Динамическое программирование. Основные концепции

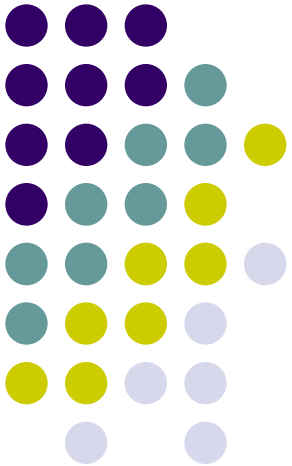
Федор Царев, Андрей Лушников

Спецкурс «Олимпиадное программирование»

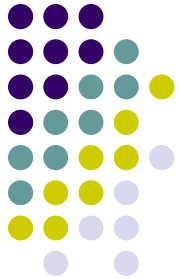
Лекция 4

09.02.2009

Санкт-Петербург, Гимназия 261

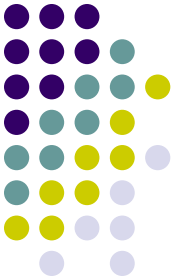


Цель лекции



- Изучить базовые идеи динамического программирования и простейшие примеры его применения
- Изучить методы реализации этих алгоритмов на языке программирования Pascal (Delphi)

Чем не является динамическое программирование

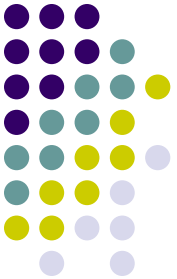


- Динамическое программирование – не метод составления программ, а метод составления алгоритмов
- Динамическое программирование не имеет ничего общего с динамической памятью

Где используется динамическое программирование?

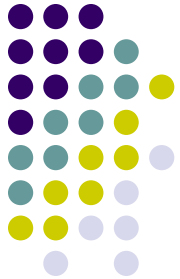
- Алгоритм обработки графов
- Алгоритмы обработки строк
- Биоинформатика
- Распознавание речи
- Оптимизация запросов к базам данных
- Обработка изображений
- …

Числа Фибоначчи



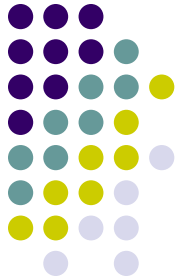
- Пример почти динамического программирования
- Fib(0) = Fib(1) = 1
- Fib(n) = Fib(n-1) + Fib(n-2), n > 1

Простой способ вычисления

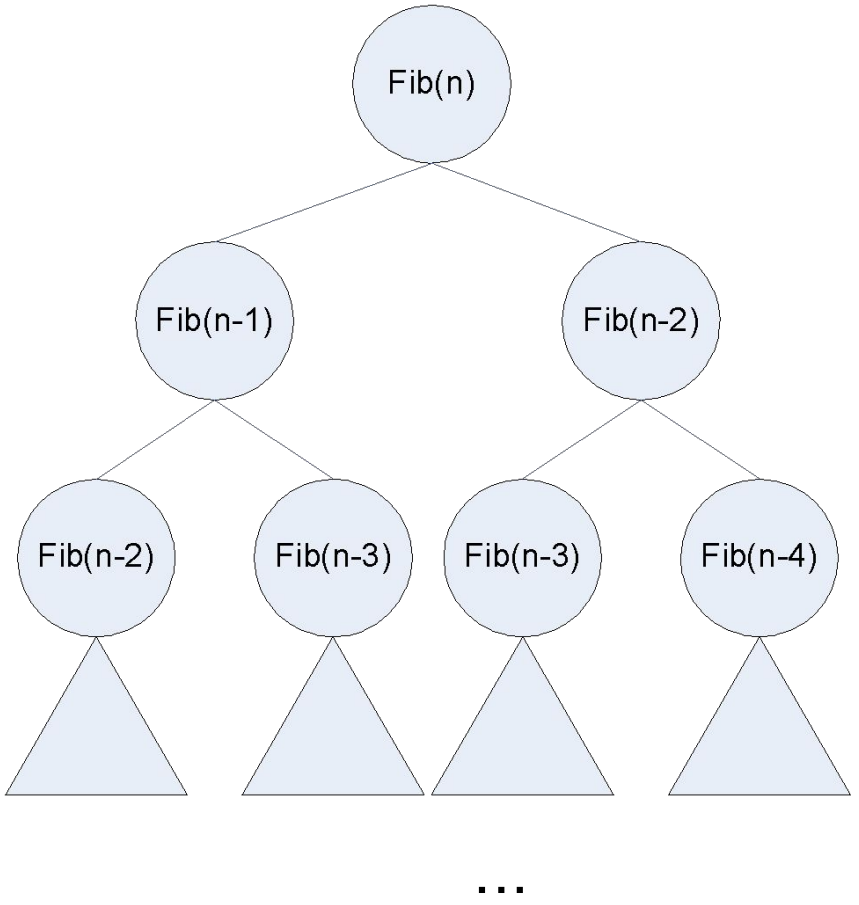


```
function fib(n : longint) : longint;
begin
  if (n < 2) then begin
   result := 1;
   exit;
  end;
  result := fib(n - 1) + fib(n - 2);
end;
```

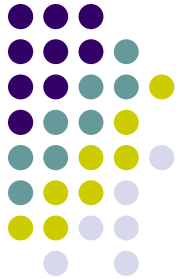
Дерево вычислений



- Медленно работает – время пропорционально значению числа Фибоначчи, которые растут примерно как 1.6^n
- Много одинаковых поддеревьев

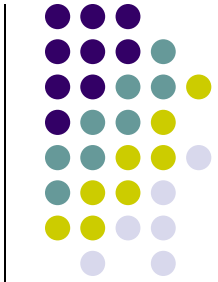


Метод запоминания ответов



- После того, как число вычислено надо его запомнить, а не считать заново
- В массиве логического типа **`calculated`** хранится информация о том, вычислено соответствующее число или нет
- В массиве **`ans`** хранится вычисленное число

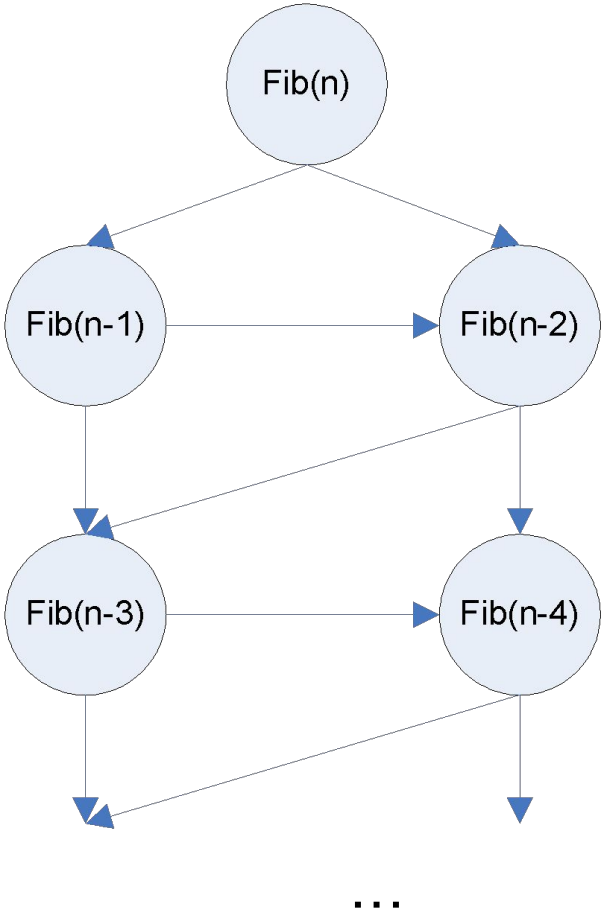
Более быстрое вычисление



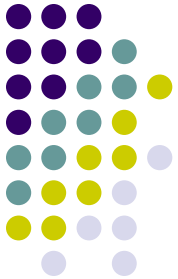
```
function fib(n : longint) : longint;
begin
  if (n < 2) then begin
   result := 1;
   exit;
  end;
  if (calculated[n]) then begin
   result := ans[n];
   exit;
  end;
  result := fib(n - 1) + fib(n - 2);
  calculated[n] := true;
  ans[n] := result;
end;
```

Ациклический граф вычислений

- Содержит n вершин
- Каждое число вычисляется ровно один раз

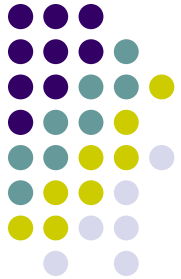


Что позволило ускорить вычисление?



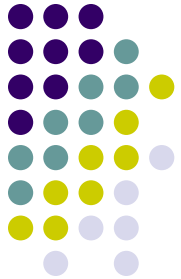
- Перекрывающиеся подзадачи (много одинаковых поддеревьев)
- Небольшое число различных подзадач (для вычисления Fib(n) – примерно n подзадач)
- Возможность записывать ответы для подзадач

Признаки возможности применения ДП



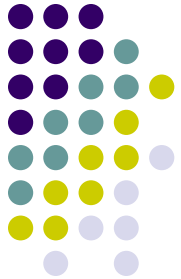
- Возможность разбиения задачи на подзадачи (метод «разделяй-и-властвуй»)
- Наличие свойства оптимальности для подзадач – оптимальный ответ для большой задачи строится на основе оптимальных ответов для меньших
- Наличие перекрывающихся подзадач

Этапы решения задачи методом динамического программирования



1. Разбиение задачи на подзадачи
2. Построение рекуррентной формулы для вычисления значения функции
3. Вычисление значения функции для всех подзадач
4. Восстановление структуры оптимального ответа

Задача о наибольшей общей подпоследовательности



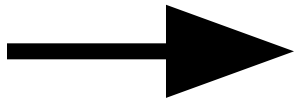
- На примере этой задачи будут рассматриваться указанные четыре этапа
- На базе этой задачи построена программа diff, которая используется в Linux для сравнения файлов
- Задача имеет приложения в биоинформатике

Постановка задачи

- Заданы две строки: $a_1a_2\ldots a_n$ и $b_1b_2\ldots b_m$
- Необходимо найти строку максимальной длины, которая встречается в обеих заданных строках как подпоследовательность

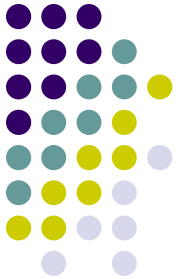
AGCAT

GAC



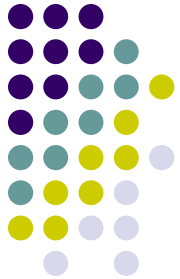
GA

Медленное решение



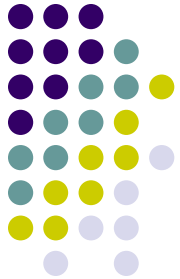
- Перебрать все подпоследовательности одной из строк и проверить их вхождение в другую строку
- Число подпоследовательностей строки длиной n – 2^n
- Поэтому время работы такого решения – $O(m2^n)$

Разбиение на подзадачи (1)



- Рассмотрим строки : $a_1a_2\ldots a_n$ и $b_1b_2\ldots b_m$
- Если последние символы совпадают ($a_n=b_m$), то их нужно включить в ответ и отбросить
- Если они различны, то нужно попробовать отбросить только a_n, а потом только b_m

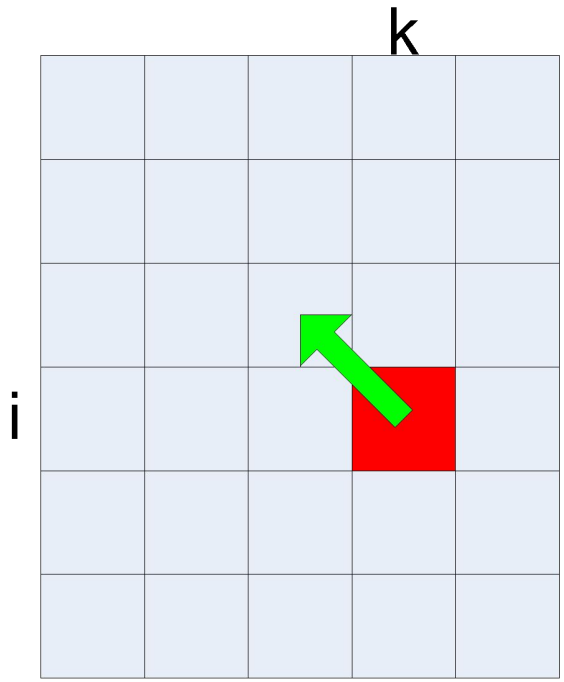
Разбиение на подзадачи (2)

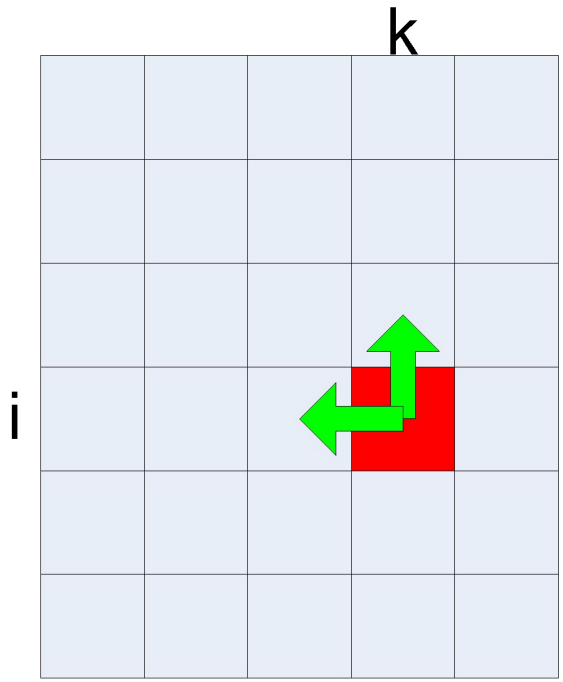


- Подзадачами являются задачи такого типа «Найти наибольшую общую подпоследвательность строк $a_1a_2\ldots a_i$ и $b_1b_2\ldots b_k$»
- Обозначим ответ (длину последовательности) на эту подзадачу как len[i][k]

Рекуррентная формула

$$\text{len[i][k]} = \begin{cases} len[i-1][k-1]+1, если\ \text{a}_\text{i} = b_k \\ \max(len[i-1][k], len[i][k-1]), если\ \text{a}_\text{i} \neq b_k \end{cases}$$



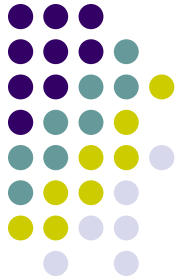


Начальные условия

- len[0][k] = 0 для всех k
- len[i][0] = 0 для всех i

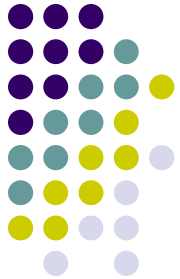
0	0	0	0	0
0				
0				
0				
0				
0				

Два метода вычисления



- «Сверху вниз» – рекурсия с запоминанием ответов
- «Снизу вверх» – заполнение таблицы

Метод «сверху вниз»

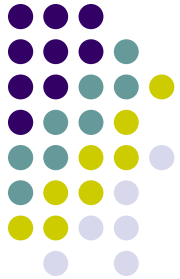


- Решение больш*их* подзадач начинается до того, как получены ответы для маленьких
- Маленькие решаются в процессе решения больших

Программа

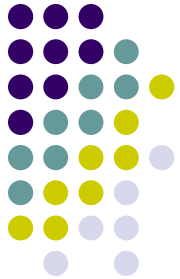
```
function calc(i, k : integer) : integer;
begin
   if (calculated[i][k]) then begin
    result := len[i][k];
    exit;
   end;
   if (i = 0) or (k = 0) then begin
    result := 0;
    exit;
   end;
   if (a[i] = b[k]) then begin
    result := calc(i - 1, k - 1) + 1;
   end else begin
    result := max(calc(i - 1, k), calc(i, k - 1));
   end;
   calculated[i][k] := true;
   len[i][k] := result;
end;
```

Преимущества и недостатки



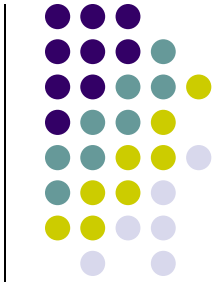
- Преимущества:
  - Достаточно просто пишется на основе рекуррентной формулы
  - Вычисляются ответы только для тех подзадач, которые действительно нужны
- Недостатки:
  - Некоторое замедление из-за накладных затрат на рекурсию

Метод «снизу вверх»



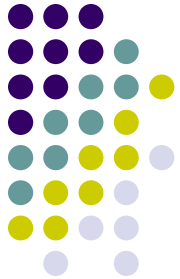
- Заполняется таблица ответов на подзадачи в порядке возрастания размера подзадачи
- Когда начинается решение большой подзадачи, меньшие уже решены

Программа



```
len[0][0] := 0;
for i := 1 to n do begin
  len[i][0] := 0;
end;
for k := 1 to m do begin
 len[0][k] := 0;
end;
for i := 1 to n do begin
  for k := 1 to m do begin
   if (a[i] = b[k]) then begin
       len[i][k] := len[i - 1][k - 1] + 1;
   end else begin
       len[i][k] := max(len[i-1][k], len[i][k-1]);
   end;
  end;
end;
```

Преимущества и недостатки



- Преимущества:
  - Требуется меньше памяти, чем при методе «сверху вниз» и отсутствует рекурсия
  - Быстрее работает в случае, когда необходимо вычислить ответы для всех подзадач
- Недостатки:
  - Порядок заполнения таблицы не всегда прост (например, может потребоваться заполнять по диагоналям)

Пример

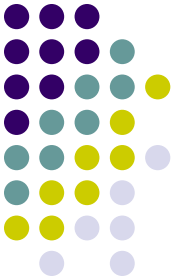
		A	G	C	A	T
	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1	1	2	2
C	0	1	1	2	2	2

Красный цвет – начальные условия

Зеленый цвет – случай $a_i = b_k$

Желтый цвет – случай $a_i \neq b_k$

Восстановление структуры оптимального ответа

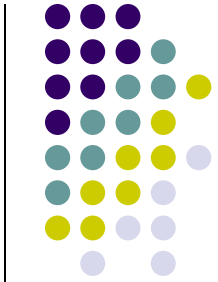


		A	G	C	A	T
	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1	1	2	2
C	0	1	1	2	2	2

Верхний путь – GA

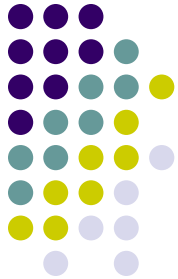
Нижний путь – AC

Программа



```
procedure restore(i, k : integer);
begin
  if (i = 0) or (k = 0) then begin
   exit;
  end;
  if (a[i] = b[k]) then begin
   restore(i - 1, k - 1);
   write(a[i]);
  end else begin
   if (len[i - 1][k] = len[i][k]) then begin
      restore(i - 1, k);
   end else begin
      restore(i, k - 1);
   end;
  end;
end;
```

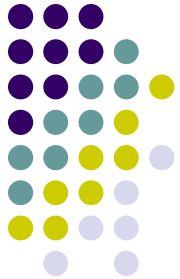
Как делать в общем случае?



- Такой метод работает в этой задаче, но не понятно, как его адаптировать к другим
- Общий метод состоит в том, чтобы запоминать, какой из вариантов в рекуррентной формуле был реализован

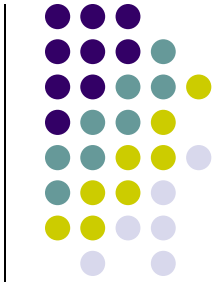
$$\text{len[i][k]} = \begin{cases} len[i-1][k-1]+1, если\ \text{a}_\text{i} = b_k \\ \max(len[i-1][k], len[i][k-1]), если\ \text{a}_\text{i} \neq b_k \end{cases}$$

Вычисление функции с запоминанием выбранного варианта



```
for i := 1 to n do begin
  for k := 1 to m do begin
   if (a[i] = b[k]) then begin
       len[i][k] := len[i - 1][k - 1] + 1;
       back[i][k] := 1;
   end else begin
       if (len[i - 1][k] > len[i][k - 1]) then begin
      len[i][k] := len[i-1][k];
      back[i][k] := 2;
       end else begin
      len[i][k] := len[i][k - 1];
      back[i][k] := 3;
       end;
   end;
  end;
end;
```

Восстановление ответа

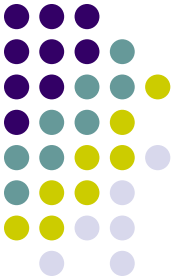


```
procedure restore(i, k : integer);
begin
  if (i = 0) or (k = 0) then begin
   exit;
  end;
  if (back[i][k] = 1) then begin
   restore(i - 1, k - 1);
   write(a[i]);
  end else if (back[i][k] = 2) then begin
   restore(i - 1, k);
  end else begin
   restore(i, k - 1);
  end;
end;
```

Упражнение – 1

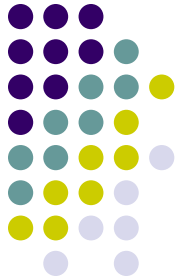
Путь с максимальной суммой. Задан прямоугольник размером *n* на *m*, в каждой клетке которого находится число. За один ход можно сдвинуться вверх или вправо. Необходимо найти путь из левого нижнего угла в правый верхний с максимальной суммой чисел в посещенных клетках

Упражнение – 2



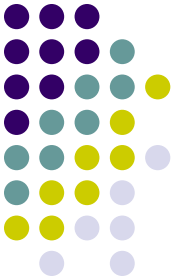
Число путей. Задан прямоугольник размером *n* на *m*, некоторые клетки которого вырезаны. За один ход можно сдвинуться вверх или вправо. Необходимо найти число путей из левого нижнего угла в правый верхний

Упражнение – 3



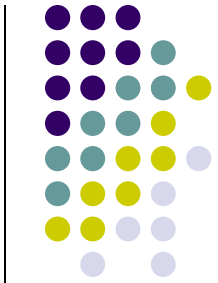
Максимальный подпалиндром. Задана строка. Необходимо найти наибольшую по длине подпоследовательность, которая является палиндромом (читается одинаково с обеих сторон)

Упражнение – 4

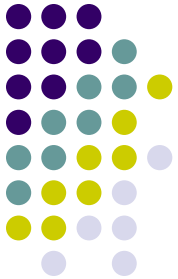


Наибольшая возрастающая подпоследовательность. Задана последовательность из *n* чисел. Необходимо найти ее наибольшую по длине подпоследовательность, числа которой расположены в возрастающем порядке

Литература



- Кормен, Лейзерсон, Ривест, Штайн «Алгоритмы. Построение и анализ», глава 15



Выводы

- Динамическое программирование – метод составления алгоритмов
- Оно применимо в случае наличия перекрывающихся подзадач
- Решение задачи методом ДП состоит из четырех этапов:
  1. Разбиение задачи на подзадачи
  2. Построение рекуррентной формулы для вычисления значения функции
  3. Вычисление значения функции для всех подзадач
  4. Восстановление структуры оптимального ответа

Спасибо за внимание!

Вопросы? Комментарии?

fedor.tsarev@gmail.com