Wstęp: obliczanie liczb Fibonacciego

Rozważmy problem obliczania liczb Fibonacciego. Pierwsze dwie liczby Fibonacciego są równe 1, a każda kolejna jest sumą dwóch poprzednich. Sekwencja liczb Fibonacciego: (0,) 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, ...

Kod do obliczania liczb Fibonacciego wprost z definicji:

```
FIBO-RECUR(n)
1  if n < 1                    // for completeness
2        return 0
3  elseif n == 1
4        return 1
5  else return FIBO-RECUR(n − 2) + FIBO-RECUR(n − 1)
```

Liczby Fibonacciego nie rosną jakoś astronomicznie szybko, jednak ich obliczanie przy użyciu powyższej procedury okazuje się bardzo nieefektywne.

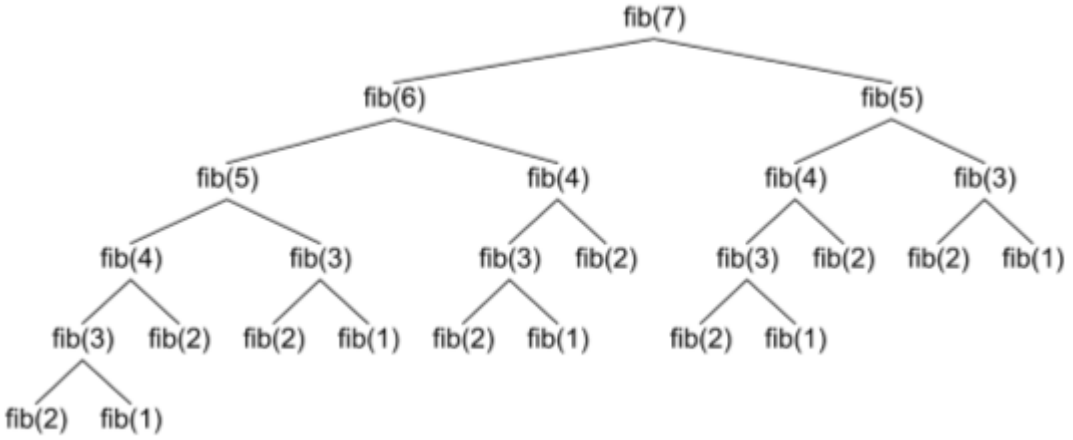
Liczby Fibonacciego — podwójna rekurencja

Przykładowe obliczenia:

```
% time python3 fibo_recur.py 40
Computing 40th Fibonacci number ...
 = 102334155
real: 0m12.930s, user: 0m12.921s, sys: 0m0.009s

% time python3 fibo_recur.py 50
Computing 50th Fibonacci number ...
 = 12586269025
real: 43m5.903s, user: 43m5.828s, sys: 0m0.004s
```

Dlaczego obliczenie 50-tej kolejnej liczby Fibonacciego zajmuje tak dużo czasu CPU? Przyczyną jest podwójna rekurencja. Niewinnie wyglądające podwójne wywołanie rekurencyjne generuje lawinę wywołań, z których większość jest powtórzeniami:



Liczby Fibonacciego — rozwiązanie iteracyjne

Oczywiście tej podwójnej rekurencji możnaby łatwo uniknąć zamieniając algorytm **rekurencyjny** na **iteracyjny**:

FIBO-ITER(n)

```
1  if n < 1                    // for completeness
2        return 0
3  elseif n == 1
4        return 1
5  fib2 = 0
6  fib1 = 1
7  for i = 2 to n
8        fib0 = fib1 + fib2
9        fib2 = fib1
10       fib1 = fib0
11 return fib0
```

```
% time python3 fibo_iter.py 50
Computing 50th Fibonacci number ...
 = 12586269025
real: 0m0.028s, user: 0m0.016s, sys: 0m0.012s
```

Ale teraz możemy naprawdę poszaleć:

```
% time python3 fibo_iter.py 500
Computing 500th Fibonacci number ...
= 139423224561697880139724382870407283950070256587697307264108962948325571 6\
22863290691557658876222521294125
real: 0m0.028s, user: 0m0.020s, sys: 0m0.008s

% time python3 fibo_iter.py 5000
Computing 5000th Fibonacci number ...
 = 3878968454388325633701916308325905312082127714646245106160597214895550136\
90440370970108229164622106694792934528588829738134831020089549829403614301 5\
69114789383642165639441069102145056341337065586562382546567007125259299038 5\
49338139288363783475189087629707120333370529231076930085180938498018038478 1\
39967488817655546537882916442689129803846137789690215022930824756663462249 2\
30718833248032803750391303529033045058427011476352422702109346376991040067 1\
41748832984228914912731040543287532980442736768229772449877498745556919077 0\
38806370468327948113589737399931101062193081490185708153978543791953056175 1\
07610530756887837660336673554452588448862416192105534574936758978490279882 3\
43510235998446639348532564119522218595630604753646454707603309024208063825 8\
49291564528762915757591423438091423029174910889841552098544324865940797935 7\
13168416928680395453095453886981146650820668628974206393234384884652409887 4\
23958738019769938203171742089322654688793640026307977800587591296713896342 1\
42525791168727556003603113705477547246046399875880469851784086743828631 25
real: 0m0.029s, user: 0m0.013s, sys: 0m0.016s
```

Liczby Fibonacciego — *spamiętywanie*

Rozwiązanie iteracyjne ma nie tylko przewagę polegającą na uniknięciu lawiny podwójnych rekurencji, ale ogólnie rozwiązania rekurencyjne wprowadzają w wykonywaniu programów dodatkowe narzuty na wywołania procedur, a w niektórych przypadkach długi łańcuch wywołań rekurencyjnych może przepełnić zakres pamięci zarezerwowany dla stosu.

Jednak rozwiązania rekurencyjne mają pewne zalety — bardzo często są bardziej czytelne i prostsze, czego obliczanie liczb Fibonacciego jest dobrym przykładem. Często gotowi bylibyśmy pogodzić się z dodatkowym narzutem wywołań rekurencyjnych, i przynajmniej na początkowym etapie eksperymentalnego uruchamiania programu posłużyć się czytelnym i prostym rozwiązaniem rekurencyjnym.

W przypadku liczb Fibonacciego nie sam mechanizm rekurencji był problemem, ale ich lawinowe duplikowanie. Jednak można uniknąć tego lawinowego duplikowania przez prosty zabieg polegający na zapamiętywaniu już raz obliczonych liczb Fibonacciego w tablicy globalnej, i unikaniu dublowanych obliczeń przez odczytanie rozwiązań z tablicy. Takie rozwiązanie nazywane jest **spamiętywaniem** (ang. *memoization*).

FIBO-RECUR-MEMO(n)

```
1  if n < 1                          // for completeness
2        return 0
3  elseif n == 1
4        return 1
5  elseif n in fibotab
6        return fibotab[n]
7  fib2 = FIBO-RECUR-MEMO(n − 2)
8  fib1 = FIBO-RECUR-MEMO(n − 1)
9  fibotab[n] = fib1 + fib2
10 return fibotab[n]
```

```
% time python3 fiboe_recur_memo.py 50
Computing 50th Fibonacci number ...
 = 12586269025
real: 0m0.029s, user: 0m0.020s, sys: 0m0.008s

% time python3 fiboe_recur_memo.py 500
Computing 500th Fibonacci number ...
 = 139423224561697880139724382870407283950070256587697307264108962948325571\
622863290691557658876222521294125
real: 0m0.029s, user: 0m0.017s, sys: 0m0.013s
```

```
% time python3 fiboe_recur_memo.py 50000
Computing 50000th Fibonacci number ...
 = 10777734893072974780279038855119480829625106769411579784902309210032744\
7
353646523049848844402047602984931943328327405495330753981733048306741483538\
71755545405198446200873464249380723258213016701908119882516186149595860854\
0
993737510653044874463782996851389325663668163313173204591893189886313559961\
26556155463897640305571514053979226012432273048290007169088637862067551770\
0
832269328087849866274058836537593758274508704744192976808834961311297128859\
...
013608175009331914291885808751962605458474604194206257224753676742372629234\
67763105426068549719178378668819786805212576177264040949511215576188269822\
3
66838153968218686762926290755720567510373245164756842944423699212491240487\
4
64281580686750806724451064512444192234336251813764582803376461209571993619\
7
36455646214921063358870308182304266593049366953768037220397037490781969011\
1
26652402029761830536425237355312\
5
real: 0m0.088s, user: 0m0.030s, sys: 0m0.056s
```

W powyższym wyniku 50000-ta liczba Fibonacciego ma 10450 cyfr i przy standardowych ustawieniach interpretera Pythona jej obliczenie przepełnia pojemność stosu wywołań procedur, oraz standardowe ustawienia konwersji liczb nieograniczonej precyzji na stringi. Ale oczywiście te ustawienia można powiększyć, i zasadniczo 50000 wywołań funkcji nie jest niczym przerażającym dla współczesnego komputera, czego dowodzi całkowity czas obliczeń poniżej 90 milisekund.

Liczby Fibonacciego — jeszcze raz rozwiązanie iteracyjne

Ideę spamiętywania można również wykorzystać w rozwiązaniu iteracyjnym:

```
FIBO-ITER-MEMO(n)
1  if n < 1
2        fibotab[0] = 0
3  elseif n == 1
4        fibotab[1] = 1
5  for i = 2 to n
6        fibotab[i] = fibotab[i − 1] + fibotab[i − 2]
7  return fibotab[n]
```

To rozwiązanie niekoniecznie jest dobre pod względem informatycznym, ponieważ zamienia kilka operacji przypisania do dwóch dodatkowych zmiennych na tablicę zawierającą wszystkie kolejne liczby Fibonacciego.

Jednak ma ono istotną zaletę — wykorzystuje wprost wzór definicyjny na liczbę Fibonacciego i nie wymaga jego konwersji na procedurę iteracyjną. Jak wkrótce zobaczymy, nie dla każdego zagadnienia taka konwersja jest równie prosta jak w tym przypadku. Zastosowanie spamiętywania bardzo ułatwia implementację tego podejścia.

Z kodu algorytmu wynika również jasno, że działa on w czasie $\Theta(n)$.

Inny problem - rozkrój pręta

Aby lepiej przybliżyć się do idei programowania dynamicznego rozważmy teraz pewne zagadnienie praktyczne. Chodzi o optymalny rozkrój pręta stalowego. Załóżmy, że pręty stalowe dostępne są hurtowo w pewnej standardowej długości, i rozważamy opcje sprzedawania ich w całości, lub po pocięciu na mniejsze odcinki. Przyjmując, że znamy ceny handlowe wszystkich możliwych długości pręta, i że możemy dowolne cięcia wykonać tanio (dokładnie przyjmujemy koszt cięcia równy zero), zależy nam na maksymalizacji zysku, czyli sumy wartości prętów pociętych minus cena oryginalnego pręta pełnej długości. Jest możliwe, że cena pręta pełnej długości jest na tyle duża, że nie opłaca się w ogóle go ciąć.

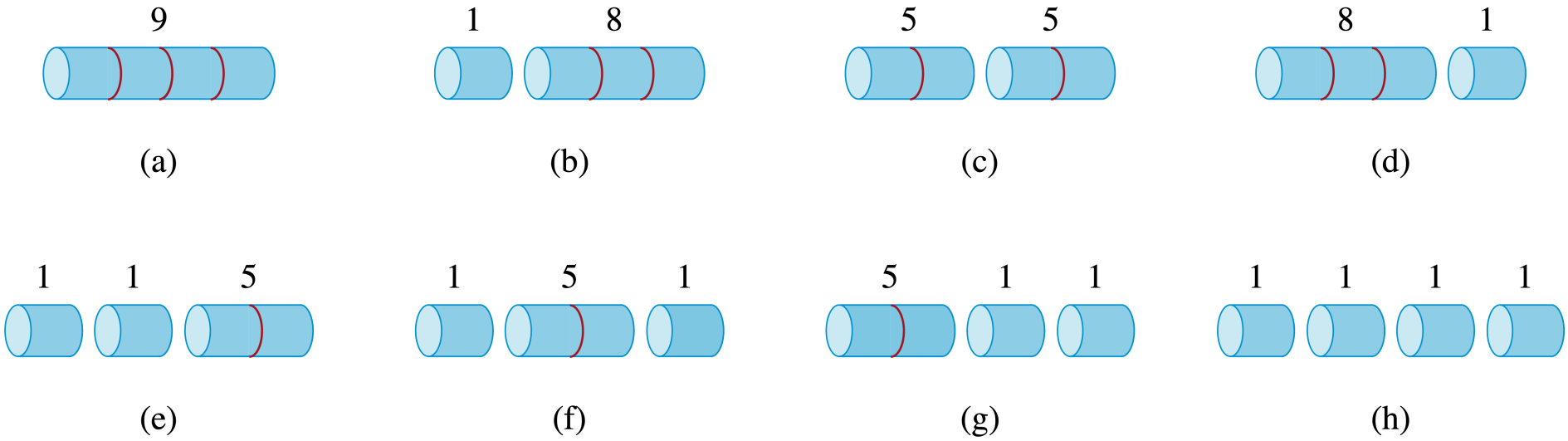
Dokładniej, mając pręt długości n i tabelę cen p_i dla $i = 1, 2, \ldots, n$ wyznacz maksymalną wartość r_n rozkroju tego pręta i sprzedaży w kawałkach całkowitoliczbowej długości.
Dla pewnych cen możliwe jest rozwiązanie bez rozkroju, to znaczy $r_n = n$.

Ponieważ w każdym punkcie i pomiędzy 1 a $n - 1$ mamy możliwość cięcia lub niecięcia, tych możliwości razem jest 2^{n-1}. Zapisując rozwiązania jako sumy długości kawałków, jeśli optymalnym rozwiązaniem jest pocięcie pręta na k kawałków długości $n = i_1 + i_2 + \cdots + i_k$ to wartość tego rozwiązania jest $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$.

Rozkrój pręta — przykład

Przykładowa tabela cen i poniżej opcje rozkroju pręta o długości 4 z wartościami poszczególnych kawałków:

długość i	1	2	3	4	5	6	7	8	9	10
cena p_i	1	5	8	9	10	17	17	20	24	30



Jak widać na rysunku, optymalny rozkrój uzyskujemy w przypadku (c) o wartości $r_4 = 10$.

Rozkrój pręta — analiza

Ponownie przyjmując przykładową tabelę cen rozkroju:

długość	i	1	2	3	4	5	6	7	8	9	10
cena	p_i	1	5	8	9	10	17	17	20	24	30

możemy obliczyć optymalne rozwiązania dla różnych wyjściowych długości pręta:

$$
\begin{aligned}
r_1 &= \text{1 dla rozwiązania } 1 = 1 \text{ (bez cięcia)} \\
r_2 &= \text{5 dla rozwiązania } 2 = 2 \text{ (bez cięcia)} \\
r_3 &= \text{8 dla rozwiązania } 3 = 3 \text{ (bez cięcia)} \\
r_4 &= \text{10 dla rozwiązania } 4 = 2 + 2 \\
r_5 &= \text{13 dla rozwiązania } 5 = 2 + 3 \\
r_6 &= \text{17 dla rozwiązania } 6 = 6 \text{ (bez cięcia)} \\
r_7 &= \text{18 dla rozwiązania } 7 = 1 + 6 \text{ lub } 7 = 2 + 2 + 3 \\
r_8 &= \text{22 dla rozwiązania } 8 = 2 + 6 \\
r_9 &= \text{25 dla rozwiązania } 9 = 3 + 6 \\
r_{10} &= \text{30 dla rozwiązania } 10 = 10 \text{ (bez cięcia)}
\end{aligned}
$$

Rozkrój pręta — dekompozycja na podproblemy

Ogólnie, możemy zapisać wartość rozwiązania dla długości n za pomocą rozwiązań dla krótszych odcinków:

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\}$$

Sens powyższego wzoru jest następujący: rozwiązanie optymalne wymaga 0 cięć (pierwszy wyraz we wzorze na maksimum), bądź wymaga co najmniej jednego cięcia na kawałki długości i oraz $n - i$, i dalszego optymalnego rozkroju otrzymanych kawałków.

Inaczej mówiąc, możemy rozwiązać problem rozkroju dla wszystkich możliwych wariantów pierwszego cięcia (od 1 do $n - 1$ w powyższym wzorze na max), i mając wartości ich optymalnych rozwiązań, wybrać optymalną sumę, lub brak cięcia (pierwszy wyraz p_n w powyższym wzorze).

W takiej sytuacji, gdy optymalne rozwiązanie problemu zawiera w sobie optymalne rozwiązania podproblemów, które można rozwiązać niezależnie, mówimy, że problem ma **optymalną podstrukturę**.

Rozkrój pręta — pseudokod rozwiązania

Dekompozycję problemu cięcia na podproblemy, wynikającą z powyższego wzoru, i wymagającą rozwiązania kaskady podproblemów, możemy nieco uporządkować i uprościć, uwzględniając pierwsze cięcie na odcinek długości i, oraz resztę pręta $n - i$, dla którego trzeba znaleźć optymalne rozwiązanie r_{n-i}:

$$r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\}$$

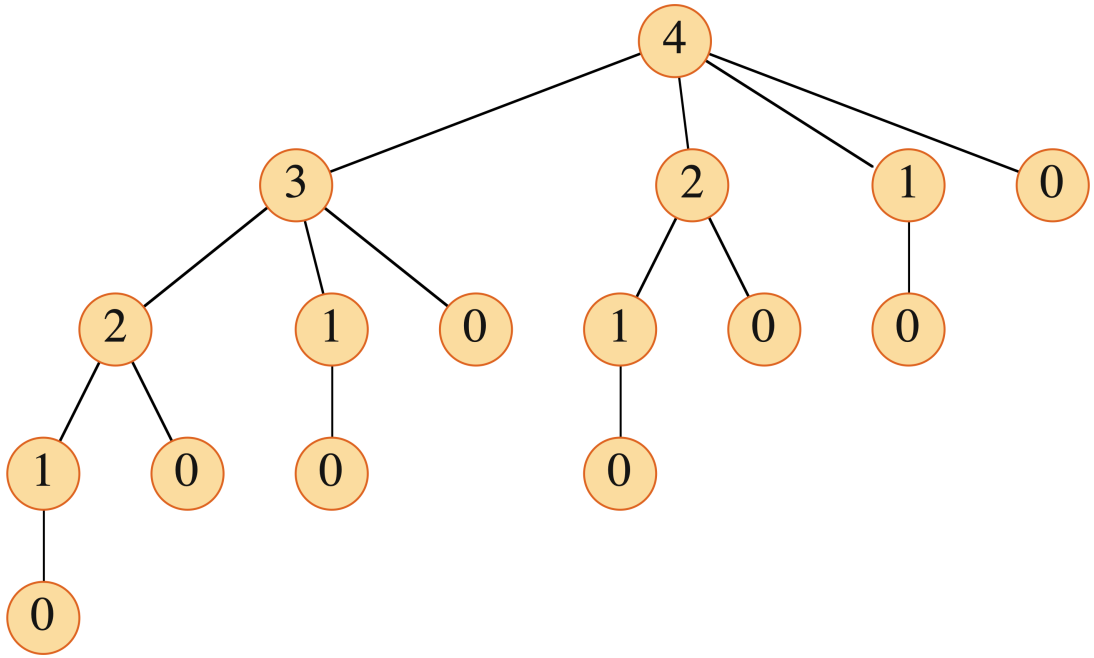
Przypadek bez cięcia otrzymujemy dla $i = n$ i r_0.

W tym ujęciu optymalne rozwiązanie uzyskujemy po podziale i rozwiązaniu tylko jednego podproblemu. Możemy to wyrazić pseudokodem:

```
CUT-ROD(p, n)
1  if n == 0
2        return 0
3  q = -∞
4  for i = 1 to n
5        q = max{q, p[i] + CUT-ROD(p, n - i)}
6  return q
```

Pseudokod procedury CUT-ROD wygląda niewinnie, jakby z pojedynczym wywołaniem rekurencyjnym, ale niestety, to wywołanie znajduje się w pętli, a więc naprawdę jest to lawina wywołań rekurencyjnych.

Drzewo wywołań rekurencyjnych dla $n = 4$:



Próba użycia tego rozwiązania dla wartości $n \geq 40$ okazuje się bardzo nieefektywna.
Czy to nam się z czymś kojarzy? Fibonacci?
Spamiętywanie na pomoc?

Programowanie dynamiczne — *podejście zstępujące*

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0 : n] be a new array        // will remember solution values in r
2  for i = 0 to n
3       r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0                        // already have a solution for length n?
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n                // i is the position of the first cut
7            q = max{q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r)}
8  r[n] = q                           // remember the solution value for length n
9  return q
```

To rozwiązanie łączy rekurencję ze spamiętywaniem, i zastępuje pierwotne rozwiązanie o eksponencjalnym czasie działania $\Theta(2^n)$ rozwiązaniem o czasie kwadratowym $\Theta(n^2)$.

Programowanie dynamiczne — *podejście wstępujące*

Możliwe jest również podejście alternatywne. Zamiast dzielić i rządzić, możemy zastosować podejście analogiczne do iteracyjnego obliczania liczb Fibonacciego, i skonstruować kompletną tabelę $r[0:n]$ rozwiązań optymalnych wszystkich długości pręta aż do n, i na końcu zwrócić $r[n]$. Rozwiązanie jest proste ponieważ ponownie wykorzystuje spamiętywanie:

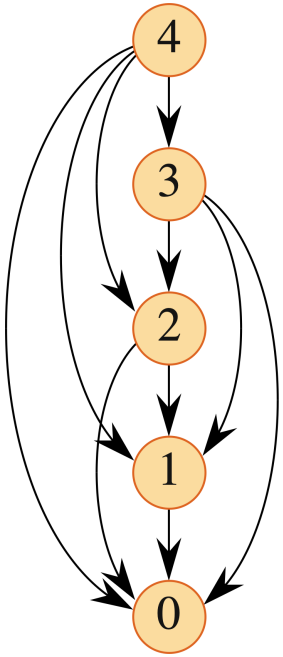
```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0:n] be a new array        // will remember solution values in r
2  r[0] = 0
3  for j = 1 to n                   // for increasing rod length j
4       q = -∞
5       for i = 1 to j              // i is the position of the first cut
6            q = max{q, p[i] + r[j - i]}
7       r[j] = q                    // remember the solution value for length j
8  return r[n]
```

W tym rozwiązaniu widać wyraźnie jego $\Theta(n^2)$ czas działania.

Programowanie dynamiczne — podsumowanie

W programowaniu dynamicznym ważne jest jasne zdefiniowanie podproblemów, których rozwiązania składają się na rozwiązanie problemu podstawowego, oraz zależności pomiędzy podproblemami i ich rozwiązaniami.

Te podproblemy i ich zależności można przedstawić na **grafie podproblemów**. Rysunek przedstawia graf podproblemów dla rozkroju pręta przy $n = 4$. Węzły grafu odpowiadają podproblemom, a łuki skierowane łączą jeden podproblem z innym podproblemem, którego rozwiązanie wpływa na rozwiązanie pierwszego podproblemu. W podejściu zstępującym (*top-down*) rozwiązując ten pierwszy podproblem rekurencyjnie wywołujemy rozwiązanie drugiego podproblemu. Graf jest w istocie spłaszczonym drzewem wywołań rekurencyjnych, które przez spamiętywanie zostały zredukowane do pojedynczego wywołania.



W podejściu wstępującym (*bottom-up*) łuk dla każdego podproblemu wskazuje inne podproblemy, które muszą być rozwiązane zanim będzie możliwe skonstruowanie rozwiązanie pierwotnego problemu.

Rozkrój pręta — zapomniany szczegół

W problemie rozkroju pręta pozostał jeden szczegół do uzupełnienia. Przedstawione algorytmy obliczają jedynie koszt optymalnego rozwiązania, ale nie dają wskazówek jak kroić pręt. Można w tym celu uzupełnić procedurę BOTTOM-UP-CUT-ROD, aby generowała nie tylko koszty, ale również odpowiadające im miejsce (pierwszego) cięcia (bo tak działa uproszczony rekurencyjny schemat rozwiązania problemu rozkroju):

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$		1	2	3	2	2	6	1	2	3	10

Aby uzyskać miejsca kolejnych cięć należy obliczyć długość reszty pręta pozostałej po pierwszym cięciu, i dla niej odnaleźć w tabelce miejsce kolejnego „pierwszego" cięcia.

Na przykład, dla $n = 7$ pierwsze cięcie wypada w miejscu $i = 1$, a dla reszty pręta długości $n = 6$ miejscem cięcia jest $i = 6$, co oznacza brak cięcia.

Uzupełniona procedura BOTTOM-UP-CUT-ROD jak również pomocnicza procedura wyświetlająca wszystkie miejsca cięcia przedstawione są poniżej.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0 : n] and s[1 : n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n                    // for increasing rod length j
 4        q = −∞
 5        for i = 1 to j              // i is the position of the first cut
 6              if q < p[i] + r[j − i]
 7                    q = p[i] + r[j − i]
 8                    s[j] = i        // best cut location so far for length j
 9        r[j] = q                    // remember the solution value for length j
10  return r and s

PRINT-CUT-ROD-SOUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3        print s[n]              // cut location for length n
4        n = n − s[n]            // length of the remainder of the rod
```

Optymalne drzewa przeszukiwań BST

Jako przykład nieco trudniejszego algorytmu, wykorzystującego programowanie dynamiczne, rozważymy algorytm budowania optymalnych drzew przeszukiwań BST. Dla przypomnienia, **optymalnym drzewem przeszukiwań** dla sekwencji kluczy $k_1, k_2, \ldots, k_n$, ze znanym rozkładem prawdopodobieństw ich wyszukiwania $p_1, p_2, \ldots, p_n$, nazywamy binarne drzewo przeszukiwań, które zapewnia minimalną oczekiwaną liczbę kroków przeszukiwania spośród wszystkich możliwych binarnych drzew przeszukiwań (uporządkowanych) zawierających te klucze.

Ta definicja wymaga pewnego uzupełnienia. Chcemy dopuścić możliwość, by na drzewie wykonywane były również wyszukiwania wartości nienależących do sekwencji kluczy (czyli wyszukiwań zakończonych porażką), i znamy również rozkład prawdopodobieństw takich wyszukiwań. Chcemy by drzewo było optymalne również dla wyszukiwania tych wartości, które nazwiemy tutaj **niekluczami** (ang. *dummy key*; w polskim tłumaczeniu podręcznika CLRS nieklucze nazywane są kluczami-imitacjami).

Nieklucze nie mają konkretnych wartości, jak klucze, np. liczbowych. Dla sekwencji n kluczy $k_1, k_2, \ldots, k_n$ przyjmujemy odpowiednią sekwencję niekluczy $d_0, d_1, d_2, \ldots, d_n$. Nieklucz d_0 reprezentuje wszystkie wartości mniejsze od klucza k_1, nieklucz d_n reprezentuje wszystkie wartości większe od k_n, natomiast dla $(i = 1, 2, \ldots, n-1)$ każdy nieklucz d_i reprezentuje wartości pomiędzy k_i i k_{i+1}.

Z założenia, dla sekwencji n kluczy $k_1, k_2, \ldots, k_n$ znamy rozkład prawdopodobieństwa ich wyszukiwań dany sekwencją prawdopodobieństw $p_1, p_2, \ldots, p_n$. Ale dla tej sekwencji kluczy mamy wynikającą z niej sekwencję $n+1$ niekluczy $d_0, d_1, d_2, \ldots, d_n$ i odpowiadający jej rozkład prawdopodobieństw wyszukiwania odpowiednich niekluczy, które oznaczymy $q_0, q_1, q_2, \ldots, q_n$. To znaczy, prawdopodobieństwo wyszukania nieklucza d_i wynosi q_i. A ponieważ każde wyszukiwanie odnosi się albo do jednego z kluczy, albo do jednego z niekluczy, to mamy:

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Wykorzystując znane rozkłady prawdopodobieństw możemy obliczyć oczekiwany koszt wyszukiwania w danym drzewie T. Zakładamy, że rzeczywisty koszt dowolnego wyszukiwania jest równy liczbie sprawdzonych wierzchołków drzewa, który jest równy głębokości znalezionego wierzchołka w drzewie, plus 1. Na przykład, gdy poszukiwany (i znaleziony) był korzeń drzewa, na głębokości 0, to koszt wyszukiwania jest 1.

$$\begin{aligned} E_T[\text{koszt wyszuk.}] &= \sum_{i=1}^{n} (\mathsf{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\mathsf{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \mathsf{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathsf{depth}_T(d_i) \cdot q_i \end{aligned}$$

Własności optymalnego drzewa przeszukiwań BST

Zauważmy najpierw, że w wynikowym drzewie klucze powinny być wewnętrznymi węzłami, a nieklucze liśćmi. Wynika to z faktu, że nieklucze nie mają konkretnych wartości, ich jakby nie ma w drzewie, one tylko wyłapują nieudane poszukiwania.

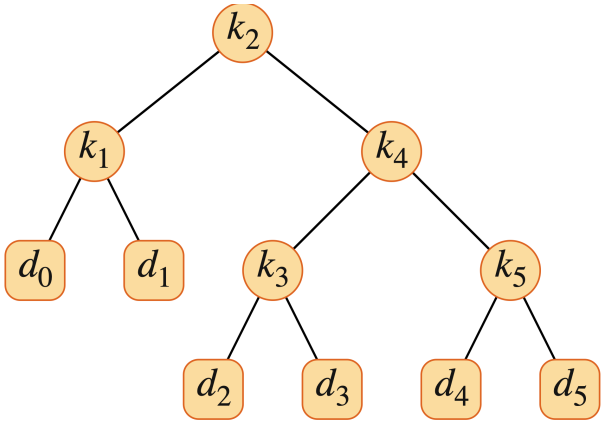
Następnie zauważmy, że problem budowy optymalnego BST ma optymalną podstrukturę, to znaczy, jeśli istnieje optymalne BST T dla pewnego zestawu kluczy, i zawiera ono w sobie poddrzewo T' zakorzenione w jakimś węźle k_r i zawierające klucze $k_i, \ldots, k_j$ i nieklucze $d_{i-1}, \ldots, d_j$, to poddrzewo T' musi być optymalne dla tego zestawu kluczy i niekluczy. Albowiem gdyby nie było ono optymalne, a optymalne było inne poddrzewo T'' z tymi samymi kluczami i niekluczami, o niższym oczekiwanym czasie wyszukiwania, to możnaby tym poddrzewem T'' zastąpić T' w drzewie T i w ten sposób zmniejszyć oczekiwany czas wyszukiwania w drzewie T.

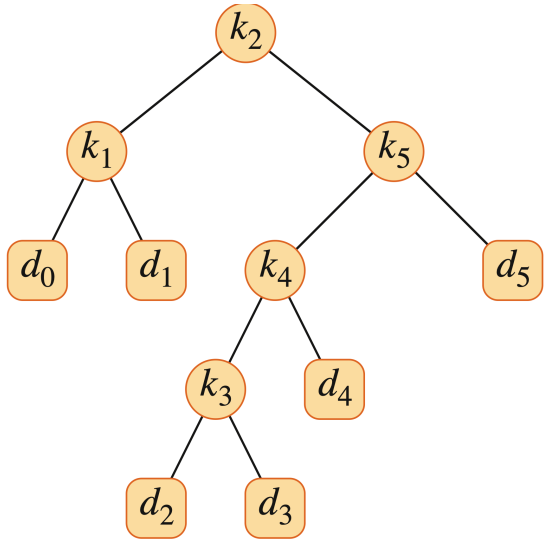
Zwróćmy jeszcze uwagę na pewną cechę poddrzew „pustych". Gdyby korzeniem (pod)drzewa zawierającego klucze $k_i, \ldots, k_j$ był klucz k_i, to wszystkie klucze tego poddrzewa znajdowałyby się w jego prawym poddrzewie, a lewe poddrzewo zawierałoby 0 kluczy, a więc było „puste". Jednak zawierałoby ono wtedy jeden nieklucz d_{i-1}.

A więc możemy o poddrzewach zawierających 0 kluczy myśleć jako o „pustych", jednak będą one w istocie zawierały jeden liść z niekluczem.

Przykład dla $n = 5$ kluczy:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10





node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

Budowa optymalnego drzewa przeszukiwań BST

Chcemy skonstruować algorytm budowy optymalnego drzewa BST za pomocą budowy jego optymalnych poddrzew. Podproblemem jest budowa drzewa optymalnego dla sekwencji kluczy $k_i, \ldots, k_j$, gdzie $i \geq 1, j \leq n, j \geq i-1$ (przypadek $j = i-1$ jest przypadkiem pustego poddrzewa).

Oznaczamy przez $e[i, j]$ oczekiwany koszt wyszukiwania w optymalnym drzewie zawierającym klucze $k_i, \ldots, k_j$. Algorytm obliczy pełną tablicę $e[i, j]$ dla $j \geq i-1, i = 1, \ldots, n+1, j = 0, \ldots, n$ (rozszerzone zakresy wartości uwzględniają sekwencje puste kluczy, generujące „puste" poddrzewa z pojedynczymi niekluczami).

Skrajne przypadki „pustych" poddrzew uzyskujemy dla każdej pary $j = i-1$ gdzie kosztem przeszukiwania jest prawdopodobieństwo dostępu do nieklucza, czyli $e[i, i-1] = q_{i-1}$.

Dla $j \geq i$ będziemy rozważali wszystkie możliwe korzenie k_r poddrzewa dla sekwencji $k_i, \ldots, k_j$, gdzie lewe poddrzewo będzie zawierało klucze $k_i, \ldots, k_{r-1}$, a prawe poddrzewo klucze $k_{r+1}, \ldots, k_j$.

Dodatkowo będziemy wykorzystywali tablicę sum prawdopodobieństw:

$$w[i, j] = \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i$$

Dla obliczeń tablicy $e[i,j]$ można wyprowadzić wzór:

$$e[i,j] = \begin{cases} q_{i-1} & \text{gdy } j = i-1 \\ \min\{e[i,r-1] + e[r+1,j] + w(i,j) : i \le r \le j\} & \text{gdy } i \le j \end{cases}$$

Przedstawiony poniżej algorytm oblicza wszystkie wartości tablic $e[i,j]$ i $w[i,j]$ (w podanych wyżej zakresach indeksów), i po jego zakończeniu za rozwiązanie będziemy uważali wartość $e[1,n]$.

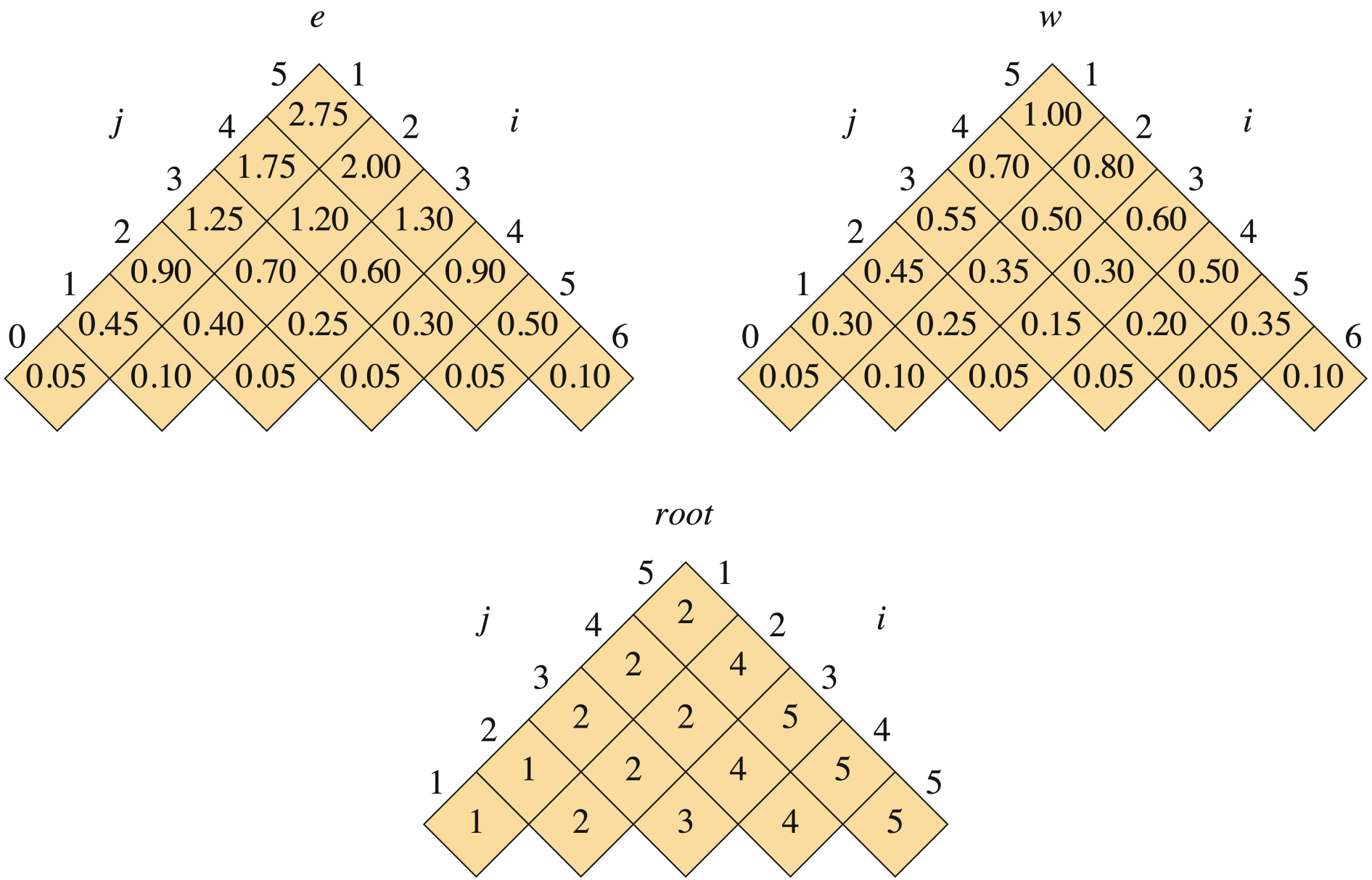
Jednak aby ułatwić odtworzenie optymalnego drzewa odpowiadającego temu rozwiązaniu, algorytm dodatkowo zbuduje tablicę $root[i,j]$ dla $i,j = 1,\ldots,n$, która w każdej pozycji zawierać będzie indeks r korzenia k_r optymalnego drzewa dla podsekwencji kluczy $k_i,\ldots,k_j$ (zatem główny korzeń całego optymalnego drzewa BST znajdziemy w pozycji $root[1,n]$).

Algorytm budowy optymalnych drzew BST

```
OPTIMAL-BST(p, q, n)
 1  let e[1 : n+1, 0 : n], w[1 : n+1, 0 : n], root[1 : n, 1 : n] be new tables
 2  for i = 1 to n+1                          // base cases
 3      e[i, i−1] = q_{i−1}
 4      w[i, i−1] = q_{i−1}
 5  for l = 1 to n
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          e[i, j] = ∞
 9          w[i, j] = w[i, j−1] + p_j + q_j
10          for r = i to j                    // try all possible roots r
11              t = e[i, r−1] + e[r+1, j] + w[i, j]    // see formula
12              if t < e[i, j]
13                  e[i, j] = t
14                  root[i, j] = r
15  return e, root
```

Jak widać, algorytm wykonuje trzy zagnieżdżone pętle w zakresie maksymalnie do n i zatem działa w czasie $\Theta(n^3)$.

Przykładowe drzewo BST — rozwiązanie



Krótkie podsumowanie — pytania sprawdzające

1. Napisz pseudokod procedury $\textsc{Display-Optimal-BST}(root, n)$, która mając daną tablicę $root$ wypisze strukturę optymalnego drzewa BST.

2. Wyznacz koszt wyszukiwania i strukturę optymalnego drzewa BST dla zestawu $n = 7$ kluczy z następującymi prawdopodobieństwami:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

Literatura i materiały pomocnicze

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, Clifford Stein: Wprowadzenie do algorytmów, PWN, 2024, rozdział 14.