Opracowanie: KLO
Klocki

1 Rozwiązanie I

Bezpośrednie podejście do rozwiązania zadania polega na sprawdzeniu wszystkich możliwości. Istnieje 2^n podzbiorów zbioru wszystkich klocków. Możemy więc po prostu przejść po nich wszystkich i dla każdego z nich sprawdzić, czy zawiera on nie więcej niż k klocków i czy ich sumaryczna masa nie przekracza s. Jeżeli dany podzbiór spełnia te warunki, to porównujemy jego masę z najlepszą dotychczas znalezioną.

Pozostaje pytanie, jak w łatwy sposób przejść po wszystkich podzbiorach. Jedno z podejść polega na napisaniu funkcji rekurencyjnej, która dla danego i generowałaby wszystkie podzbiory zbioru $\{i, i+1, \ldots, n\}$.

Tutaj jednak zaprezentujemy inną metodę, polegającą na pętli po wszystkich liczbach n-bitowych. Każdą z tych liczb możemy interpretować jako podzbiór, do którego i-ty należy klocek wtedy i tylko wtedy, gdy ma ona i-ty bit równy 1.

Pseudokod tego rozwiązania wygląda następująco:

```
function ile_elementow(zbior)
begin
    ile := 0;
    for i := 1 to n do
        if bit(i, zbior) = 1 then
            ile := ile + 1;
    return ile;
end

function suma_elementow(zbior)
begin
    suma := 0;
    for i := 1 to n do
        if zapalone(i, zbior) then
            suma := suma + masy[i];
    return suma;
end

wczytaj_dane();
odpowiedz := 0;
for zbior := 0 to 2^n - 1 do  { 2^n to "dwa do potęgi n-tej" }
begin
    suma := suma_elementow(zbior);
    if (ile_elementow(zbior) <= k) and (suma <= s) then
        odpowiedz := max(odpowiedz, suma);
end
wypisz(odpowiedz);
```

Warunek sprawdzający, czy i-ty bit liczby jest jedynką, można zapisać w języku C/C++ następująco:

```
if ((1<<i) & liczba) ...
```

zaś w Pascalu następująco:

```
if (1 shl i) and liczba then ...
```

Ponieważ iterujemy w tym rozwiązaniu po wszystkich 2^n podzbiorach i dla każdego z nich wykonujemy n operacji, to złożoność czasowa rozwiązania wynosi $O(n \cdot 2^n)$. To rozwiązanie otrzymywało 50% punktów.

2 Rozwiązanie II

Aby przyspieszyć nasz algorytm, zauważmy, że w poprzednim rozwiązaniu niepotrzebnie sprawdzamy wszystkie podzbiory. Możemy się bowiem ograniczyć tylko do tych, które mają co najwyżej k elementów, a innych w ogóle nie sprawdzać.

W tym celu musimy już zastosować rekurencję. W każdym wywołaniu funkcji rekurencyjnej będziemy sprawdzać, czy nie wybrano jeszcze k elementów. Jeżeli tak, to przerwiemy dalsze rekurencyjne wywołania i porównamy otrzymany wynik z najlepszym dotychczas znalezionym.

Przykładowy pseudokod:

```
ile := 0;
wynik := 0;

function podzbiory(i)
begin
    if (ile = k) or (i = n + 1) then
    begin
        { sprawdzamy czy znalezione rozwiązanie jest lepsze od dotychczas najlepszego }
        if (suma > wynik) and (suma <= s) then
            wynik := suma;
    end else
    begin
        { wybieramy klocek o numerze 'i' }
        ile := ile + 1;
        suma := suma + t[i];
        podzbiory(i + 1);

        ile := ile - 1;
        suma := suma - t[i];
        { nie wybieramy klocka o numerze 'i' }
        podzbiory(i + 1);
    end;
end;

wczytaj_dane();
podzbiory(1);
```

Złożoność czasowa tego rozwiązania wynosi $O(2^n)$, ponieważ tym razem dla każdego wygenerowanego zbioru wykonujemy stałą liczbę operacji (liczba klocków i suma ich mas są liczone na bieżąco).

3 Rozwiązanie III

Do poprzedniego rozwiązania można jeszcze dodać pewną optymalizację. Oprócz ucinania rekursji po wygenerowaniu zbioru o k elementach, możemy ją ucinać także w momencie, kiedy suma mas wybranych klocków przekroczy s. Dodatkowo posortowanie elementów nierosnąco spowoduje, że suma elementów przekroczy liczbę s prawdopodobnie szybciej niż w przypadku nieposortowanego ciągu. W dalszym ciągu złożoność czasowa rozwiązania wynosi jednak $O(2^n)$, jednak to rozwiązanie uzyskuje już ok. 80% punktów.

4 Rozwiązanie IV (wzorcowe)

Rozwiązanie wzorcowe stosuje metodę programowania dynamicznego. Polega ona na wykorzystaniu rozwiązań mniejszych podproblemów do znalezienia rozwiązania całego problemu.

Każdy podproblem określmy dwoma liczbami: i, j. Nazwijmy go $d[i][j]$. Rozwiązaniem podproblemu $d[i][j]$ ma być odpowiedź na pytanie, ilu co najmniej klocków spośród i pierwszych trzeba użyć, aby ich suma wynosiła dokładnie j. Jak można to szybko zrobić?

Rozpatrzmy klocek o numerze i: możemy go albo użyć, albo nie.

- Jeżeli nie wybierzemy i-tego klocka, to rozwiązaniem jest wynik podproblemu $d[i-1][j]$.
- Jeżeli wybierzemy i-ty klocek, to rozwiązaniem jest wynik podproblemu $d[i-1][j-masy[i]]$ zwiększony o jeden.

Zatem wzorem na rozwiązanie problemu jest:

$$d[i][j] = \min(d[i-1][j],\ d[i-1][j-masy[i]]+1).$$

Aby przy rozwiązywaniu podproblemu dla klocków $1..i$ móc korzystać z rozwiązań dla mniejszej liczby klocków, rozwiązania obliczamy kolejno dla coraz większych wartości i.

Nie interesują nas sumy większe od s, zatem wystarczy, że policzymy rozwiązanie dla każdej sumy z przedziału $[0..s]$. Ostatecznie otrzymamy rozwiązania wszystkich $n \cdot s$ podproblemów.

Na koniec wystarczy znaleźć największą taką liczbę w ($w \leqslant s$), że $d[n][w] \leqslant k$. Będzie to oznaczać, że jesteśmy w stanie otrzymać sumę w, wybierając nie więcej niż k klocków spośród wszystkich n. A o to przecież chodziło w zadaniu!

Jest do rozwiązania jeszcze jeden problem. Otóż tablica $d[1..n][1..s]$ ma $n \cdot s$ komórek, czyli maksymalnie 30 000 000. W każdej z nich będzie przechowywana liczba całkowita, która zajmuje 4 bajty. Tak więc w sumie tablica ta zajmie w pamięci około 120 MB. Jest to znacznie więcej niż limit 32 MB ustalony dla tego zadania.

Okazuje się jednak, że tablica dwuwymiarowa nie jest w rozwiązaniu do niczego potrzebna. W rzeczywistości wystarczy utrzymywać tablicę jednowymiarową $d[1..s]$, którą będziemy aktualizować kolejno dla wszystkich klocków.

W tym celu dla każdego klocka i przechodzimy wszystkie masy j w kolejności *od największej do najmniejszej*. Każde pole aktualizujemy zgodnie z wcześniejszym wzorem:

$$d[j] = \min(d[j],\ d[j - masy[i]] + 1).$$

Kolejność iteracji po masach (zmienna j) zmieniamy po to, aby przy liczeniu $d[j]$ dla i-tego klocka korzystać z wyników $d[j - masy[i]]$ policzonych dla klocka $i - 1$, a nie z poprzednich wyników dla klocka i.

```
{ wczytanie danych }
wczytaj(n, k, s);
for i := 1 to n do
    read(masy[i]);

{ inicjalizcja tablicy wyników }
d[0] := 0;
for i := 1 to s do
    d[i] := n + 1; { jeżeli nie da się uzyskać sumy, to wpisujemy n + 1 }

{ obliczenie rozwiązań podproblemów }
for i := 1 to n do
    for j := s downto masa[i] do { istotne jest, aby tablicę wypełniać od tyłu }
        if d[j] > d[j - masy[i]] + 1 then
            d[j] := d[j - masy[i]] + 1;

{ znalezienie maksymalnej sumy }
wynik := 0;
for i := s downto 0 do
    if d[i] <= k then
    begin
        wynik := i;
        break;
    end;

wypisz(wynik);
```

Złożoność czasowa tego rozwiązania wynosi $O(n \cdot s)$, ponieważ główna (najbardziej pracochłonna) część algorytmu składa się z pętli przechodzącej po $O(s)$ liczbach, zagnieżdżonej w pętli iterującej po $O(n)$ liczbach.