Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Вятский государственный гуманитарный университет»

Дополнительная подготовка школьников по дисциплине «Информатика и информационные технологии»

Учебный модуль
Динамическое программирование

О. А. Пестов

Киров
2011

СОДЕРЖАНИЕ

1. Введение

В историческом плане понятие *динамическое программирование* введено Ричардом Беллманом (Richard Bellman) в 1950-х годах прошлого века и определяло раздел прикладной математики под названием *исследование операций*. Круг исследуемых задач был достаточно четко обозначен. Это методы поиска оптимальных (наилучших) решений в задачах управления системами, но с одной, если так можно выразиться, особенностью. Как изменение самой системы во времени, так и процесс управления системой, допускал разбивку по времени на этапы (шаги). Отсюда и термин «динамическое» – изменяющееся во времени. Но так (опять же) можно представить (описать) практически любую систему управления. Особенность динамического программирования заключалась в том, что допускалась разбивка процесса на фиксированные промежутки времени и, в целом, оптимальное решение задачи как бы складывалось из оптимальных решений на каждом из промежутков (этапе, шаге). Термин «программирование» (dynamic programming) в данном случае никоим образом не связан с разработкой программ для компьютера, так же как, например, и «линейное программирование» (linear programming). Он означает нечто другое, а именно, строго заданную последовательность операций (арифметических, логических) по нахождению оптимального решения. Фактически это алгоритм решения задачи.

В ходе дальнейшего развития, но уже не раздела прикладной математики, а информатики в целом с понятием динамическое программирование произошла некая трансформация. Оно очерчивает вполне определенный метод проектирования алгоритмов, который не ограничивается только задачами оптимизации. Естественно этот метод не универсален, он работоспособен для вполне определенного класса задач. В идее разбивки задачи на подзадачи и компоновки решения задачи из решений подзадач нет ничего нового – это универсальный метод и по-другому действовать нельзя. Смысл (суть) заложена в использованных терминах «разбивка» и «компоновка». Задача должна допускать разбивку на перекрывающиеся подзадачи того же вида (типа) так, чтобы её решение как бы складывалось, компоновалось из решений подзадач. Другими словами, должны быть выведены (определены) функциональные зависимости (рекуррентные соотношения) между решениями подзадач. Тогда, начиная, например, с небольших задач, мы постепенно получаем решения всё бо́льших задач и, наконец, доходим до решения исходной задачи.

Особенность динамического программирования как метода заключается и в том, что каждая подзадача решается только один раз. Результат её решения запоминается и затем, при решении следующих подзадач, используется как данное. Итак, два ключевых момента «связь» и «запоминание».

2. Простые задачи

2.1. Числа Л. Фибоначчи

Знаменитая последовательность чисел 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, … определяется тем, что каждое очередное число является суммой двух предыдущих. Обозначим n-е ($n \geq 0$) число в последовательности как F_n (в приведенном ряде чисел F_{10}=55). Тогда последовательность определяется следующим рекуррентным соотношением: $F_n = F_{n-2} + F_{n-1}$ ($n > 1$) и начальными значениями F_0=0, F_1=1.

Сформулируем задачу: по заданному числу n вычислить F_n.

Рекурсивный вариант решения задачи очевиден.

```
function fib(n : integer) : integer;
begin
  if (n < 2) then
    fib := n
  else
    fib := fib(n-1)+fib(n-2);
end;
```

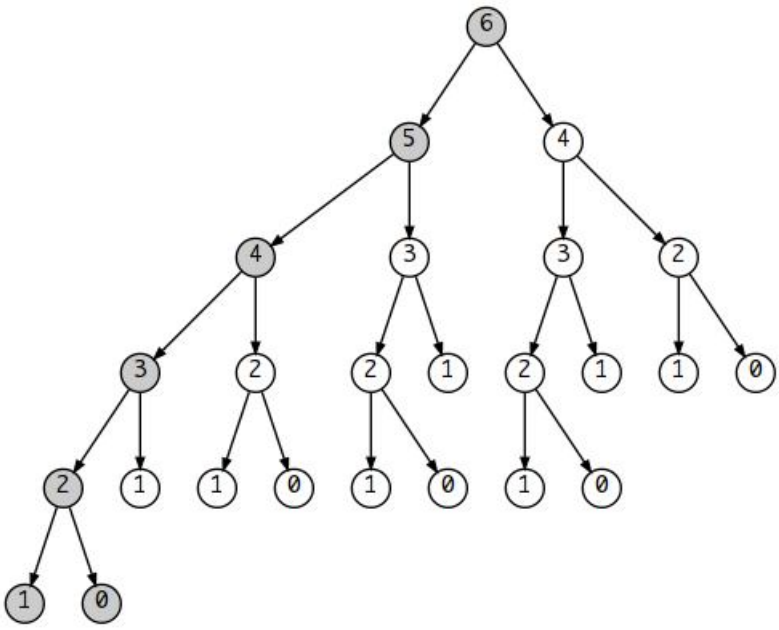

Рис.1. Последовательность вызова функций в процессе вычисления `fib(6)`.

Данный алгоритм имеет экспоненциальную временную сложность $O(2^n)$.

Рассмотрим другой вариант решения задачи. Определим понятие *подзадачи*. В данном случае это вычисление меньшего числа Фибоначчи. Для

каждого i, зная результат решения подзадач для i–1 и i–2, то есть числа F_{i-1} и F_{i-2} мы решаем очередную подзадачу, и так до значения n. При этом каждая подзадача решается только один раз. Другими словами, мы «проходим», только по «кружкам», выделенным темным цветом на рис. 1. Предположим, что результаты решения подзадач хранятся в массиве `F` (табл. 1).

Таблица 1. Значения первых чисел Фибоначчи

I	0	1	2	3	4	5	6	7	8	9	10	11	…
F[i]	0	1	1	2	3	5	8	13	21	34	55	89	…

Запись логики решения имеет следующий вид.

```
procedure fib(n : integer);
var i : integer;
begin
  f[0] := 0;
  f[1] := 1;
  for i := 2 to n do
    f[i] := f[i-1]+f[i-2];
  writeln(f[n]);
end;
```

Временная сложность вычислений O(n).

Итак, причем здесь динамическое программирование. Мы определили понятие подзадача, мы показали как из решения подзадач «складывается» решение исходной задачи. Каждую подзадачу мы решали только один раз, запоминая результаты в массиве. В результате этих действий получен выигрыш по времени и первоначальное решение с экспоненциальной временной сложность трансформировалось в решение с линейной сложностью.

2.2. Биномиальные коэффициенты или нахождение числа сочетаний

Действия, аналогичные действиям из предыдущего раздела, можно выполнять при вычислении числа сочетаний – способов выбрать k элементов из n ($n \geq 0$, $0 \leq k \leq n$). Известно следующее равенство $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$.

Из этого соотношения следует рекурсивная реализация.

```
function cnk(n, k : integer) : integer;
begin
  if (k = 0) or (n = k) then
    cnk := 1
  else
    cnk := cnk(n-1,k)+cnk(n-1,k-1);
end;
```

Прорисуем рекурсивные вызовы `cnk` при вычислении, например, C_5^3. Не выделенные темным цветом вершины дерева соответствуют повторным вызовам функции. Очевидно, что с ростом n и k количество повторных вызовов возрастает, и способ вычисления C_n^k становится неэффективным.

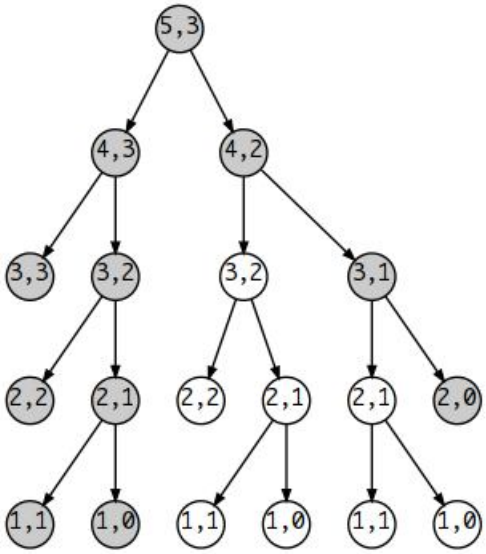

Рис. 2. Последовательность вызовов функции `cnk` при вычислении C_5^3

Другой способ решения опять же вытекает из приведенного равенства. Оно фактически описывает связь подзадач, если в последнее понятие трактовать как вычисление числа сочетаний для меньших значений n и k. Но каждая подзадача в этом случае решается один раз и результат её решения запоминается.

```
procedure cnk(n, k : integer);
var i, j : integer;
begin
  for i := 1 to n do
  begin
    c[i, 0] := 1;
    c[i, i] := 1;
```

```
      for j := 1 to i-1 do
        c[i, j] := c[i-1, j]+c[i-1,j-1];
    end;
    writeln(c[n, k]);
end;
```

Данный метод имеет временную сложность $O(n^2)$ и требует объем памяти $O(n^2)$. При этом он не совершенен в том смысле, что выполняет дополнительные вычисления, возможно ненужные, для заданных n и k. Например, если $n = 5$ и $k = 3$, то нет смысла находить C_5^2 или C_4^1. Тем не менее, они будут найдены.

2.3. Задача о Черепашке

Дана прямоугольная таблица, в клетках которой записаны целые числа. Черепашка находится в левой верхней клетке и ей необходимо попасть в правую нижнюю клетку. За один ход Черепашка может переместиться в соседнюю нижнюю или правую клетку. Требуется найти путь Черепашки с максимальной суммой элементов.

34	17	27	43	53
31	30	42	49	11
18	31	21	24	55
43	41	36	58	53

Рис. 3. Пример поля для Черепашки

Полный перебор вариантов – универсальный способ решения. Рассмотрим принципы реализации перебора и его потенциальные возможности. Пусть дана таблица 4×5 (рис. 3). Любой путь состоит из трех перемещений вниз и четырех перемещений вправо, то есть длина пути равна семи. Другими словами, дано семь позиций, из них три выбирается для перемещений вниз, оставшиеся четыре – для перемещений вправо. Количество способов выбора трех перемещений из семи C_7^3 определяет количество путей Черепашки. В общем случае – C_{n+m-2}^{n-1}.

Для рассматриваемого примера у Черепашки 35 путей. Однако их количество растет очень быстро с ростом размера таблицы. Так для поля 20×18 различных путей будет 8 597 496 600, а для поля 31×30 уже 59 132 290 782 430 712. Т. е. возможности полного перебора вариантов ограничены.

Примечание. Мы условно отождествляем задачу нахождения пути с задачей нахождения максимальной суммы, ибо, как будет показано, нахождение самого пути по сформированной сумме требует линейного времени.

Рассмотрим другой способ решения задачи. Определим подзадачу как ту же самую задачу, но для таблицы меньшего размера, и «свяжем» решения для подзадач с решением исходной задачи. Для таблицы 4×5 подзадачи – это решения для таблиц с размером 1×1, 1×2, 1×3, 1×4, 1×5, 2×1, 2×2, 2×3, 2×4, 2×5 и так далее до 4×4. Результаты решений запоминаем в отдельном массиве *F*.

34	51	78	121	174
65	95	137	186	197
83	126	158	210	265
126	167	203	268	321

Рис. 4. Результаты решений подзадач для примера на рис. 3 . В клетке (i, j) записано решение для подзадачи $i \times j$ – максимальная стоимость пути из левой верхней клетки до клетки (i, j).

Для таблиц 1×2, 1×3, 1×4, 1×5 (движение Черепашки вправо), стоимость считается однозначно (сумма стоимости клеток), так как Черепашка может попасть единственным образом в эти клетки. Аналогично для таблиц 2×1, 3×1, 4×1 – движение Черепашки вниз. Рассмотрим таблицу 2×2. У Черепашки два способа попадания – или справа, или сверху. Выбираем тот, который дает максимальную сумму, и фиксируем результат. Аналогично и для таблицы 2×3 (рис. 5).

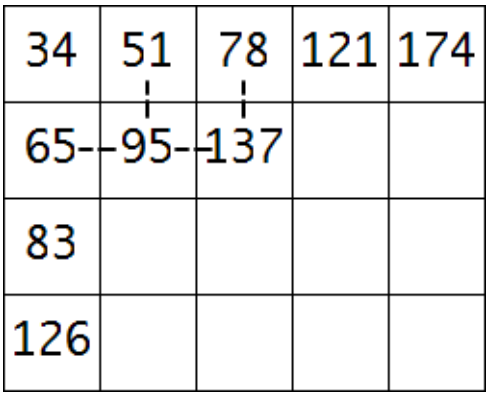

Рис. 5. Принцип решения подзадач

В программе описанная логика выглядит следующим образом:

```
procedure solve;
var i, j : integer;
begin
  f[1, 1] := a[1, 1]; {a - исходный массив}
  for i := 2 to n do
    f[i, 1] := f[i-1, 1]+a[i, 1];
  for j := 2 to m do
    f[1, j] := f[1, j-1]+a[1, j];
  for i := 2 to n do
    for j := 2 to m do
      f[i, j] := max(f[i-1, j], f[i, j-1])+a[i, j];
end;
```

После полного вычисления F мы находим стоимость пути Черепашки (для рассматриваемого примера она равна 321), но не сам путь. Для нахождения пути следует выполнить «обратный просмотр» массива F, начиная с клетки (n, m). Его суть: из значения `f[i,j]` вычитаем `a[i,j]` и смотрим, которое из двух чисел `f[i-1,j]` или `f[i,j-1]` равно полученному числу. Осуществляем переход по равенству и продолжаем до тех пор, пока не будет достигнута клетка (1,1). Естественно, что следует предусмотреть ситуации наличия одной соседней клетки.

Временная сложность решения $O(n \cdot m)$. Для таблицы $n = 300$, $m = 300$ потребуется меньше 100 000 шагов.

Итак, очередной раз обратимся к понятию динамического программирования. Наилучшее (оптимальное) решение для всей задачи состоит из наилучших решений подзадач. Любой подпуть Черепашки, являющийся частью пути, дающего максимальную стоимость, является наилучшим в своей подзадаче, то есть для таблицы меньшего размера.

3. Основной принцип

3.1. Вычисляемая функция, рекуррентные соотношения

При решении задач методом динамического программирования требуется найти способ формирования новых решений на основе уже найденных. Это должно быть некое правило, определяющее каким образом решение задачи выражается через решения ее подзадач. В силу того, что все подзадачи имеют одинаковую структуру (топологию), у нас есть основания полагать, что это правило будет общим.

Введем новое понятие. Назовем *вычисляемой функцией f*, отображение множества задач во множество соответствующих решений. То есть, $f(v)$ это решение задачи v. Таким образом, мы связали нахождение решений с вычислением значений некоторой функции. Зачем это нужно? Оказывается, что в терминах этой функции, способ формирования новых решений обычно записывается как рекуррентное соотношение. Пример для задач из первой главы:

1) числа Л. Фибоначчи. Здесь вычисляемая функция определяется, следующим образом: $f(i)$ – значение i-го числа Фибоначчи. Верно рекуррентное соотношение $f(n) = f(n–1) + f(n–2)$.

2) Биномиальные коэффициенты. По аналогии, подзадачи это нахождение биномиальных коэффициентов для всевозможных параметров n и k. Поэтому вычисляемая функция в данном случае также от двух параметров: $f(n,k)= C_n^k$. Рекуррентное соотношение нам известно:

$$f(n, k) = f(n–1, k) + f(n–1, k–1).$$

3) Черепашка. Здесь $f(i, j)$ – это сумма элементов на оптимальном пути до клетки (i, j) и рекуррентное соотношение имеет вид $f(i, j) = \max\big(f(i-1, j), f(i, j-1)\big) + a[i, j]$.

Итак, множество решаемых задач определено. Известна взаимосвязь задач – как получать новые решения на основе уже найденных. Как теперь организовать порядок решения подзадач? В неявном виде эта информация прописана в рекуррентных соотношениях. Благодаря им, мы можем выбрать очередность решения подзадач. Вычисления необходимо организовывать таким образом, чтобы когда решается задача, все подзадачи, от которых она зависит, были уже решены.

Возникает вопрос, а каким образом находить решения задач, которые не зависят от других? Такие задачи называются *базовыми*. Именно на их основе, будут решаться все оставшиеся задачи, и вычисляться ответ. Принято решения базовых задач называть *начальными условиями*.

3.2. Общая схема

Обобщим вышесказанное, обозначим те условия, при которых задача решается методом динамического программирования, и рассмотрим общую схему решения.

1) Задача должна допускать разбивку на подзадачи аналогичной структуры (аналогичной «топологии»), но меньшей размерности. В результате определяется множество решаемых задач. Речь может не идти о задаче в её первоначальной формулировке. Исходная задача, возможно, решается с помощью решенных подзадач. Так в задаче о Черепашке находится стоимость оптимального (максимального) пути, но не сам путь. Для поиска последнего требуется предпринять дополнительные усилия.

2) На множестве решаемых задач определяем вычисляемую функцию f. Вычисление новых значений f выполняется на основе уже найденных, то есть определяются рекуррентные соотношения (зависимость между задачами) связывающие решения задач.

3) Определяются тривиальные подзадачи (начальные условия), то есть задачи, имеющие наименьшую размерность.

4) Наилучшее (оптимальное) решение любой подзадачи определенного размера конструируется из наилучших (оптимальных) решений подзадач меньшей размерности. Это положение требует обоснования в каждом отдельном конкретном случае.

5) При решении различных подзадач согласно логике (зависимостями между задачами) приходится неоднократно обращаться за результатами решения подзадач меньшей размерности. Результаты решения подзадач следует запоминать, а не решать подзадачи многократно. Структуры данных для запоминания (обычно таблицы) должны быть разумного объема.

3.3. Пример решения задачи

Необходимо найти количество способов представления числа n в виде суммы натуральных слагаемых. При этом представления, отличающиеся порядком слагаемых, считаются одинаковыми. Так, для n=5 существует 7 разбиений:

$$5=1+1+1+1+1$$
$$5=1+1+1+2$$
$$5=1+1+3$$
$$5=1+2+2 \quad .$$
$$5=1+4$$
$$5=2+3$$
$$5=5$$

Первое, что нам нужно сделать, это определить вычисляемую функцию. Самая простая идея – сказать, что $f(i)$ – количество разбиений числа i на слагаемые. Тогда ответ задачи – просто $f(n)$. Но сможем ли мы найти закон, который позволит нам вычислить новое значение функции на основе ранее вычисленных значений? Другими словами, как, зная $f(1), f(2), \ldots, f(i-1)$, найти $f(i)$? Оказывается, что простого соотношения не существует. Но если видоизменить функцию, добавив еще один аргумент, и определить $f(i, k)$, как количество разбиений числа i на слагаемые, *где максимальное слагаемое равно* k, то рекуррентное соотношение найдется легко. Действительно, если максимальное слагаемое равно k, а общая сумма i, то остается разбить число $i-k$ на слагаемые, при этом максимальное слагаемое должно быть меньше или равно k. Переберем, каким может быть максимальное слагаемое в разбиении числа $i-k$ и получим соотношение: $f(i,k)=\sum_{j=1}^{k} f(i-k,j)$, которое верно при $i > k$. Начальные условия задаются следующим образом:

$$f(i,k)=\begin{Bmatrix} 0, & если\ i<k \\ 1, & если\ i=k \end{Bmatrix}.$$

А общий порядок вычислений описывается тремя вложенными циклами:

```
for i := 1 to n do
  for k := 1 to i-1 do
  begin
    f[i, k] := 0;
    for j := 1 To k Do
      f[i, k] := f[i, k] + f[i-k, j];
  end;
```

Чтобы найти ответ – количество разбиений числа n на слагаемые, мы должны перебрать все возможные максимальные слагаемые и сложить соответствующие значения функции f. Временная сложность решения $O(n^3)$.

На самом деле, решение можно ускорить, если по-другому задать вычисляемую функцию. Пусть у нас теперь $f(i, k)$ – это количество разбиений числа i на слагаемые, где максимальное слагаемое меньше или равно k. В этом случае возможны два варианта:

- максимальное слагаемое меньше k, тогда число разбиений это $f(i, k–1)$;
- максимальное слагаемое равно k, тогда число разбиений это $f(i–k, k)$.

Получаем рекуррентное соотношение: $f(i, k) = f(i, k–1) + f(i–k, k)$, которое верно, если $i \geq k$ и $k > 0$. Если $i < k$, то $f(i, k) = f(i, i)$. Начальные условия задаются следующим образом: $f(0,k) = 1$.

Порядок вычислений – это два вложенных цикла:

```
var n, i, k : integer;
    f : array[0..20, 0..20] of integer;

begin
  read(n);
  f[0, 0] := 1;
  for i := 0 to n do
  begin
    for k := 1 to i do
      f[i, k] := f[i, k-1] + f[i-k, k];
    for k := i+1 to n do
      f[i, k] := f[i, i];
  end;
  writeln(f[n, n]);
end.
```

А ответ – это значение $f(n, n)$. Временная сложность решения $O(n^2)$.

4. Задания для самостоятельного решения

1. Дана лестница из n ступенек. За один шаг можно подниматься или на следующую ступеньку или через одну. Напишите программу, определяющую количество различных способов, которыми можно подняться на n-ую ступеньку.

2. На вершине лестницы, содержащей n ($0<n<31$) ступенек, находится мяч, который начинает прыгать по ним вниз, к основанию. Мяч может прыгнуть на следующую ступеньку, на ступеньку через одну или через 2. (То есть, если мячик лежит на 8-й ступеньке, то он может переместиться на 5-ую, 6-ую или 7-ую). Определить число возможных «маршрутов» мячи с вершины лестницы на землю.

3. Напишите программу, которая находит количество последовательностей из 0 и 1 без двух единиц подряд.

4. На каждой из n+2 ступенек лестницы записано целое число, причем на первой и последней ступеньке записано число ноль. На первой ступеньке стоит человек, которому необходимо подняться на последнюю ступеньку. За один шаг он может подниматься на любое число ступенек, не превосходящее k. Подсчитаем сумму всех чисел, написанных на ступеньках, на которые наступил человек. Требуется найти наибольшее возможное значение этой суммы.

5. Верно ли, что рекуррентная формула для подсчета количества различных укладок плитками размера 1×2 коридора размера 2×n, имеет вид:

$$f_n = f_{n-1} + 2 \cdot f_{n-2}.$$

6. Предположим, что в задаче о Черепашке некоторые клетки поля являются запрещенными (на них невозможно становиться). Решите задачу, учитывая это ограничение.

7. Черепашка перемещается не из клетки в клетку, а по ребрам клеточного поля из точки А в точку Б (пример на рис. 6). За один ход разрешается пройти по одному ребру вправо или вверх. Ребра имеют вес – стоимость перемещения. Найти путь Черепашки с максимальной суммарной стоимостью.

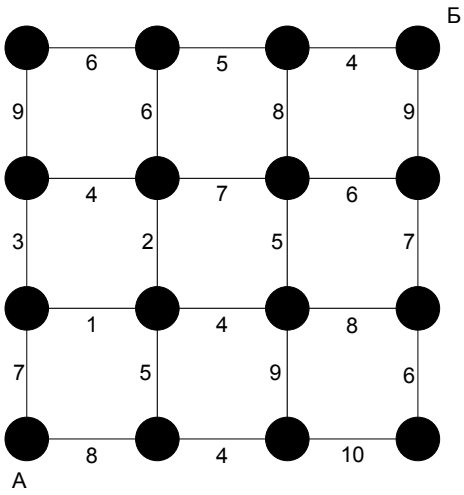

Рис. 6. Пример поля

8. Черепашка может начинать свой путь из любой клетки первой строки поля и заканчивать в любой клетке *n*-й строки. Черепашке разрешено перемещаться из клетки в любую из трех соседних (с учетом ограничений), находящихся в строке с номером на единицу большем, чем текущий номер. Найти путь Черепашки с максимальной стоимостью.

Литература

1. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы. М.: Вильямс, 2007.
3. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: Построение и анализ. – 2-е изд. – М.: Издательский дом «Вильямс», 2005.
4. Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы. Теория и практика. – М.: Мир, 1980.
5. Шень А. Программирование: теоремы и задачи. – 3-е изд. – М.: МЦНМО, 2007.