Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Робототехники и комплексной автоматизации

КАФЕДРА Системы автоматизированного проектирования (РК-6)

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

Студент	Никифорова Ирина Андреевна
Группа	РК6-71б
Тип задания	лабораторная работа
Тема лабораторной работы	Разбиения натуральных чисел
Вариант	13

Студент ________________ **Никифорова И. А.**
подпись, дата *фамилия, и.о.*

Преподаватель ________________ **Родионов С.В.**
подпись, дата *фамилия, и.о.*

Оценка ________________________________

Москва, 2019 г.

Оглавление

Задание на лабораторную работу

Определить суммарное количество разбиений *p(n)* для заданного натурального числа **21**. Сгенерировать все возможные разбиения целого числа **9** на различное количество частей *m*. Для генерации применить алгоритмы, предложенные Гинденбургом и Эрлихом.

Алгоритм Гинденбурга

Данный алгоритм порождает разбиения натуральных чисел в порядке увеличения количества слагаемых, при этом разбиения с одинаковым числом слагаемых перечисляются в лексикографическом порядке.

Алгоритм перечисления разбиений числа n имеет следующие шаги:

0. Начальное разбиение состоит только из самого числа n:

$$p = \{ n \}$$

1. Проверка: является ли текущее разбиение максимальным в своей разрядности m:

$$p_m - p_1 \leq 1$$

2. Если условие 1 **не** выполняется, то
   1. Текущее разбиение просматривается справа налево и ищется самый правый элемент p_i , который отличается от последнего не менее, чем на 2:

      $$p_i - p_m \geq 2$$

   2. К найденному слагаемому p_i прибавляется единица.
   3. Всем последующим слагаемым, кроме последнего, присваивается новое значение p_i :

      $$p_j = p_i\,,\ j = (i + 1)\ \ldots\ (m - 1)$$

   4. Последнему присваивается необходимый остаток для дополнения суммы:

      $$p_m = n - \sum_{k=1}^{m-1} p_k$$

3. Если условие 1 верно, то выполняется переход к следующей серии лексикографически упорядоченных разбиений на $(m + 1)$ слагаемых следующим образом:

$$p_1 = 1,\ \ldots\ , p_m = 1, p_{m+1} = (n - m)$$

Алгоритм Эрлиха

В данном алгоритме разбиения представляются в виде мультимножеств элементов. Такие мультимножества записываются в следующем виде:

$n = k_1 \bullet p_1 + k_2 \bullet p_2 + \ldots + k_m \bullet p_m$,

где p - уникальный элемент мультимножества, k - его кратность, т.е. количество повторений в мультимножестве.

Например, число 9 в такой нотации можно записать так:

$9 = \mathbf{1} \bullet \underline{3} + \mathbf{2} \bullet \underline{2} + \mathbf{2} \bullet \underline{1}$, где

{ $\underline{3}$, $\underline{2}$, $\underline{1}$ } - уникальные элементы мультимножества, {**1**, **2**, **2** } - кратности каждого из них соответственно.

Такая запись семантически равна следующим:

$9 = \mathbf{1} \bullet \underline{3} + \mathbf{2} \bullet \underline{2} + \mathbf{2} \bullet \underline{1} = \{ 1\ 1\ 2\ 2\ 3 \} = (3) + (2 + 2) + (1 + 1)$.

Алгоритм Эрлиха представляет из себя следующий набор итераций:

0. Начинается алгоритм с n единиц, то есть:

$p_0 = n \bullet 1$.

1. Рассматривается кратность последнего мультислагаемого на соответствие следующим двум случаям:

(а) $k_m > 1$ или (б) $k_m == 1$

2. Если верно условие (1.а), то выполняются следующие действия:

1) Кратность k_m последнего слагаемого мультимножества уменьшается на 2

2) Добавляется одно слагаемое типа p_{m+1}

3) Считается разность r между исходным уменьшаемым мультислагаемым и добавленными элементами, то есть:

$r = k_m * p_m - p_{m+1}$

4) Найденная разность r разбивается на r единиц, то есть записывается в кратность последнего мультислагаемого

3. Если же верно условие (1.б), то:

1) Добавляется одно мультислагаемое, на единицу большее предпоследнего присутствующего в мультимножестве, то есть на единицу увеличивается кратность элемента $p_{m-1} + 1$.

2) Исключаются два последних мультислагаемых.

3) Считается число, представленное сейчас мультимножеством. Оно вычитается из целевого числа n, а остаток записывается в кратность единиц, как в предыдущем пункте.

Листинг программы

```
#include <iostream>
#include <vector>

#define A 21    // to count fragmentations
#define B 9     // to view fragmentations
#define M -1    // number of parts, -1 for all

using namespace std;

int count_fragmentations(int n, int k) {
   if (k == 0) {
       if (n == 0) {
           return 1;
       }
       return 0;
   }

   if (k > n) {
       return count_fragmentations(n, n);
   }

   return count_fragmentations(n, k - 1) + count_fragmentations(n - k, k);
}

// create new fragmentation with Gindenburg algorithm
// by previous fragmentation and the whole number
vector<int> get_next_fragmentation_Gindenburg(vector<int> prev, int n) {
   int m = prev.size();
   vector<int> p = prev;
```

```
    if (p[m - 1] - p[0] <= 1) {
        for (int i = 0; i < m; i++) {
            p[i] = 1;
        }
        p.push_back(n - m);

        return p;
    }

    int i;
    for (i = m - 1; i >= 0; i--) {
        if (p[m - 1] - p[i] >= 2) {
            break;
        }
    }

    ++p[i];
    for (int j = i + 1; j <= m - 2; j++) {
        p[j] = p[i];
    }

    p[m - 1] = n;
    for (int k = 0; k <= m - 2; k++) {
        p[m - 1] -= p[k];
    }

    return p;
}

// print fragmentation in Gindenburg form
void print_gindenburg(vector<int> p, int i) {
    cout << "p_" << i << " = { ";
```

```
    for (int e : p) {
        cout << e << " ";
    }
    cout << "}" << endl;
}

// Gindenburg algorithm
void Gindenburg(int n, int m) {
    vector<int> p_prev = {n};
    if (p_prev.size() == m || m == -1) {
        print_gindenburg(p_prev, 1);
    }

    for (int i = 0; i < count_fragmentations(n, n) - 1; i++) {
        vector<int> p = get_next_fragmentation_Gindenburg(p_prev, n);

        if (p.size() == m || m == -1) {
            print_gindenburg(p, i + 2);
        }

        p_prev = p;
    }

    cout << endl;
}

// get index of last non zero element
int i_last_non_zero(vector<int> k, int before) {
    for (int i = before - 1; i >= 0; i--) {
        if (k[i] != 0) {
            return i;
        }
```

```
    }

    return -1;
}

// get the whole number by fragmentation in Ehrlich form
int num(vector<int> fragmentation, int n) {
    int num = 0;
    for (int i = 0; i < fragmentation.size(); i++) {
        if (fragmentation[i] != 0) {
            num += (n - i) * fragmentation[i];
        }
    }

    return num;
}

// create new fragmentation with Ehrlich algorithm
// by previous fragmentation and the whole number
vector<int> get_next_fragmentation_Ehrlich(vector<int> k_prev, int n) {
    vector<int> k = k_prev;

    int m = i_last_non_zero(k, k.size());

    if (k[m] > 1) {
        int ost = k[m] * (n - m) - (n - m + 1);
        k[m] = 0;
        k[m - 1] += 1;
        k[k.size() - 1] = ost;
    } else if (k[m] == 1) {
        int prev_non_zero = i_last_non_zero(k, m);
        k[prev_non_zero - 1] += 1;
```

```
        k[prev_non_zero] = 0;
        k[m] = 0;
        k[k.size() - 1] += n - num(k, n);
    }

    return k;
}

// print fragmentation in Ehrlich form
void print_ehrlich(vector<int> p, int i, int n) {
    cout << "p_" << i << " = { ";
    for (int j = 0; j < p.size(); j++) {
        if (p[j] != 0) {
            cout << p[j] << " • " << n - j << " + ";
        }
    }
    cout << "\b\b}" << endl;
}

// count part in fragmentation in Ehrlich form
int count_parts_ehrlich(vector<int> p) {
    int parts = 0;
    for (int e : p) {
        parts += e;
    }

    return parts;
}

// Ehrlich algorithm
void Ehrlich (int n, int m) {
    vector<int> p(n, 0);
```

```
    p[n - 1] = n;
    if (count_parts_ehrlich(p) == m || m == -1) {
        print_ehrlich(p, 1, n);
    }

    for (int i = 0; i < count_fragmentations(n, n) - 1; i++) {
        p = get_next_fragmentation_Ehrlich(p, n);

        if (count_parts_ehrlich(p) == m || m == -1) {
            print_ehrlich(p, i + 2, n);
        }
    }

    cout << endl;
}

int main() {
    cout << "p[" << A << "][" << A << "] = " << count_fragmentations(A, A) <<
endl << endl;

    cout << "Gindenburg algorithm:" << endl;
    Gindenburg(B, M);

    cout << "Ehrlich algorithm:" << endl;
    Ehrlich(B, M);

    return 0;
}
```

Результаты работы программы

p[21][21] = 792	
Gindenburg algorithm:	Ehrlich algorithm:
p_1 = { 9 }	p_1 = { 9 • 1 }
p_2 = { 1 8 }	p_2 = { 1 • 2 + 7 • 1 }
p_3 = { 2 7 }	p_3 = { 2 • 2 + 5 • 1 }
p_4 = { 3 6 }	p_4 = { 3 • 2 + 3 • 1 }
p_5 = { 4 5 }	p_5 = { 4 • 2 + 1 • 1 }
p_6 = { 1 1 7 }	p_6 = { 1 • 3 + 6 • 1 }
p_7 = { 1 2 6 }	p_7 = { 1 • 3 + 1 • 2 + 4 • 1 }
p_8 = { 1 3 5 }	p_8 = { 1 • 3 + 2 • 2 + 2 • 1 }
p_9 = { 1 4 4 }	p_9 = { 1 • 3 + 3 • 2 }
p_10 = { 2 2 5 }	p_10 = { 2 • 3 + 3 • 1 }
p_11 = { 2 3 4 }	p_11 = { 2 • 3 + 1 • 2 + 1 • 1 }
p_12 = { 3 3 3 }	p_12 = { 3 • 3 }
p_13 = { 1 1 1 6 }	p_13 = { 1 • 4 + 5 • 1 }
p_14 = { 1 1 2 5 }	p_14 = { 1 • 4 + 1 • 2 + 3 • 1 }
p_15 = { 1 1 3 4 }	p_15 = { 1 • 4 + 2 • 2 + 1 • 1 }
p_16 = { 1 2 2 4 }	p_16 = { 1 • 4 + 1 • 3 + 2 • 1 }
p_17 = { 1 2 3 3 }	p_17 = { 1 • 4 + 1 • 3 + 1 • 2 }
p_18 = { 2 2 2 3 }	p_18 = { 2 • 4 + 1 • 1 }
p_19 = { 1 1 1 1 5 }	p_19 = { 1 • 5 + 4 • 1 }
p_20 = { 1 1 1 2 4 }	p_20 = { 1 • 5 + 1 • 2 + 2 • 1 }
p_21 = { 1 1 1 3 3 }	p_21 = { 1 • 5 + 2 • 2 }
p_22 = { 1 1 2 2 3 }	p_22 = { 1 • 5 + 1 • 3 + 1 • 1 }
p_23 = { 1 2 2 2 2 }	p_23 = { 1 • 5 + 1 • 4 }
p_24 = { 1 1 1 1 1 4 }	p_24 = { 1 • 6 + 3 • 1 }
p_25 = { 1 1 1 1 2 3 }	p_25 = { 1 • 6 + 1 • 2 + 1 • 1 }
p_26 = { 1 1 1 2 2 2 }	p_26 = { 1 • 6 + 1 • 3 }
p_27 = { 1 1 1 1 1 1 3 }	p_27 = { 1 • 7 + 2 • 1 }
p_28 = { 1 1 1 1 1 2 2 }	p_28 = { 1 • 7 + 1 • 2 }
p_29 = { 1 1 1 1 1 1 1 2 }	p_29 = { 1 • 8 + 1 • 1 }
p_30 = { 1 1 1 1 1 1 1 1 1 }	p_30 = { 1 • 9 }