez 3 i 1 liczba podzielna przez 5.

Do oceny oddajesz plik tekstowy podzielnosc.txt zawierający w dziewięciu kolejnych wierszach dziewięć liczb, po jednej w wierszu (pierwsze trzy wiersze powinny zawierać liczby liczb z pliku liczby1.txt podzielnych odpowiednio przez 2, 3 i 5; kolejne trzy wiersze powinny zawierać liczby liczb z pliku liczby2.txt podzielnych odpowiednio przez 2, 3 i 5, a ostatnie trzy wiersze liczby liczb z pliku liczby3.txt podzielnych odpowiednio przez 2, 3 i 5) oraz pliki (zapisz nazwy tych plików), które zawierają komputerową realizację Twoich obliczeń.

Schemat punktowania

Łączna liczba punktów za zadanie - 9 punktów, w tym za poszczególne czynności:

Za poprawne wyniki dla pliku 1 – 2 punkty.

Za poprawne wyniki dla pliku 2 – 3 punkty.

Za poprawne wyniki dla pliku 3 – 4 punkty.

Zadanie 16. Podzielność (0–10) - rozwiązanie

Pliki z danymi, plik programu źródłowego oraz plik podzielnosc.txt zawierający odpowiedzi znajdują się w folderze PODZIELNOSC.

Komentarz

Jedna z metod rozwiązania tego zadania mogła by polegać na przytoczeniu i wykorzystaniu własności podzielności liczb binarnych. O ile własność podzielności przez 2 jest oczywista i powszechnie znana – najmniej znaczącą cyfrą takiej liczby musi być 0 – to już własności podzielności liczby binarnych przez inne liczby nie są tak naturalne i znane. Dla każdego dzielnika taka własność byłaby pewnie inna i prowadziłaby do algorytmu właściwego tylko dla tej własności. Przedstawimy algorytm, który jest jednakowy dla wszystkich dzielników z dokładnością do parametru, którym jest właśnie dzielnik.

Jest naturalne rozwiązanie, które pozwala stwierdzić, czy dana, dodatnia liczba binarna b, jest podzielna całkowicie przez dodatnią (dziesiętną) liczbę całkowitą p. Wystarczy obliczyć (dziesiętną) wartość d liczby b i sprawdzić, czy p dzieli całkowicie d. W tym celu wykorzystujemy dostępny w większości języków programowania operator mod obliczania reszty z dzielenia liczb całkowitych i sprawdzamy, czy d mod p = 0. Do zamiany liczby binarnej b na jej odpowiednik dziesiętny d można zastosować schemat Hornera. Prawda jakie to proste? Jest tylko jeden mały problem. W kolejnych plikach z danymi są coraz większe liczby. W pliku liczby1.txt długość zapisu każdej z liczb jest nie większa niż 12 i do zapisu każdej takiej liczby wystarczają dwa bajty, do zapisu jednej liczby z pliku liczby2.txt potrzeba już 4 bajtów, natomiast do zapisu liczby z pliku liczby3.txt może być potrzebnych 25 bajtów na każdą z nich. Jeśli język programowania użyty do rozwiązania pozwala reprezentować tak duże liczby i umożliwia wykonywanie na nich podstawowych arytmetycznych – dodawanie, mnożenie i branie modulo – to powyżej opisane rozwiązanie jest wystarczające. Często jednak języki programowania nie umożliwiają bezpośredniego operowania na bardzo dużych liczbach. Wówczas pozostaje albo zaprogramować własną arytmetykę dużych liczb, albo, jak w tym przypadku, skorzystać z pewnych własności używanych operacji arytmetycznych. Tutaj skorzystamy z własności operacji modulo brania reszty z dzielenia, które to własności pozwalają nam operować tylko na resztach z dzielenia przez dzielnik p, a nie na całych liczbach. Każda taka reszta jest nie większa od p. Dwie podstawowe, wykorzystywane przez nas własności są następujące:

Dla dodatnich liczb całkowitych a, b i p mamy:

1. (a + b) mod p =( a mod p + b mod p) mod p
2. (a\*b) mod p = ((a mod p) \* (b mod p)) mod p

Tak więc do policzenia reszty z dzielenia przez p dodatniej liczby całkowitej d, której wartość jest taka sama jak wartość liczby binarnej b, należy po prostu zastosować schemat Hornera obliczania wartości d z zapisu b pamiętając, żeby wszystkie obliczenia wykonywać modulo p. Oto funkcja zapisana w języku programowania C++, która dla liczby binarnej b podanej w postaci zero-jedynkowego napisu i dodatniego, całkowitego podzielnika p, oblicza resztę z dzielenia wartości dziesiętnej liczby b przez p.

int Reszta(string b, int p){

const char zero='0';

 int dl = b.length(); /\*obliczenie długości zapisu liczby b\*/

 int d = 0; /\* po zakończeniu obliczeń wartością d będzie wartość dziesiętna liczby b modulo p \*/

/\* schemat Hornera modulo p \*/

for (int j = 0; j<dl; j++){

 int cyfra = b[j] - zero;/\* odzyskanie kolejnej cyfry liczby b poczynając od najbardziej znaczącej \*/

 d=(d\*2 + cyfra) % p; /\* % jest operatorem modulo w C++ \*/

}

return d;

}

Należy zwrócić jeszcze uwagę na drobiazg, jakim jest odzyskiwanie wartości liczbowych kolejnych cyfr zapisu liczby b. Liczba b jest zadana jako napis złożony ze znaków ‘0’ i ‘1’. Każdy ze znaków ‘0’ lub ‘1’ do obliczeń arytmetycznych należy zamienić odpowiednio na liczbę 0 lub 1. Do tego celu najprościej wykorzystać fakt, że cyfry ‘0’, ‘1’, …, ‘9’ są kodowane kolejnymi liczbami naturalnymi poczynając od kodu znaku ‘0’. Tak więc, gdy od kodu cyfry odejmiemy kod cyfry ‘0’, to otrzymamy liczbę odpowiadającą tej cyfrze. W przedstawionym powyżej rozwiązaniu wykorzystujemy ten fakt – wartość wyrażenia b[j] - zero jest obliczana z wykorzystaniem kodów cyfr b[j] i zero.

Poniżej przedstawiamy program Podzielnosc zapisany w języku C++, który oblicza żądane wyniki dla jednego pliku. Dla przykładu, żeby uzyskać wyniki dla pliku liczby1.txt można program wykonywalny Podzielnosc uruchomić w katalogu zawierającym plik liczby1.txt następującą komendą z wiersza poleceń:

Podzielnosc < liczby1.txt > wyniki1.txt

Wyniki znajdą się w pliku tekstowym wyniki1.txt w tym samym katalogu. Na koniec trzeba pamiętać, żeby wyniki obliczeń dla wszystkich trzech plików umieścić w jednym pliku tekstowym podzielnosc.txt zgodnie z opisem w treści zadania.

/\* Program Podzielnosc \*/

#include <iostream>

using namespace std;

int Reszta(string b, int p){

const char zero = '0';

 int dl = b.length(); /\* obliczenie długości zapisu liczby b \*/

 int d = 0; /\* po zakończeniu obliczeń wartością d będzie wartość dziesiętna liczby b modulo p \*/

/\* schemat Hornera modulo p \*/

for (int j = 0; j < dl; j++){

 int cyfra = b[j] - zero; /\* odzyskanie kolejnej cyfry liczby b poczynając od najbardziej znaczącej \*/

 d = (d\*2 + cyfra) % p; /\* % jest operatorem modulo w C++ \*/

}

return d;

}

int main(){

string liczba;

const int n = 1000; /\* rozmiar danych \*/

int podz\_2 = 0, podz\_3 = 0, podz\_5 = 0; /\* podz\_p - liczba liczb podzielnych przez p \*/

for (int i = 0; i < n; i++){

 cin >> liczba; /\* wczytanie kolejnej liczby; \*/

 if (Reszta(liczba,2) == 0) podz\_2++;

 if (Reszta(liczba,3) == 0) podz\_3++;

 if (Reszta(liczba,5) == 0) podz\_5++;

}

cout << "podzielne przez " << 2 << " : " << podz\_2 << "\n";

cout << "podzielne przez " << 3 << " : " << podz\_3 << "\n";

cout << "podzielne przez " << 5 << " : " << podz\_5 << "\n";

return 0;

}

Zadanie 17. Bloki trójkowe (0–12)

Niech n będzie dodatnią liczbą całkowitą i niech a1, a2, …, an będzie ciągiem nieujemnych liczb całkowitych. Dla pary liczb i, j takich, że , blokiem b(i,j) nazywamy podciąg kolejnych elementów ciągu a z pozycji od i do j, czyli ai, ai+1, …, aj. Długością bloku nazywamy liczbę jego elementów. O bloku, którego suma elementów jest podzielna przez 3 mówimy, że jest blokiem trójkowym.

Przykład:

W ciągu 0,0,2,3,2,1,2 najdłuższym blokiem trójkowym jest b(4,6) = 3,2,1.

W plikach tekstowych bloki1.txt, bloki2.txt i bloki3.txt zapisano ciągi odpowiednio 1000, 30000 i 1000000 nieujemnych liczb całkowitych mniejszych od 10 000. W każdym pliku liczby zapisano w kolejnych wierszach, po jednej liczbie w każdym wierszu.

Dla każdego pliku z danymi wyznacz długość najdłuższego bloku trójkowego w ciągu zapisanym w tym pliku.

Przykład:

Dla danych z pliku z 7 liczbami:

0

0

2

3

2

1

2

długość najdłuższego bloku trójkowego wynosi 3.

Do oceny oddajesz plik(i) pliki tekstowe wyniki1.txt, wyniki2.txt, wyniki3.txt, gdzie każdy z nich zawiera liczbę równą długości najdłuższego bloku trójkowego w ciągach zapisanych odpowiednio w plikach bloki1.txt, bloki2.txt i bloki3.txt oraz pliki (zapisz nazwy tych plików), które zawierają komputerową realizację Twoich obliczeń.

Schemat punktowania:

Liczba punktów za zadanie - 12 pkt

Za poprawne wyniki dla pliku 1 – 3 punkty.

Za poprawne wyniki dla pliku 2 – 4 punkty.

Za poprawne wyniki dla pliku 3 – 5 punktów.

Zadanie 17. Bloki trójkowe (0–12) – rozwiązanie

Pliki z danymi, plik programu źródłowego oraz pliki wynikowe zawierające odpowiedzi znajdują się w folderze BLOKI TROJKOWE.

Komentarz

Uważny czytelnik szybko zauważy, że proste rozwiązanie jest ukryte w treści zadania. Wystarczy przejrzeć wszystkie bloki, dla każdego bloku zsumować jego elementy, sprawdzić, czy otrzymana suma jest podzielna przez 3 i spośród wszystkich bloków spełniających ten warunek wybrać najdłuższy. Oto fragment algorytmu zapisany w pseudo-języku C++, będący realizacją tego prostego pomysłu:

najdluzszy = 0; /\* długość najdłuższego z dotychczas przejrzanych bloków trójkowych\*/

for (i = 1; i <= n; i++)

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

 /\* obliczamy sumę elementów w bloku b(i,j) \*/

 suma = 0;

 for (k = i; k <= j; k++)

 suma = suma + ak; // (#)

 /\* sprawdzamy, czy suma jest podzielna przez 3, czyli czy reszta z dzielenia suma przez 3 daje 0 \*/

 if (suma % 3 == 0){

 /\* jeśli suma jest podzielna przez 3 i blok b(i,j) \*/

 /\* jest dłuższy od dotychczas najdłuższego bloku \*/

 /\* trójkowego, to zapamiętujemy jego długość \*/

 dl = j – i + 1;

 if (dl >= najdluzszy)

 najdluzszy = dl;

 }

 }

/\* najdluzszy jest długością najdłuższego bloku trójkowego w ciągu a \*/

Jedyną dobrą cechę powyższego rozwiązania jest jego prostota i to, że w ogóle mamy jakiekolwiek rozwiązanie. Poza tym przedstawione rozwiązanie ma same wady. Po pierwsze musimy pamiętać o tym, żeby arytmetyka języka programowania, którego używamy, umożliwiała operowanie na liczbach pojawiających się w obliczeniach. Zauważmy, że największa liczba jaka mogłaby się w obliczeniach dla opisanych danych wynosi 1 000 000 \* 9 999 = 9 999 000 000. W języku programowania C++ największa liczba typu int ze znakiem ma wartość 2 147 483 647, a bez znaku 4 294 967 295. Tak jest, gdy typ int jest 32-bitowy. Dla 16-bitowego typu int odpowiednie wielkości są znacząco mniejsze. Oczywiście można by użyć typów pozwalających na operowanie na dużo większych liczbach, ale za chwilę okaże się, że nie jest to konieczne. Drugą wadą powyższego rozwiązania jest to, że musimy wielokrotnie przeglądać te same elementu ciągu, a co za tym idzie najlepiej byłoby cały ciąg wczytać do tablicy. To w przypadku największego pliku wymaga tablicy o 1 000 000 elementów. W przypadku języka programowania C++ i 32-bitowego typu int wymaga to 4 000 000 bajtów, czyli około 4 gigabajtów pamięci. Nie jest to może dużo dla współczesnych komputerów, ale co gdyby dane liczyły nie milion, a miliard, bilion elementów? Trzecią wadą, chyba najistotniejszą, jest powolność zaproponowanego algorytmu. Zastanówmy się, ile łącznie dodawań wykonamy w kroku (#). Nietrudno zauważyć, że tych dodawań jest tyle, ile wynosi łączna suma długości wszystkich przedziałów. Tę wielkość łatwo oszacować z dołu. Przedziałów o długości co najmniej n/3 jest co najmniej (n/3)2. A zatem łączna suma wszystkich przedziałów wynosi co najmniej (n/3)3. To dla n = 1 000 000 daje więcej niż 3\*1016. Komputer wykonujący 109 dodawań na sekundę spędzałby na rozwiązywaniu naszego zadania więcej niż 3\*107 sekund, a to jest więcej niż 8 tysięcy godzin. To jest trochę za długo jak na czas przeznaczony na rozwiązywanie zadań maturalnych!

Jeden ze sposobów poszukiwania lepszych, szybszych algorytmów polega na przyjrzeniu się rozwiązaniu, które już mamy w ręku i zastanowieniu się, czy nie prowadzi ono do wykonywania wielu zbędnych operacji. Tak jest właśnie w tym przypadku. Zauważmy, że dla każdych dwóch bloków b(i,j-1) i b(i,j), różniących się tylko jednym elementem aj, liczymy niezależnie dwie sumy ai+ ai+1 + ... + aj-1 oraz ai+ ai+1 + ... + aj, a przecież, żeby dostać drugą sumę wystarczy do pierwszej dodać tylko aj. Jaki zysk! Zamiast j-i+1 dodawań wykonujemy tylko jedno. Ten pomysł pozwala nam natychmiast zaproponować następujący algorytm:

najdluzszy = 0; /\*długość najdłuższego z dotychczas przejrzanych bloków trójkowych \*/

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

 /\* liczymy sumy elementów w blokach o początkach na pozycji i wartością suma będzie suma elementów ostatniego przetworzonego bloku; inicjalnie suma == 0 \*/

 suma = 0;

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

 /\* obliczamy sumę elementów w bloku b(i,j) suma jest równa sumie elementów w bloku b(i,j-1) plus aj \*/

 suma = suma + aj; // (#)

 /\* sprawdzamy, czy suma jest podzielna przez 3, czyli czy reszta z dzielenia suma przez 3 daje 0 \*/

 if (suma % 3 == 0){

 /\* jeśli jest podzielna przez 3 i blok b(i,j) jest dłuższy od dotychczas najdłuższego bloku trójkowego, to zapamiętujemy jego długość \*/

 dl = j – i + 1;

 if (dl >= najdluzszy)

 najdluzszy = dl;

 }

 }

/\* najdluzszy jest długością najdłuższego bloku trójkowego w ciągu a \*/

Ile tym razem wykonujemy dodawań (#)? Nietrudno zauważyć, że tyle, ile jest bloków. Żeby policzyć sumę elementów w bloku b(i,j), wykonujemy tylko jedno dodawanie – do sumy elementów z bloku b(i,j-1) dodajemy aj. Ile jest wszystkich bloków? Bloków o początku na pozycji 1 jest n, bloków o początku na pozycji 2 jest n-1, bloków o początku na pozycji 3 jest n-2, itd. Tak więc bloków o początku na pozycji i jest n-i+1. Wszystkich bloków jest
n + n-1 + ... + 1 = n(n-1)/2. Dla n = 1 000 000 ta wartość wynosi 499 999 500 000. A zatem komputer wykonujący 109 dodawań na sekundę wykonałby nasze zadanie w około 500 sekund, czyli w około 7 minut. Natomiast dla ciągu o długości 30 000 odpowiedź dostalibyśmy natychmiast.

W przypadku rozpatrywanego zadania myślenie algorytmiczne może dać jeszcze lepsze efekty. Najpierw pozbądźmy się problemu dużych liczb. To łatwe. W wierszu (#) wystarczy sumować modulo 3. Inaczej mówiąc, w zmiennej suma zamiast sumy elementów pamiętamy resztę z dzielenia tej sumy przy dzieleniu przez 3. Więcej o wykonywaniu operacji arytmetycznych modulo napisaliśmy w komentarzu do zadania Podzielność. Przy tym podejściu wiersz (#) miałby postać: suma = (suma + aj) % 3;// (#). Natomiast instrukcja warunkowa if (suma % 3 == 0) przybrałaby postać: if (suma == 0). W ten sposób poradziliśmy sobie z problemem dużych liczb, ale czasowa złożoność obliczeniowa naszego algorytmu pozostała bez zmian. Następujące spostrzeżenia pozwolą przyśpieszyć poszukiwanie najdłuższego bloku trójkowego. Dla każdego k = 1, 2, …, n oznaczmy przez sk sumę pierwszych k elementów w ciągu a. Innymi słowy sk jest sumą elementów w bloku b(1,k). Dla wygody przyjmijmy, że mamy też element a0 = 0 i w naturalny sposób weźmy s0 = 0. Zauważmy teraz, że suma elementów w bloku b(i,j), 1 ≤ i≤ j ≤n , jest równa sj – si-1. Dla naszych celów wartości s wystarczy liczyć modulo 3. W takim przypadku wartością sk może być tylko 0, 1 lub 2. Teraz najważniejsze: Blok b(i,j) jest blokiem trójkowym wtedy i tylko, gdy sj oraz si-1 mają taką samą wartość 0, 1, lub 2. Powyższe spostrzeżenie daje bardzo proste rozwiązanie naszego zadania. Dla każdej wartości w = 0, 1, 2 poszukujemy pierwszej i ostatniej pozycji, dla której wartości sum s są takie same. Najbardziej odległe pozycje wyznaczają długość najdłuższego bloku trójkowego. Pozostaje jeszcze pytanie, czy zawsze taki blok istnieje. Osobom o zainteresowaniach bardziej matematycznych proponujemy wykazanie, że w każdym ciągu o długości co najmniej 3 taki blok musi istnieć. Dla naszych potrzeb przyjmijmy, że blok długości 0 jest blokiem trójkowym. Wówczas, jeśli wynikiem działania naszego algorytmu jest 0, oznacza to, że żaden blok w danym ciągu nie jest trójkowy. Poniżej przedstawiamy program napisany w języku C++, który konkretyzuje opisane powyżej idee.

#include <iostream>

#include <algorithm>

using namespace std;

int najdluzszy\_blok(){

int poczatki = {0,-1,-1};

/\*poczatki[w] - pierwsza pozycja, dla której suma s jest równa w -1 oznacza, że takiej pozycjijeszcze nie znaleziono \*/

int suma\_mod\_3 = 0;

int liczba\_wczytanych = 0;

int element;

int najdluzszy;

while (cin>> element){

liczba\_wczytanych++;

suma\_mod\_3 = (suma\_mod\_3 + element) % 3; //(#)

if (poczatki[suma\_mod\_3] == -1)

 poczatki[suma\_mod\_3] = liczba\_wczytanych;

else

 najdluzszy = max(najdluzszy, liczba\_wczytanych

– poczatki[suma\_mod\_3])

}

return najdluzszy;

}

int main(){

cout << "Dl bloku: " << najdluzszy\_blok() << endl;

return 0;

}

Na koniec zauważmy, że w tym algorytmie liczba dodawań (#) wynosi tylko n i na dodatek nie musieliśmy najpierw wczytać całego ciągu do tablicy.