Programowanie dynamiczne

#programowanie_dynamiczne
#metoda_zstępująca
#metoda_wstępująca
#sizeof

Ciekawą klasę algorytmów stanowią metody oparte na *programowaniu dynamicznym* (ang. **dynamic programming**, wymawiaj: *dajnamik programing*). Nazwa tej klasy metod nie pochodzi bynajmniej od samego faktu programowania komputerów, ale od sposobu konstrukcji algorytmu, opierającego się na podziale rozwiązywanego problemu na mniejsze podproblemy, rozwiązaniu tychże (często w sposób rekurencyjny), a następnie na połączeniu otrzymanych rozwiązań w ostateczne, całościowe rozwiązanie.*

Programowanie dynamiczne zilustrujemy przykładem z naszej ulubionej Bitolandii, gdzie królewscy urzędnicy zamierzają wprowadzić system opłat za korzystanie z dróg przez pojazdy obywateli, czyli tak zwane *myto* (ang. **toll**). Nie jest to jednak takie bardzo proste, gdyż tabela opłat została narzucona przez parlament, a z kolei miłościwie panujący monarcha (oby żył wiecznie!) oczekuje jak największych wpływów do królewskiej szkatuły.

Załóżmy, że w tym szczęśliwym kraju nie ma dróg dłuższych niż tuzin bitomil (bm)† i że długość każdej drogi wyraża się liczbą całkowitą. Parlament zarządził następującą tabelę opłat w bitalarach (bt) w zależności od długości przejechanego odcinka drogi (każdy odcinek jest wyceniany oddzielnie):

Długość (bm)	0	1	2	3	4	5	6	7	8	9	10	11	12
Myto (bt)	0	1	3	6	7	8	9	12	12	13	13	15	16

Urzędnicy mają następujące narzędzie prawne, które może znacząco podnieść dochody króla: otóż mogą na drodze postawić bramki do poboru opłat, tym samym dzieląc ją na dwa oddzielne odcinki. Na przykład odcinek o długości 12 (za który wypadłaby opłata 16) mogą podzielić na dwa odcinki po 6, z których każdy przyniesie po 9 bitalarów, czyli łącznie opłata wyniesie 18. Te krótsze odcinki też zresztą można dalej podzielić – ma to sens, jeśli przyniesie to dodatkowy zysk.

Zatem zadanie jest postawione: w jaki sposób można podzielić odcinek drogi o długości L tak, aby przejazd przez niego wiązał się z jak największą opłatą? Chyba widać, że sposobów podziału takiego odcinka może być naprawdę sporo i pasowałoby sprawdzić wszystkie możliwe podziały, aby wybrać ten najbardziej lukratywny. Od razu możemy jednak zauważyć, że wiele spośród tych podziałów będzie sobie równoważnych – na przykład odcinek o długości 5 możemy podzielić na $1+4$ albo $4+1$, co wyjdzie na jedno.

*Pokrewne metody tworzenia algorytmów, jak *dziel i zwyciężaj* (ang. **divide and conquer**, wymawiaj: *diwajd end konker*, z akcentem na *a*) oraz algorytmy zachłanne (ang. **greedy**, wymawiaj: *gridy*, z długim *i*) zostaną omówione w oddzielnych podrozdziałach.

†Cokolwiek by ta jednostka znaczyła.

Metoda „naiwna”

Spróbujmy wypisać wszystkie istotnie różne podziały odcinka o długości $L = 5$ wraz ze spodziewanymi opłatami:

Podział	Łączna opłata
$5 + 0$	8
$4 + 1$	8
$3 + 2$	9
$3 + 1 + 1$	8
$2 + 2 + 1$	7
$2 + 1 + 1 + 1$	6
$1 + 1 + 1 + 1 + 1$	5

Jak widać, najwyższą opłatę (9) uzyskamy przy podziale $3 + 2$.

No, nie było aż tak źle.[‡] Ale jak wygenerować takie wszystkie rozkłady? Najłatwiej zrobić to przy pomocy funkcji rekurencyjnej (nazwiemy ją `toll_count()`). Oznaczmy przez *toll*[ ] tablicę zawierającą opłaty za poszczególne długości odcinków (czyli dolny wiersz tabelki z poprzedniej strony, opatrzony nazwą „Myto”):

```
int main()
{
  int toll[] = {0, 1, 3, 6, 8, 8, 9, 12, 12, 13, 13, 15, 16};
  cout << toll_count(toll, 5) << endl;
}
```

Teraz zajmiemy się funkcją `toll_count()`. Posiada ona dwa argumenty: tabelę opłat i długość interesującego nas odcinka L (w naszym przykładzie nie może ona być większa niż 12).

```
int toll_count(int toll[], int L)
{
  . . .
}
```

Zwracamy uwagę, że jeśli tablica występuje jako argument funkcji, wtedy nie podaje się jej rozmiaru.

Generowanie wszystkich podziałów można usystematyzować według długości pierwszego odcinka, licząc od początku drogi: jeśli na przykład ten początkowy odcinek ma długość i (gdzie $1 \leqslant i \leqslant L$), wtedy na pozostałą część drogi pozostanie dystans $L - i$. O ile opłata za pierwszy odcinek będzie wynosić po prostu $toll[i]$, to do obliczenia opłaty (maksymalnej) za pozostałą część drogi należy wywołać rekurencyjnie funkcję `toll_count` z drugim argumentem równym właśnie $L - i$. Łączna opłata to suma za pierwszy odcinek i rekurencyjnie wyliczona opłata za pozostałą część. Jeśli sprawdzimy wszystkie warianty długości pierwszego odcinka (dla i od 1 do L) i wybierzemy najwyższy łączny wynik, to mamy znalezioną poszukiwaną maksymalną opłatę za całą drogę.

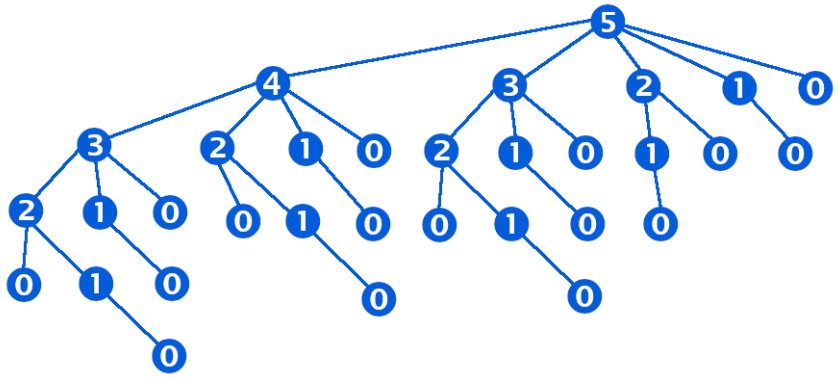
[‡]Podział $5 + 0$ oznacza, że na drodze nie ma żadnej bramki.

Możemy już napisać naszą funkcję według powyższego schematu (pamiętamy, że funkcje rekurencyjne zwykle zaczyna się od warunku zakończenia rekurencji – tutaj stanowi o tym $L = 0$):

```
int toll_count(int toll[], int L)
{
  if(L == 0) return 0;
  int profit = -1;
  for(int i = 1; i <= L; i++)
    profit = max(profit, toll[i] + toll_count(toll, L - i));
  return profit;
}
```

Zmienna *profit* oznacza proponowaną opłatę za całość drogi.

Jak to zwykle bywa, zapis rekurencyjny jest bardzo zwięzły, ale czy aby na pewno jest to optymalny algorytm? Przypomnijmy sobie, jak obliczaliśmy liczby Fibonacci'ego (podrozdział *O rekurencji*) – obliczanie wprost z rekurencyjnego wzoru powodowało wielokrotne obliczanie tych samych wielkości, co jest oczywiście skrajną nieefektywnością. Tak jest również w przypadku powyższej funkcji. Jeśli wywołamy ją z przykładowym argumentem $L = 5$, wówczas dostajemy następujący schemat wywołań rekurencyjnych (kółeczka oznaczają kolejne instancje funkcji `toll_count()`):



Funkcja `toll_count()` z argumentem $L = 1$ jest wywoływana 8 razy, a z argumentem $L = 0$ – aż 16 razy! Czyli że mamy tutaj – o zgrozo! – złożoność wykładniczą. Trzeba coś z tym zrobić: w pierwszym podejściu posłużymy się techniką zwaną *spamiętywaniem* (ang. **memoization**, wymawiaj: *memoizejszn*, z akcentem na drugie *e*).[§]

Metoda zstępująca ze spamiętywaniem

W tej metodzie przyjmujemy zasadę, że jeżeli opłata za odcinek o danej długości L jest już policzona, wtedy powinna być przechowywana w pomocniczej tablicy, z której możemy ją pobrać bez potrzeby rekurencyjnego wywoływania funkcji. Napiszemy nieco inną wersję funkcji `toll_count()` – będzie ona w dalszym ciągu rekurencyjna, ale zdecydowanie bardziej „inteligentna". Nazwiemy ją `toll_mem()`.

[§]Wspominaliśmy o niej w podrozdziale *O rekurencji*.

Zaczniemy od programu głównego, praktycznie takiego samego, jak w poprzedniej sekcji, dodamy jednak wspomnianą pomocniczą tablicę *result*[ ], gdzie przechowamy wyniki cząstkowych obliczeń:

```
int main()
{
  int toll[] = {0, 1, 3, 6, 8, 8, 9, 12, 12, 13, 13, 15, 16};
  int result[13];
  fill(result, result + 13, -1);
  cout << toll_mem(toll, 5, result) << endl;
}
```

Na początku tablicę *result*[ ] wypełniamy wartością -1, która jako żywo nie może wystąpić jako wynik obliczeń. Magiczna liczba 13 to oczywiście rozmiar tablicy *toll*[ ] w naszym przykładzie.[¶]

Teraz piszemy funkcję `toll_mem()`:

```
int toll_mem(int toll[], int L, int result[])
{
  if(result[L] >= 0) return result[L];
  int profit;
  if(L == 0) profit = 0;
  else
  {
    profit = -1;
    for(int i = 1; i <= L; i++)
      profit = max(profit, toll[i] + toll_mem(toll, L - i, result));
  }
  result[L] = profit;
  return profit;
}
```

Jeśli zatem przyjdzie nam obliczyć nową wartość opłaty (zmienna *profit*), wtedy wpisujemy ją w odpowiednie miejsce w tablicy *result*[ ] i na następny raz – jeśli zajdzie potrzeba – będzie jak znalazł.

Uważny czytelnik może mieć (przynajmniej) trzy wątpliwości: po pierwsze, dlaczego takie tablice jak *toll*[ ] czy *result*[ ] nie są globalne, co uprościłoby odwołania do funkcji. Już pisaliśmy o tym, że należy raczej unikać zmiennych globalnych, gdyż wtedy pojawia się ryzyko, że jakaś funkcja zmieni w niekontrolowany sposób ich wartość (na przykład na skutek błędu literowego w nazwie zmiennej).

Po drugie, czy używanie tablicy jako argumentu funkcji nie obniża jej efektywności? Otóż nie, gdyż w takiej sytuacji do wywoływanej funkcji przekazywany jest jedynie wskaźnik do początku obszaru zajmowanego przez tablicę, a to przy obecnych architekturach komputerów oznacza przekaz zaledwie 4 bajtów. To pikuś!

[¶]Istnieje sprytny sposób, aby nie wpisywać konkretnego rozmiaru w deklaracji tablicy *result*[ ], a mianowicie można ją zadeklarować tak: `int result[sizeof(toll)/sizeof(int)];`. Operator `sizeof` zwraca rozmiar danej struktury lub typu danych, wyrażony w bajtach.

Po trzecie, czy funkcja `toll_mem()` może istotnie zmienić zawartość tablicy w nadrzędnej funkcji (tutaj: `main()`)? Odpowiedź jest twierdząca, patrz punkt drugi. Skoro funkcja „wie", gdzie jest tablica, wówczas może operować na jej oryginale (tutaj chodzi o tablicę *result*[ ]).

A skąd taki tytuł sekcji? *Metoda zstępująca* polega na rozważaniu podproblemów od największych w dół do najmniejszych. Potem następuje „zwijanie" rekurencji z powrótem do góry. A jaka jest szacunkowa złożoność tej metody? Ilość obrotów pętli `for` w kolejnych wywołaniach funkcji układa się w szereg arytmetyczny, którego łączna suma jest rzędu L^2. W porównaniu z wykładniczą złożonością metody naiwnej mamy więc zasadniczy postęp.

Metoda wstępująca

Podejście w tej metodzie jest nieco inne: zaczniemy od rozpracowania najkrótszych odcinków i w oparciu o spamiętane wyniki cząstkowe będziemy stopniowo budować rozwiązania coraz większych podproblemów. Co ciekawe, funkcja, którą napiszemy, nie będzie w ogóle rekurencyjna! (Nazwiemy tę funkcję `toll_up()`.)

Program główny będzie praktycznie taki sam, przy czym nawet nie musimy inicjalizować tablicy *result*[ ], poza jej zerową komórką:

```
int main()
{
  int toll[] = {0, 1, 3, 6, 8, 8, 9, 12, 12, 13, 13, 15, 16};
  int result[13];
  result[0] = 0;
  cout << toll_up(toll, 5, result) << endl;
}
```

A oto i sama funkcja obliczająca maksymalną opłatę dla drogi o długości L:

```
int toll_up(int toll[], int L, int result[])
{
  for(int p = 1; p <= L; p++)
  {
    int profit = -1;
    for(int i = 1; i <= p; i++)
      profit = max(profit, p[i] + result[p - i]);
    result[p] = profit;
  }
  return result[L];
}
```

Zmienna p oznacza rozmiar podproblemu, czyli długość odcinka. Jak widać, zaczynamy od $p = 1$ i idziemy stopniowo do góry, zapisując najwyższą opłatę w tablicy *result*[ ]. Przy kolejnym obiegu pętli po p wszystkie krótsze odcinki mamy już wycenione, więc nie musimy odwoływać się rekurencyjnie.

Złożoność metody wstępującej jest taka sama, jak metody zstępującej (czyli kwadratowa), z tym, że odwołania rekurencyjne kosztują co nieco (czasu i pamięci), więc mamy tu remis ze wskazaniem na metodę nierekurencyjną.

Jaki podział wybrać?

OK, wiemy już na ile maksymalnie możemy skasować kierowcę za przejechanie drogi o długości L, ale królewscy urzędnicy domagają się konkretnej informacji, w których miejscach postawić bramki na trasie. Odpowiedź na to pytanie nie musi być jednoznaczna, co widać choćby we wspomnianym przykładzie dla $L = 5$: obydwa podziały $2 + 3$ oraz $3 + 2$ dadzą ten sam wynik. (Pewność możemy mieć jedynie co do największej kwoty do zapłaty.)

Urzędnikom wystarczy wskazanie jednego przykładowego optymalnego podziału. W tym celu nieco wzbogacimy funkcję `toll_up()`: dla każdego podproblemu znajdziemy odpowiadający mu punkt optymalnego podziału. (Nazwiemy tę funkcję `toll_point()`.) Wypisaniem punktów podziału zajmie się funkcja `print_toll_point()`.

Zaczniemy od głównego programu, w którym dołożymy tablicę $point[\,]$:

```
int main()
{
  int toll[] = {0, 1, 3, 6, 8, 8, 9, 12, 12, 13, 13, 15, 16};
  int result[13], point[13];
  result[0] = 0;
  cout << toll_point(toll, 5, result, point) << endl;
  print_toll_point(5, point);
}
```

Funkcja `toll_point()` prezentuje się następująco:

```
int toll_point(int toll[], int L, int result[], int point[])
{
  for(int p = 1; p <= L; p++)
  {
    int profit = -1;
    for(int i = 1; i <= p; i++)
      if(profit < p[i] + result[p - i]))
      {
        profit = p[i] + result[p - i]);
        point[p] = i;
      }
    result[p] = profit;
  }
  return result[L];
}
```

Tym razem musieliśmy zrezygnować z wygodnej funkcji `max()`, gdyż jeśli zachodzi warunek w instrukcji warunkowej, wtedy musimy wykonać dwie operacje: uaktualnienie wartości zmiennej $profit$ oraz zapisanie punktu podziału w zmiennej $point[p]$. Zauważmy, że może to zachodzić wiele razy, więc ostatecznie $point[p]$ będzie zawierać punkt podziału dla maksymalnego $profit$-u (i o to chodzi).

Jeszcze została nam do napisania funkcja wypisująca punkty wyznaczające optymalny podział drogi:

```
void print_toll_point(int L, int point[])
{
  for(int p = L; p > 0; p -= point[p])
    cout << point[p] << ' ';
  cout << endl;
}
```

Pierwsza wypisana liczba to punkt podziału całej drogi o długości L, potem posuwamy się w kierunku coraz krótszych odcinków.

To jeszcze nie koniec na temat programowania dynamicznego, mamy jeszcze sporo do obgadania.