Programowanie dynamiczne

Jeżeli podproblemy, na które został podzielony problem główny, nie są niezależne to w różnych podproblemach wykonywane są wiele razy te same obliczenia, warto jest wtedy zastosować ulepszenie tej metody- programowanie dynamiczne. Wyniki obliczeń są zapamiętywane w tablicy pomocniczej, która jest wykorzystywana w kolejnych krokach algorytmu, co eliminuje potrzebę wielokrotnego wykonywania tych samych obliczeń. Prowadzi to do widocznego obniżenia złożoności obliczeniowej. Przykładem może być obliczanie symbolu Newtona. Rekurencyjna funkcja wyznaczająca ten współczynnik ma złożoność wykładniczą. Po zastosowaniu programowania dynamicznego złożoność maleje do O(n2). Programowanie dynamiczne polega więc na wykonania obliczeń każdego podproblemu tylko raz i zapamiętaniu jego wyniku w tabeli. W każdym kolejnym kroku można z tej tabeli korzystać. Programowanie dynamiczne jest zazwyczaj stosowane w rozwiązywaniu problemów optymalizacyjnych, prowadzi to często do wyznaczenia kilku równoznacznych, optymalnych rozwiązań. Taka metoda tworzenia algorytmów znalazła zastosowanie m.in. w rozwiązywaniu problemu plecakowego, w optymalnym mnożeniu ciągu macierzy. Jest także stosowana w automatach do kawy przy wydawaniu reszty w taki sposób, by monet było najmniej.

U Jarka też to było w problemie komiwojażera. Mieliśmy znaną najlepszą drogę do tej pory przebytą i na tej podstawie liczyliśmy dalszą.

To chyba najlepiej na przykładach łyknąć

Przykład 1

Dyskretny problem plecakowy

Jest N przedmiotów. Każdy waży w_i kilogramów i kosztuje kosztuje c_i. Możemy unieść maksymalnie K kilogramów. Które przedmioty wziąć, aby łączna wartość zabranych towarów była jak największa?

Rozwiązanie

Niech wynik[n, k] oznacza rozwiązanie dla pierwszych n przedmiotów i plecaka rozmiaru k. Znając wynik dla mniejszych danych możemy obliczyć go dla większych danych. Konkretnie jest tak:
wynik[n, k] = max(wynik[n-1, k], wynik[n-1, k-w_i]+c_i, wynik[n, k-1])
Możemy bowiem albo nie brać n-tego przedmiotu i zapełnić k kilogramów poprzednimi przedmiotami (wynik[n-1, k]), albo wziąć n-ty przedmiot i zapełnić k-w_i kilogramów poprzednimi przedmiotami (wynik[n-1, k-w_i]+c_i), albo pozostawić ostatni kilogram plecaka pusty i popatrzeć co się mieści w k-1 kilogramach (wynik[n, k-1]). Rozwiązanie to działa w czasie O(NK).

Oto program, który to liczy:

```
{w tablicach w oraz c mamy dane przedmiotów}
for a := 0 to K do
   wynik[0, a] := 0;
for p := 1 to N do
   for a := 0 to K do begin
            wynik[p, a] := wynik[p-1, a];
            if (a>=1) and (wynik[p, a-1]>wynik[p, a]) then
               wynik[p, a] := wynik[p, a-1];
            if (a>=w[p]) and (wynik[p-1, a-w[p]]+c[p]>wynik[p, a]) then
               wynik[p, a] := wynik[p-1, a-w[p]]+c[p];
   end;
{w wynik[N, K] mamy rozwiązanie}
```

Przykład 2

Rozpatrzmy rekurencyjny algorytm obliczający ciąg Fibonacciego:

Fib(0)= 0, Fib(1)=1, Fib(n)= Fib(n-1)+Fib(n-2) dla n=2.....

```
int Fib( int n ) {
  if( n < 1 )
    return 0;
  if( n == 1 )
    return 1;
  return Fib(n-1) + Fib(n-2);
}
```

Nietrudno zauważyć, że algorytm jest nieefektywny, bo przy obliczeniu np. Fib(5) liczy (wciecia oznaczają kolejne poziony rekurencji).

```
[Fib(5)]  = Fib(4) + Fib(3)
  [Fib(4)] = Fib(3) + Fib(2)
  [Fib(3)] = Fib(2) + Fib(1)
    [Fib(3)] = Fib(2) + Fib(1)
    [Fib(2)] = Fib(1) + Fib(0)
    [Fib(2)] = Fib(1) + Fib(0)
      [Fib(2)] = Fib(1) + Fib(0)
```

Na przykład wartość Fib(2) jest wyznaczana aż trzykrotnie.

Algorytm rekurencyjny można poprawić wykorzystując tak zwane "zapamiętywanie" (spamiętywanie, ang. mnemonization): wykorzystujemy w tym celu tablicę zawierającą już obliczone wartości:

```
int fib[MAX];
```

inicjujemy ją, aby oznaczyć wszystkie wartości jako "nieznane"

```
for( i= 0; i < MAX; i++ )
  fib[i]= -1;
```

i piszemy funkcję, która wykorzystuje tablicę:

```
int Fib( int n ) {
  if( n < 1 )
    return 0;
  if( n == 1 )
    return 1;
  if( fib[n] == -1 )
    fib[n]= Fib(n-1) + Fib(n-2);
  return fib[n];
}
```