Лабораторная работа № 6

Тема: *Разработка эффективных алгоритмов. Динамическое программирование.*

Цель: *ознакомиться с понятием «динамическое программирование», изучить алгоритмы работы динамического программирования, научиться применять полученные знания на практике.*

1 Краткая теория

1.1 ОБЩИЕ ПОНЯТИЯ.

Что такое динамическое программирование?

Работу разработчика часто можно сравнить с решением головоломок. Как в настоящей головоломке, разработчику приходится тратить существенные ресурсы не столько на реализацию конкретного решения, сколько на выбор оптимального подхода. Иногда задача решается легко и эффективно, а порой — только полным перебором всех возможных вариантов. Такой подход часто называют *наивным решением*. Он имеет существенный минус — временные затраты.

Представим хакера, который пытается взломать какой-то пароль перебором всех комбинаций символов. Если пароль допускает 10 цифр, 26 маленьких букв, 26 больших букв и 32 специальных символа (например, значок доллара), то для каждого символа в пароле есть 94 кандидата. Значит, чтобы взломать перебором пароль, состоящий из одного символа, потребуется 94 проверки. Если в пароле два символа — 94 кандидата на первую позицию, 94 кандидата на вторую позицию — то придется перебрать аж 94*94 = 8836 возможных пар. Для пароля из десяти символов потребуется уже перебор 94^10 комбинаций.

Если обобщить, то для взлома пароля с произвольной длиной N требуется O(94^N) операций. Такие затраты часто называют «*экспоненциальными*»: появление каждой новой позиции влечёт за собой увеличение количества операций в 94 раза. Взлом пароля может показаться чем-то экзотическим, но задачи, требующие полного перебора всех вариантов — совсем не экзотика, скорее угрюмая реальность.

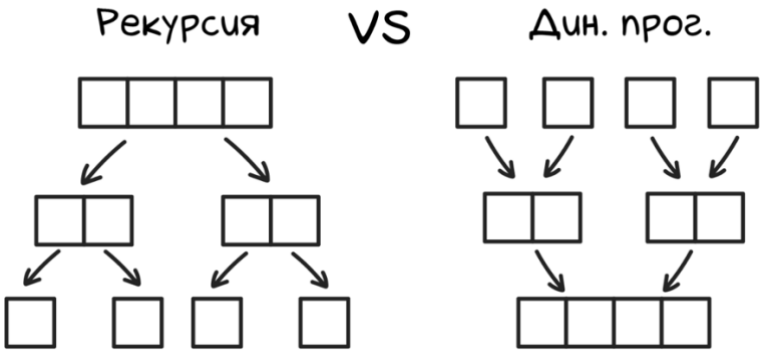
Экспоненциальное время — это долго. Даже O(2^N) — это просто непозволительно долго. Настолько долго, что никому и в голову не придет запустить подобный алгоритм даже на простых данных — ведь на решение такой задачи с сотней элементов потребуются тысячи, миллионы или миллиарды лет вычислений. А в реальной жизни нужно решать задачи с намного большим количеством элементов. Как же быть?

Дело в том, что многие задачи без эффективного алгоритма решения можно решить за привлекательное время с помощью одной хитрости — **динамического программирования**.

Динамическое программирование?

Динамическое программирование — это особый подход к решению задач. Идея заключается в том, что оптимальное решение зачастую можно найти, рассмотрев все возможные пути решения задачи, и выбрать среди них лучшее.

Работа динамического программирования очень похожа на рекурсию с запоминанием промежуточных решений — такую рекурсию еще называют *мемоизацией*. Рекурсивные алгоритмы, как правило, разбивают большую задачу на более мелкие подзадачи и решают их. Динамические алгоритмы делят задачу на кусочки и вычисляют их по очереди, шаг за шагом наращивая решения. Поэтому динамические алгоритмы можно представить как рекурсию, работающую снизу вверх.



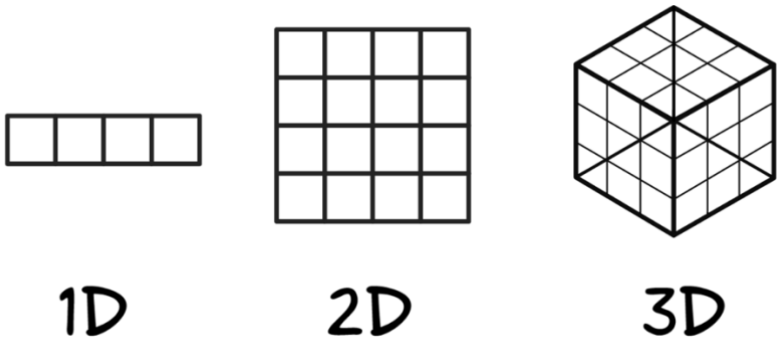
Магия динамического программирования заключается в умном обращении с решениями подзадач. «Умный» в этом контексте значит «не решающий одну и ту же подзадачу дважды». Для этого решения мелких подзадач должны где-то сохраняться. Для многих реальных алгоритмов динамического программирования этой структурой данных является таблица.

В самых простых случаях эта таблица будет состоять только из одной строки — аналогично обычному массиву. Эти случаи будут называться *одномерным динамическим программированием*, и потреблять O(n) памяти. Например, алгоритм эффективного вычисления чисел Фибоначчи использует обычный массив для запоминания вычисленных промежуточных результатов. Классический рекурсивный алгоритм делает очень много бессмысленный работы — он по миллионному разу рассчитывает то, что уже было рассчитано в соседних ветках рекурсии.

В самых распространённых случаях эта таблица будет выглядеть привычно и состоять из строчек и столбиков. Обычная таблица, похожая на таблицы из Excel. Это называется *двумерным динамическим программированием*, которое при n строках и n столбцах таблицы потребляет O(n*n) = O(n^2) памяти. Например, квадратная таблица из 10 строк и 10 столбцов будет содержать 100 ячеек. Чуть ниже будет подробно разобрана именно такая задача.

Бывают и более запутанные задачи, использующие для решения трехмерные таблицы, но это редкость — решение задачи с использованием трехмерной таблицы зачастую просто нельзя себе позволить. Небольшая двухмерная таблица на 1024 строки и 1024 столбца может потребовать несколько

мегабайт памяти. Трехмерная таблица с такими же параметрами будет занимать уже несколько гигабайт.



Что нужно, чтобы решить задачу динамически, помимо ее исходных данных? Всего три вещи:

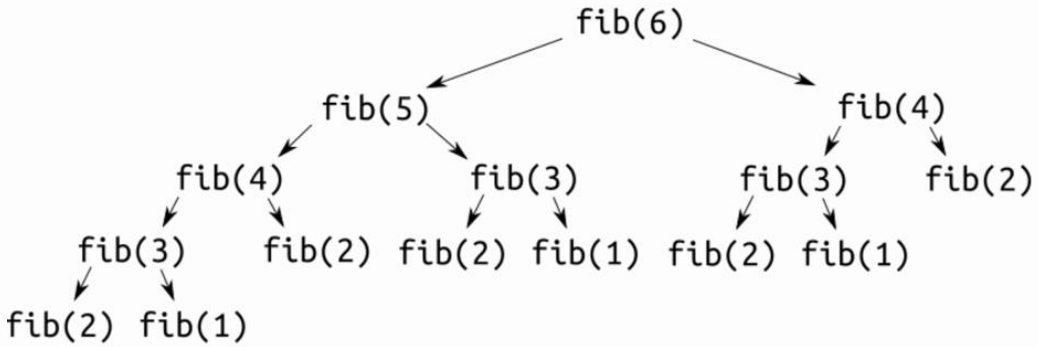
- Таблица, в которую будут вноситься промежуточные результаты. Один из них будет выбран в конце работы алгоритма в качестве ответа.
- Несколько простых правил по заполнению пустых ячеек таблицы, основанных на значениях в уже заполненных ячейках. Универсального рецепта тут нет и к каждой задаче требуется свой подход.
- Правило выбора финального решения после заполнения таблицы.

Связь между динамическим программированием и рекурсией. Последовательность Фибоначчи

Реализация расчёта n-го числа Фибоначчи через рекурсию:

```
1    int fib(int n) {
2        if (n <= 2) {
3            return 1;
4        } else {
5            return fib(n - 1) + fib(n - 2);
6        }
7    }
```

Данная реализация не подходит для использования на практике из-за слишком высокой сложности. Откуда же берётся эта сложность? Просто рассмотрим дерево вызовов функции *fib* для *n*=6:



Для подсчёта *fib(6)* функцию *fib* пришлось вызвать 15 раз, хотя логично, что хватило бы шести. Вся проблема заключается в том, что для

некоторых *x* функция *fib(x)* будет вызываться больше одного раза, и каждый раз высчитываться рекурсивно заново. Очевидно, что для оптимальной работы значения функции нужно сохранять для последующего использования. ДП – один из способов такой оптимизации.

Решение задачи о последовательности Фибоначчи с помощью ДП:

```
#include <bits/stdc++.h>

using namespace std;

int fib[100001];

int main() {
    int n;      //N <= 100000
    cin >> n;

    fib[1] = fib[2] = 1;     //начальные значения

    for (int i = 3; i <= n; i++) {
        fib[i] = fib[i - 1] + fib[i - 2];    //формула перехода
    }

    cout << fib[n];    //вычисление ответа
}
```

Идея проста: для вычисления *fib(n)* нам нужны значения *fib(n−1)* и *fib(n−2)*. Будем считать *fib* в порядке возрастания *n*, сохраняя результаты в массив.

В большинстве случаев ДП характеризуется тремя главными параметрами:

- **Начальные значения.** Аналогично крайним случаям в рекурсивных функциях.
- **Формула перехода.** Описывает рекурсивную зависимость.
- **Вычисление ответа.** В некоторых случаях ответ может быть не последним значением, а суммой или максимумом по значениям.

Путь в матрице

Задачи на поиск оптимального пути в матрице, наверное, самые классические, после задач на последовательность Фибоначчи. В таких задачах каждой клетке в матрице присвоено некоторое число, и нужно найти путь между двумя клетками с максимальной или минимальной суммой.

5	2	3	-2	-2
-1	4	1	-3	10
6	-2	4	-5	0
12	-8	-5	3	6

Приведём решение такой задачи. Будем искать путь между левой верхней и правой нижней клетками с максимальной суммой, если ходить можно только вниз или вправо. Для решения задачи используем следующее ДП: *dp[i][j]* - максимальная сумма, которую мы можем набрать, дойдя до клетки (i,j). Опишем ДП:

- Начальные значения: *dp[0][0]=c[0][0]* (c - исходная матрица). Мы находимся в клетке (0,0), значит мы ещё не двигались, то есть собранная нами сумма равна значению в этой клетке.
- Формула перехода: *dp[i][j]=max(dp[i−1][j],dp[i][j−1])+c[i][j]*. Мы можем перейти в клетку (i,j) либо сверху, либо слева. Выгоднее перейти из той, в которую мы до этого пришли с большей суммой.
- Ответ: *dp[n−1][m−1]*.

Реализация на C++:

```cpp
#include <bits/stdc++.h>

using namespace std;

int c[1000][1000];
int dp[1000][1000];

int main() {
    int n, m;
    cin >> n >> m;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> c[i][j];
        }
    }

    dp[0][0] = c[0][0];

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (i || j) {   //цикл не должен заходить в клетку (0, 0)
                dp[i][j] = INT_MIN;      //код рассчёта максимума получился
                                         //достаточно длинным из-за дополнитель
                                         //проверок на выход за границы матрицы

                if (i - 1 >= 0) {
                    dp[i][j] = max(dp[i][j], dp[i - 1][j] + c[i][j]);
                }

                if (j - 1 >= 0) {
                    dp[i][j] = max(dp[i][j], dp[i][j - 1] + c[i][j]);
                }
            }
        }
    }

    cout << dp[n - 1][m - 1];
}
```

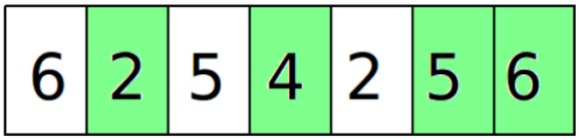
При реализации ДП всегда нужно быть уверенным, что все значения, необходимые для вычисления текущего, уже были вычислены.

Наибольшая возрастающая последовательность

! Собственно, этот алгоритм - плохой пример ДП. Он приведён только для решения задачи за O(NlogN). Даже у опытных программистов могут возникать серьёзные сомнения, причислять ли его к классу ДП !

Ещё одна классическая задача на ДП. Её формулировка следующая: задана последовательность из N чисел. Нужно удалить из неё минимальное число элементов, чтобы оставшиеся составляли строго (в других версиях - нестрого) возрастающую последовательность.

Например, рассмотрим последовательность 6,2,5,4,2,5,66,2,5,4,2,5,6. Мы можем вычеркнуть из неё три числа (первая 66, первая 55 и вторая 22) и получить возрастающую последовательность 2,4,5,62,4,5,6, длина которой 44. Она является оптимальной, нельзя получить возрастающую подпоследовательность большей длины.



Для решения задачи будем использовать ДП следующего вида: *dp[i]* - длина наибольшей возрастающей подпоследовательности, оканчивающейся числом *a[i]*. Опишем это ДП:

- Начальные значения: отсутствуют.

Достаточно редкий случай, когда формула перехода позволяет решать ДП без начальных значений.

- Формула перехода:

$$dp[i] = \max(1, \max_{\substack{0 \le j < i \\ a[j] < a[i]}} dp[j] + 1)$$

- Можно либо начать новую последовательность длиной 11, либо продолжить любую из уже начатых, последний элемент которого строго меньше текущего. Из всех таких последовательностей выгодно выбрать, разумеется, самую длинную (это записано во втором аргументе max).
- Ответ

$$: \max_i dp[i].$$

Последовательность не обязательно должна заканчиваться последним элементом.

Возможно, реализация будет немного понятнее:

```
#include <bits/stdc++.h>
using namespace std;
int a[100000];
int dp[100000];

int main() {
```

```
    int n;
    cin >> n;

    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    for (int i = 0; i < n; i++) {
       //Мы можем начать новую подпоследовательность
        dp[i] = 1;

        for (int j = 0; j < i; j++) {
            if (a[j] < a[i]) {
              //Или продолжить уже начатую.
                 dp[i] = max(dp[i], dp[j] + 1);
            }
        }
    }

    int ans = *max_element(dp, dp + n);       //C++11
    cout << ans;
}
```

Как можно заметить, сложность такого решения $O(N^2)$. Существует другое решение этой задачи, не такое тривиальное, имеет сложность $O(NlogN)$. Его можно охарактеризовать как ДП (с натяжкой), но оно значительно отличается от примеров выше. Разберём его.

Будем идти по последовательности слева направо, поддерживая массив d, где $d[i]$ - минимальный последний элемент среди всех возможных возрастающих подпоследовательностей длиной i. Если таковых не существует, то примем $d[i]=\infty$. Можно достаточно тривиально доказать, что массив d будет строго возрастающим:

> По определению $d[i]$ - *минимальный* последний элемент среди *всех* подпоследовательностей длиной i. Значит, i-ый элемент любой подпоследовательности длиной больше i не меньше, чем $d[i]$. Следовательно, не может существовать строго возрастающей подпоследовательности длиной $i+1$, такой что её последний элемент меньше либо равен $d[i]$, что и требовалось доказать.

Пусть мы обрабатываем очередной элемент $a[i]$, и хотим с его помощью продлить некоторые последовательности. С помощью бинарного поиска найдём в массиве d первый такой индекс j, что $a[i]<d[j]$. Утверждается, что элемент $a[j]$. может эффективно продолжить только последовательность длиной $j-1$. Доказательство:

> $a[i] < d[j]$ по определению, а массив d строго возрастает. Чтобы продлить некоторую последовательность, $a[i]$ должен быть строго больше её последнего элемента. Значит, $a[i]$ не может продлить ни одну последовательность длиной $\geq j$.
>
> Чтобы эффективно продлить последовательность длиной k должно выполняться условие $a[i] < d[k+1]$. Но $a[i]$ по определению больше либо равен $d[j-1]$, а значит и всем предыдущим значениям. Значит, $a[i]$ не может продлить ни одну последовательность длиной $< j-1$
>
> Следовательно, если $a[i]$ может эффективно продлить какую-либо последовательность, то её длина равна $j-1$.

Реализация на C++:

```
#include <bits/stdc++.h>
using namespace std;
const int INF = 1000000007;      //"бесконечность"
int a[100000];
int d[100001];

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    for (int i = 2; i <= n; i++) {
        d[i] = INF;
    }
    d[1] = a[0];    //В качестве начального значения, обработаем первый
элемент, и запишем его как минимальный для длины 1.
    for (int i = 1; i < n; i++) {

       //не забываем сдвинуть индексы.
        int j = upper_bound(d + 1, d + n + 1, a[i]) - d;
       //если a[i] строго больше предыдущего элемента, а не равен ему
        if (j == 1 || a[i] > d[j - 1]) {
       //попытаемся с его помощью улучшить ответ для длины j
            d[j] = min(d[j], a[i]);
        }
    }
//выводим максимальное i, для которого d[i] не равно бесконечности.
    for (int i = n; i > 0; i--) {
        if (d[i] != INF) {
            cout << i;
            break;
        }
    }
}
```

У вас мог возникнуть вопрос: где в этом решении ДП? На самом деле, *d*, хоть и неявно, является рекурсивной функцией, так как *d[i]* зависит от *d[i−1]*. Её отличие от нормальных рекурсивных функций заключается в нетрадиционном порядке пересчета - переборе *a[i]* с обновлением промежуточных значений *d[j]*, вместо традиционного перебора *dp[i]*.

! Дополнительный материал !

1. Зачем изучать алгоритмы:
2. Динамическое программирование для начинающих:
3. Динамическое программирование:
4. Учебник по ДП: создание эффективных программ на Python:
5. Python и ДП на примере задачи о рюкзаке

2 Задания.

Обязательное 1: *Каждой вершине дерева присвоено некоторое число (возможно отрицательное). Нужно найти в дереве путь с максимальной суммой вершин (начальные и конечные вершины могут быть произвольными).*

! Контрольные вопросы !

1. Принцип работы динамического программирования.
2. Какими тремя главными параметрами характеризуется ДП в большинстве случаев?

Содержание отчёта

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Ответы на контрольные вопросы.
6. Выводы.