

Комбинаторно генериране

Въведение

Въпреки че пълното изчерпване е много бавно, все пак в някои задачи се налага да прибегнем до него. В такъв случай е важно да разполагаме с бързи алгоритми за генериране на различни видове комбинаторни обекти.

Структурата на рекурсивните алгоритми обикновено следва структурата на рекурентното уравнение за преброяване на комбинаторните обекти.

Итеративните алгоритми имат следния общ вид:

```
combObj = Init
do
{
    print(combObj)
} while (Next(combObj))
```

Функцията **Init** построява първия обект според някаква зададена наредба. Функцията **Next** построява следващия обект (ако го има) и връща истина; а ако няма следващ обект, връща лъжа.

Освен пълното изчерпване има още две важни задачи:

- ранжиране — пресмятане на поредния номер на комбинаторен обект;
- деранжиране — построяване на комбинаторен обект по пореден номер.

Тези две операции компресират и декомпресират комбинаторни обекти. Ранжирането е перфектна хеш-функция (тоест инекция).

Приемаме, че поредните номера започват от 1, както броим в ежедневието. (Обаче формулите и програмният код са малко по-прости, ако броим от 0.)

Цел

За всяка алгоритмична задача е важно бързодействието на алгоритъма. Общото време за построяване на N комбинаторни обекта е от порядък $\Omega(N)$ — тривиална долна граница по размера на изхода. При това се интересуваме само от времето за построяване на обектите, но не и от времето за тяхната по-нататъшна обработка (която символично ще сведем до отпечатването им). Ето защо не умножаваме $\Omega(N)$ по мощността на комбинаторните обекти. Отнапред не е ясно дали време от порядък $\Theta(N)$ ще стигне за построяването на N комбинаторни обекта: може би има още по-голяма долна граница.

Често времето $\Theta(N)$ се оказва достатъчно за N комбинаторни обекта, тоест амортизираната времева сложност за построяването на един обект има асимптотичен порядък $\Theta(N)/N = \Theta(1)$. От друга страна, обикновено е невъзможно (и ненужно) да получим максимална времева сложност $\Theta(1)$ за построяването на всеки комбинаторен обект, поради което сме доволни, ако постигнем амортизирана времева сложност $\Theta(1)$.

Подмножества

Всяко подмножество на n -елементно множество може да се представи с n -мерен характеристичен вектор (който е двоичен низ, т.е. редица от битове): k -тата компонента на характеристичния вектор е единица тогава и само тогава, когато k -тият елемент на даденото множество принадлежи на подмножеството. Например празното подмножество \emptyset се представя чрез нулевия вектор.

Рекурсивен алгоритъм

За всеки елемент на даденото множество разполагаме с две възможности: да го включим или да не го включим в подмножеството. Оттук произтича рекурентно уравнение за броя N на подмножествата на n -елементно множество: $N(n) = N(n-1) + N(n-1)$, т.е. $N(n) = 2 \cdot N(n-1)$, за всяко цяло число $n > 0$. Начално условие: $N(0) = 1$. От двете равенства следват формулата $N = 2^n$ и рекурсивният алгоритъм, показан тук.

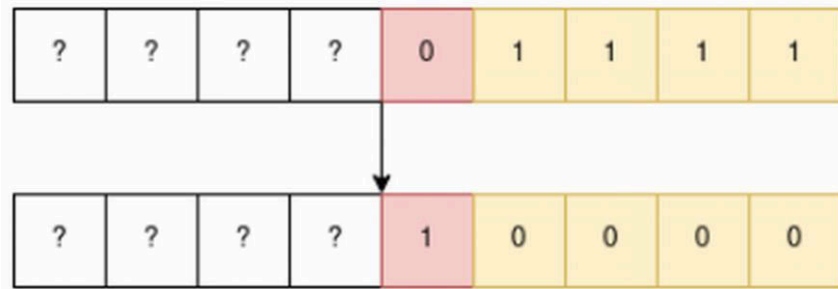
```
void generateAllSubsets(const int* arr, int len, int* bitString, int pos)
{
    if (pos == len)
        print(arr, len, bitString);
    else
    {
        bitString[pos] = 0;
        generateAllSubsets(arr, len, bitString, pos + 1);
        bitString[pos] = 1;
        generateAllSubsets(arr, len, bitString, pos + 1);
    }
}
```

При първото извикване на функцията **generateAllSubsets** ѝ се подават следните стойности на параметрите: **pos** = 0, **len** = n (броя на елементите на даденото множество); дължината на всеки от масивите **arr** и **bitString** трябва да бъде n ; съдържанието на **bitString** в този миг няма значение; масивът **arr** трябва да съдържа елементите на даденото множество (те могат да бъдат от произволен тип, но тук за простота е прието, че са цели числа).

Времевата сложност на този рекурсивен алгоритъм удовлетворява следното линейно-рекурентно уравнение: $T(n) = 2 \cdot T(n-1) + \Theta(1)$. Тук множителят 2 съответства на двете рекурсивни обръщания, а свободният член е времето за останалата работа. С помощта на характеристично уравнение намираме $T(n) = \Theta(2^n) = \Theta(N)$, тоест алгоритъмът поражда всичките подмножества възможно най-бързо. Следователно едно отделно подмножество се поражда за амортизирано константно време: $\Theta(N) / N = \Theta(1)$.

Итеративен алгоритъм

Идеята на итеративния алгоритъм е да разглеждаме всеки двоичен низ като запис на цяло неотрицателно число в двоичната бройна система. Тогава следващият низ се получава чрез събиране с единица: обхождаме двоичния низ отдясно наляво, заменяйки единиците с нули, до срещане на нула, и я заменяме с единица (ако не срещнем нула, правим извод, че няма следващ двоичен низ).



Описан на езика за програмиране Си-плюс-плюс, този алгоритъм изглежда по следния начин:

```
bool nextObj(std::vector<int>& bitString)
{
    int i = bitString.size() - 1;

    while (i >= 0 && bitString[i] == 1)
        bitString[i--] = 0;

    if (i < 0)
        return false;

    bitString[i] = 1;

    return true;
}
```

Оттук получаваме итеративен алгоритъм за генериране на подмножествата на дадено множество.

```
void generateAllSubsetsIter(const std::vector<int>& v)
{
    std::vector<int> bitString(v.size()); // 0 0 0 0 0 ... 0

    do
    {
        printSubset(bitString, v);
    } while (nextObj(bitString));
}
```

Времевата сложност е равна по порядък на броя на проверките за край на цикъла **while** от функцията **nextObj**. Проверките със стойност *лъжа* са по една за всяко излизане от цикъла, тоест за всяко изпълнение на **nextObj**, тоест по една за всяко подмножество — общо 2^n за всички подмножества. Проверките със стойност *истина* за бит № k (броен отдясно наляво) са толкова, колкото са двоичните вектори, чиито последни k бита са единици, тоест 2^{n-k} ; а общо за всичките n бита на вектора броят на проверките със стойност *истина* се намира чрез сумиране по всевъзможните стойности на $k \in \{1; 2; \dots; n\}$:

$$2^n + \sum_{k=1}^n 2^{n-k} = 2^n + 2^{n-1} + 2^{n-2} + \dots + 8 + 4 + 2 + 1 = 2^{n+1} - 1 = 2 \cdot 2^n - 1 = \Theta(2^n).$$

Затоа времевата сложност за генериране на всички подмножества е равна на $T(n) = \Theta(2^n) = \Theta(N)$, тоест тя е оптимална. Едно подмножество се генерира с амортизирана времева сложност $\Theta(N) / N = \Theta(1)$.

Код на Грей

Ако промяната на битовете на двоичния низ е трудна (скъпа) операция (например ако трябва да се превключват механично n електрически ключа), става нужно да генерираме множествата (двоичните низове) по такъв начин, че всяко множество да се получава от предишното с добавяне или премахване на единствен елемент (а всеки двоичен низ да се получава от предишния низ чрез промяна на точно един бит). На това изискване отговаря кодът на Грей. В този код последният и първият низ също се различават само в един бит.

За $n = 1$ редицата от двоични низове е следната: 0; 1.

За $n > 1$ редицата от низове се получава от редицата за $n - 1$ с дописване на нейно копие, чиито членове са подредени в обратен ред, и пред всеки низ от оригинала се добавя 0, а пред всеки низ от копието се добавя 1. Примери:

— За $n = 2$: 00; 01; 11; 10.

— За $n = 3$: 000; 001; 011; 010; 110; 111; 101; 100.

Кодът на Грей може да се програмира, както е показано тук.

```
void generateAllSubsets(const int* arr, int len, int* bitString, int pos)
{
    if (pos == len)
        print(arr, len, bitString);
    else
    {
        generateAllSubsets(arr, len, bitString, pos + 1);
        bitString[pos] = 1 - bitString[pos];
        generateAllSubsets(arr, len, bitString, pos + 1);
    }
}
```

Отначало функцията се извиква с **pos** = 0, **len** = n , **bitString** = $\vec{0}$.

Анализът на времевата сложност на този алгоритъм е съвсем същият като анализа на първия алгоритъм. Отново $T(n) = 2 \cdot T(n-1) + \Theta(1)$, защото има две рекурсивни обръщения, а времето за нерекурсивната част от работата е константа (няма цикли). Решението на уравнението е $T(n) = \Theta(2^n) = \Theta(N)$, тоест алгоритъмът поражда всичките подмножества възможно най-бързо, а едно отделно подмножество се поражда за амортизирано константно време: $\Theta(N)/N = \Theta(1)$.

Може да възникне предположението, че щом новият алгоритъм променя само един бит при прехода от всеки двоичен низ към следващия, то всеки низ се поражда за максимално (а не амортизирано) време $\Theta(1)$. Но това не е вярно, както можем да забележим от голямото сходство между програмния код на двата рекурсивни алгоритъма: при първия от тях времето за пораждаване на един двоичен низ не е $\Theta(1)$ в най-лошия случай, защото два поредни низа понякога се различават дори в n позиции; вторият алгоритъм е твърде подобен като структура, а това прави предположението неправдоподобно. Ако все пак искаме да разберем откъде идва забавянето при алгоритъма с кода на Грей, да забележим следното: въпреки че всеки път алгоритъмът променя един бит, може да има нужда от $\Theta(n)$ операции, за да реши точно кой бит да промени. Ето защо порядък $\Theta(1)$ притежава не максималната, а само амортизираната времева сложност за генериране на един двоичен низ.

Кодът на Грей има много приложения. В предаването на цифрови сигнали той намалява броя на грешките. В уреди за измерване на ъгли почти премахва възможността за грешка при отчитане на стойност; цикличността му също е предимство. При адресиране на памет намалява разхода на ел. енергия.

Ранжиране

За всеки n -мерен двоичен вектор пресмятането на поредния му номер относно лексикографската наредба се извършва по изключително лесен начин: записът на поредния номер, намален с единица, в двоичната бройна система (допълнен отляво с нули, така че дължината на записа да стане равна на n) съвпада с двоичния вектор:

$$\text{rank}(b_1 b_2 b_3 \dots b_n) = 1 + \sum_{k=1}^n b_k \cdot 2^{n-k}.$$

Аналогично, подмножествата на n -елементно множество се ранжират според поредните номера на своите характеристични вектори.

Деранжиране

Подмножество на n -елементно множество намираме по поредния номер A , тълкувайки записа на числото $A-1$ в двоичната позиционна бройна система като характеристичен вектор на подмножеството.

Пермутации

Искаме да генерираме всички пермутации на n елемента без повторения. Тази задача също може да се реши по два начина — рекурсивно и итеративно.

Рекурсивен алгоритъм

За първия елемент на пермутацията разполагаме с общо n възможности: той може да е всеки от дадените n елемента, последван от всевъзможните пермутации без повторение на останалите $n - 1$ елемента. Оттук съставяме следното рекурентно уравнение за броя N на пермутациите на n елемента без повторение: $N(n) = n \cdot N(n - 1)$, което важи за всяко цяло число $n > 0$. Начално условие: $N(0) = 1$. От двете равенства следват формулата $N = n!$ и рекурсивният алгоритъм, показан тук.

```
void generatePermutations(int* arr, int len, int index)
{
    if (index >= len)
        print(arr, len);
    else
    {
        for (int i = index; i < len; i++)
        {
            std::swap(arr[index], arr[i]);
            generatePermutations(arr, len, index + 1);
            std::swap(arr[index], arr[i]);
        }
    }
}
```

При първото извикване на функцията **generatePermutations** се подават следните стойности на параметрите: **index** = 0, **len** = n (броя на елементите); масивът **arr** съдържа някаква пермутация на n елемента без повторение (те могат да са от произволен тип; тук за простота е прието, че са цели числа).

Анализ на времевата сложност: Тялото на цикъла се изпълнява n пъти, където $n = \text{len} - \text{index}$ е броят на оставащите елементи (които трябва да бъдат разместени по всички възможни начини). Ето защо за сложността $T(n)$ важи рекурентното уравнение $T(n) = n \cdot T(n - 1)$, от което след развиване намираме $T(n) = \Theta(n!) = \Theta(N)$. Това е времето за генериране на всички пермутации. Следва, че амортизираната времева сложност за генериране на една пермутация е равна на $\Theta(N) / N = \Theta(1)$, т.е. алгоритъмът е оптимален по бързодействие.

Основно предимство на описания рекурсивен алгоритъм е простотата му. Недостатък на алгоритъма е, че не подрежда пермутациите лексикографски. Това прави итеративният алгоритъм, който предстои да разгледаме.

Итеративен алгоритъм

Следният алгоритъм генерира пермутациите на n елемента без повторение в лексикографски ред.

```
bool next(std::vector<int>& arr)
{
    if(arr.size() <= 1)
        return false;

    int k = arr.size() - 2;

    while(k >= 0 && arr[k] > arr[k+1])
        k--;

    if(k < 0)
        return false;

    size_t j = arr.size() - 1;

    while(arr[k] > arr[j])
        j--;

    std::swap(arr[k], arr[j]);

    size_t r = arr.size() - 1;

    size_t s = k + 1;

    while(r > s)
        std::swap(arr[r--], arr[s++]);

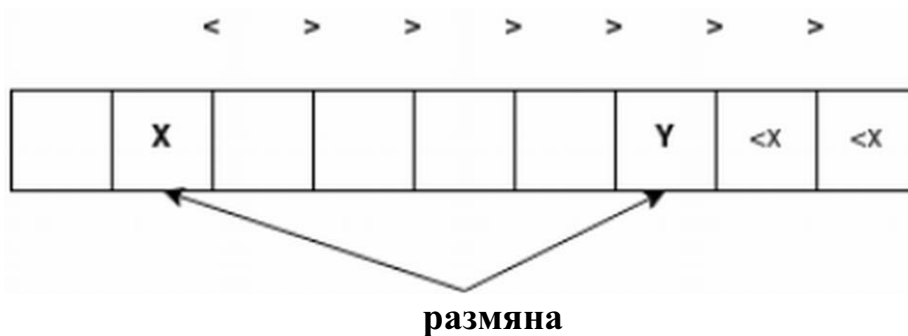
    return true;
}
```

Алгоритъмът обхожда масива отдясно наляво и спира при първия елемент, който е по-малък от предишния (тоест по-малък е от десния си съсед). Намереният елемент се увеличава минимално и обходената част от масива (надясно от мястото, на което алгоритъмът е спрял) се попълва по такъв начин, че да се получи най-малка пермутация относно лексикографската наредба. С други думи, елементите на тази част от масива трябва да се подредят във възходящ ред, но тъй като непосредствено преди това те са подредени в намаляващ ред, достатъчно е да бъде обърната разглежданата част от масива.

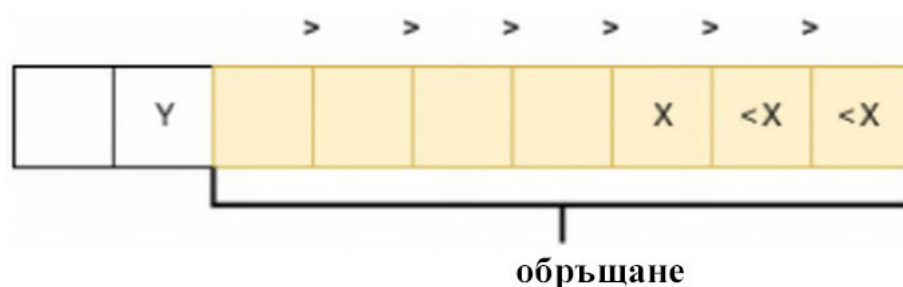
Работата на алгоритъма може да се онагледява така:

1) Обхождаме масива отляво наляво и спираме при първия елемент X , който е по-малък от десния си съсед.

2) Обхождаме изследваната част от масива и спираме новото обхождане при първия елемент $Y > X$.



3) Разменяме елементите X и Y .



4) Обръщаме частта от масива, изследвана при първото обхождане.



С това е получена следващата пермутация.

Второто обхождане винаги е успешно, тоест елемент Y винаги съществува. Това следва от избора на елемента X . Обаче първото обхождане може да е неуспешно, тоест може да не съществува подходящ елемент X . Това е признак, че елементите на текущата пермутация са подредени в намаляващ ред, поради което тя е последна относно лексикографската наредба, затова няма следваща пермутация.

Анализ на времевата сложност на итеративния алгоритъм за генериране на пермутации без повторения: Времето за генериране на следваща пермутация не е константа: в общия случай то е правопрпорционално на частта от масива, изследвана при първото обхождане, тоест то има вида cm , където c е времето за обработка на един елемент, а m е дължината на споменатата част от масива. Броят на разместванията на два елемента също е правопрпорционален на m , а оттам — и на времето.

И така, времето за едно изпълнение на процедурата **next** е равно на cm . Остава да сумираме тези времена по всички пермутации; така ще получим времето за генериране на всичките $n!$ пермутации. За да пресметнем сбора, групираме равните събираеми.

Колко пермутации завършват с намаляваща редица от точно m члена, тоест колко пермутации имат вида $\dots X < a_1 > a_2 > a_3 > \dots > a_m$? Ясно е, че множеството от стойности на последните $m + 1$ елемента можем да изберем по C_n^{m+1} начина. След това стойността на най-левия от тези елементи (тоест X) можем да изберем по m начина (той може да е всеки освен най-големия измежду избраните елементи). Другите m от тях подреждаме в намаляващ ред по единствен начин. Освен това има още $n - m - 1$ елемента наляво от X , които можем да разместваме по всички възможни начини; броят им е равен на $P_{n-m-1} = (n - m - 1)!$. Най-накрая съчетаваме начините от различните етапи всеки с всеки, затова използваме правилото за умножение:

$$C_n^{m+1} \cdot m \cdot (n - m - 1)! = \frac{n!}{(m+1)!(n-m-1)!} \cdot m \cdot (n - m - 1)! = \frac{n! m}{(m+1)!}.$$

Толкова са пермутациите, завършващи с намаляваща редица от точно m члена. Получената формула важи за всяко цяло число m от 1 до $n - 1$ включително. Тя не важи за $m = n$, защото тогава не съществува елементът X , но този случай е достатъчно лесен: има само една пермутация, чиито елементи са наредени в намаляващ ред.

Сумираме произведенията на тези бройки по времената за обработка, за да получим времето за генериране на всички пермутации:

$$T(n) = cn \cdot 1 + \sum_{m=1}^{n-1} \frac{n! m}{(m+1)!} \cdot cm = cn \cdot 1 + cn! \cdot \sum_{m=1}^{n-1} \frac{m^2}{(m+1)!} = cn! \cdot \Theta(1) = \Theta(n!).$$

Последната крайна сума е равна на $\Theta(1)$, т.е. тя е ограничена отгоре, защото съответният безкраен ред е сходящ. Това следва от принципа за сравняване на безкрайни редове с неотрицателни членове:

$$\sum_{m=1}^{\infty} \frac{m^2}{(m+1)!} \leq \sum_{m=1}^{\infty} \frac{m(m+1)}{(m+1)!} = \sum_{m=1}^{\infty} \frac{1}{(m-1)!} = \sum_{m=0}^{\infty} \frac{1}{m!} = e.$$

Забележка: В пресмятанията предполагахме, че $n > 1$, което не е проблем, защото асимптотичните оценки поначало се правят при $n \rightarrow \infty$.

Времето за генериране на всички пермутации на n елемента без повторение се получава равно на $T(n) = \Theta(n!) = \Theta(N)$. То е оптимално по порядък. Затова амортизираната времева сложност за генериране на една пермутация е константна: $\Theta(N) / N = \Theta(1)$.

Ранжиране

Поредният номер на пермутация в лексикографската наредба се пресмята по формулата

$$\text{rank}(\pi_1 \pi_2 \pi_3 \dots \pi_n) = 1 + \sum_{k=1}^{n-1} (n-k)! \left| \left\{ i: k < i \leq n, \pi_i < \pi_k \right\} \right|.$$

Номерацията започва от единица. Сумата в дясната страна представлява броят на пермутациите преди дадената пермутация в лексикографската наредба. Преброяването е по видове, като k -тият вид пермутации са тези, в които първата разлика между тях и дадената пермутация се намира на място № k при сравняване отляво надясно.

Пример: Поредният номер на пермутацията $(3 ; 5 ; 1 ; 4 ; 2)$ е равен на $1 + 2 \cdot 4! + 3 \cdot 3! + 0 \cdot 2! + 1 \cdot 1! = 1 + 48 + 18 + 0 + 1 = 68$.

Деранжиране

Тази операция се извършва пак по горната формула, но в обратна посока. Например коя е пермутацията № 94 на числата 1, 2, 3, 4 и 5 ? По-удобно е да работим с номерация, започваща от нулата (тогава липсва събираемостта 1 във формулата по-горе), затова вадим 1 от 94 и получаваме 93.

Делим 93 на $4! = 24$ и получаваме частно 3 и остатък 21. Частното определя поредния (първия) елемент на пермутацията: той е четвъртото ($3 + 1 = 4$) неизползвано досега число, тоест 4. Остатъкът 21 е новият ранг (броен от 0).

Делим 21 на $3! = 6$ и получаваме частно 3 и остатък 3. Частното 3 показва, че поредният (вторият) елемент на пермутацията е четвъртото неизползвано досега число (измежду 1, 2, 3 и 5), тоест 5. Остатъкът 3 е новият ранг.

Делим 3 на $2! = 2$ и получаваме частно 1 и остатък 1. Частното 1 показва, че поредният (третият) елемент на пермутацията е второто неизползвано досега число (измежду 1, 2 и 3), тоест 2. Остатъкът 1 е новият ранг.

Делим 1 на $1! = 1$ и получаваме частно 1 и остатък 0. Частното 1 показва, че поредният (четвъртият) елемент на пермутацията е второто неизползвано досега число (измежду 1 и 3), тоест 3. Остатъкът 0 е новият ранг.

Делим 0 на $0! = 1$ и получаваме частно 0 и остатък 0. Частното 0 показва, че поредният (петият) елемент на пермутацията е първото неизползвано досега число (единствен избор: 1), тоест 1.

Следователно търсената пермутация е $(4 ; 5 ; 2 ; 3 ; 1)$.

Комбинации

Разглеждаме комбинациите без повторения на n елемента от k -ти клас. Искаме да съставим алгоритъм, който да генерира всички тези комбинации. Тази задача също може да се реши по два начина — рекурсивно и итеративно. Но преди да съставим алгоритъма, трябва да уточним каква структура от данни ще използваме за представяне на комбинациите.

Понеже комбинациите са множества, те се представят като двоични низове (това са техните характеристични вектори). Обаче при голямо n и малко k това представяне е твърде неикономично: k -елементно множество се представя чрез низ с дължина $n \gg k$. Затова ще представяме комбинациите чрез масив с k елемента. Понеже две комбинации се различават само по елементите си, но не и по техния ред, то ще пазим елементите на масива в нарастващ ред. Без ограничение ще смятаме, че елементите са целите числа от 1 до n вкл. (в противен случай просто номерираме елементите с числата от 1 до n).

Рекурсивен алгоритъм

Добре известна е формулата за броя N на комбинациите без повторение на n елемента от k -ти клас ($0 \leq k \leq n$):

$$N = C_n^k = \frac{n!}{k!(n-k)!}.$$

В сила е тъждеството:

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}.$$

То може да се разглежда като рекурентно уравнение и съчетано с подходящи начални условия ($C_n^0 = 1$; $C_n^k = 0$ при $k > n$), води до рекурсивен алгоритъм:

- 1) Генерираме комбинациите без повторение на $n - 1$ елемента от клас k .
- 2) На последното място (тоест на място № k) записваме числото n .
- 3) Генерираме комбинациите без повторение на $n - 1$ елемента от клас $k - 1$.

Дъно на рекурсията: Когато k достигне нулата, алгоритъмът отпечата текущата комбинация. Ако n стане по-малко от k , алгоритъмът не прави нищо и веднага се връща в съседното по-плитко равнище на рекурсията.

Анализ на времевата сложност: Да означим с $T(n; k)$ времето за работа на рекурсивния алгоритъм за генериране на комбинации. Функцията $T(n; k)$ удовлетворява рекурентното уравнение

$$T(n; k) = T(n - 1; k) + T(n - 1; k - 1) + a,$$

където $a = \text{const.}$ е общото време за изпълнение на стъпка № 2 и за проверка дали е достигнато дъното на рекурсията.

Начални условия: $T(n; 0) = T(n; n + 1) = b = \text{const.}$ (времето за дъното на рекурсията; то може да се приеме за едно и също в двата случая).

С помощта на математическа индукция по n се доказва, че

$$T(n; k) = (a + b) C_{n+1}^k - a.$$

При оценка на порядъка пренебрегваме константните множители и събираеми:

$$T(n; k) \asymp C_{n+1}^k = \frac{n+1}{n-k+1} C_n^k \asymp C_n^k = N,$$

като последното асимптотично равенство е в сила, при условие че

$$\frac{n+1}{n-k+1} = \Theta(1) \text{ при } n \rightarrow \infty.$$

Условието се изпълнява от различни k и със сигурност от всички $k \leq n/2$. Не се изпълнява само от такива k , които са близо до n , но тогава $k > n/2$ и можем да заменим комбинациите от клас k с комбинации от клас $n-k$, защото едните са допълнения на другите (а пък $n-k$ вече е по-малко от $n/2$).

И така, генерирането на всичките N комбинации изразходва време $\Theta(N)$, затова генерирането на една комбинация има амортизирана времева сложност $\Theta(N)/N = \Theta(1)$. С други думи, бързодействието на рекурсивния алгоритъм е оптимално по порядък.

Итеративен алгоритъм

Ако искаме да генерираме комбинациите в азбучен ред (лексикографски), трябва да започнем от най-малката комбинация, тоест $\{1; 2; 3; \dots; k\}$, а всяка следваща да образуваме, като в текущата търсим най-десния елемент, който може да се увеличи, и да го увеличим възможно най-малко, а елементите след него да са минимални (да образуват аритметична прогресия с разлика 1).

```
bool nextObj(std::vector<size_t>& v, size_t n) // k = v.size()
{
    int j = v.size() - 1;

    while (j >= 0 && v[j] == n - v.size() + j + 1) //(*)
        j--;

    if (j < 0)
        return false;

    v[j]++;

    for (size_t i = j + 1; i < v.size(); i++) //(**)
        v[i] = v[i - 1] + 1;

    return true;
}
```

Времевата сложност е равна по порядък на броя на проверките, извършени на редовете (*) и (**). На втория ред се извършват $k + 1$ по-малко проверки на условието за край на цикъл (заради последната комбинация), поради което времевата сложност на алгоритъма е равна по порядък на броя на проверките, извършени на реда (*). Броят им е различен за различните комбинации: $m + 1$, ако комбинацията завършва на точно m максимални елемента:

$$\{ \dots ; x ; n - m + 1 ; \dots ; n - 2 ; n - 1 ; n \}, \quad x \leq n - m - 1.$$

Събираме бройките на проверките, като групираме равните събираеми. За целта е нужно да пресметнем колко пъти се среща събираемото $m + 1$, тоест колко от комбинациите завършват на точно m максимални елемента. Броят на тези комбинации се пресмята така: останалите $k - m$ техни елемента се избират произволно измежду числата $1, 2, 3, \dots, n - m - 1$. Ето защо броят на тези комбинации е равен на $C_{n-m-1}^{k-m} = C_{n-m-1}^{n-k-1}$.

Общият брой проверки, извършени на реда (*), е равен на

$$\begin{aligned} \sum_{m=0}^k (m+1) C_{n-m-1}^{n-k-1} &= \sum_{m=n-k-1}^{n-1} (n-m) C_m^{n-k-1} = \\ &= \sum_{m=n-k-1}^{n-1} [(n+1) - (m+1)] C_m^{n-k-1} = \\ &= \sum_{m=n-k-1}^{n-1} (n+1) C_m^{n-k-1} - \sum_{m=n-k-1}^{n-1} (m+1) C_m^{n-k-1} = \\ &= (n+1) \sum_{m=n-k-1}^{n-1} C_m^{n-k-1} - \sum_{m=n-k-1}^{n-1} (n-k) C_{m+1}^{n-k} = \\ &= (n+1) \sum_{m=n-k-1}^{n-1} C_m^{n-k-1} - (n-k) \sum_{m=n-k-1}^{n-1} C_{m+1}^{n-k} = \\ &= (n+1) \sum_{m=n-k-1}^{n-1} C_m^{n-k-1} - (n-k) \sum_{m=n-k}^n C_m^{n-k} = \\ &= (n+1) (C_n^{n-k} - C_{n-k-1}^{n-k}) - (n-k) (C_{n+1}^{n-k+1} - C_{n-k}^{n-k+1}) = \\ &= (n+1) C_n^k - (n-k) C_{n+1}^k = \\ &= \frac{(n+1)!(n-k+1)}{k!(n-k+1)!} - \frac{(n+1)!(n-k)}{k!(n-k+1)!} = \frac{(n+1)!}{k!(n-k+1)!} = C_{n+1}^k. \end{aligned}$$

Ето защо $T(n; k) \asymp C_{n+1}^k$ е времето за генериране на всички комбинации.

Амортизирана времева сложност за генериране на една комбинация:

$$\frac{T(n; k)}{N} \asymp \frac{C_{n+1}^k}{C_n^k} \asymp \frac{n+1}{n+1-k}.$$

Последната дроб е $\Theta(1)$, т.е. бързодействието е оптимално, за всички $k \leq n/2$ (както и за други k). А при $k > n/2$ можем вместо комбинациите от клас k да генерираме техните допълнения (те са от клас $n - k < n/2$).

Ранжиране

Поредният номер на комбинация в лексикографската наредба е равен на

$$\text{rank}(a_1 a_2 a_3 \dots a_n) = 1 + \sum_{i=0}^{k-1} \left(C_{n-a_i}^{k-i} - C_{n+1-a_{i+1}}^{k-i} \right), \text{ където } a_0 = 0.$$

Номерацията започва от единица. Сумата в дясната страна представлява броят на комбинациите преди дадената комбинация в лексикографската наредба: i -тият вид предшественици съвпадат с текущата комбинация в първите i места, на позиция № $i + 1$ съдържат число $v \in (a_i; a_{i+1})$, а другите $k - i - 1$ места са комбинация от клас $k - i - 1$ на $n - v$ елемента ($v + 1, v + 2, \dots, n$).

Деранжиране

Тази операция се извършва чрез обръщане на горната формула. Например коя е осемдесетата комбинация от трети клас на целите числа от 1 до 10 вкл.?

С числото 1 започват $C_9^2 = 36$ комбинации. Дотук са $36 < 80$ комбинации.

С числото 2 започват $C_8^2 = 28$ комбинации. Дотук са $64 < 80$ комбинации.

С числото 3 започват $C_7^2 = 21$ комбинации. Дотук са $85 \geq 80$ комбинации.

Следователно осемдесетата комбинация започва с числото 3 и е 16-та поред измежду комбинациите, започващи с 3. Затова сега търсим комбинация № 16 от втори клас на целите числа от 4 до 10 вкл. Номерът $16 = 80 - 64$.

С числото 4 започват $C_6^1 = 6$ комбинации. Дотук са $6 < 16$ комбинации.

С числото 5 започват $C_5^1 = 5$ комбинации. Дотук са $11 < 16$ комбинации.

С числото 6 започват $C_4^1 = 4$ комбинации. Дотук са $15 < 16$ комбинации.

С числото 7 започват $C_3^1 = 3$ комбинации. Дотук са $18 \geq 16$ комбинации.

Следователно шестнадесетата комбинация започва с числото 7 и е първата измежду комбинациите, започващи със 7 (тя е първата, защото $16 - 15 = 1$). Затова следващият елемент на комбинацията е числото 8, тоест тя е $\{7; 8\}$. А търсената комбинация от трети клас е $\{3; 7; 8\}$.