Wstęp do programowania imperatywnego
kartkówka z rekurencji

Treść zadania:

Tablica A zawiera posortowane rosnąco liczby naturalne reprezentujące pewien n-elementowy zbiór **A**, $n > 0$. Napisz funkcję

```
RóżnePodzbiory(const A:Array[1..n] of Integer; S:Integer):Integer;
```

która wyznaczy liczbę różnych podzbiorów zbioru **A** o sumie elementów równej S. Określ złożoność czasową i pamięciową podanego algorytmu.

Opis rozwiązań studentów

Zadanie rozwiązywało 84 studentów. Można je podzielić na następujące kategorie:

- Prace, w których nie widać nic, co można uznać za fragment poprawnego myślenia o problemie. Do tej kategorii wliczają się wszystkie rozwiązania, których autorzy myśleli, że znaleźli algorytm działający w czasie wielomianowym (chyba, że wynikało to z błędnego policzenia czasu) — są to rozwiązania, które sprawdzały wszystkie spójne podciągi, być może z jakimiś modyfikacjami (np. wyrzucały elementy, jeśli po dodaniu wartości suma nie mieściła się w S). Rozwiązań takich było 37. Zdobyły one 0 punktów.

- Rozwiązania, w których było widać coś w kierunku prawidłowego rozwiązania, ale wciąż było od tego daleko. Najlepszymi rozwiązaniami w tej kategorii były rozwiązania typu: *jeśli* $S = 0$ *to zwróć 1, jeśli* $S < 0$ *to zwróć 0, w przeciwnym przypadku zwróć* $\sum_i RoznePodzbiory(A, S - A[i])$. Rozwiązania takie nie uwzględniały tego, że mogą wziąć tą samą liczbę kilka razy do tego samego podzbioru, a także uwzględniały kilka razy ten sam podzbiór, jeśli elementy różniły się kolejnością (np. dla $S = 11$, tablicy $(3, 4)$ program taki by znalazł 3 rozwiązania: $3+4+4$, $4+3+4$ i $4+4+3$). Innym ciekawym pomysłem było stworzenie tablicy wielkości 2^N, która była wypełniana wszystkimi możliwymi do uzyskania sumami (w tym rozwiązaniu niestety były błędy). Rozwiązań tego typu było 25, dostawały one od 1 do 5 punktów.

- Rozwiązania, w których dla każdego elementu po kolei decydujemy, czy go wziąć do sumy, czy nie — rozdzielamy się zatem na 2 gałęzie wywoływane rekurencyjnie (sprawdzające drugi element itd.). Poniżej jest krótka implementacja takiego rozwiązania w Pascalu. (Zmienna t określa tu wartośc sumy, którą musimy uzyskać z jeszcze nieprzebadanych elementów.)

```
function
RóżnePodzbiory(const A:Array[1..n] of Integer; S:Integer):Integer;

function Rek(i, t: Integer): Integer;
begin
  if i = 0 then
    if t = 0 then Rek := 1
    else Rek := 0
  else
    Rek := Rek(i-1, t-A[i]) + Rek(i-1, t)
  end;

begin
  RóżnePodzbiory := Rek(n, S)
  end;
```

Jest to rozwiązanie poprawne, pojawiło się u 10 studentów. W zależności od ilości błędów dostawali oni od 5 do 10 punktów.

- Rozwiązania, w których wybieramy i – numer największego elementu, który znajdzie się w naszym podzbiorze – a następnie dobieramy pozostałe elementy spośród elementów o **niższych** numerach (aby uniknąć błędu z przykładu powyżej). Poniżej krótka implementacja w Pascalu.

```
function
RóżnePodzbiory(const A:Array[1..n] of Integer; S:Integer):Integer;

function Rek(i, t: Integer): Integer;
var pom: Integer;
begin
  if t = 0 then pom := 1 {*}
  else pom := 0;

  for j:=1 to i do pom := pom + Rek(j-1, t-A[j]);
  Rek := pom
  end;

begin
  RóżnePodzbiory := Rek(n, S)
  end;
```

  To rozwiązanie również jest poprawne i pojawiło się u 12 studentów. Dostali oni od 6 do 9 punktów (nie było całkowicie poprawnego rozwiązania tego typu). Jeden ze studentów zrobił wariant polegający na spamiętywaniu wartości dla par (i, t), dla których wynik policzyliśmy już wcześniej (niestety z błędem).

Powyższe rozwiązania są napisane w maksymalnie prosty sposób. Oczywiście chodzić po tablicy można było w dowolną stronę (i obie strony były stosowane przez studentów). Studenci zwykle próbowali pewnych optymalizacji (np. jeśli już wiemy, że suma dotychczasowa przekracza sumę, którą chcieliśmy uzyskać, to nie sprawdzamy danej gałęzi). Jednak mimo takich optymalizacji czas działania wciąż był $O(2^n)$, a pamięć liniowa.

Wśród błędów popełnianych w prawie poprawnych rozwiązaniach najczęstsze były następujące:

- różnego rodzaju błędy „rachunkowe"
- brak policzonego czasu (albo źle policzony) – 1 punkt karny; wielu studentów pisało, że rozwiązanie drugiego rodzaju działa w czasie $O(n!)$ – w rzeczywistości działa ono w czasie $O(2^n)$, ale nie było to traktowane jako błąd, bo (zgodnie z informacją podaną na wykładzie) optymalizacja czasu nie była istotna, a program działający w czasie 2^n działa też w czasie $O(n!)$ (gdyż notacja O oznacza *co najwyżej*); zdarzały się słowne określenia złożoności czasowej, typu *wykładnicza*, *duża*, *ogromna* albo *potworna*
- źle policzony albo brak policzonego kosztu pamięciowego (niektórzy uważali, że pamięć jest stała – w rzeczywistości do pamięci należy wliczyć wysokość stosu, która jest liniowa) – 1 punkt karny
- źle rozważane przypadki szczególne , w których jedna z liczb występująca w ciągu jest zerem, albo w których mamy uzyskać sumę 0 – 1 punkt karny
- Drobne błędy w składni Pascala nie były karane – a także te trochę poważniejsze, jak pisanie funkcji pomocniczej we wnętrzu bloku begin...end, albo przekazywanie fragmentu tablicy w wywołaniu rekurencyjnym jako $A[i+1..n]$); jednak traktowanie zmiennej lokalnej w procedurze wywoływanej rekurencyjnie tak, jakby była to ta sama zmienna w każdym wywołaniu, już było uznawane za błąd

- brak rozważonego przypadku, gdy rekurencja dochodzi do końca (gdy już trzeba zwrócić konkretny wynik, a nie rozgałęziać się dalej) – przykładem błędu w tej kategorii, który wystąpił w rozwiązaniach, jest napisanie po prostu $pom := 0$ zamiast if-a oznaczonego $\{*\}$ w programie powyżej (1-2 punkty)
- brak jednej z gałęzi (np. w rozwiązaniu pierwszego rodzaju, obecna tylko ta, w której i-ty element bierzemy do sumy, a brak tej, w której go nie bierzemy) – jeśli wyglądało na to, że student mniej więcej miał dobry pomysł (np. napisał, że program działa w czasie $O(2^n)$) to kara nie była bardzo duża (2-3 punkty)
- kilka razy zdarzył się błąd, że gdy znaleźliśmy w tablicy element, którego wartość była dokładnie równa wartości brakującej do otrzymania pożądanej sumy (t w rozwiązaniu powyżej), to zwracaliśmy 1, mimo, że być może dało się jeszcze uzyskać t jako sumę mniejszych elementów (1 punkt)