Temă

1 Fractional Knapsack Problem

Pentru algoritmul de sortare am ales HeapSort, avantajul său fiind că poate fi oprit imediat după ce am terminat de umplut rucsacul (când c devine 0).

```
min(x, y) { return x < y ? x : y; }
swap(out x, out y) { z = x; x = y; y = z; }

n = 6;
c = 10;
v = [7, 4, 2, 9, 4, 5];
g = [3, 3, 1, 1, 2, 1];

heapSize = n;
heap = [0 | i from [0..n]];
for (i = 1; i <= n; i++)
    heap[i] = i - 1;

gt(x, y) modifies v, g {
    return v[x] * g[y] > v[y] * g[x];
}

heapify(pos) modifies heap, heapSize {
    lft = 2 * pos;
    rgh = 2 * pos + 1;
    max = pos;
    if (lft <= heapSize && gt(heap[lft], heap[max])) max = lft;
    if (rgh <= heapSize && gt(heap[rgh], heap[max])) max = rgh;
    if (max != pos) {
        swap(heap[pos], heap[max]);
        heapify(max);
    }
}

ans = 0;
for (i = n / 2; i >= 1; i--)
    heapify(i);
for (i = n; i >= 1 && c > 0; i--) {
    swap(heap[1], heap[i]);
    heapSize--;
    heapify(1);
    now = heap[i];
    ans += 1.0 * min(g[now], c) / g[now] * v[now];
    c -= min(g[now], c);
}
print(ans);
```

2 Set Partition Problem

Input. O mulțime $S \subset \mathbb{N}$ de cardinal n.

Output. `YES` dacă există o partiție (A, B) a lui S cu proprietatea că $\sum_{x \in A} x = \sum_{y \in B} y$; `NO` altfel.

Domeniu. (A, B) se numește *partiție* a lui S dacă $A, B \subset S$, $A, B \neq \varnothing$, $A \cap B = \varnothing$ și $A \cup B = S$.

2.1 Relația de recurență

Fie $s = \sum_{x \in S} x$. Notăm cu $\mathrm{dp}[i][j] \in \{\texttt{false}, \texttt{true}\}$ posibilitatea de a forma o submulțime a lui S, de sumă j $(0 \leq j \leq \lfloor s/2 \rfloor)$, din primele sale i $(0 \leq i \leq n)$ elemente. Obținem următoarea relație de recurență:

$$\mathrm{dp}[i][j] = \begin{cases} \texttt{true} & \text{dacă } i = 0, j = 0 \\ \texttt{false} & \text{dacă } i = 0, j > 0 \\ \mathrm{dp}[i-1][j] \ \texttt{or} \ \mathrm{dp}[i-1][j - S_{i-1}] & \text{dacă } i > 0, j \geq S_{i-1} \\ \mathrm{dp}[i-1][j] & \text{altfel} \end{cases}$$

Răspunsul problemei este afirmativ dacă s este par și $\mathrm{dp}[n][s/2]$ are valoarea `true`.

2.2 Algoritm recursiv (cu memoizare)

```
set = [6, 1, 8, 3, 2, 4];
sum = 0;
for (i = 0; i < set.size(); i++)
    sum += set[i];
dp = [[-1 | j from [0..sum / 2]] | i from [0..set.size()]];

getDP(i, j) modifies set, dp {
    if (dp[i][j] == -1) {
        if (i == 0)
            dp[i][j] = (j == 0 ? 1 : 0);
        else {
            dp[i][j] = getDP(i - 1, j);
            if (j >= set[i - 1])
                dp[i][j] |= getDP(i - 1, j - set[i - 1]);
        }
    }
    return dp[i][j];
}
print(sum % 2 == 0 && getDP(set.size(), sum / 2) == 1 ? "YES" : "NO");
```

2.3 Algoritm iterativ

```
set = [6, 1, 8, 3, 2, 4];
sum = 0;
for (i = 0; i < set.size(); i++)
    sum += set[i];

if (sum % 2 == 1)
    print("NO");
else {
    dp = [[false | j from [0..sum / 2]] | i from [0..set.size()]];
    dp[0][0] = true;
    for (i = 1; i <= set.size(); i++)
        for (j = 0; j <= sum / 2; j++) {
            dp[i][j] = dp[i - 1][j];
            if (j >= set[i - 1])
                dp[i][j] = dp[i][j] || dp[i - 1][j - set[i - 1]];
        }
    print(dp[set.size()][sum / 2] ? "YES" : "NO");
}
```

2.4 Algoritm iterativ cu $O(s)$ memorie

Cum fiecare stare a dinamicii (cu $i > 0$) se bazează pe valori de pe linia precedentă și

$$\mathrm{dp}[i-1][j] = \texttt{true} \to \mathrm{dp}[i][j] = \texttt{true},$$

putem scăpa de prima dimensiune a tabloului dp, construind linia curentă direct din linia precedentă. Astfel, vom seta la `true` eventualele valori $\mathrm{dp}[j]$ pentru care $\mathrm{dp}[j - S_{i-1}] = \texttt{true}$. Cum $j - S_{i-1} < j$, indicii j trebuie parcurși în ordine descrescătoare.

```
set = [6, 1, 8, 3, 2, 4];
sum = 0;
for (i = 0; i < set.size(); i++)
    sum += set[i];

if (sum % 2 == 1)
    print("NO");
else {
    dp = [false | j from [0..sum / 2]];
    dp[0] = true;
    for (i = 1; i <= set.size(); i++)
        for (j = sum / 2; j >= set[i - 1]; j--)
            dp[j] = dp[j] || dp[j - set[i - 1]];
    print(dp[sum / 2] ? "YES" : "NO");
}
```