Poszukiwanie najlepszej sumy w ciągu

Najlepsza suma – specyfikacja:

Dane: $\mathrm{int\ a[n]}$

Wynik: $\mathrm{najlepsza} = \max(\{0\} \cup \{\mathrm{s[i,j]}: 1 \le \mathrm{i} \le \mathrm{j} \le \mathrm{n}\})$,

gdzie: $\mathrm{s[i,j]} = \mathrm{a[i]} + \mathrm{a[i+1]} + \cdots + \mathrm{a[j]}$.

Przykład
$\mathrm{a} = \{2, 3, -2, 3, -5, -1, 2, -1, 3, -2, 4, -1, 2, -1\}$
$\mathrm{najlepsza} = 7$ (począwszy od elementu szóstego do trzynastego – indeksujemy od zera)

Łatwo zauważyć, że gdy wszystkie liczby są dodatnie, rozwiązanie jest trywialne, bo najlepszym przedziałem jest wówczas cała tablica, a najlepszą sumą jest suma wszystkich elementów tablicy. Zawsze należy starać się znaleźć pierwsze (także nieoptymalne), ale poprawne (działające) rozwiązanie, bo wtedy można je doskonalić.

Rozwiązanie pierwsze wynika wprost ze specyfikacji, tj. należy przejść po wszystkich możliwych przedziałach i spośród wszystkich sum w tych przedziałach należy wybrać najlepszą.

Rozwiązanie 1.
Korzystając z opisu zadania obliczamy kolejne sumy z przedziałów $\lfloor \mathrm{i}, \mathrm{j} \rfloor$ i wybieramy największą z nich (lub 0, gdy zbiór jest pusty).

Algorytm 1.

```
najlepsza = 0; //najlepsza suma pustego przedziału
for(i = 0; i < n; i++) // ustawienie początku przedziału
   for(j = i; j < n; j++) // ustawienie końca przedziału
      {
       suma = 0; // inicjalizacja bieżącej sumy
       for(k = i; k <= j; k++) // pętla sumująca w zadanym przedziale
          suma = suma + a[k];   // sumowanie liczb w przedziale
                                // jest to operacja dominująca
       if(suma > najlepsza) // jeśli znajdziemy lepszą sumę
          najlepsza = suma; // to przypisujemy ją do najlepszej
      }
```

Operacją dominującą w tym algorytmie jest ustalanie sumy w poszczególnych przedziałach. Kolejnym krokiem jest więc ustalenie liczby dodawań w całym algorytmie. Mamy tutaj n dodawań związanych z sumowaniem całej tablicy (zmienna suma zainicjowana jest na 0), dwa dodawania $(n-1)$-elementowe, trzy dodawania $(n-2)$-elementowe i tak dalej, a na końcu mamy jeszcze n dodawań jednoelementowych (nawet jeśli zbiór jest jednoelementowy, to i tak trzeba wykonać jedno dodawanie, bo zmienna suma zainicjowana jest na 0). Inicjowanie zmiennej suma w tym algorytmie na 0 nie jest błędem, bo i tak nie ma to praktycznie żadnego wpływu na rząd wielkości realizowanej funkcji.

Oznaczmy liczbę operacji dominujących jako $T_1(n)$, która w rzeczywistości jest sumą długości wszystkich przedziałów:

$$\mathrm{T_1(n)} = \mathrm{n} + 2(\mathrm{n}-1) + 3(\mathrm{n}-2) + \cdots + \mathrm{n} = \frac{\mathrm{n(n+1)(n+2)}}{6}$$

Zatem przedstawiony algorytm działa w czasie n^3, czyli sześciennym, a zatem jest bardzo czasochłonny oraz zasobochłonny. Pora więc zastanowić się nad jego ulepszeniem.

Możemy podać rekurencyjny wzór na $\mathrm{T_1(n)}$, dlatego że zbiór wszystkich przedziałów możemy podzielić na przedziały, które kończą się na pozycji co najwyżej przedostatniej plus przedziały, które kończą się na pozycji ostatniej. Jeśli tak będziemy rozumowali, to

$$\mathrm{T_1(n)} = \mathrm{T_1(n-1)} + \textit{suma długości przedziałów, które kończą się na pozycji ostatniej}$$

$T_1(n-1)$ – suma długości przedziałów, które kończą się na pozycji co najwyżej przedostatniej

Suma długości przedziałów, które kończą się na pozycji ostatniej, to przedział długości n, przedział długości $(n-1)$, przedział długości $(n-2)$ i na końcu przedział długości jeden. Jeśli to zsumujemy, to otrzymamy wzór

$$\frac{n(n+1)}{2}$$

Zatem

$$T_1(n) = T_1(n-1) + \frac{n(n+1)}{2}$$

Korzystając z zapisu symbolem Newtona, można zapisać, że

$$\frac{n(n+1)}{2} = \binom{n+1}{2}$$

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

Na przykład dla $n = 100$ otrzymamy

$$\frac{100(100+1)}{2} = 5050$$

$$\binom{n+1}{2} = \frac{101!}{2!\,(101-2)!} = \frac{101!}{2!\,99!} = \frac{100 \cdot 101}{2!} = 5050$$

Jeśli więc rozpiszemy powyższą rekurencję, to otrzymamy:

$$T_1(n) = \binom{2}{2} + \binom{3}{2} + \cdots + \binom{n+1}{2}$$

Do obliczenia wartości tej sumy można użyć trójkąta Pascala

						1						
					1		**1**					
				1		**2**		**1**				
			1		**3**		**3**		**1**			
		1		**4**		**6**		**4**		**1**		
	1		**5**		**10**		**10**		**5**		**1**	

Zastanówmy się teraz, gdzie w trójkącie Pascala leży ciąg liczb $\binom{2}{2} + \binom{3}{2}$ i tak dalej.

$$\binom{2}{2} + \binom{3}{2} + \binom{4}{2} + \binom{5}{2} = 1 + 3 + 6 + 10$$

Zatem liczby te leżą na przekątnej oznaczonej kolorem żółtym.

						1						
					1		1					
				1		2		1				
			1		3		3		1			
		1		4		6		4		1		
	1		5		10		10		5		1	

Nie ma też potrzeby sumowania wyróżnionych liczb, bo wynik tego sumowania znajduje się poniżej i oznaczony jest na czerwono.

						1						
					1		1					
				1		2		1				
			1		3		3		1			
		1		4		6		4		1		
	1		5		10		10		5		1	
1		6		15		20		15		6		1

Jak ustalić zatem ogólny wzór na sumę elementów takiego ciągu? Skoro wynik leży na przekątnej poniżej, to znaczy, że sumujemy aż do $(n+1)$ wiersza i w związku z tym będzie on miał postać następującą:

$$\binom{n+2}{3}$$

W tym wypadku można zatem uzyskać bardzo dokładny wzór na liczbę operacji dominujących (sumowań). Niestety, gdybyśmy chcieli analizować w ten sposób każdy algorytm, byłoby to bardzo trudne, czasami wręcz niemożliwe. W związku z tym w algorytmice posługujemy się notacją asymptotyczną, która daje wyniki przybliżone, ale wystarczająco dokładne do prowadzenia analiz i porównań różnych algorytmów.

Użyteczne notacje:

f, g – funkcje ze zbioru liczb naturalnych w zbiór liczb rzeczywistych

Mówimy, że funkcja f jest co najwyżej rzędu funkcji g (co zapisujemy następująco)

$$f(n) = O\big(g(n)\big)$$

(Z punktu widzenia matematyki powinno być $f(n) \subset O\big(g(n)\big)$, ale w algorytmice przyjął się znak równości)

gdy istnieje stała rzeczywista c i stała naturalna n_0 taka, że dla każdej liczby naturalnej $n \geq n_0$

$$f(n) \leq c \cdot g(n)$$

Zatem wartość funkcji $f(n)$ da się od góry ograniczyć przez wartość wyrażenia $c \cdot g(n)$. Można więc powiedzieć, że od pewnego momentu badaną funkcję $f(n)$ można szacować od góry przez funkcję $g(n)$.

Podobną notację można wprowadzić dla dolnego ograniczenia. W tym przypadku mówimy, że funkcja f jest co najmniej rzędu funkcji g (co zapisujemy)

$$f(n) = \Omega\big(g(n)\big)$$

gdy istnieje stała rzeczywista c i stała naturalna n_0 taka, że dla każdej liczby naturalnej $n \geq n_0$

$$f(n) \geq c \cdot g(n)$$

Zatem wartość funkcji $f(n)$ da się od dołu ograniczyć przez wartość wyrażenia $c \cdot g(n)$. Można więc powiedzieć, że od pewnego momentu badaną funkcję $f(n)$ można szacować od dołu przez funkcję $g(n)$.

Jeśli jednocześnie $f(n) = O\big(g(n)\big)$ oraz $f(n) = \Omega\big(g(n)\big)$, to mówimy, że funkcja f jest dokładnie rzędu funkcji g i zapisujemy

$$f(n) = \Theta\big(g(n)\big)$$

W przypadku analizowanego rozwiązania 1 algorytmu mamy

$$T_1(n) = \Theta(n^3)$$

Stała c dla ograniczenia górnego wynosi 1, a dla dolnego $\frac{1}{6}$.

Pesymistyczna złożoność czasowa algorytmu jest funkcją rozmiaru danych i definiuje się ją następująco:

$$T(n) = \sup\{t(d) : \mathrm{d} - \text{dane rozmiaru n}\}$$

gdzie $t(d)$ jest liczbą operacji dominujących wykonywanych przez algorytm dla danych d. W tym wypadku rozmiarem danych jest liczba n, czyli długość ciągu. Ten algorytm będzie też miał cały czas taką samą złożoność czasową, niezależną od wartości wprowadzanych liczb, o ile tylko ciąg będzie miał taką samą długość.

Supremum (sup), a inaczej kres górny, oznacza najmniejsze z ograniczeń górnych (o ile istnieje) zbioru liczbowego. Przeciwieństwem jest infimum (inf), inaczej kres dolny, który oznacza największe z ograniczeń dolnych (o ile istnieje) zbioru liczbowego.

Algorytmy analizuje się najczęściej pod kątem złożoności pesymistycznej, ponieważ chcemy wiedzieć, jak algorytm zachowuje się w najgorszym możliwym przypadku.

Teraz nadszedł właściwy czas, by zastanowić się, jak poprawić ten algorytm. Zauważmy, co złego dzieje się w tym algorytmie i co możemy tutaj poprawić. Przebiegamy tutaj po wszystkich możliwych przedziałach, a wszystkich możliwych przedziałów jest rzędu n^2 i w każdym możliwym przedziale dokonuje sumowania, które jest rzędu n. Mamy więc kwadratową liczbę przedziałów, a w każdym przedziale dokonujemy jeszcze sumowania. Wykonujemy więc redundantne operacje. Jeśli na przykład obliczamy sumę w przedziale od pozycji 2 do 6, a następnie od 2 do 7, to zapominamy wynik pierwszego sumowania i sumujemy od nowa dla nowego przedziału, a przecież dla poprzedniej sumy wystarczy dodać wartość siódmego elementu.

Rozwiązanie 2.

$$suma[i, j+1] = suma[i, j] + a[j+1]$$

Mamy zatem pomysł na nowy algorytm.

Algorytm 2.

```
najlepsza = 0;
for(i = 0; i < n; i++)
   {
    suma = 0;
    for(j = i; j < n; j++)
       {
        suma = suma + a[j]; //operacja dominująca
           if(suma > najlepsza)
              najlepsza = suma;
       }
   }
```

Liczba operacji dominujących równa jest liczbie przedziałów i wynosi

$$T_2(n) = \frac{n(n+1)}{2}$$

Zatem algorytm 2 działa w czasie n^2 (czyli kwadratowym) i jest o wiele lepszy od algorytmu 1.

Czy da się jeszcze bardziej ulepszyć ten algorytm? Da się, na przykład za pomocą rozwiązania „indukcyjnego", które ma częste zastosowanie dla zadań z ciągami. Należy odpowiedzieć na pytanie, w jaki sposób z rozwiązania dla tablicy $a[0 \dots i-1]$, otrzymać rozwiązanie dla $a[0 \dots i]$?

Rozwiązanie 3.
Przeglądamy nasz ciąg od strony lewej do prawej. Załóżmy, że obliczyliśmy najlepszą sumę w przedziale $[0 \dots i-1]$. Przechodzimy do kolejnego elementu i-tego oraz zastanawiamy się, jak ten element wpłynie na najlepszą sumę w przedziale $[0 \dots i]$?

Niezmiennik:
`najlepsza` – jest najlepszą sumą w tablicy $a[0 \dots i-1]$,
`najlepsza_prawa` – jest najlepszą sumą w tablicy $a[0 \dots i-1]$ dla przedziałów o prawym końcu w $i-1$.

Algorytm 3.
```
najlepsza = 0, najlepsza_prawa = 0;
for(i = 0; i < n; i++)
   {
    najlepsza_prawa = max(0, najlepsza_prawa + a[i]);
    if(najlepsza_prawa > najlepsza)
       najlepsza = najlepsza_prawa;
   }
```

Ten algorytm działa w czasie n, czyli liniowym i jest najlepszym (optymalnym) rozwiązaniem przedstawionego problemu przy danych założeniach.

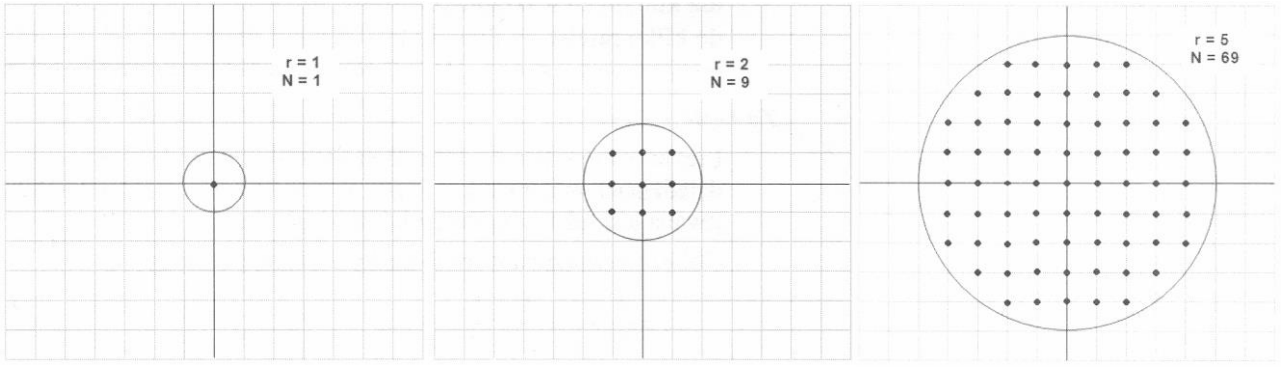
Z5. (nieobowiązkowe)
Na podstawie powyższych rozważań napisz w języku C#, (Python, C++ lub Java) program, który precyzyjnie porównuje przedstawione algorytmy pod kątem złożoności obliczeniowej oraz dokładnego czasu ich wykonania. Do analizy porównawczej wykorzystaj dane zawarte w plikach: `najlepsza_1.txt`, `najlepsza_2.txt` oraz `najlepsza_3.txt`. Opisz tok rozumowania w pliku `PDF`. Rozwiązania niepoprawne lub częściowo poprawne nie będą honorowane.

Termin przesłania rozwiązania zadania **Z1** upływa w dniu **14 kwietnia 2023 r.** (o północy)

Ćwiczenia do wykonania na zajęciach

C4.
Napisz program, który pozwoli użytkownikowi wprowadzić n liczb całkowitych $a_0, a_1, \cdots, a_{n-1}$ $(5 \leq n \leq 25)$, a następnie wyszuka wśród nich trójkę o indeksach x, y, z $(0 \leq x < y < z < n)$, których iloczyn $a_x \cdot a_y \cdot a_z$ jest maksymalny. Na wyjściu powinna pojawić się liczba całkowita równa wartości największego iloczynu.

C5.
Dane jest koło o środku w układzie współrzędnych i promieniu r. Punkt kratowy to punkt, którego współrzędne w układzie kartezjańskim są liczbami całkowitymi (patrz: rysunki).



Poprawne wyniki:
$r = 1 \Rightarrow N = 1$

$r = 2 \Rightarrow N = 9$

$r = 2{,}1 \Rightarrow N = 13$

$r = 2{,}5 \Rightarrow N = 21$

$r = 5 \Rightarrow N = 69$

$r = 100 \Rightarrow N = 31\,397$

Na podstawie powyższych rozważań napisz program w języku C# (Java, Python, C++), który oblicza liczbę N wszystkich punktów kratowych leżących wewnątrz tego koła. Mile widziane rozwiązania rekurencyjne.