Урок 4

Динамическое программирование. Поиск возвратом

Динамическое программирование

- Пример. Задача ЕГЭ
- Количество маршрутов
- Количество маршрутов с препятствиями

Наибольшая общая подпоследовательность

- Решение

Поиск с возвратом

- Описание метода
- Использование метода
- Задача о восьми ферзях

Домашнее задание

Дополнительные материалы

Используемая литература

Динамическое программирование

Динамическое программирование в теориях управления и вычислительных систем — способ решения сложных задач путём разбиения их на более простые подзадачи.

Ключевая идея в динамическом программировании достаточно проста. Как правило, чтобы решить поставленную задачу, требуется решить отдельные части задачи (подзадачи), после чего объединить решения подзадач в одно общее решение. Часто многие из этих подзадач одинаковы. Подход динамического программирования состоит в том, чтобы решить каждую подзадачу только один раз, сократив тем самым количество вычислений. Это особенно полезно в случаях, когда число повторяющихся подзадач экспоненциально велико.

Метод динамического программирования сверху — это простое запоминание результатов решения тех подзадач, которые могут повторно встретиться в дальнейшем. Динамическое программирование снизу включает в себя переформулирование сложной задачи в виде рекурсивной последовательности более простых подзадач.

Пример. Задача ЕГЭ

У исполнителя Калькулятор две команды, которым присвоены номера:

1. Прибавь 1.
2. Умножь на 2.

Сколько есть программ, которые число 1 преобразуют в число 16?

Эта задача решается довольно легко, если составить рекуррентное соотношение.

P(1) = 1
P(N) = P(N - 1) + P(N / 2), при N > 1, если N кратно 2
P(N) = P(N - 1), при N > 1, если N не кратно 2

Попробуйте запрограммировать его сами. Здесь можно использовать как рекурсивный, так и не рекурсивный способ.

Ответ для проверки: 36.

План решения задач

1. Решение задачи для маленьких ограничений.
2. Ввести обозначения, что такое Pi — сформулировать, какую величину мы считаем.
3. Получить рекуррентное соотношение.
4. Определить, при каких ограничениях работает эта формула, и выписать начальные значения.
5. Определить порядок вычислений.
6. Определить, где лежит вычисленное значение.

Количество маршрутов

Пусть за один ход королю разрешается передвинуться на одну клетку вниз или вправо. Необходимо определить, сколько существует различных маршрутов, ведущих из левого верхнего угла в правый

нижний. Будем считать, что положение короля задается парой чисел (a, b), где a задаёт номер строки, а b — номер столбца. Строки нумеруются сверху вниз от 0 до n−1, а столбцы — слева направо от 0 до m−1. Таким образом, первоначальное положение короля — клетка (0, 0), а конечное — клетка (n−1, m−1). Пусть W(a, b) — количество маршрутов, ведущих в клетку (a, b) из начальной клетки. Запишем рекуррентное соотношение. В клетку (a, b) можно прийти двумя способами: из клетки (a, b − 1), расположенной слева, и из клетки (a − 1, b), расположенной сверху от данной. Поэтому количество маршрутов, ведущих в клетку (a, b), равно сумме количеств маршрутов, ведущих в клетку слева и сверху от неё. Получили рекуррентное соотношение:

```
W(a, b) = W(a, b − 1) + W(a − 1, b)
```

.

Это соотношение верно при a > 0 и b > 0. Зададим начальные значения: если a = 0, то клетка расположена на верхнем краю доски, и прийти в неё можно единственным способом — двигаясь только влево, поэтому W(0, b)=1 для всех b. Аналогично, W(a, 0) = 1 для всех a. Создадим массив W для хранения значений функции, заполним первую строку и первый столбец единицами, а затем заполним все остальные элементы массива. Поскольку каждый элемент равен сумме значений, стоящих слева и сверху, заполнять массив W будем по строкам сверху вниз, а каждую строку — слева направо.

```
#include <stdio.h>
#define N 3
#define M 3
void Print2(int n, int m, int a[N][M])
{
    int i, j;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < m; j++)
            printf("%4d", a[i][j]);
        printf("\n");
    }
}

int main(int argc, char *argv[])
{
    int A[N][M];
    int i, j;
    for(j = 0; j < M; j++)
        A[0][j] = 1; // Первая строка заполнена единицами
    for(i = 1; i < N; i++)
    {
        A[i][0] = 1;
        for(j = 1; j < M; j++)
            A[i][j] = A[i][j-1] + A[i-1][j];
    }
    Print2(N, M, A);
    return 0;
}
```

Количество маршрутов с препятствиями

Пусть некоторые клетки на доске являются «запретными»: король не может ходить на них. Карта запретных клеток задана при помощи массива Map[n][m]: нулевое значение элемента массива

означает, что данная клетка запрещена, единичное значение означает, что в клетку можно ходить. Массив Map считывается программой после задания значений n и m. Король может ходить только вниз или вправо. Для решения этой задачи придётся изменить рекуррентное соотношение с учётом наличия запрещенных клеток. Для запрещённой клетки количество ведущих в неё маршрутов будем считать равным 0. Получим:

W(a,b) = W(a - 1, b) + W(a, b - 1), если Map[a][b] = 1,
W(a,b) = 0, если Map[a][b] = 0.

Также надо учесть, что для клеток верхней строки и левого столбца эта формула некорректна, поскольку для них не существует соседней сверху или слева клетки.

Попробуйте решить задачу самостоятельно.

Наибольшая общая подпоследовательность

Задача нахождения наибольшей общей подпоследовательности (англ. longest common subsequence, LCS) — задача поиска последовательности, которая является подпоследовательностью нескольких последовательностей (обычно двух). Часто задача определяется как поиск всех наибольших подпоследовательностей. Это классическая задача информатики, которая имеет приложения, в частности, в задаче сравнения текстовых файлов (утилита diff), а также в биоинформатике.

Подпоследовательность можно получить из некоторой конечной последовательности, если удалить из последней некоторое множество её элементов (возможно пустое). Например, BCDB является подпоследовательностью последовательности ABCDBAB. Будем говорить, что последовательность Z является общей подпоследовательностью последовательностей X и Y, если Z является подпоследовательностью как X, так и Y. Требуется для двух последовательностей X и Y найти общую подпоследовательность наибольшей длины. Заметим, что НОП может быть несколько.

Обратите внимание: подпоследовательность отличается от подстроки. Например, если есть исходная последовательность ABCDEF, то ACE будет подпоследовательностью, но не подстрокой, а ABC будет как подпоследовательностью, так и подстрокой.

Решение

Вначале найдём длину наибольшей подпоследовательности. Допустим, мы ищем решение для случая (n1, n2), где n1, n2 — длины первой и второй строк. Пусть уже существуют решения для всех подзадач (m1, m2), меньших заданной. Тогда задача (n1, n2) сводится к меньшим подзадачам следующим образом.

Рекурсивное решение:

```
int lcs_length(char * A, char * B)
{
    if (*A == '\0' || *B == '\0') return 0;
    else if (*A == *B) return 1 + lcs_length(A + 1, B + 1);
    else return max(lcs_length(A + 1, B), lcs_length(A, B + 1));
}
```

Представленный алгоритм позволяет найти длину максимальной последовательности.

Решение этой задачи очень напоминает задачу о поиске длины пути. Также её можно свести к решению с помощью матрицы.

		G	E	E	K	B	R	A	I	N	S
	0	0	0	0	0	0	0	0	0	0	0
G	0	**1**	1	1	1	1	1	1	1	1	1
E	0	1	**2**	2	2	2	2	2	2	2	2
E	0	1	2	**3**	3	3	3	3	3	3	3
K	0	1	2	3	**4**	4	4	4	4	4	4
M	0	1	2	3	4	4	4	4	4	4	4
I	0	1	2	3	4	4	4	4	**5**	5	5
N	0	1	2	3	4	4	4	4	5	**6**	6
D	0	1	2	3	4	4	4	4	5	6	6
S	0	1	2	3	4	4	4	4	5	6	**7**

Предлагаем вам самостоятельно разработать алгоритм решения, используя матрицу вместо рекурсии.

Поиск с возвратом

Поиск с возвратом, бэктрекинг (англ. backtracking) — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве M. Как правило, он позволяет решать задачи, в которых ставятся вопросы типа: «Перечислите все возможные варианты …», «Сколько существует способов …», «Есть ли способ …», «Существует ли объект…» и т. п.

Описание метода

Рекурсия используется также для поиска лучшего решения сложных задач. В алгоритмах с возвратом создаются тестовые варианты, которые просчитывают возможность справиться с поставленной задачей. Если программа приходит к выводу, что предлагаемый способ не приведет к нужному результату, данный тестовый вариант отбрасывается и вверху по стеку вызовов ищется другой.

Такой метод очень удобен: он позволяет строить частичное решение и определять, приведет ли оно к полному. Благодаря такому подходу вы можете прекратить дальнейшее усовершенствование пробного варианта и вернуться на несколько шагов назад, чтобы продолжить работу оттуда.

```
// Исследуем тестовое решение.
// Возвращаем false, если его нельзя развить до полного решения.
// Возвращаем true, если рекурсивный вызов SearchSolution  приводит к полному решению.
Boolean: SearchSolution(Solution: test_solution)
// Если можно сделать вывод, что данное частичное решение
// не приводит к полному, возвращаем false.
  If <test_solution не может решить проблему> Then Return false
// Если это полное решение, возвращаем true.
  If <test_solution — полное решение> Then Return true

// Расширяем частичное решение.

 Loop <Проходим по всем возможным расширениям в test_solution.>

   <Расширяем test_solution.>

// Рекурсивно проверяем, ведет ли это к решению.
    If (SearchSolution(test_solution)) Then Return true
// Это расширение не ведет к решению. Отменяем расширение.
   <Отменяем расширение.>

 End Loop

// Если мы дошли до этой строки, данное частичное решение
// не приводит к полному.
Return false
End SearchSolution
```

Алгоритм SearchSolution берет в качестве параметра любые необходимые данные, чтобы проработать частичное решение, и возвращает true, если оно приводит к полному.

Первым делом осуществляется проверка, имеет ли частичное решение право на существование. Если оно не приводит к полному решению, алгоритм возвращает false и вызывает метод LeadsToSolution, который завершает текущий тест и приступает к новому. Если предлагаемое решение допустимо, алгоритм циклически проходит по всем возможным его расширениям до получения конечного результата. Для каждого расширения алгоритм рекурсивно ссылается на самого себя, чтобы определить, будет ли оно работать. Если рекурсивный вызов возвращает false, расширение не годится: оно отменяется, и осуществляется следующая попытка с новым расширением. Если алгоритм перепробовал все возможные расширения и не нашел среди них целесообразного, он возвращает false, чтобы вызов SearchSolution закрыл тестовое решение.

Использование метода

Классическим примером использования алгоритма поиска с возвратом является задача о **восьми ферзях**. Её формулировка такова: «Расставить на стандартной 64-клеточной шахматной доске 8 ферзей так, чтобы ни один из них не находился под боем другого». Сперва на доску ставят одного ферзя, а потом пытаются поставить каждого следующего ферзя так, чтобы его не били уже установленные ферзи. Если на очередном шаге такую установку сделать нельзя, возвращаются на шаг назад и пытаются поставить ранее установленного ферзя на другое место.

Задача о восьми ферзях

В решении мы используем рекурсивный подход.

Запускаем Функцию Поиска при N=1

Начало функции SearchSolution(1)

1. Если расстановка (CheckBoard) не подходит, не ставим ферзя.
2. При N=9 ферзи расставлены (т.е. девятого ферзя ставить никуда не нужно). Решение найдено. Заканчиваем поиск. Идем к шагу 7.
3. Ищем решение. Запускаем цикл перебора клеток доски (row и col).
4. Ставим ферзя на доску в позицию row,col, сообщаем об этом вызовом SearchSolution(N+1). Идем к шагу 1.
5. Если ферзь не подошёл (SearchSolution вернул 0), убираем его (board[row][col]=0).
6. Проверяем следующую позицию (идём к шагу 3).
7. Выходим из функции.

Здесь для проверки полезно написать функцию CheckBoard, которая проверяет расположение ферзей. Самих ферзей можно расставлять в двухмерном массиве 8x8, который первоначально нужно заполнить 0.

```
#include <stdio.h>
#include <math.h>
#define N 8
#define M 8
// Доска для ферзей.
// 0 - клетка пустая
// число - номер ферзя
int board[N][M];

int SearchSolution(int n);
int CheckBoard();
int CheckQueen(int x, int y);
void Print(int n, int m, int a[N][M]);
void Zero(int n, int m, int a[N][M]);
void Pause(int key);

int main()
{
   Zero(N, M, board);
   SearchSolution(1);
   printf("\n\n");
   Print(N, M, board);
   getch();
   return 0;
}

int SearchSolution(int n)
{
   // Если проверка доски возвращает 0, то эта расстановка не подходит
   if (CheckBoard() == 0) return 0;
   // 9 ферзя не ставим. Решение найдено
   if (n == 9) return 1;
   int row;
   int col;
   for(row = 0; row < N; row++)
```

```
        for(col = 0; col < M; col++)
        {
            if (board[row][col]==0)
            {
                // Расширяем test_solution
                board[row][col]=n;
                // Рекурсивно проверяем, ведет ли это к решению.
                if (SearchSolution(n+1)) return 1;
                // Если мы дошли до этой строки, данное частичное решение
                // не приводит к полному.
                board[row][col]=0;
            }
        }
    return 0;
}
// Проверка всей доски
int CheckBoard()
{
    int i, j;
    for(i = 0; i < N; i++)
        for(j = 0; j < M; j++)
            if (board[i][j] != 0)
                if (CheckQueen(i, j) == 0)
                    return 0;
    return 1;
}
// Проверка определённого ферзя
int CheckQueen(int x, int y)
{
    int i, j;
    for(i = 0; i < N; i++)
        for(j = 0; j < M; j++)
// Если нашли фигуру
            if (board[i][j] != 0)
                if (!(i == x && j == y)) // Если это не наша фигура
                {
// Лежат на одной вертикали или горизонтали
                    if (i – x == 0 || j – y == 0)
                        return 0;
                    // Лежат на одной диагонали
                    if (abs(i - x) == abs(j - y))
                        return 0;
                }
// Если дошли до сюда, то всё в порядке
    return 1;
}
// Выводим доску на экран
void Print(int n, int m, int a[N][M])
{
    int i, j;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < m; j++)
            printf("%4d", a[i][j]);
        printf("\n");
    }
}
// Очищаем доску
```

```
void Zero(int n, int m, int a[N][M])
{
    int i, j;
    for(i = 0; i < n; i++)
        for(j = 0; j < m; j++)
            a[i][j] = 0;
}
void Pause(int key)
{
    if (key == 1)
        getch();
    else
        for (int i = 0; i < 100000000; i++);
}
```

Домашнее задание

1. *Количество маршрутов с препятствиями. Реализовать чтение массива с препятствием и нахождение количество маршрутов.

Например, карта:

3 3

1 1 1

0 1 0

0 1 0

2. Решить задачу о нахождении длины максимальной последовательности с помощью матрицы.

3. ***Требуется обойти конём шахматную доску размером NxM, пройдя через все поля доски по одному разу. Здесь алгоритм решения такой же как и в задаче о 8 ферзях. Разница только в проверке положения коня.

Дополнительные материалы

1. Задача о восьми ферзях на Википедии
2. Динамическое программирование. Д.П. Кириенко
3. Сайт К.Ю. Полякова. Задачи ЕГЭ (рекурсивные алгоритмы)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Род Стивенс, Алгоритмы. Теория и практическое применение. Издательство«Э», 2016.
2. Никлаус Вирт, Алгоритмы и структуры данных. Новая версия для Оберона, ДМК, Москва, 2010.
3. Пол Дейтел, Харви Дейтел. C для программистов. С введением в C11, ДМК, Москва, 2014.