Основы программирования

Комбинаторные алгоритмы

Комбинаторные алгоритмы

Исследуемые объекты представлены дискретными математическими структурами (множествами, графами).

Требуется найти ответ на вопросы типа:

- существует ли способ…
- сколько существует способов…
- найти все решения…
- найти лучшее (в смысле некоторого критерия) решение.

При этом обычно не существует аналитического решения и не заданы правила вычислений.

Задачи, требующие перебора вариантов решения – комбинаторные.

Перестановки

Пример комбинаторной задачи: нахождение всех перестановок натуральных чисел от 1 до n:

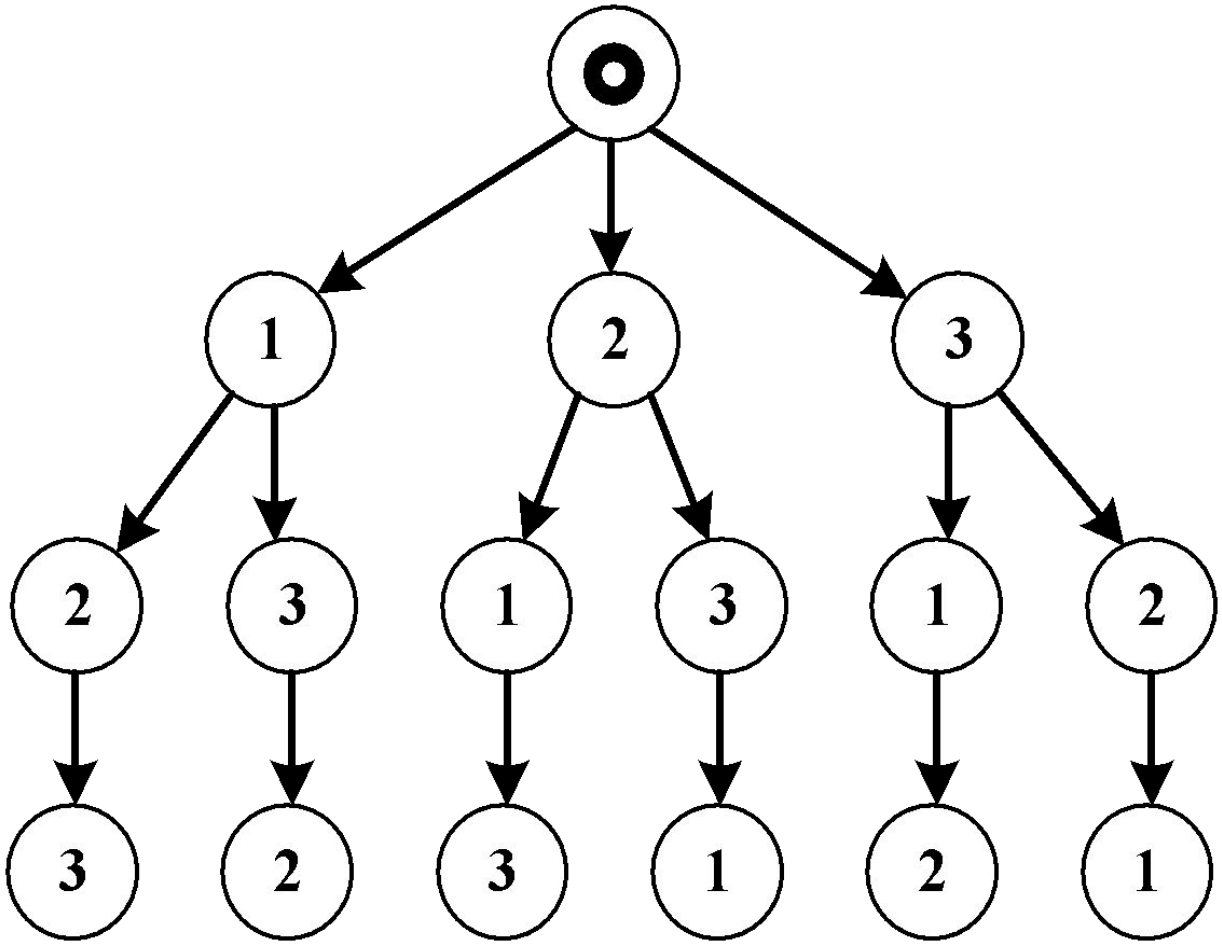
1) первое число – любое из чисел 1, …, n;

2) второе число – любое из чисел 1, …, n, кроме первого числа;

3) третье число – одно из чисел, которое не выбрано первым или вторым, и т.д.

Всего существует $n\,(n-1)\ldots 2\cdot 1 = n!$ вариантов перестановок.

Дерево решений для $n=3$



Перестановки

В реальных задачах могут потребоваться различные перестановки n произвольных объектов. Для этого проще всего использовать «косвенный» метод: переставлять местами не сами объекты, а их номера в наборе.

Построим, соответственно, алгоритм генерации всех перестановок чисел 0…n-1.

Идея рекурсивного алгоритма:

- основным параметром является номер текущей позиции в перестановке k ($0 \le k < n$)
- начиная с k=0, последовательно ставим на позицию k все числа, которые пока не использованы в перестановке, и производим рекурсивный вызов для k+1 (т.е. генерируем все возможные варианты для позиций k+1…n-1).

Алгоритм генерации всех перестановок

В алгоритме используются 2 массива (их проще сделать глобальными, чтобы постоянно не передавать параметры в рекурсивную функцию):

- целочисленный массив **`Per`** длины **`n`** содержит текущую построенную часть перестановки
- массив **`Inc`** длины **`n`** содержит признаки включения чисел в перестановку (**`bool`** или **`int`**), например **`Inc[i]=true`**, если число **`i`** включено в перестановку, и **`Inc[i]=false`**, если нет.

Вызов функции генерации перестановок **`permut`**:

```
for (i = 0; i < n; i++) Inc[i] = false;
permut(0);
```

Алгоритм генерации всех перестановок

```
void permut(int k)
{
  for (int i = 0; i < n; i++)
    if (!Inc[i])
    {
      Per[k] = i; Inc[i] = true;
      if (k == n-1) OUTPUT_PERMUTATION();
      else permut(k+1);
      Inc[i] = false;
    }
}
```

Число рекурсивных вызовов: $O(n!)$

Сочетания

Сочетания из n элементов по m – это всевозможные подмножества $1 \le m \le n$ элементов множества длины n. Порядок расположения элементов не важен, т.е. при использовании их номеров и m=3 последовательности (1,3,6), (3,1,6), (6,3,1) представляют одно и то же сочетание.

Следовательно, сочетания номеров элементов (в глобальном массиве **Comb**) выгодно генерировать в возрастающем порядке, и массив **Inc** не нужен:

- **min(Comb[k+1]) = Comb[k] + 1**
- **max(Comb[k]) = n – m + k**

Количество различных сочетаний из n по m:

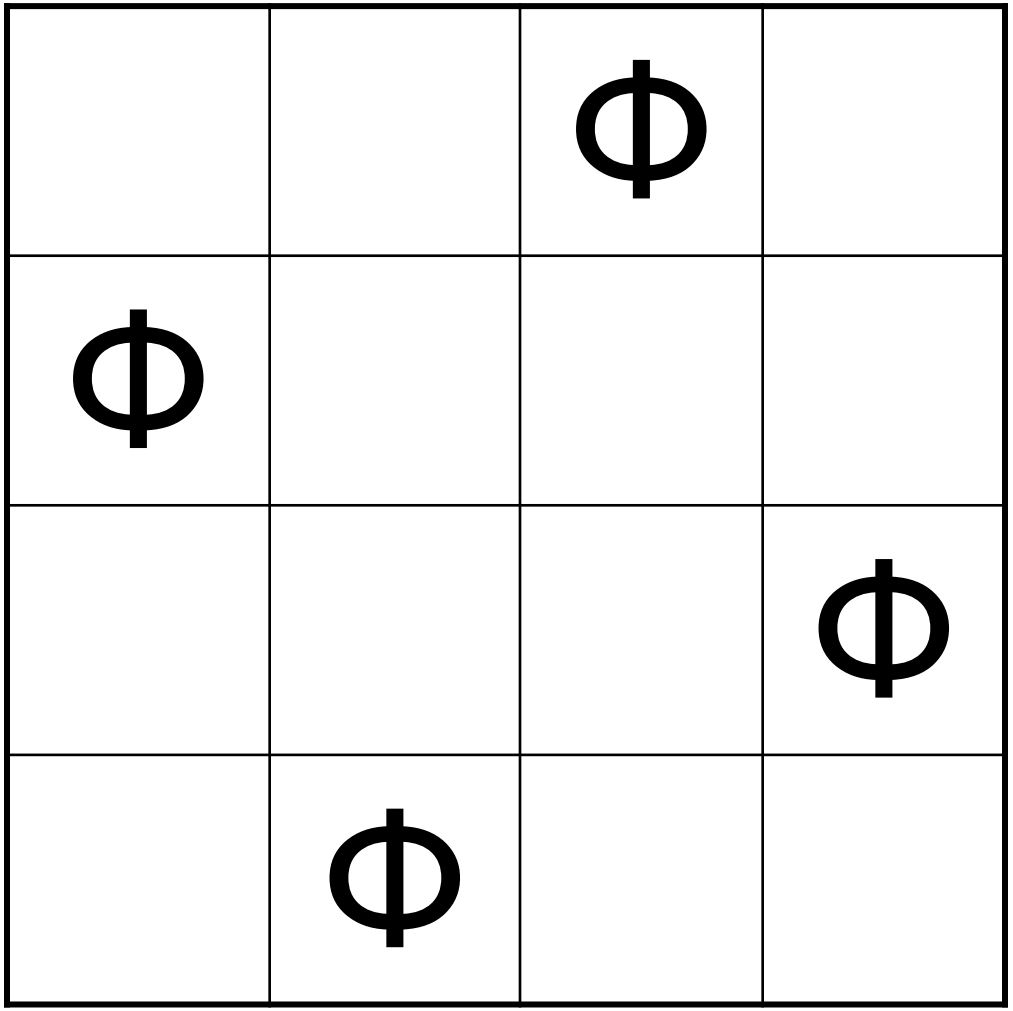
$$C_n^m = A_n^m / m! = \frac{n!}{m!(n-m)!}$$

Алгоритм генерации всех сочетаний

```
void combinat(int k)
{
  int i = (!k)? 0 : Comb[k-1]+1;
  for (; i <= n-m+k; i++)
  {
    Comb[k] = i;
    if (k == m-1) OUTPUT_COMBINATION();
    else combinat(k+1);
  }
}
```

Вызов: `combinat(0);`

Задача о ферзях
Пример для доски 4x4



Задача о ферзях

Основные требования при поиске решения любой комбинаторной задачи:

- найти удобную форму для представления информации;
- найти **эвристики** (совокупности приемов и правил решения практических задач), позволяющие заранее отсекать невыполнимые варианты.

Число проверяемых вариантов для 8 ферзей:

- без учета совпадения вертикалей и горизонталей всего $C_{64}^{8} \approx 4.4$ млрд.
- с учетом расстановки только в разных горизонталях (или только в разных вертикалях) $8^{8} \approx 16.8$ млн.

Задача о ферзях

- с учетом горизонталей и диагоналей – для элементов на одной диагонали константами являются величины:

(№ столбца – № строки)

0	1	2	3
-1	0	1	2
-2	-1	0	1
-3	-2	-1	0

(№ столбца + № строки)

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

Для 8 ферзей проверяется всего 8!=40320 вариантов.

Гамильтоновы циклы и пути

Гамильтонов цикл в неориентированном графе:

- начинается в произвольной вершине **a**
- проходит по ребрам через все вершины графа по одному разу
- завершается в вершине **a**.

Если в графе найдутся такие 2 вершины **a** и **b**, что переходя из **a** по ребрам можно попасть в **b**, обойдя все остальные вершины по одному разу, то в графе существует гамильтонов путь из **a** в **b**.

Для графов нет явных аналитических условий существования гамильтонова цикла/пути, поэтому решение можно найти только путем перебора вариантов путей.

Гамильтоновы циклы и пути

Любой гамильтонов цикл/путь задается некоторой перестановкой номеров вершин графа. Получать все перестановки, а затем проверять, не соответствуют ли они некоторому пути в графе, невыгодно.

Необходимо как можно раньше отсекать варианты, которые не соответствуют путям в графе, когда переход из предыдущей вершины в следующую невозможен.

Для упрощения алгоритма добавим в класс **`MGraph`** 2 дополнительных поля:

- **`int *Path`** – массив, в котором будут сохраняться пути
- **`bool *Inc`** – массив отметок пройденных вершин.

Рекурсивная функция **`ham_loops`** – поиск всех гамильтоновых циклов в графе – это просто модификация функции генерации всех перестановок.

Алгоритм поиска всех гамильтоновых циклов

```
void MGraph::ham_loops(int k)
{ int i, j;
  for (i = 0; i < vernum; i++)
    if (!Inc[i] && mat[Path[k-1]][i])
    {
      Path[k] = i; Inc[i] = true;
      if (k == vernum-1)
      { if (mat[i][0]) PROCESS_LOOP(); }
      else ham_loops(k+1);
      Inc[i] = false;
    }
}
```

Трудоемкость: $O(n!)$

Обертка для рекурсивной функции

Для метода **ham_loops** необходимо заранее подготовить 2 массива и задать начальную вершину пути. Поэтому **ham_loops** лучше сделать приватным методом и добавить public-обертку для него:

```
void hamilton_loops()
{
  Path = new int[vernum];
  Inc = new bool[vernum];
  for (int i = 0; i < vernum; i++)
    Inc[i] = false;
  Path[0] = 0; Inc[0] = true;
  ham_loops(1);
  delete [] Inc;
  delete [] Path;
}
```

Формализация комбинаторных задач

Общие условия:

- задано $M = \{a_1, a_2, \dots, a_n\}$ – множество элементов решения
- решение $S^{(m)} = \{a^{(1)}, a^{(2)}, \dots, a^{(m)}\},\ 1 \le m \le n,\ a^{(j)} \in M,$ – это обычно не просто подмножество M, а комплекс, в котором важен порядок следования элементов $a^{(j)}$
- множество всех возможных вариантов решений R (возможных комплексов элементов из M) очень велико и не может быть построено целиком, поэтому все решения $S \in R$ отыскиваются последовательно
- не каждый вариант может оказаться решением задачи.

Существуют 2 основных подхода к решению комбинаторных задач: бэктрекинг и метод решета.

Бэктрекинг (поиск с возвратом)

- Полное решение $S^{(m)} = \{a^{(1)}, a^{(2)}, \dots, a^{(m)}\}$ формируется рекурсивно на основе последовательности частичных решений $S^{(1)} = \{a^{(1)}\}$, $S^{(2)} = \{a^{(1)}, a^{(2)}\}$,…, $S^{(m)} = \{a^{(1)}, a^{(2)}, \dots, a^{(m)}\}$: при очередном рекурсивном вызове к текущему частичному решению добавляется новый элемент – один из множества допустимых.

Если из текущего решения $S^{(i)}$ невозможно построить никакое последующее $S^{(i+1)}$ (тупик) или $S^{(i)}$ уже является полным решением, то производится рекурсивный возврат к предыдущему частичному решению $S^{(i-1)}$ и выбирается следующий допустимый элемент $a^{(i)}$.

Бэктрекинг позволяет перебрать все варианты полных решений без повторов.

Общий вид алгоритма поиска с возвратом

Пусть **S** – текущее решение, **M** – множество элементов решений, **a** – один из эл-тов M):

```
поиск(S)
{
  while (существует_подходящий_элемент(M,S,a))
  {
    добавить_к_текущему_решению(S,a);
    if (полное_решение(S)) вывод_решения(S);
    else if
      (возможен_дальнейший_поиск(S)) поиск(S);
    удалить_из_текущего_решения(S,a);
  }
}
```

Метод решета

Производится не последовательный расчет допустимых решений, а исключение вариантов, которые не являются решениями (если таких много и они исключаются просто).

Решето Эратосфена для нахождения простых чисел $\leq n$:

- битовую строку длины n заполнить нулями
- $\forall$ нулевого бита с номером $i: 2 \leq i \leq \sqrt{n}$ установить в 1 все биты с номерами $ki \leq n,\ k \geq 2$.

Трудоемкость составляет $O(n\sqrt{n})$ – это гораздо быстрее, чем проверять остатки от деления.

Задача о корзине с яйцами

Найти минимальное n, для которого выполняется:

$$n \bmod 2 = n \bmod 3 = \cdots = n \bmod 6 = 1,\ n \bmod 7 = 0\ .$$

Решение: $n - 1$ делится нацело на 2, 3, 4, 5 и 6, следовательно, $n - 1 = 60k$.