Лекция 6. Комбинаторный перебор и рекурсия, алгоритмы STL для организации перебора.

Переборные задачи. Классы сложности задач P и NP. Рекурсивные алгоритмы. Дерево рекурсивных вызовов на примере рекурсивного вычисления чисел Фибоначчи. Пример нерекурсивного и рекурсивного алгоритма вычисления a^n со сложностью O(n) и O(log n). Комбинаторные объекты: перестановки, сочетания, размещения, сочетания с повторениями, размещения с повторениями. Перебор перестановок: рекурсивный и нерекурсивный алгоритмы. Генерация m-размещений без повторений и с повторениями. Генерация всех m-элементных подмножеств n-элементного множества (генерация m-сочетаний без повторений и с повторениями). Треугольник Паскаля. Правильные скобочные последовательности. Числа Каталана. Перебор разложений числа в сумму. Алгортимы STL для организации перебора. Примеры олимпиадных задач.

Общее представление

С содержательной точки зрения комбинаторный анализ представляет собой совокупность задач, связанных с нахождением числа объектов, обладающих определенным перечнем свойств. Например, всем известные задачи о разложении числа в сумму всевозможных слагаемых, о генерации всех m-элементных подмножеств n-элементного множества, о раскраске вершин графа, о «счастливых билетах», являются комбинаторными задачами. К основным комбинаторным объектам относятся: перестановки, размещения без повторений и с повторениями, сочетания. Математические понятия о данных объектах, алгоритмы их генерации будут рассмотрены в этой лекции.

При решении комбинаторных переборных задач удобно использовать рекурсивные алгоритмы. Поэтому рассмотрение темы будет начато с изложения логики работы рекурсивных алгоритмов на конкретных примерах. Кроме того, в Visual Studio представлены алгоритмы STL для организации перебора, которые могут помочь в быстром решении несложных олимпиадных задач.

Переборные задачи. Классы сложности P и NP

Задачи, которые можно решать при помощи алгоритмов, имеющих полиномиальную сложность $O(n^k)$, относятся к задачам *класса сложности P* (так называемым, быстро разрешимым задачам). В обозначениях полиномиальной сложности $O(n^k)$, n – это размер задачи, определяемый входными данными, k- некоторое положительное целое число. Задачи, которые решаются при помощи полного перебора, относятся к классу задач класса NP. Для этих задач не удается найти полиномиальный алгоритм решения, поэтому их также называют *трудноразрешимыми*. В большинстве своем это переборные задачи, которые характеризуются экспоненциальным множеством вариантов, среди которых нужно найти решение. Такие задачи могут решаться при помощи полного перебора, что возможно только для небольших размеров задачи, в противном же случае - с ростом размера задачи число вариантов быстро растет, и задача становится практически неразрешимой методом полного перебора.

Примерами задач класса P являются задачи: умножения матриц, например, со сложностью $O(n^3)$; сортировки массивов; нахождения эйлерова цикла в графе из m ребер (обход графа, начиная с какой-то вершины и заканчивая в ней самой, таким образом, чтобы каждое ребро было посещено по одному разу). *Примерами задач класса NP* являются задачи: коммивояжера (коммивояжер хочет объехать все города, побывав в каждом ровно по одному разу, и вернуться в город, из которого начато путешествие. Известно, что переезд из города i в город j стоит c(i,j) рублей, требуется найти путь минимальной стоимости.

Одна из проблем теоретической информатики заключается в том, верно ли равенство P=NP? Неформально говоря, постановка задачи заключается в следующем: можно ли быстро решить всякую переборную задачу, или, иначе говоря, можно ли найти полиномиальный алгоритм для задач класса NP. В настоящее время таких алгоритмов указать не удалось. Если для некоторой задачи удалось доказать ее NP-полноту, то есть основания считать ее практически неразрешимой. В этом случае лучше построить приближенный алгоритм решения задачи.

Рекурсивные алгоритмы

Рассмотрим рекурсивные алгоритмы на примере задачи вычисления a^n - степени числа. Первый способ вычисления основан на однопроходном итерационном алгоритме:

```
long long  p=1;
for (int i=1; i<=n;++i)
          p*=a;
```

Сложность работы алгоритма $O(n)$.

Второй способ вычисления степени основан на рекурсии. Рекурсия базируется на следующем рекуррентном соотношении: $a^0 = 1, a^n = a * a^{n-1}$. То есть, каждый раз функция рекурсивно вызывает саму себя с уменьшенным на единицу значением показателя степени. Таким образом, для вычисления a^n потребуется n рекурсивных вызовов.

```
long long power(int a,int n){
     if(n==0)
          return 1;  //Выход из рекурсии в случае нулевой степени
     else
          return a*power(a,n-1); //Рекурсивный вызов
}
```

Оказывается, существует и более оптимальный алгоритм, основанный на рекуррентном соотношении

$$a^n = \begin{cases} a * a^{n-1}, \text{если } n - \text{нечетное} \\ \left(a^{n/2}\right)^2, \text{если } n - \text{четное} \end{cases}$$

```
int power(int a,int n){
     if(n==0) return 1;  //Выход из рекурсии
     else
          if (n&1)       //Степень нечетная
               return a*power(a,n-1);
          else           //Степень четная
               return power(a*a, n/2);
}
```

Оценим количество рекурсивных вызовов при вычислении a^{100}. В дальнейших рассуждениях потребуется двоичное представление показателя степени - $100_{10}=1100100_2=64+32+4$.

$a^{100} = (a^2)^{50} = (a^4)^{25} = (a^4)^{24} * a^4 = (a^8)^{12} * a^4 = (a^{16})^6 * a^4 = (a^{32})^3 * a^4 =$
$= (a^{32})^2 * a^{32} * a^4 = a^{64} * a^{32} * a^4.$

Как видим, потребовалось 8 рекурсивных вызовов, а в двоичной записи показателя 100 семь разрядов: $\lceil log 100 \rceil = 7$. Всего же, в общем случае будет не более $2 \log n$ рекурсивных вызовов при вычислении степени. Таким образом, сложность работы алгоритма $O(\log n)$.

Использование стека в работе рекурсивных функций. Рекурсивные функции при каждом рекурсивном вызове используют *рекурсивный стек* – область памяти, в которую заносятся значения всех локальных переменных функции в момент рекурсивного обращения. Каждое такое обращение формирует один слой данных стека. При завершении вычислений по конкретному об-

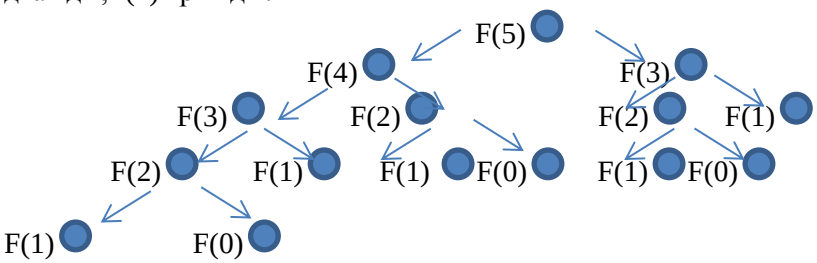
ращению, из стека считывается соответствующий ему слой данных, и локальные переменные восстанавливаются, снова принимая значения, которые они имели в момент обращения. Максимальное количество слоев рекурсивного стека, заполняемых при конкретном вычислении значения рекурсивной функции, называют *глубиной рекурсивных вызовов*. Количество элементов полной рекурсивной траектории всегда не меньше глубины рекурсивных вызовов. Эта величина не должна превосходить максимального размера стека используемой вычислительной среды.

Вычисление чисел Фибоначчи

В разделе динамического программирования мы рассматривали вычисление чисел Фибоначчи при помощи рекуррентных соотношений, которые служили основой для базы и перехода динамики: F[0]=0, F[1]=1, F(i)=F(i-1)+F(i-2). Вычисление чисел Фибоначчи можно реализовать и рекурсивно.

```
int fib(int n){
     if (n<=1)
          return n;
     else
          return fib(n-1)+fib(n-2);
}
```

Для данной задачи рекурсивные вызовы образуют бинарное дерево, по которому можно оценить количество рекурсивных вызовов. На рисунке изображены рекурсивные вызовы для f(5), всего потребуется 14 рекурсивных вызовов. При вычислении нерекурсивным алгоритмом f(5) будет получено линейно за 5 итераций. Рекурсия в данном случае неэффективна, так как многие числа Фибоначчи вычисляются несколько раз, например на рисунке видно, что f(3) вычисляется дважды, f(2) трижды.



Обозначим за T(n) - количество строк (операций), выполняемых рекурсивной процедурой fib(). T(n)=2 для n<=1. T(n)=T(n-1)+T(n-2)+3, при n>1. То есть, T(n)>=F(n). К примеру, T(100)>3*10^20. Тактовая частота процессора 1GHz позволяет компьютеру производить 10^9 операций в секунду. Легко проверить, что рекурсивное вычисление числа F(100) в этом случае потребует более десяти тысяч лет.

Можем сделать вывод о необходимости правильного выбора алгоритмов для эффективного решения той или иной задачи.

Комбинаторные объекты. Основные математические определения

Перестановкой из n элементов (например, чисел $1, 2, 3, \dots, n$) называется каждый упорядоченный набор из этих элементов. Для примера рассмотрим все перестановки при $n = 3$: $\{1,2,3\}, \{1,3,2\}, \{2,1,3\}, \{2,3,1\}, \{3,1,2\}, \{3,2,1\}$, получаем 6 вариантов. Количество перестановок из n элементов - $P_n = n!$ Действительно, пусть имеется множество $A = \{1, 2, 3, \dots, n-1, n\}$, где $|A| = n$. Первый элемент множества можно выбрать n способами для включения его в перестановку, второй элемент - $n-1$ способом, и так далее до последнего элемента, который можно выбрать одним способом. Получим по правилу комбинаторного произведения формулу вычисления количества перестановок.

$$P_n = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1 = n!$$

Размещением из n элементов по m называется упорядоченный набор из m различных элементов некоторого n – элементного множества. Различают размещения без повторений и размещения с повторениями. В размещениях с повторениями элементы исходного множества могут включаться в набор несколько раз. Заметим, что перестановками являются размещения без повторений из n элементов по n.

Так, при при $n = 4$ и $m = 2$ получим следующие 12 размещений без повторений:
$\{1,2\},\{1,3\},\{1,4\},\{2,1\},\{2,3\},\{2,4\},\{3,1\},\{3,2\},\{3,4\},\{4,1\},\{4,2\},\{4,3\}$.

Приведем размещения с повторениями для $n = 3$, $m = 2$ (9 вариантов):
$\{1,1\},\{1,2\},\{1,3\},\{2,1\},\{2,2\},\{2,3\},\{3,1\},\{3,2\},\{3,3\}$.

Число размещений без повторений из n по m обозначается как A_n^m, и равно: $A_n^m = \frac{n!}{(n-m)!}$. Действительно, первый объект можно выбрать n способами, второй - $(n-1)$ способом, последний m-й объект – $(n-m+1)$ способом. Получим по правилу комбинаторного произведения формулу размещений без повторений.

$$A_n^m = n \times (n-1) \times (n-2) \times (n-m+1) = \frac{n!}{(n-m)!}.$$

Число размещений с повторениями из n по m обозначается как $\overline{A_n^m}$, и равно $\overline{A_n^m} = n^m$. Действительно, каждый элемент подмножества можно выбрать ровно n способами, а всего элементов - m.

Сочетанием из n элементов по m, называется набор из m различных элементов некоторого n – элементного множества, причем их порядок следования в наборе не важен. Различают сочетания без повторений и сочетания с повторениями. В сочетаниях с повторениями элементы исходного множества могут включаться в набор несколько раз

Рассмотрим сочетания без повторений для $n = 4, m = 2$: $\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\}$. Получили 6 сочетаний.

Числом сочетаний из n по m называется число всех m -элементных подмножеств - элементного множества и обозначается так: C_n^m, и равно $C_n^m = \frac{n!}{m!(n-m)!}$. Эта формула выводится из формулы размещений: $A_n^m = \frac{n!}{(n-m)!}$, но поскольку в сочетаниях не важен порядок следования элементов, то мы делим количество размещений на $m!$ – количество всевозможных способов, сколькими можно изменить порядок следования элементов набора из m элементов.

Число сочетаний без повторений из n по m равно биномиальному коэффициенту $\binom{n}{m}$. Напомним, что биномиальными коэффициентами называются коэффициенты в разложении бинома Ньютона: $(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$.

Для вычисления биномиальных коэффициентов можно использовать треугольник Паскаля, по бокам и на вершине которого стоят единицы, а для построения треугольника используется тождество: $\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}$. Приведём первые строки треугольника Паскаля:

```
                    1
                1       1
            1       2       1
        1       3       3       1
    1       4       6       4       1
```

Каждая ячейка задаётся двумя координатами – номером строки – n и номером столбца в этой строке – m. Таким образом, число, стоящее в ячейке с координатами $(n;m)$ равно C_n^m. Нумерация строк и столбцов начинается с нуля.

Приведем пример для сочетаний с повторениями для $n = 3, m = 2$: $\{1,1\}$, $\{1,2\}$, $\{1,3\}$, $\{2,2\}$, $\{2,3\}$, $\{3,3\}$. Количество сочетаний с повторениями можно вычислить по формуле

$\overline{C_n^m} = C_{n+m-1}^m$. Формула выводится на основе следующих соображений. Возьмем множество X={1, 2, 3, 4, 5,…}. Составим из элементов множества *X* мультимножество из m элементов (элементы в мультимножестве могут повторяться). Пусть в мультимножество входят элементы: $a_1 \le a_2 \le ... \le a_m$. Прибавим к каждому элементу, начиная со второго, числа 1, 2, 3, …, m-1. Получим строгое соотношение: $a_1 < a_2 + 1 < a_3 + 2 < ... \le a_m + m - 1$. Этот набор чисел есть набор из более широкого множества чисел, которое включает в себя (n+m-1) чисел. Из этого множества мы выбираем m чисел для составления сочетаний - C_{n+m-1}^m. Таким образом, получаем формулу $\overline{C_n^m} = C_{n+m-1}^m$.

Комбинаторные объекты. Алгоритмы генерации

На практике часто возникают задачи генерации комбинаторных объектов, а также задачи нахождения объекта по его номеру и задачи определения номера комбинаторного объекта. Рекурсивные программы являются удобным способом порождения комбинаторных объектов, и позволяют прозрачно и сжато записать алгоритм решения. В то же время, любой рекурсивный алгоритм тратит время на запоминание данных в незавершенных вызовах (используется рекурсивный стек) и поэтому не всегда является эффективным. Каждый рекурсивный алгоритм может быть представлен нерекурсивным итерационным образом, в этом разделе будут рассмотрены и рекурсивные, и нерекурсивные реализации алгоритмов генерации комбинаторных объектов. Для перестановок рассмотрены также задачи получения перестановки по ее номеру и номера для заданной перестановки.

Перестановки. Генерация всех перестановок. Рекурсивный алгоритм

Дано множество чисел {1, 2, 3,..,n}. Необходимо вывести на экран все перестановки этого множества. Очередная перестановка чисел (в общем виде каких-либо объектов) хранится в массиве a[]. Индексация элементов начинается с нуля. t – индекс элемента, который включается в очередную перестановку. Рекурсивная процедура печатает очередную полученную перестановку, когда t станет равным n-1 (набрали n элементов). Для того, чтобы сгенерировать очередную перестановку, необходимо обменять местами текущий включенный элемент в перестановку со всеми последующими элементами массива (цикл перебора элементов для обмена начинается с индекса t, чтобы не терять первоначальные перестановки после добавления нового элемента). Сложность алгоритма оценивается как *O(n!)*. Используется дополнительная память $O(n)$.

```
void generate (int t){
     if (t==n-1){ //Вывод очередной перестановки
          for (int i=0;i<n;++i)
               cout<<a[i]<< " ";
               cout<<endl;
     }
     else{
          for (int j=t;j<n;++j){   //Запускаем процесс обмена
               swap(a[t],a[j]);    //a[t] со всеми последующими
               t++;
               generate(t);        //Рекурсивный вызов
               t--;
               swap(a[t],a[j]);
          }
     }
}
```

В основной программе первый рекурсивный вызов делаем от нуля (добавляем нулевой элемент).

```
generate(0);
```

Программа выведет всевозможные n! перестановок. Пример вывода для n=3.

1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 2 1, 3 1 2.

Заметим, что программа генерирует перестановки не в лексикографическом порядке. Для различного рода задач необходим именно лексикографический порядок генерации. Две перестановки находятся в лексикографическом порядке, если первые их элементы могут совпадать, а начиная с некоторого индекса, элемент во второй перестановке больше, чем элемент в первой перестановке. Например, перестановки 12453 и 12534 находятся в лексикографическом порядке.

Перестановки. Генерация всех перестановок. Нерекурсивный алгоритм

Рассмотрим нерекурсивный алгоритм генерации перестановок в лексикографическом порядке.

1. Последовательность элементов просматривается с конца до тех пор, пока не будет встречен первый элемент, такой что *a[i]<a[i+1]*.
2. В «хвосте» последовательности, состоящем из элементов, расположенных за найденным элементом, производим поиск минимального элемента min, большего, чем a[i].
3. Меняем местами a[i] и найденный элемент min.
4. Сортируем хвост последовательности.

Такой алгоритм позволяет получить все перестановки в лексикографическом порядке. Проиллюстрируем на примере *n=5* рассмотренный алгоритм.

i=0	i=1	i=2	i=3	i=4	Комментарии
1	2	3	4	5	Определили, что a[3]<a[4]. Меняем его места с минимальным, большим a[3] в «хвосте» последовательности. То есть, поменяли 4 и 5
1	2	3	5	4	a[2]<a[3]. Min=4. Меняем 3 и 4. Сортируем «хвост» последовательности, то есть элементы 3 и 5.
1	2	4	3	5	
1	2	4	5	3	Найден первый элемент с конца, меньший последующего – это a[2]=4. В «хвосте» последовательности {5, 3} ищем минимальный min>a[2]. Min=5. Меняем 4 и 5. Получаем новый «хвост» {4, 3}.Сортируем «хвост». Получаем перестановку {1, 2, 5, 3, 4}.
1	2	5	3	4	

Генерация очередной перестановки реализована в виде функции NextPermutation(). Функция возвращает true, если была сгенерирована очередная перестановка и false, если очередной перестановки в лексикографическом порядке сгенерировать невозможно (конец работы программы).

```
bool NextPermutation(){
     for (int i = n - 2; i >= 0; i--){
               if (a[i] < a[i + 1]){
               int min_val = a[i + 1],  min_id = i + 1;
               for (int j = i + 2; j < n; j++)
                    if (a[j] > a[i] && a[j] < min_val){
                         min_val = a[j];
                         min_id = j;
                    }
               swap(a[i], a[min_id]);
               sort(a.begin() + i+1, a.end());
               return 1;
               }
     }
          return 0;
}
```

В основном тексте программы для генерации всех перестановок можно использовать цикл while(), работающий до тех пор, пока не удастся сгенерировать очередную перестановку.

```
while (NextPermutation()){
        for (int i=0;i<n;++i) cout<<a[i]<< " ";
        cout<<endl;
```

Определение перестановки по номеру и номера перестановки

Задача определения перестановки по ее номеру. Рассмотрим всевозможные перестановки из n первых натуральных чисел. Таких перестановок n!. Требуется по номеру перестановки вывести ее на экран. Договоримся, что нумерация перестановок начинается с 1 (единицы).

Например, при n=5 получаем 120 перестановок с номерами от 1 до 120. Допустим, нужно вывести на экран перестановку с номером num=110. Проведем следующие рассуждения. Все перестановки можно разбить на группы. Выделим n=5 групп перестановок по их первой цифре – от 1 до 5.

1 в начале. Перестановки 1****	4!=24 перестановки с 1 в начале. Номера 1-24
2 в начале. Перестановки 2****	24 перестановки с 2 в начале. Номера 25-48
3 в начале. Перестановки 3****	24 перестановки с 3 в начале. Номера 49-72
4 в начале. Перестановки 4****	24 перестановки с 4 в начале. Номера 73-96
5 в начале. Перестановки 5****	24 перестановки с 5 в начале. Номера 97-120

Таким образом, можно найти номер группы, к которой относится перестановка – 110/24=4. Значит, *первая цифра* в перестановке – dig=5. Номер перестановки в группе (а также и новое число для поиска второй цифры) Np=110%24=14. Перестановки вида 5**** в свою очередь разбивается на 4 группы с вторыми цифрами из множества {1,2,3,4}. Количество перестановок в каждой из четырех групп - (n-1)!=3!=6. Продолжаем алгоритм и далее, используя информацию о группе перестановки и номеру перестановки в группе, однозначно задающую текущую цифру перестановки. Заведем вспомогательный массив digit[], значения которого digit[i]=1, если цифра i уже была использована в перестановке. В перестановку будем брать dig-ю по счету свободную цифру.

i - номер цифры в искомой перестановке dig – номер свободной цифры, которую будем ставить на позицию i в перестановку Np – номер перестановки в группе	Шаблон P=***** до определения цифры на позиции i и после ее определения	Массив digit[], хранящий информацию об использовании цифр в перестановке

i=1;
dig=110/24+1=5;
Np=110%24=14
	P=***** (4!=24)
	P=5****
		{1,2,3,4,~~5~~} – множество цифр для второй *

i=2;
dig=14/6+1=3;
Np=14%6=2
	P=5**** (3!=6)
	P=53***
		{1,2,~~3~~,4,~~5~~} – множество цифр для третьей *

i=3; dig=2/2=1; Np=2%2=0; Np=6 (если остаток равен нулю, то единицу к dig не добавляем, номер перестановки в группе делаем последним)	P=53*** (2!=2) P=531**	{~~1~~,2,~~3~~,4,~~5~~} – множество цифр для четвертой *
i=4; dig=6/1=6; Np=6%1=0; Np=6	P=531** (1!=1) P=5314*	{~~1~~, 2,~~3~~,~~4~~,~~5~~} – множество цифр для пятой *
На пятом шаге алгоритма получим P=53142		

В программе удобно завести вспомогательный массив факториалов, например

```
int fact[13] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880,
3628800, 39916800, 479001600};
```

Сама программа выглядит следующим образом.

```
for (int i=n; i>0;--i){
     Np=num%fact[i-1]; //Определение номера группы перест.
     d=num/fact[i-1];  //Определение индекса текущей цифры
     if (Np) d++; else Np+=fact[i-1];
     num=Np;
     int pos=0;
     for (int j=1;j<=n;++j){//Определение текущей цифры
          if (!digit[j]) pos++;
          if (pos==d) {
               digit[j]=1;
               res.push_back(j); //Формирование перестановки
               break;
          }
     }
```

Ответ программы – перестановка по ее номеру, хранится в массиве res. Сложность алгоритма - $O(n^2)$.

Задача определения номера перестановки по заданной перестановке. Требуется по заданной перестановке P вывести на экран ее номер. Нумерация перестановок начинается с 1 (единицы). Количество цифр в перестановке – n.

Рассмотрим идею решения на конкретном примере. n=5, p=53142. Номер данной перестановки сформируем следующим образом

p=53142. Первый элемент перестановки, данной в условии – 5. Перестановки, начинающиеся на цифры {1, 2, 3, 4} находятся перед данной нам в условии. Их количество 4*4!=96. Значит, номер перестановки, заданной в условии > 96. Следующий элемент перестановки – 3. Значит, до этого идут перестановки, у которых вторая цифра может быть 1 или 2. Следовательно, минимально возможный номер перестановки, заданной в условии > 4*4!+2*3!. Следующий элемент перестановки – 1 (минимально возможный). То есть, номер перестановки не меняется – минимальный номер заданной в условии перестановки остается прежним > 4!*4+3!*2+2!*0. Следующий элемент перестановки – 4. Напомним, что это на этом месте могли быть не занятые цифры– {2,4}. В перестановке используется 4 – второй не занятый элемент. Следовательно, минимально возможный номер перестановки, заданной в условии > 4!*4+3!*2+2!*0+1!*1. С учетом последнего элемента получаем - 4!*4+3!*2+2!*0+1!*1+0!*1=110 – номер перестановки.

Цифра i	i=2	i=3	i=4	i=5	Итог
Количество перестановок до i цифры	4!*4	4!*4+3!*2	4!*4+3!*2+2!*0	4!*4+3!*2+2!*0++1!*1	4!*4+3!*2+2!*0++1!*1+0!*1=110
Группа перестановки (с учетом занятых цифр)	5-я	3-я	1-я	4-я	2-я
Занятые цифры после вычисления	12345	12345	~~12345~~	~~12345~~	~~12345~~

Итак, получаем ответ – 110. Это номер перестановки p=53142. Сложность ма - $O(n^2)$.

Алгоритмы STL для генерации перестановок

Удобным преимуществом библиотеки STL является наличие в ней реализованных алгоритмов, в том числе и относящихся к комбинаторному перебору. Для использования алгоритмов требуется наличие заголовка <algorithm>. Приведем некоторые из них.

Алгоритм	Описание. Пример (msdn.microsoft.com)

next_permutation(it1, it2)	Генерирует следующую перестановку в указанном подмножестве контейнера, которое задается итераторами it1, it2.
	`Вектор v1 (-3 -2 -1 0 1 2 3).`
	`После первого next_permutation, вектор v1 выглядит следующим образом: v1 = (-3 -2 -1 0 1 3 2).`
	Функция next_permutation() возвращает true, если следующая лексикографическая перестановка сгенерирована, в противном случае – false (следующей лексикографической перестановки не существует).

prev_permutation(it1, it2)	Генерирует предыдущую перестановку в указанном подмножестве контейнера, которое задается итераторами it1, it2.
	`Вектор v1 = (-3 -2 0 3 -1 2 1).`
	`После prev_permutation вектора v1, вектор v1 выглядит следующим образом: v1 = (-3 -2 0 3 -1 1 2).`

is_permutation(it11, it12, it21, it22)	Функция возвращает true, когда указанные подмножества можно переупорядочить так, чтобы они стали совпадающими. В противном случае функция возвращает false.

В нерекурсивном алгоритме генерации всех m-размещений из n элементов потребуется использование алгоритма next_permutation().

Генерация всех m-размещений из n элементов. Рекурсивный и нерекурсивный алгоритмы

Рекурсивный алгоритм. В массиве a[] будет хранится очередное размещение. Вспомогательный массив used[] используется для пометки элемента, как уже взятого в размещение. Суть алгоритма в том, что в очередное размещение добавляются по очереди всевозможные элементы, которые еще не были использованы в размещении. Пример реализации функции generate, получающий на вход номер очередного элемента, включаемого в размещение.

```
void generate(int num){
     if (num == m){ //Если размещение готово, то печатаем его
          for (int i = 0; i < m; i++)
               cout<<a[i]<<" ";
               cout<<endl;
          }
     for (int i = 1; i <= n; i++)
          if (!used[i]){ //Добавляем еще не взятый элемент
               used[i] = 1, a.push_back(i);
               generate(num + 1);
               used[i] = 0, a.pop_back();
          }
}
```

Вызов функции из основного текста

```
     generate(0); //Начинаем добавлять элемент на нулевое место
```

Программа выведет для n=3, m=2 следующие перестановки: 1 2, 1 3, 2 1, 2 3, 3 1, 3 2.

Нерекурсивный алгоритм генерации следующего размещения. Работа алгоритма основывается на генерации всевозможных перестановок первых натуральных чисел от 1 до n и умении «взять» новое размещение из перестановки, «удалив» лишние элементы из «хвоста» перестановки. Рассмотрим пример для n=4 всех перестановок и всех размещений при m=3.

Перестановки. n=4	Размещения. n=4, m=3
1234	123
1243	124
1324	132
1342	134
1423	142
1432	143

По таблице хорошо видно, что размещения можно получать из перестановок, «удаляя» хвост новой перестановки и оставив только первые m элементов в ней. Новое размещение получается из той перестановки, в которой будет меняться элемент, стоящий на m –месте. Это происходит тогда, когда за m-м элементом следуют элементы в порядке убывания. Получаем способ генерации нового размещения: добавим к очередному размещению неиспользованные элементы в порядке убывания, сгенерируем новую перестановку, возьмем первые m элементов и получим новое размещение. Приведем фрагмент программы генерации размещений.

```
bool generate(){
     int k=m;
     for (int i=n-1;i>=0;--i){
          if(!used[i]){
               a[k]=i+1;
               k++;
          }
     }
     if (next_permutation(a.begin(),a.end())){
          for (int i=m;i<n;++i) used[a[i]-1]=0;
          for (int i=0;i<m;++i) used[a[i]-1]=1;
          return 1;
     }
     else
          return 0;
}
```

Начальные присваивания в основном тексте программы

```
for (int i=0;i<n;++i)a[i]=i+1; //Берем последовательные эл.
     for (int i=0;i<m;++i)used[i]=1; //Помечаем первые m
     while (generate()){//Если функция генерирует перестановку
          for (int j=0;j<m;++j)cout<<a[j]; //печатаем ее
               cout<<endl;
     }
```

Сложность алгоритма генерации всех размещений оценивается как O(n!/(n-k)!).

Генерация всех m-сочетаний n элементов. Рекурсивный и нерекурсивный алгоритмы

Рекурсивная функция генерации сочетаний. Сочетания из n=4 элементов по m=3 будут представлять собой следующий набор подмножеств n-элементного множества: 123, 124, 134, 234. Реализация в виде рекурсивной функции generate() использует переменные: last – последний добавляемый элемент в сочетание и num – количество элементов в сочетании. Если сочетание сформировано (num=m), то выводим его на экран. В противном случае добавляем новый элемент в сочетание, следующий за последним добавленным элементом, и снова вызываем рекурсивно функцию generate().

```
void generate(int num , int last){
     if (num == m){
```

```
            for (int i = 0; i < m; i++)
                cout<<a[i];
                cout<<endl;
      }
      for (int i = last + 1; i <= n; i++){
                a.push_back(i);
                generate(num + 1, i);
                a.pop_back();
      }
}
```

Основной текст программы вызывает функцию с начальными параметрами 0,0.

```
generate(0 , 0);
```

На экране будут выведены для n=5, m=3 следующие сочетания: 123, 124, 125, 134, 135, 145, 234, 235, 245, 345.

Нерекурсивный алгоритм генерации сочетаний. В основе алгоритма используется следующая идея: просматривая сочетания справа налево, находим первый элемент, который можно увеличить. Этот элемент увеличиваем (берем следующий возможный элемент), а все элементы после него, заменяем на натуральные числа, следующие за новым элементом. Зададимся вопросом – какой элемент можно увеличить в сочетании?. Если n=5, m=3, то максимальное лексикографическое сочетание 345. Значит, на месте m может стоять максимальный элемент n, на месте (m-1) может стоять максимальный элемент (n-1). То есть, на месте i может стоять максимальный элемент (n - m + i). Нумерация элементов в массиве a[] - с единицы.

```
bool Next(){
      for (int i = m; i >= 1; i--)
            if (a[i] < n - m + i)    {
            a[i]++;
            for (int j = i + 1; j <= n; j++)
                        a[j] = a[j - 1] + 1;
                  return 1;
                                    }
      return 0;
}
```

Сложность алгоритма генерации всех сочетаний оценивается как O(n!/(k!*(n-k)!)).

Представление числа в виде суммы целых положительных слагаемых

Необходимо перечислить все представления целого положительного числа n в виде суммы последовательности невозрастающих целых положительных слагаемых.

Для n=5 программа выведет на экран следующие представления числа пять в виде сумммы.
1+1+1+1+1=5, 2+1+1+1=5, 2+2+1=5, 3+1+1=5, 3+2=5, 4+1=5.

Программа решения задачи будет оперировать массивом слагаемых a[], состоящим из n элементов, где n – максимальное количество слагаемых. Максимальное количество слагаемых достигается в том случае, когда все слагаемые – единицы. Пусть a[1], a[2], a[3],..,a[last] – невозрастающая последовательность целых чисел, сумма которых не превосходит числа n. Сумма элементов последовательности хранится в переменной s= a[1]+ a[2]+ a[3]+… +a[last], где last – индекс последнего взятого элемента. Если сумма стала равна заданному числу n, то выводим последовательность чисел. В противном случае, пробуем подобрать следующее слагаемое j, начиная с единицы, которое не больше последнего слагаемого j<= a[last], и которое не увеличит сумму, так что-

бы она не стала больше n, т.е. j<= (n-sum). Взяв очередное слагаемое, комбинируем последовательность дальше. Функция представления числа в виде суммы приведена ниже.

```
void generate(int last,int sum){
     if (sum==n) {
          for (int i=0;i<last;++i)
               cout<<a[i]<<"+";
          cout<<a[last]<<"="<<n<<endl;
     }
     else
          for (int j=1; j<=min(a[last],n-sum);++j){
//Берем следующие слагаемые, начиная с 1, но не больше a[last]
               last++;
               a[last]=j;
               generate(last,sum+j);
               last--;
          }
}
```

В основном тексте программы необходимо вызвать функцию для различных первых слагаемых из представления числа в виде суммы:

```
for (int j=1; j<n;++j){
     a[0]=j;
     generate(0,j); //Индекс последнего элемента, сумма
}
```

Аналогично можно составить алгоритм представления числа в виде суммы неубывающих слагаемых.

Правильные скобочные последовательности. Числа Каталана

Проверка скобочной последовательности с целью определения ее правильности была проделана нами в лекции 1 при изучении темы «Стек». В этом разделе мы хотим решить следующую задачу: *сгенерировать все правильные скобочные последовательности, состоящие из n пар скобок*. Напомним, что скобочная последовательность правильная, если для каждой позиции скобочной последовательности выполняется правило - число открывающихся скобок в последовательности больше или равно числу закрывающихся скобок, а общее число открывающихся скобок для всей скобочной последовательности равно числу закрывающихся.

Нерекурсивный алгоритм генерации правильных скобочных последовательностей

Пусть n=3 пар скобок. Рассмотрим все соответствующие правильные скобочные последовательности из 6 скобок. ()()(), ()(()), (())(), ((())), (()()).

Генерацию очередной скобочной последовательности будем делать следующим образом:

- просматриваем последовательность справа налево до первой комбинации скобок)(. Такая комбинация будет в любой последовательности, кроме последней;
- найденную комбинацию надо заменить на ();
- подсчитываем с начала строки, на сколько число открывающихся скобок больше числа закрывающихся и дописываем это количество закрывающихся скобок;
- если длина строки еще не стала равна 2*n, то ее необходимо дополнить нужным количеством пар скобок ().

Рекурсивный алгоритм генерации правильных скобочных последовательностей

Лексикографическим порядком правильных скобочных последовательностей будем называть следующий их порядок: ((())), (()()), (())(), ()(()), ()()() (приведен пример для n=3 – трех пар скобок). Рекурсивная функция получает следующие параметры: n – количество пар скобок, s – строка, представляющая собой скобочную последовательность, op_br – количество открывающихся скобок в текущей последовательности, cl_br – количество закрывающихся скобок в текущей последовательности. Функция печатает очередную скобочную последовательность, если уже использовано n пар скобок. Если можно поставить открывающуюся скобку (не все открывающиеся скобки использованы), то ставим открывающуюся скобку. Закрывающуюся скобку ставим в том случае, если ее можно поставить, то есть количество отрывающихся скобок больше количества закрывающихся.

```
void generate(int n, string s, int op_br, int cl_br){
     if (op_br+cl_br==2*n)
          cout<<s<<endl;
     if (op_br<n)
          generate(n, s+'(',op_br+1,cl_br);
     if(op_br-cl_br>0){generate(n,s+')',op_br,cl_br+1);

}
```

Количество правильных скобочных последовательностей. Числа Каталана

Количество правильных скобочных последовательностей определяется числом Каталана C_n, применяющегося во многих комбинаторных задачах. Для вывода формулы будем рассматривать X – произвольную правильную скобочную последовательность длины 2n. Она начинается с открывающейся скобки, ей соответствует определенная закрывающаяся скобка. Найдем ее и представим исходную последовательность как X=(A)B, где A и B – тоже правильные скобочные последовательности. Пусть длина последовательности A равна 2k, тогда правильных скобочных последовательностей для A - C_k. Тогда длина последовательности B – (2n-2k-2)=2(n-k-1), а правильных скобочных последовательностей для B - C_{n-k-1}. Всевозможные комбинации правильных скобочных последовательностей для k=0..n-1 и дадут общее количество правильных последовательностей. То есть, получаем рекуррентную формулу:

$C_n = C_0 C_{n-1} + C_1 C_{n-2} + C_2 C_{n-3} + \cdots + C_{n-1} C_0$.

Начальные значения чисел Каталана: $C_0 = 1$, $C_1 = 1$ (ноль скобок и одну пару скобок можно расставить одним способом), $C_2 = 2$. Последующие значения приведены в таблице.

n	0	1	2	3	4	5	6	7	8	9
C_n	1	1	2	5	14	42	132	429	1430	4862

Числа Каталана применяются и в других задачах комбинаторики, с их помощью можно вычислить:

- количество бинарных деревьев с заданным количеством листьев,
- количество различных триангуляций выпуклого многоугольника диагоналями,
- количество разбиений на пары четного числа точек на окружности непересекающимися хордами.

Задания

1. Опишите алгоритм генерации всех m-сочетаний с повторениями из множества первых n натуральных чисел.
2. Опишите алгоритм генерации всех m-размещений с повторениями из множества первых n натуральных чисел.

3. Опишите алгоритмы получения сочетания по его номеру и, наоборот, номера сочетания по самому сочетанию.
4. Опишите алгоритмы получения размещения по его номеру и, наоборот, номера сочетания по самому сочетанию.
5. Изобразите дерево рекурсивных вызовов для рекурсивного вычисления степени алгоритмом сложности O(log n). В качестве конкретного примера возьмите вычисление a^{150}.
6. Проведите триангуляцию выпуклого шестиугольника – разбиение его непересекающимися диагоналями на треугольники. Подсчитайте количеств различных вариантов. Вычислите это количество при помощи соответствующего числа Каталана.
7. Предложите решение следующей задачи. Даны шахматные доски с размером 2, 3, 4,…, в которых закрашены все квадраты северо-западнее главной диагонали. Требуется провести ладью из левого нижнего угла в правый верхний. Причем, двигаться можно только на север и на восток, не заходя при этом на закрашенные клетки. Спрашивается, сколько существует различных путей ладьи на доске со стороной n?

Литература

Введение в теоретическую информатику. Курс Александра Шеня. Сайт stepic.org

Зимняя школа по программированию. – Харьков: ХНУРЭ, 2014.

Леонтьев В.К. Комбинаторика и информатика. Часть 1. Комбинаторный анализ: учебное пособие/ В.К.Леонтьев. – М.: МФТИ, 2015.

Окулов С.М. Программирование в алгоритмах /С.М.Окулов. – М.: БИНОМ. Лаборатория знаний, 2002.

Основы перечислительной комбинаторики. Курс Александра Омельченко. Сайт stepic.org

Генерация сочетаний. http://e-maxx.ru/algo/generating_combinations

Сайт вики-конспектов университета ИТМО. http://neerc.ifmo.ru/wiki