Лекция 6. . Типология алгоритмических методов. Переборы. Динамическое программирование

К большому *классу P* относят задачи, решение которых получается за *полиномиальное время* $O(n^{const})$. На практике, как правило, const ≤ 5. В этом случае обычно считают, что задача решается достаточно *эффективно*.

Имеется также обширный класс (их известно более двух тысяч), который включает так называемые *NP-полные* задачи и не менее сложные задачи, к которым они сводятся (в некотором смысле, худшие из них), называемые поэтому *NP-трудными*. Этот класс содержит много практически важных задач.

Все NP-полные задачи эквивалентны по вычислительной сложности в том смысле, что если одна из них имеет эффективное, т.е. полиномиальное, решение, то и все они имеют эффективные решения. Если же будет доказано, что хоть одна NP-полная задача не решается проще, чем перебором, т.е. решается только алгоритмом со *сложностью экспоненциальной* (c^n, n^n, n!) или выше, то это будет относиться ко всем NP-полным задачам.

Тем не менее, несмотря на многолетние усилия специалистов, ни для одной из них до сих пор не удалось найти эффективного решения. Возникла не доказанная до сих пор, но весьма правдоподобная гипотеза (называемая P≠NP), что ни одну такую задачу нельзя решить эффективно [13, 59].

Другими словами, разумно в качестве естественнонаучной гипотезы считать, что *точное решение любой NP-полной задачи невозможно получить алгоритмом полиномиальной сложности***.** Поэтому про такие задачи обычно говорят, что они не решаются иначе, чем перебором.

Кроме NP-полных задач, для точного решения которых найдены только экспоненциальные алгоритмы (а возможность создания полиномиальных алгоритмов остается под вопросом), существуют задачи, которые заведомо можно решать только за *экспоненциальное время* $O(const^n)$. В таких случаях нет надежды на возможность получения точного решения задач требуемого для практики размера за приемлемое время, независимо от будущего прогресса в производительности компьютеров.

Если же задача за приемлемое время точно не решается, но решать ее все равно надо, приходится использовать *приближенный алгоритм* – такой алгоритм, для которого известна оценка точности. *Эвристическим* называют алгоритм, зарекомендовавший себя на практике, основанный на здравом смысле, догадках и т. п., для которого неизвестно, насколько получаемое решение далеко от точного.

6.1. Базисные методы разработки алгоритмов

Базисные схемы используют некоторые типовые шаблоны, применяемые для построения различных алгоритмов, например: итерацию, рекурсию, построение цикла на основе инварианта, разработку сверху вниз, рекурсивный

спуск, принцип “разделяй и властвуй”, ретроспективный анализ, перебор вариантов, динамическое программирование, индуктивное вычисление функций, автоматное программирование, жадные алгоритмы и др [7, 10, 13, 20, 33, 35 – 38, 59, 75].

Метод *итераций* (повторений) – это общая идея цикла. В первой части учебника [67] даны некоторые рекомендации по использованию циклов в рамках методики структурного программирования.

Другим способом описания в алгоритме повторяющихся действий является *рекурсия,* позволяющая во многих случаях, например, для данных рекурсивной структуры, быстро сформулировать компактный алгоритм, который, однако, как правило, требует больше памяти и времени, чем итеративный (циклический) алгоритм. Применение рекурсии поэтому бывает оправдано обычно только в тех случаях, когда трудно построить итеративный алгоритм.

Много полезных примеров *построения циклов на основе инварианта* дано в книгах [3, 75]; см. также раздел о методах сортировки в учебнике [67]. Этот подход позволяет строить программу параллельно с обоснованием ее корректности.

Метод *сверху вниз* означает движение в направлении от объекта, изображаемого корнем дерева, к его частям – в сторону листьев. Этот термин может характеризовать организацию (направленность) как процесса разработки алгоритма, так процесса его выполнения. Примеры обоих вариантов этого подхода есть в обеих частях данного учебника. В частности, в разделе 7 рассматривается *рекурсивный спуск* – метод анализа и обработки алгоритмом его входных данных в направлении сверху вниз: от более крупных конструкций к их составным частям.

Принцип “*разделяй и властвуй*” также может определять характер и процесса построения алгоритма (например, методом сверху вниз), и процесса его работы. В этом случае сначала выполняется *декомпозиция* (разделение) исходной задачи на подзадачи, а затем *композиция* (объединение) решений подзадач для получения решения исходной задачи.

Термин “ретроспективный” означает движение назад, к прошлому, в отличие от слова “перспективный” - направленный к будущему. *Ретроспективный анализ* (отрабатывание назад) – это ход рассуждений при разработке алгоритма или его действия при решении задачи в направлении (обратном обычному) от требуемого результата задачи к ее исходным данным.

Решение ряда задач ищется методом *перебора вариантов*. Такой перебор зачастую требует экспоненциального времени, что во многих случаях неприемлемо для задач практического размера.

Метод *динамического программирования* использует идею *обобщения* или *погружения* – сводит исходную задачу к решению семейства подзадач (включающего исходную) и запоминанию их ответов, чтобы не тратить время на их повторное получение (экономия времени благодаря дополнительной памяти). Такой подход в ряде случаев позволяет эффективно решать задачи, для

которых переборный алгоритм требует экспоненциального времени, и его можно рассматривать также как способ сокращения перебора.

Индуктивное вычисление функции предоставляет некоторый аппарат для построения однопроходных алгоритмов обработки входных последовательностей данных.

Автоматное программирование (программирование с явным выделением состояний) – это одно из новых направлений в технологии программирования, в котором процесс решения задачи рассматривается как последовательность переходов абстрактного автомата из одного состояния в другое. Алгоритм описывается в виде абстрактного автомата, задаваемого правилами (таблицей, графом) перехода из каждого возможного состояния в другие в зависимости от входных данных. Ранее подобный подход применялся в основном лишь для построения программ синтаксического и лексического анализа текста в трансляторах (см. раздел 7).

Жадным называют алгоритм решения **задачи оптимизации,** старающийся сделать максимально выгодным каждый шаг (рассчитывающий развитие ситуации не более чем на один “ход” вперед). Естественно, что во многих случаях при таком подходе получается лишь локально оптимальное решение, не являющееся оптимальным в целом, но иногда этот метод срабатывает.

Примером жадного алгоритма является алгоритм Дейкстры для определения во взвешенном орграфе расстояний от заданной вершины до всех остальных (раздел 4.4).

6.2. Методы перебора

Для поиска решения многих задач требуется перебор элементов некоторого конечного множества, заведомо содержащего все возможные варианты решения. Целями такого перебора может быть

- поиск *хотя бы одного* решение задачи;
- получение *всех* решений задачи;
- нахождение *оптимального* решения задачи (наилучшего по некоторому критерию);
- подсчет *количества* решений задачи.

В ряде случаев необходим и возможен полный перебор всех вариантов. В некоторых ситуациях этот перебор удается сократить.

6.2.1. Полный перебор

При *полном переборе* решение задачи включает просмотр всех элементов некоторого множества. При этом очередной элемент строится либо напрямую, заново, “с нуля” (*прямой перебор)*, либо получается из предыдущего элемента

(в некотором порядке). Примерами второго подхода являются алгоритмы получения перестановок и сочетаний из раздела 6.3.

Метод решета состоит в том, что из множества возможных вариантов убираются все неподходящие кандидаты и остаются только решения задачи.

Пример 6.1. Решето Эратосфена. Среди целых чисел от 1 до K требуется отыскать все простые числа, т. е. числа, которые делятся без остатка только на себя и единицу. Выписываются все целые числа от 1 до K. Из них вычеркивается каждое второе число после числа 2, каждое третье число после числа 3 и т. д. (каждое n-е число после числа n, где n - наименьшее из оставшихся чисел). Останутся только нужные простые числа.

6.2.2. Перебор с возвратом

Перебор с возвратом **(***поиск с возвращением, бэктрекинг*) представляет собой общий метод организации исчерпывающего перебора с отсечением заведомо неприемлемых вариантов. Перебор с возвратом используется для нахождения *всех* возможных способов, *хотя бы одного* способа или *наилучшего* способа решения задачи. Решение задачи ищется как последовательность шагов - элементов решения

e[0], e[1], ... , e[k]

(k < R, R может быть неизвестно). Каждый элемент решения e[k] выбирается путем перебора из множества альтернатив (вариантов) A[k], в общем случае зависящего от предыдущих элементов

e[0], ... , e[k-1].

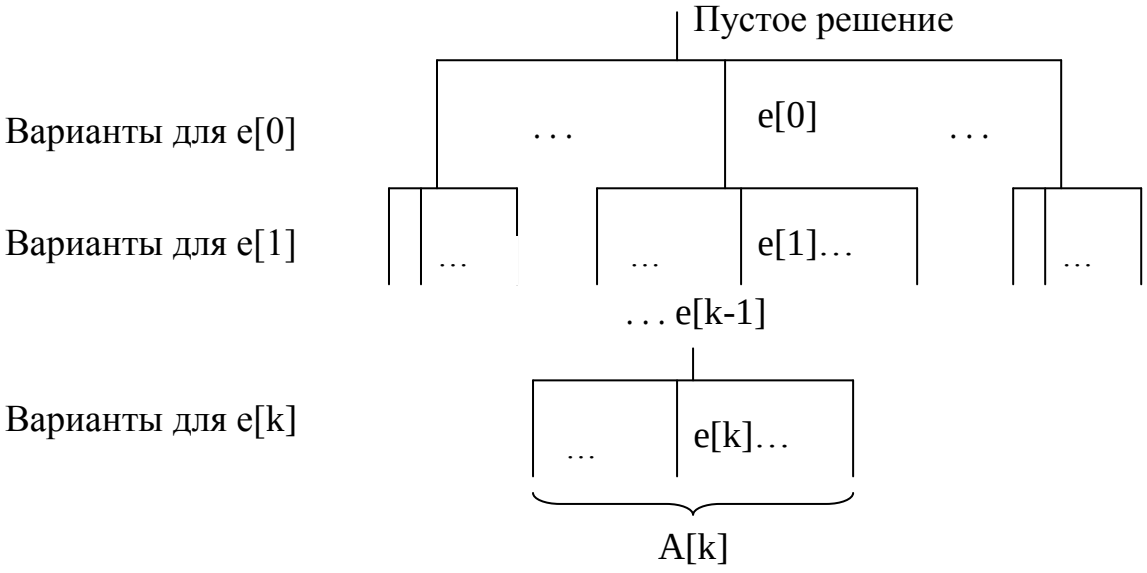


Рис. 6.1. Дерево вариантов решения при поиске с возвращением

В случае тупика, когда для элемента e[k] все альтернативы из множества A[k] уже рассмотрены (или A[k] пусто), происходит возвращение (откат), по-английски – backtracking, на один элемент назад, рассматривается следующая

альтернатива для элемента a[k-1] и т.д. Таким образом, происходит *обход в глубину дерева вариантов* решения (рис. 6.1).

Поскольку последний полученный элемент удаляется первым, элементы решения e[0], ... , e[k] образуют стек (алг. 6.1).

Поиск с возвращением легко описывается рекурсивно (алг. 6.1): перебор расширений частичного решения (первоначально пустого) состоит из получения всех вариантов очередного шага и перебора их расширений. Явный стек и возвращения отсутствуют: они скрыты в механизме рекурсии.

Если достаточно получить одно решение задачи, после нахождения первого решения поиск прекращается.

Алгоритм 6.1. Рекурсивный поиск с возвращением всех решений задачи

```
/* Рекурсивный перебор всех решений, являющихся          */
/* расширением последовательности e[0], ..., e[k-1]      */
void Backtracking (ШАГ e[], int k)
{   if (e[0]...e[k-1] - решение)
        Запомнить e[0]...e[k-1];
    for (каждого допустимого варианта шага v из A[k])
    { e[k] = v;                  /* Вперед: v - в стек    */
       Backtracking (e, k+1);
    }
}           . . .

#define R 100           /* Максимальная длина решения     */
ШАГ x[R];               /* Вектор текущего решения        */
   . . .
Backtracking (x, 0);    /* Поиск всех решений задачи      */
```

В алгоритме 6.2. приведен итеративный вариант перебора с возвратом.

Алгоритм 6.2. Итеративный поиск с возвращением всех решений задачи

```
k = 0;
while (k >= 0)                  /* Стек не пуст                                  */
{  if (в  A[k] существует еще не рассмотренный вариант  v
       очередного шага решения)          /* Можно продолжать решение          */
   {  e[k] = v;                 /* Вперед: v - в стек                            */
       if  (e[0]  . . . e[k] - решение)
       {    Запомнить e[0]...e[k];
            k--;                /*  или   k++;  в  зависимости от задачи         */
       }else
            k++;
    }
    else k--;                   /* Назад: удаление верхнего элемента стека       */
}
```

Для случая, когда возможные варианты каждого шага решения перенумерованы и известно их количество (возможно, зависящее от предыдущих шагов), итеративный алгоритм 6.2 можно конкретизировать (алг. 6.3), используя следующие обозначения:

A[k,m] - m-й вариант k-го шага решения ($0 \le m < N[k]$),
N[k] - количество вариантов k-го шага решения задачи,
e[k] - номер варианта k-го шага текущего частичного решения,
k - указатель стека e.

Алгоритм 6.3. Поиск с возвратом всех решений задачи

```
k=0; e[0]=0;                    /* Начальный вариант e[0]                    */
while (k > 0);                  /* Стек не пуст                              */
{  while (e[k]<N[k] && e[k] противоречит задаче)
     e[k]++;                    /* Отбросить заведомо неверное решение       */
  if (e[k]<N[k])                /* A[0,e[0]]...A[k,e[k]] - часть решения     */
  {  /*          Вперед: очередной вариант шага e[k] - в стеке               */
     if (A[0,e[0]]...A[k,e[k]] - полное решение)
     {  Запомнить решение A[0,e[0]]...A[k,e[k]];
           Изменить k и e[k];   /* Назад или вперед, зависит от задачи       */
     }else
     {  k++;
        e[k]=0;                 /* Начальный вариант следущего шага          */
     }
  }
  else                          /* Назад                                     */
  {  k--;                       /* Удалить e[k] из стека                     */
     e[k]++;                    /* Следующий вариант e[k]                    */
  }
}
```

Методы ускорения поиска с возвращением:

- *поиск с ограничениями (отсечение ветвей)* - отбрасывание бесперспективных поддеревьев; частный случай (*метод ветвей и границ)* - отбрасывание худших, чем ранее найденные, частичных решений;

- *слияние ветвей* (склеивание) - исключение повторного просмотра одинаковых (изоморфных) поддеревьев;

- *переупорядочение поиска*: начинать с более перспективных вариантов и вершин с меньшим числом сыновей. Это - эвристический прием. *Эвристика* – это метод, основанный на здравом смысле, не обоснованный формально.

Примеры перебора вариантов для задач на графах рассмотрены в разделах 4.5 и 4.6.

Пример. 6.2. Задача о восьми ферзях. На шахматной доске размером 8×8 полей требуется расставить максимальное число мирных ферзей (не находящихся под боем друг друга). Ферзь – это фигура, которая держит под боем другие фигуры, находящиеся с ним на одной горизонтали, вертикали или диагонали на любом расстоянии.

Эта классическая головоломка известна с 1848 г. Ее решал, в частности, великий математик К. Гаусс. Были найдены отдельные расстановки, но много лет никто не знал, сколько их всего существует. С помощью компьютера методом бэктрекинга за доли секунды можно получить все 92 решения.

На каждой вертикали (и горизонтали) может находиться только один ферзь, и поэтому на доске можно расставить не более 8 мирных ферзей.

Поскольку никакой формулы расстановок, видимо, не существует, задачу приходится решать методом перебора. Полный перебор потребует проверки всех возможных сочетаний из 64 по 8 полей - всего $C^8_{64} = (64 \cdot 63 \cdot 62 \cdot \ldots \cdot 57)/8!$, т.е. около $4.4 \cdot 10^9$ вариантов расстановки 8 ферзей.

Перебор можно сократить, если ставить k-го ферзя на одно из 8 полей k-й вертикали (или горизонтали). Тогда в полном переборе остается $8^8 = 2^{24} = 16777216$, т.е. в 260 раз меньше вариантов. Перебор с возвратом еще в 8160 раз сократит это количество до проверки 2056 узлов дерева вариантов.

Решением задачи будет массив f из 8 чисел, в котором f[k] равно номеру горизонтали от 1 до 8, на которой расположен k-й ферзь (стоящий на k-й вертикали). Ферзи выставляются на доску по одному так, чтобы не быть под боем уже стоящих ферзей. Для каждого ферзя имеется 8 пронумерованных вариантов размещения. Поэтому удобно использовать алгоритм 6.3. В этой задаче механизм поиска с возвратом очень нагляден, и ее рассматривают во многих учебниках программирования [15, 16, 22, 49 и др.].

6.3. Подсчет и порождение комбинаторных объектов

В ряде задач, например, при поиске решения методом перебора, требуется получить один за другим все элементы некоторого множества. Каждый из полученных элементов чаще всего сразу же используется для решения задачи и потом не сохраняется. В этих случаях удобно иметь подпрограмму получения элемента множества, следующего в определенном порядке за данным элементом.

Рассмотрим получение последовательностей из целых чисел от 1 до n. Будем получать их в лексикографическом (*словарном*) порядке. Последовательность A предшествует последовательности B в *лексикографическом порядке*, если для некоторого j их начальные отрезки длины j равны, а (j+1)-й член в последовательности A меньше.

Пример 6.2. Генерация перестановок. Требуется получить все n! *перестановок* чисел 1, ..., n, т.е. последовательностей длины n, в которые по

одному разу входит каждое из чисел 1, ..., n. Первой в лексикографическом порядке будет перестановка <1, 2, ..., n>, последней - <n, ..., 2, 1>. Пусть в массиве x с индексами от 0 до n-1 находится последняя найденная перестановка. Составить подпрограмму получения следующей перестановки в этом же массиве.

Решение представлено в алгоритме 6.4. Чтобы новая перестановка была непосредственно следующей, например, за перестановкой

2 1 4 6 5 3

необходимо, не изменяя предыдущих членов, минимально увеличить как можно более близкий к концу массива член x[k].

Алгоритм 6.4. Получение в массиве x длины n следующей в лексикографическом порядке перестановки чисел 1, 2, ..., n. Значение: 1 - перестановка получена, 0 - нет больше перестановок

```
int sl_perest (int x[], int n)
{  int i, j, k;
   for (k=n-2; k>=0 && x[k]>x[k+1]; k--);
   if(k<0) return 0;     /* Перестановка x[0]...x[n-1] последняя */
                         /* x[k] < x[k+1] > ... > x[n-1]         */
   for (j=n-1; x[j]<x[k]; j--);
                         /* x[k+1]>...>x[j]>x[k] >...>x[n-1]     */
   i=x[j]; x[j]=x[k]; x[k]=i;            /* Обмен x[k] <--> x[j] */
   for (k++,j=n-1; k<j; k++,j--)
   {  i=x[j]; x[j]=x[k]; x[k]=i;         /* Обмен x[k] <--> x[j] */
   }
   return 1;
}
```

Это можно сделать, обменяв x[k] с наименьшим из превышающих его последующих членов x[j] (поскольку предыдущие не изменяются). Для этого необходимо найти наибольшее такое k, что x[k] < x[k+1] > ... > x[n-1]. Затем отыскивается наибольшее j > k такое, что x[j]>x[k]. При этом справедливы неравенства:

x[k] < x[k+1] > x[k+2] > ... > x[j] > x[k] > x[j+1] >...>x[n-1].

В данном примере x[k]=4, x[j]=5 (подчеркнуты). После обмена получим:

2 1 5 6 4 3

Затем необходимо расположить следующие за x[k] числа (подчеркнуты):

2 1 5 6 4 3

так, чтобы перестановка была наименьшей, т.е. по возрастанию. Поскольку они убывают, достаточно расположить их в обратном порядке:

2 1 5 3 4 6.

В главной программе можно просмотреть все перестановки чисел от 1 до n, например, следующим образом:

```
#define NMAX 10
int x[NMAX];
                . . .
for (j=0; j<n; j++)  x[j]=j+1;          /* 1, 2, 3, ..., n   */
do
   Использование перестановки x[0], ..., x[n-1];
while (sl_perest(x, n));
```

Например, следущий фрагмент программы выведет построчно в алфавитном порядке все 5! = 120 перестановок букв A, B, C, D, E.

```
int k[5];  char t[]="ABCDE";
                . . .
for (j=0; j<5; j++)  k[j]=j+1;          /* 1, 2, 3, ..., 5   */
do
{  putchar('\n');
   for(j=0; j<5; j++) putchar(t[k[j]-1];
}while (sl_perest(k, 5));
```

Пример 6.3. Генерация сочетаний. Требуется получить все k-элементные подмножества множества {1, ..., n}, т.е. *сочетания* из указанных n чисел по k чисел. Чтобы каждое подмножество имело ровно одно представление, будем перечислять его элементы в возрастающем порядке. Таким образом, необходимо в лексикографическом порядке перечислить все возрастающие последовательности длины k из чисел 1 , ..., n. Например, при n = 5, k = 2 получим:

1, 2; 1, 3; 1, 4; 1, 5; 2, 3; 2, 4; 2, 5; 3, 4; 3, 5; 4,5.

Первая последовательность: 1, 2, ..., k; последняя последовательность: (n-k+1), ..., (n-1), n.

Пусть в массиве x с индексами от 0 до n-1 находится последнее найденное сочетание. Составить подпрограмму получения следующего сочетания в этом же массиве. ***Решение*** дано в алгоритме 6.5.

Алгоритм 6.5. Получение в массиве x следующего в лексикографическом порядке сочетания из 1, 2, ..., n по k чисел - возрастающей последовательности длины k из чисел 1, ..., n. Значение: 1 - сочетание получено, 0 - нет больше сочетаний

```
int sl_sochet (int x[], int n, int k)
{  int j;
   for (j=k-1; j>=0 && x[j]>n-k+j; j--);
   if(j<0) return 0;   /* Сочетание x[0]...x[k-1] последнее */
   x[j]++;
   for (j++; j<k; j++)  x[j]=x[j-1]+1;
   return 1;
}
```

Чтобы новое сочетание было непосредственно следующим в лексикографическом порядке, в текущем сочетании ищется самый правый член x[j], который можно увеличить (не достигший максимального значения n-k+1+j). Этот член увеличивается на 1, а всем последующим членам

присваиваются наименьшие возможные значения, т. е. возрастающие с шагом 1 числа (т. к. последовательность должна быть возрастающей).

В главной программе можно просмотреть в лексикографическом порядке все сочетания из чисел 1, ..., n по k штук, например, следующим образом:

```
#define NMAX 10
int x[NMAX];
                . . .
for (j=0; j<k; j++)  x[j]=j+1;          /* 1 2 3 ... k       */
do
   Использование сочетания x[0]...x[k-1];
while (sl_sochet (x, n, k));
```

Пример 6.4. Требуется подсчитать число последовательностей длины n (n≥0) из нулей и единиц, не содержащих двух единиц подряд (назовем такую последовательность “правильной”).

Решение. Обозначим искомую величину K(n). Если правильная последовательность длины n заканчивается нулем, то ее началом может быть любая правильная последовательность длины n-1, а их число равно K(n-1).

Если же в конце правильной последовательности длины n расположена единица, то перед ней может быть только ноль, а перед ним может находиться любая правильная последовательность длины n-2. Количество таких вариантов равно K(n-2). Отсюда получим рекуррентную формулу для искомой величины

$$K(0) = 1, \quad K(1)=2, \quad K(j) = K(j-1) + K(j-2) \quad \text{для } 1 < j \le n.$$

Таким образом, для получения K(n) необходимо знать K(n-1) и K(n-2), а для их вычисления нужно, в конечном счете, знать значения K(j) для всех j от 0 до n-3 (это – так называемые *числа Фибоначчи*). Будем вычислять эти числа по возрастанию j. Для вычисления очередного значения K достаточно помнить два предыдущих; обозначим их K1 и K2. Получим следующий алгоритм.

```
if (n==0)   K=1;
else   if (n==1)   K=2;
else
{  K2=1;  K1=2;
    for (j=2; j<=n; j++)
    {  K=K2+K1;
        K2=K1;  K1=K;
    }
}
```

Чтобы вычислять K(0) и K(1) по общей формуле (избавиться от краевого эффекта приемом “*кайма*” - частным случаем метода *обобщения*), продолжим эту последовательность в сторону меньших аргументов и будем считать K(-2)=0, K(-1)=1. Тогда алгоритм будет проще (но чуть медленнее):

```
K2=0;  K1=1;
for (j=0; j<=n; j++)
{  K=K2+K1;
    K2=K1;  K1=K;
}
```

Пример 6.5. Вычисление числа сочетаний. Обозначим $C(n, k) = C_n^k$ - число сочетаний из n объектов по k (число k-элементных подмножеств n-элементного множества).

Решение 1. Можно воспользоваться формулой

$$C(n,k) = n! / (k! * (n-k)!)$$

Однако при этом возможно переполнение, т. к. значения факториалов значительно превышают число сочетаний.

Решение 2. Если выбрать некоторый элемент n-элементного множества и отдельно подсчитать k-элементные множества, включающие и не включающие этот элемент, то получится известное соотношение

$$C(n, k) = C(n-1,k-1)+C(n-1,k), \qquad (6.1)$$

. Таблица значений C(n,k) имеет вид так называемого треугольника Паскаля:

```
      1
    1   1
  1   2   1
1   3   3   1
    …
```

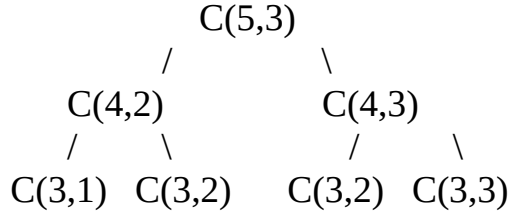
В нем каждый элемент, кроме крайних единиц, равен сумме двух стоящих над ним.

Составим таблицу значений функции C(n,k), заполняя ее для n = 0, 1, 2,..., пока не дойдем до интересующего нас элемента.

Таблица занимает место порядка n^2, но его можно сократить до n, если заметить, что для вычисления следующей строки треугольника Паскаля нужна только предыдущая. Время работы в обоих случаях порядка n^2.

Рекурсивная программа на основе формулы (6.1) потребует существенно большего времени, поскольку вызов C(n,k) сводится к двум вызовам для C(n-1,…), а они - к четырем вызовам для C(n-2,…) и т. д. Таким образом, время оказывается экспоненциальным (порядка 2^n). Используемая рекурсивной версией память равна произведению глубины рекурсии (n) на количество памяти, используемое одним экземпляром процедуры (константа), т.е. пропорциональна n.

Кардинальный выигрыш во времени при переходе от рекурсивной версии к нерекурсивной связан с тем, что в рекурсивном варианте одни и те же вычисления происходят много раз. Например, вызов C(5,3) в конечном счете порождает два вызова C(3,2):



Заполняя таблицу, мы каждую клетку заполняем только однажды - отсюда и экономия. Это - метод динамического программирования.

6.4. Динамическое программирование

Динамическое программирование может рассматриваться как развитие идеи ретроспективного анализа.

Общая схема рассуждений следующая. Сначала выявляются подзадачи, решение которых позволило бы получить решение исходной задачи. Обычно используются рекуррентные соотношения. Затем таким же образом определяются подзадачи, необходимые для решения выявленных подзадач и т.д. В результате образуется семейство задач уменьшающегося размера, включающее исходную задачу.

Разрабатываемый алгоритм решает сформулированные задачи от меньших к большим и запоминает найденные ответы (в массивах). Эти ответы используются в алгоритме для решения последующих задач, включая исходную задачу.

Этот прием использовался в примерах 6.4 и 6.5 для подсчета комбинаторных объектов. Он применим в тех случаях, когда объем хранимой информации оказывается не слишком большим. Обычно это бывает, когда удается обойтись не более чем двумерными массивами.

Преимущество динамического программирования в экономии времени - если уж подзадача решена, то ее ответ где-то хранится и никогда не вычисляется заново. Метод может оказаться неприменимым, если требуемая для ответов память превысит допустимые возможности.

Методом динамического программирования, в частности, можно находить кратчайшие пути и расстояния между вершинами графа или орграфа (алгоритм Флойда), его транзитивное замыкание и компоненты связности (алгоритм Уоршалла) – см. Лекция 4, раздел 4.4.

Пример 6.6. Задача "Треугольник" [32]. Составить программу вычисления наибольшей суммы чисел, расположенных на пути от вершины до основания числового треугольника, например (нужный путь подчеркнут):

```
      7
    3   8          - каждый шаг - вниз по диагонали влево или вправо;
  8   1   0        - число строк треугольника от 2 до 100;
2   7   4   4      - треугольник состоит из чисел от 0 до 99.
4 5   2   6   5
```

Ответ: 30

Входной файл для этого примера:	Выходной файл:
5 - количество строк треугольника	30

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

Решение. Типичная задача для метода динамического программирования. Обозначим:

A[i,j] - j-е число i-й строки треугольника;

S[i,j] - наибольшая сумма чисел на пути от вершины A[1,1] до A[i,j]

Очевидны рекуррентные соотношения:

S[1,1] = A[1,1]

S[i,j] = max(S[i-1,j], S[i-1,j-1]) + A[i,j] $(2 \le j \le i \le N)$

Полагаем S[i,0] = S[i,i+1] = 0 $(1 \le i \le N)$ - прием "кайма".

Искомый ответ равен max S[N,j] для $1 \le j \le N$.

Алгоритм: по исходному треугольнику строить треугольник из чисел S[i,j]. Можно размещать S в исходном треугольнике A. Ответом будет максимальное число последней строки. Для приведенного в задаче примера числа S[i,j] будут иметь вид:

S[i,j]		A[i,j]
0 7 0	Исходный треугольник:	7
0 10 15 0		3 8
0 18 16 15 0		8 1 0
0 20 25 20 19 0		2 7 4 4
0 24 30 27 26 24 0		4 5 2 6 5

Полный перебор путей потребовал бы времени порядка $O(2^N)$, что совершенно нереально для N=100 (если рассматривать 10^6 вариантов в секунду, то для 2^{100} вариантов потребуется свыше $3 \cdot 10^{16}$ лет, что в миллионы раз превышает возраст Вселенной).

Усложнение задачи: определить не только сумму чисел оптимального пути, но и сам путь. Возможны разные решения:

1. От полученного максимального числа последней строки двигаться назад, переходя на каждом шаге от S[i,j] к числу max(S[i-1,j], S[i-1,j-1]).
2. В дополнительном треугольнике P при вычислении каждого числа S[i,j] запоминать, откуда оно получилось: из S[i-1,j] или из S[i-1,j-1], например:

P[i,j] =1, если S[i-1,j] > S[i-1,j-1], и 0 в противном случае.

Тогда можно будет проследить путь наверх от максимума последней строки чисел S[i,j]. Подобный метод используется для получения кратчайших путей при обходе графа в ширину (алг. 4.3), в алгоритме Дейкстры (4.11) и Флойда (алг. 4.12).

Пример 6.7. Задача “(Много)значащие нули” [86]. Даны целые десятичные числа N, B и K ($N \le 2 \cdot 10^9$, $2 \le B \le 10$, $0 \le K \le 100$). Найти

количество целых чисел от 1 до N, запись которых в системе счисления с основанием B содержит K значащих нулей. Например, в диапазоне от 1 до 30 имеется пять чисел с двумя значащими нулями в троичной системе счисления: 100, 200, 1001, 1002, 1010.

Решение (метод динамического программирования). Для B-ичного представления чисел обозначим:

kch(N,J) - количество чисел из множества 1..N, имеющих J нулей;
kzn(N) - количество нулей в B-ичной записи числа N,
D - количество цифр числа N.

Представим N в виде N = B * Q + R, где Q = N / B (целая часть частного), R = N % B (0 ≤R<B). Тогда B-ичная запись числа N примет вид числа Q, к которому справа приписана цифра R. Поэтому kzn(N) = kzn(Q) + kzn(R). Отсюда: kzn(N) = kzn(Q*B+R) = 0 при 0 < N < B (одна ненулевая цифра R),

kzn(N) = kzn(Q*B+R) = kzn(Q) + 1 при R=0, (1)
kzn(N) = kzn(Q*B+R) = kzn(Q) при R>0

Кроме того kch(N,J) = 0 при J < 0 или J ≥ D,
kch(N,J) = старшая цифра N при J = D-1 (числа из одной цифры от 1 до старшей цифры N и D-1 нулей).

Числа 1, 2, 3, ..., N=Q*B+R, Q*B+R+1, …, Q*B+B-1 разобьем на множества в зависимости от значения младшей цифры R=N%B:

R	Множество чисел				Слагаемые kch(N,J) - количества чисел от 1 до N=Q*B+R, имеющих J значащих нулей		
> 0	1,	2,	…,	B-1	B-1 при J=0,	0	при J>0
0	B,	2*B,	…,	Q*B	0 при J=0,	kch(Q,J-1)	при J>0
1	B+1,	2*B+1,	…,	Q*B+1			
2	B+2,	2*B+2,	…,	Q*B+2			
		. . .			(B-1)* kch(Q,J)		
R	B+R,	2*B+R,	…,	Q*B+R=N	Чисел		
R+1	B+R+1,	2*B+R+1,	…,	Q*B+R+1			
		. . .		. . .			
B-1	B+B-1,	2*B+B-1,	…,	Q*B+B-1			

отнять (B-1-R) чисел, если они имеют по kzn(Q)=J нулей

Отсюда получим

kch(N,J) = kch(Q,J-1)+(B-1)*kch(Q,J) - (B-R-1)*(kzn(Q)=J) при J>0, (2)
= B-1 + (B-1)*kch(Q,0) - (B-R-1)*(kzn(Q)=0) при J=0,
= 0 при J < 0 или J ≥ D (количество цифр числа N),
= старшая цифра N при J = D-1,

где выражение (kzn(Q)=J) равно 1 при kzn(Q) = J или 0 в противном случае.

Таким образом, задача для заданных N чисел сводится к задаче для N / B чисел, которая, в свою очередь, сводится к задаче размера, в B раз меньшего, и

т. д. до меньшего чем B размера, равного старшей цифре N (операция N/B удаляет младшую цифру числа N).

Значения размера задачи N, его младшей цифры R и функций kzn и kch удобно хранить в массивах: N[1]=N, N[i]=N[i-1] / B, R[i]=N[i] % B; kzn[i]=kzn(N[i]), kch[i,j]=kch(N[i],j).

Роль Q для N[i] играет N[i+1]=N[i] / B. Тогда в программе после замены Q на i+1 формулы (2) примут вид

kch[i,j] = kch[i+1,j-1] + (B-1)*kch[i+1,j] - (B-R[i]-1)*(kzn[i+1]=j) при j > 0 (3)
= B-1 + (B-1)*kch[i+1,0] - (B-R[i]-1)*(kzn[i+1]=0) при j = 0
= N[D] при i+j = D
= 0 при i+j > D

Пример. N=7436, B=3, K=4. Ответ = kch(7436,4) = kch[1,4] = 997 (подчеркнут).

kch[i,j] = kch[i+1,j-1] + 2*kch[i+1,j] - (2-R[i]) * (kzn[i+1]=j) при j > 0
= 2 + 2*kch[i+1,0] - (2-R[i]) * (kzn[i+1]=0) при j = 0
= N[D] при i+j = D
= 0 при i+j > D

kch[i,j] = kch(N[i],j)

$N[i]_3$	N[i]	R[i]	kzn[i]	i	j= 0	1	2	3	4	5	6	7	8
101012102	7436	2	3	1	510	1538	2118	1772-0	997	386	99	15	1 0
10101210	2478	0	3	2	254	642	740-2	517	240	73	13	1	0
1010121	826	1	2	3	126	258	242-1	138	51	11	1	0	
101012	275	2	2	4	62	98	72-0	33	9	1	0		
10101	91	1	2	5	30	34	20-1	7	1	0			
1010	30	0	2	6	14	12-2	5	1	0				
101	10	1	1	7	6	4-1	1	0					
10	3	0	1	8	4-2	1	0						
1	1	1	0	9	1	0	D=9						

Отсюда алгоритм:

```
Ввод N[1], B, K;
for (i=1; N[i]>=B; i++)  {  N[i+1] = N[i] / B;  R[i] = N[i] % B;  }
D=i;                              // если N[i] =B*Q+R[i], то N[i+1]=Q,
i=1..D-1
kzn[D]=0;  kch[D,0] = N[D];   kch[D,1]=0;
for (i=D-1; i>0; i--)  {
    f (R[i]==0) kzn[i]=kzn[i+1]+1; else kzn[i]=kzn[i+1];
    kch[i,0] = B-1 + (B-1)*kch[i+1,0] - (B-R-1) * (kzn[i+1]==0);
}
for (j=1; j<=K; j++) {
    kch[D-j,j]=N[D];  kch[D-j,j+1]=0;
    for (i=D-j-1; i>0; i--)
        kch[i,j] = kch[i+1,j-1] + (B-1)*kch[i+1,j] - (B-R[i]-
1)*(kzn[i+1]==j);
}
Вывод N[1,K];
```

Пример 6.8. Задача "Книги" [86]. На полке стоят N книг разных авторов. Справа или слева берется книга и ставится на другую полку справа или слева

от стоящих там книг. Вначале на другой полке книг нет. Найти число способов поставить таким способом на другую полку все книги в лексикографическом порядке по кодам ASCII фамилий авторов и возрастанию номеров томов одного и того же автора (способы отличаются порядком взятия книг).

Входной файл - расположение книг на первой полке слева направо (фамилии авторов):

N (0 < N ≤ 50)
Фамилия_1 [номер_тома_1] [ ] - необязательность присутствия
. . .
Фамилия_N [номер_тома_N]

Фамилия содержит от 1 до 40 символов ASCII, кроме пробелов. Номер тома (от 1 до 200), если он есть, отделен от фамилии одним пробелом. В списке книг нет одинаковых строк.

Выходной файл - найденное количество способов.

Примеры выходного файла	Выходной файл
3 Dumas 13 Dumas 3 Gomer	3
3 Dumas 3 Dumas 13 Gomer	2

Решение. Метод - динамическое программирование. Обозначим:

n - количество книг (количество фамилий авторов);
k[i,j] - количество способов получения алфавитного порядка фамилий, если i фамилий взято от начала списка (слева) и j фамилий взято от конца - справа ($0 \le i, j \le n$);
imin[i] - индекс "минимальной" (начальной по алфавиту) фамилии из первых (левых) i фамилий;
imax[i] - индекс "максимальной" фамилии из левых i фамилий;
jmin[j] - индекс "минимальной" фамилии из правых j фамилий;
jmax[j] - индекс "максимальной" фамилии из правых j фамилий.

Список "i + j" - упорядоченную по алфавиту последовательность из i первых (левых) и j последних (правых) заданных фамилий - можно получить добавлением i-й от начала фамилии к списку "(i-1) + j" или добавлением j-й от конца фамилии к списку "i + (j-1)", если добавляемая фамилия по алфавиту "меньше" или "больше" всех ранее взятых ("меньше" обоих минимумов или "больше" обоих максимумов). Отсюда:

k[i,j] = 0 + k[i-1,j] при fam[i] < (fam[imin[i-1]], fam[jmin[j]])

или fam[i] > (fam[imax[i-1]], fam[jmax[j]])
+ k[i,j-1] при fam[j] < (fam[imin[i]], fam[jmin[j-1]])
или fam[j] > (fam[imax[i]], fam[jmax[j-1]]),

k[i,0] = 0 + k[i-1,j] при fam[i] < (fam[imin[i-1]], fam[jmin[j]])
или fam[i] > (fam[imax[i-1]], fam[jmax[j]])

k[0,j] = 0 + k[i,j-1] при fam[j] < (fam[imin[i]], fam[jmin[j-1]])
или fam[j] > (fam[imax[i]], fam[jmax[j-1]]),
k[0,0] = 1

После взятия n книг возникает список "i + j", где j = n - i, и ответом задачи является деленная пополам сумма чисел k[i,j] по диагонали i + j = n матрицы k, т. к. при таком подсчете каждая последняя n-я книга учитывается дважды. Как имеющая некоторый номер i от начала она входит в количество k[i,n-i], и в то же время под номером n-i+1 от конца - в количество k[i-1,n-i+1] (для i от 1 до n) - см. рис. 6.2.

Примечание. Для достижения алфавитного порядка в последнюю очередь должна выбираться либо первая по алфавиту фамилия fam[imin[n]], либо последняя по алфавиту фамилия fam[imax[n]]. Поэтому на диагонали i+j=n ненулевыми могут быть лишь четыре элемента k[i,j] в строках с индексами: imin[n], imin[n]-1, imax[n], imax[n]-1 или три элемента, когда два из этих индексов совпадают (при imin[n] и imax[n], отличающихся на единицу).

```
                    jmin[j]= 10  9  9  9  9  5  5  5  5  5
                    jmax[j]=  0  9  8  7  6  6  6  6  6  6
                           j=  0  1  2  3  4  5  6  7  8  9
fam[i] imin[i] imax[i]  i     k[i,j]
       10        0       0    1  1  1  1  1  1  0  0  0  0
IVANOV  1        1       1    1  2  3  4  5  5  0  0  0
CALOV   2        1       2    1  3  6 10 15 15  0  0
BUGROV  3        1       3    1  4 10 20 35 35  0
ALOV    4        1       4    1  5 15 35 70 70
AIZOV   5        1       5    1  6 21 56 126
YANOV   5        6       6    1  6 21 56
SIDOROV 5        6       7    0  0  0      При таком порядке фамилий
NIKULIN 5        6       8    0  0         будет максимальное число
MORGUN  5        6       9    0            способов для n = 9
126 = ответ <-- сумма/2
```

Рис. 6.2. Решение задачи о перестановке книг

6.5. Индуктивное вычисление функции

Функцию $F(x_1, \ldots, x_n)$, где $x_1, \ldots, x_n$ – некоторая входная последовательность, называют *индуктивной* (индуктивно вычислимой), если ее значение $F(x_1, \ldots, x_n, x_{n+1})$ можно вычислить, зная только $F(x_1, \ldots, x_n)$ и x_{n+1} [36].

Функцию F называют *индуктивным расширением* функции F1, если F индуктивна и по ее значению $F(x_1, \ldots, x_n)$ можно вычислить $F1(x_1, \ldots, x_n)$.

Основная идея состоит в следующем. Если функция не индуктивна, т.е. для вычисления ее следующего значения недостаточно знать только предыдущее значение и очередной член последовательности, то надо запоминать какую-то информацию о предыдущих членах последовательности. Естественно, что объем дополнительной информации желательно минимизировать. Индуктивное вычисление функций рассмотрим на примере.

Пример 6.9. Максимальная длина возрастающей подпоследовательности за время O(n*log(n)).

Рассмотрим задачу [19]. Дана последовательность целых чисел x[1],..., x[n]. Найти максимальную длину k ее возрастающей подпоследовательности (число действий порядка n*log(n)). Подпоследовательность получается вычеркиванием некоторых членов входной последовательности.

Решение [75]**.** Искомая функция не индуктивна, но имеет индуктивное расширение в виде функции "максимальная длина k возрастающей подпоследовательности" с дополнительной информацией, включающей числа u[1],...,u[k],

где u[i] - минимальный из последних членов возрастающих подпоследовательностей длины i. Очевидно, $u[1] \le \ldots \le u[k]$. (6.3)

При добавлении нового члена последовательности x изменяются u и k:

```
k=1;  u[1]=x[1];
// инвариант: "k и u соответствуют описанию  (6.3)"
for (j=2; j<=n; j++)
{   Найти i - наибольший индекс от 1 до k, такой что u[i]<x[j];
     если таких нет, i=0 ;
     if (i == k)  {  k++; u[k+1]=x[j]; }
     else u[i+1]=x[j];              // i<k, u[i] < x[j] <= u[i+1]
}
```

Для определения i используется идея двоичного поиска. Цель: $u[i] < x[j] \le u[i+1]$ (в инварианте условно полагаем u[0] равным минус бесконечности, а u[k+1] - плюс бесконечности):

```
// Найти i - наибольший индекс от 1 до k, такой что u[i]<x[j];
//           если таких нет, i=0 ;
  i=0; r=k+1;
  // инвариант: u[i] < x[j] <= u[r],  r>i
  while (r-i != 1)
  {  s = (i+r)/2;                    // i < s < j     (s=i+(r-i)/2)
     if (u[s] >= x[j]) r = s;
     else              i = s;        // u[s] < x[j]
  }
```

6.6. Задачи

6.1. Какие абстрактные структуры данных и структуры хранения можно использовать для представления множества из 1000 людей и отношения "A и B знакомы между собой"?

6.2. Какие абстрактные структуры данных и структуры хранения можно использовать для представления следующей информации:

а) план посадочных мест кинотеатра или самолета с учетом их занятости;

б) таблица среднего роста как функция веса, пола и возраста;

в) различные подмножества в множестве из M человек;

д) текущая позиция игры в шашки;

е) перечень студентов-задолжников факультета с указанием не сданных зачетов и экзаменов;

ж) план трамвайных маршрутов города с указанием времени проезда между остановками.

6.3. Решето Эратосфена. Среди целых чисел от 1 до K требуется отыскать все простые числа, т. е. числа, которые делятся без остатка только на себя и единицу. Составить программу с использованием метода решета Эратосфена (пример 6.1).

6.4. Дан граф (орграф). Составить описание данных для его представления в виде матрицы смежности и фрагмент программы (подпрограмму) поиска какого-либо пути от вершины A к вершине B.

Указание: использовать перебор с возвратом по алг. 6.3 (обход графа в глубину) или обход графа в ширину на основе алг. 4.4 и 4.9.

6.5. Напечатать все возможные способы представления натурального числа N суммой натуральных чисел. Перестановка слагаемых не считается новым способом.

Например, для N=4 возможным результатом будет:

4 = 4
4 = 3 + 1
4 = 2 + 2
4 = 2 + 1 + 1
4 = 1 + 1 + 1 + 1

Указание: использовать перебор с возвратом.

6.6. Составить фрагмент программы (подпрограмму) поиска начинающегося с заданной точки пути выхода из лабиринта. План лабиринта задан прямоугольной матрицей: пути прохода - нулями, стены - единицами.

а) Использовать перебор с возвратом по алг. 6.3 (обход графа в глубину).

б) Использовать волновой алгоритм (обход графа в ширину).

6.7. Составить программу решения задачи о восьми мирных ферзях (пример. 6.2).

6.8. Составить программу решения задачи из примера 6.6.

6.9. Составить программу решения задачи из примера 6.7.

6.10. Составить программу решения задачи из примера 6.8.

6.11. Задача "Сумма цифр" [86]. Найти в порядке возрастания K минимальных чисел натурального ряда 1, 2, 3, ..., каждое из которых в системе счисления с основанием B имеет сумму цифр, равную S. *Входной файл* содержит три целых числа B, K и S, разделенных пробелами ($2 \le B \le 10$, $0<K\le250$, $0<S\le100$). *Выходной файл* должен содержать искомую последовательность из K натуральных чисел в системе счисления с основанием B, разделенных пробелами и/или символами перевода строки.

Пример входного файла	*Выходной файл*
9 3 7	7 16 25

6.12. Задача "Полосатый кросс" [86]. Группе туристов необходимо за кратчайшее время совершить переход из точки с координатами (xa, ya) в точку (xb, yb). Прямые, параллельные оси X, разделяют местность на полосы разной проходимости. В области, определяемой неравенством $y \le B_1$, можно двигаться со скоростью V_0; в полосе, определяемой неравенствами $B_1 \le y \le B_2$ - со скоростью V_1 и т. д. В области, определяемой неравенством $y \ge B_n$, можно двигаться со скоростью V_n ($(B_1 \le B_2 \le ... \le B_n$; $1 \le n \le 10$ $V_i > 0$, $i = 1, \ldots, n$). *Входной файл* содержит разделенные пробелами целые числа, не превышающие 100 по абсолютной величине: xa, ya, xb, yb, n, V_0, B_1, V_1, ... B_n, V_n. *Выходной файл* должен содержать найденное минимальное время с двумя дробными разрядами (с погрешностью не более 0,005).

Примеры

Входной файл	*Выходной файл*
0 1 10 1 1 2 0 2	5.00
0 1 10 1 1 10 0 2	1.98