Рекурсия

Рекурсивным называется способ построения объекта (понятия, системы, описание действия), в котором определение объекта включает аналогичные объекты (понятие, систему, действие) в виде составных частей.

Рекурсивная функция: вызов самой себя прямо или косвенно:

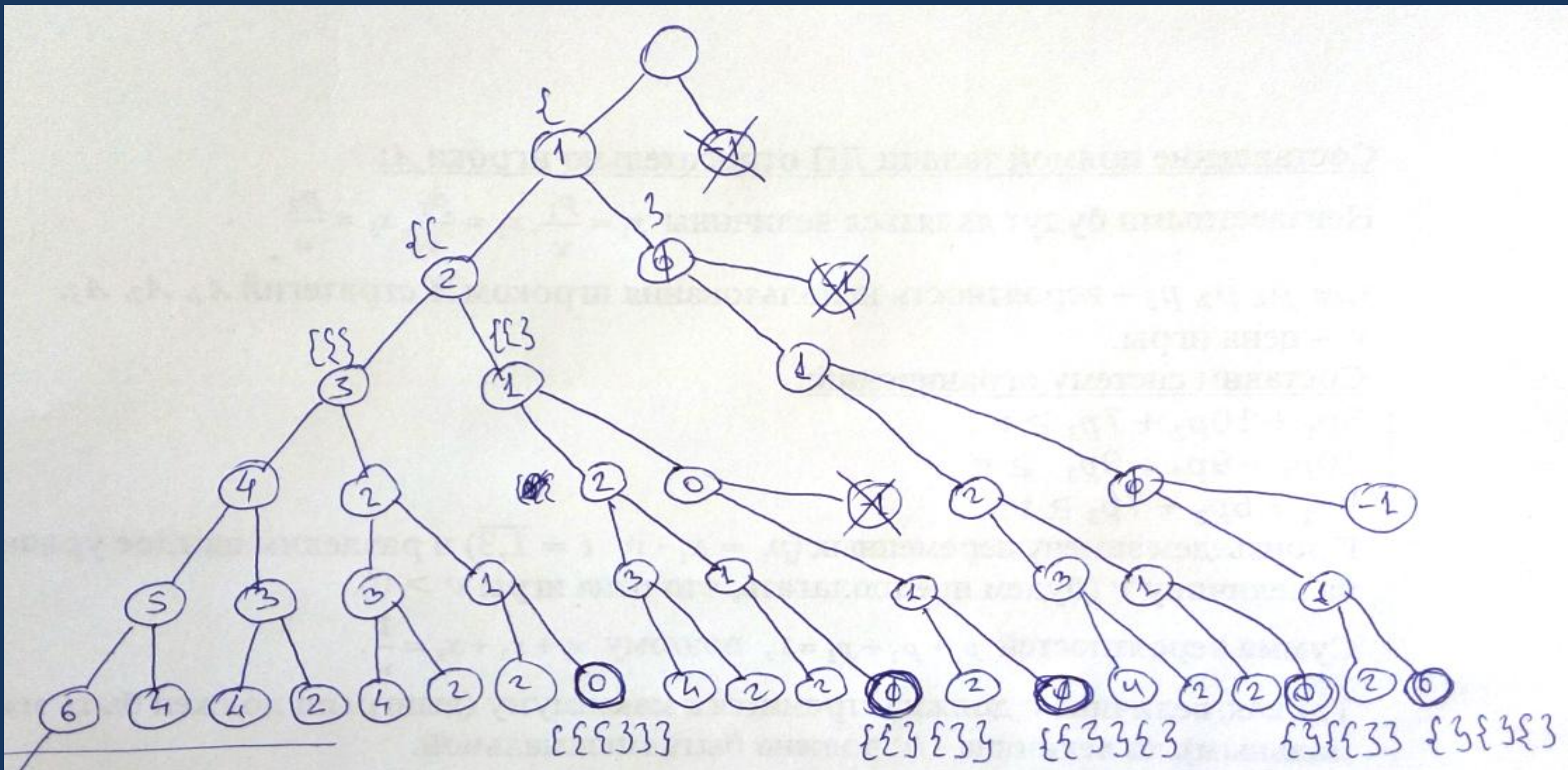
- Неограниченная рекурсия – переполнение стека
- Единственный вызов – линейная рекурсия, цикл
- Множественный вызов (несколько явных вызовов или вызов в цикле) – ветвящаяся рекурсия, дерево вызовов

Простой пример

Строки с «правильными» парами скобок

Правильные – это:

- Количество открывающихся равно количеству закрывающихся
- Закрывающихся не может быть больше открывающихся **в любой части строки** (уровень вложенности)



Строки с «правильными» парами скобок

```
public class brackets {
    static int count=0;
    static int step(char s[],int sz, int n,int over){
        if (over<0)                // Уровень вложенности отрицательный
            return 0;
        if (sz==n){                // Строка построена
            if (over==0){          // Уровень=0, одинаковое кол-во
                System.out.println(s);
                count++;           // Глобальный (общий) счетчик
                return 1;          // Одни вариант обнаружен
                }
            return 0;
            }
        int cnt=0;
        s[n]='{';                  // Вариант с открывающейся
        cnt += step(s,sz, n: n+1, over: over+1);
        s[n]='}';                  // Вариант с открывающейся
        cnt += step(s,sz, n: n+1, over: over-1);
        return cnt;                // Обратное накопление результата
        }
    public static void main(String s[]){
        int N=8;
        char cc[]=new char[N];
        count=0;
        int count2 = step(cc,N, n: 0, over: 0);
        System.out.println(count2+" "+count);
    }
}
```

Заповеди:

- думай только в рамках текущего шага: до и после то же самое
- шаг рекурсии – добавление одной из скобок (2 варианта) – получаются 2 задачи
- завершение рекурсии – достигнут результат-строка (+/-)
- Накопление результата – глобальное, прямое, обратное

Строки с «правильными» парами скобок

Тестирование

```
{{}}
{}{}
4 2 2
```

```
{{{}}}
{{}{}}
{{}}{}
{}{{}}
{}{}{}
6 5 5
```

```
{{{{}}}}
{{{}{}}}
{{{}}{}}
{{{}}}{}
{{}{{}}}
{{}{}{}}
{{}{}}{}
{{}}{{}}
{{}}{}{}
{}{{{}}}
{}{{}{}}
{}{{}}{}
{}{}{{}}
{}{}{}{}
8 14 14
```

```
{{}}{{}}{}
{{}}{}{{}}
{{}}{}{}{}
{}{{{{}}}}
{}{{{}{}}}
{}{{{}}{}}
{}{{{}}}{}
{}{{}{{}}}
{}{{}{}{}}
{}{{}{}}{}
{}{{}}{{}}
{}{{}}{}{}
{}{}{{{}}}
{}{}{{}{}}
{}{}{{}}{}
{}{}{}{{}}
{}{}{}{}{}
10 42 42
```

```
{}{}{}{}{}{{}}{{}}{}
{}{}{}{}{}{{}}{}{{}}
{}{}{}{}{}{{}}{}{}{}
{}{}{}{}{}{}{{{{}}}}
{}{}{}{}{}{}{{{}{}}}
{}{}{}{}{}{}{{{}}{}}
{}{}{}{}{}{}{{{}}}{}
{}{}{}{}{}{}{{}{{}}}
{}{}{}{}{}{}{{}{}{}}
{}{}{}{}{}{}{{}{}}{}
{}{}{}{}{}{}{{}}{{}}
{}{}{}{}{}{}{{}}{}{}
{}{}{}{}{}{}{}{{{}}}
{}{}{}{}{}{}{}{{}{}}
{}{}{}{}{}{}{}{{}}{}
{}{}{}{}{}{}{}{}{{}}
{}{}{}{}{}{}{}{}{}{}
20 16796 16796
```

```
{}{}{}{}{}{}{}{}{}{}{{{}}}
{}{}{}{}{}{}{}{}{}{}{{}{}}
{}{}{}{}{}{}{}{}{}{}{{}}{}
{}{}{}{}{}{}{}{}{}{}{}{{}}
{}{}{}{}{}{}{}{}{}{}{}{}{}
26 742900 742900
```

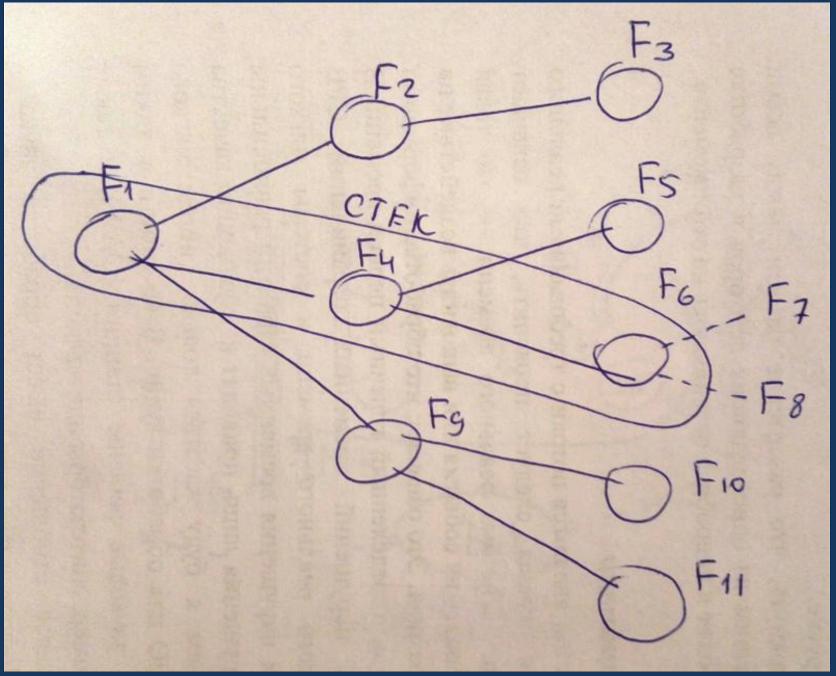
```
}{{}{}}
}{{}}{}
{}{}{}{}{}{}{}{}{}{}{}{}{{}}
{}{}{}{}{}{}{}{}{}{}{}{}{}{}
28 2674440 2674440
```

Верхняя оценка перебора для дерева с 2 вариантами – 2^N

Свойство рекурсивной функции

- Функция не вызывает сама себя, а **порождает экземпляр (**метафора)
- Фрейм стека – формальные (фактические) параметры, локальные переменные, точка возврата
- Экземпляр функции:
    - Разделяемый код
    - Фрейм стека («замороженное» состояние) на каждый вызов
    - Глобальные данные **-** разделение между экземплярами функций (вызовами)

•Дерево вызовов, стек – текущая цепочка экземпляров



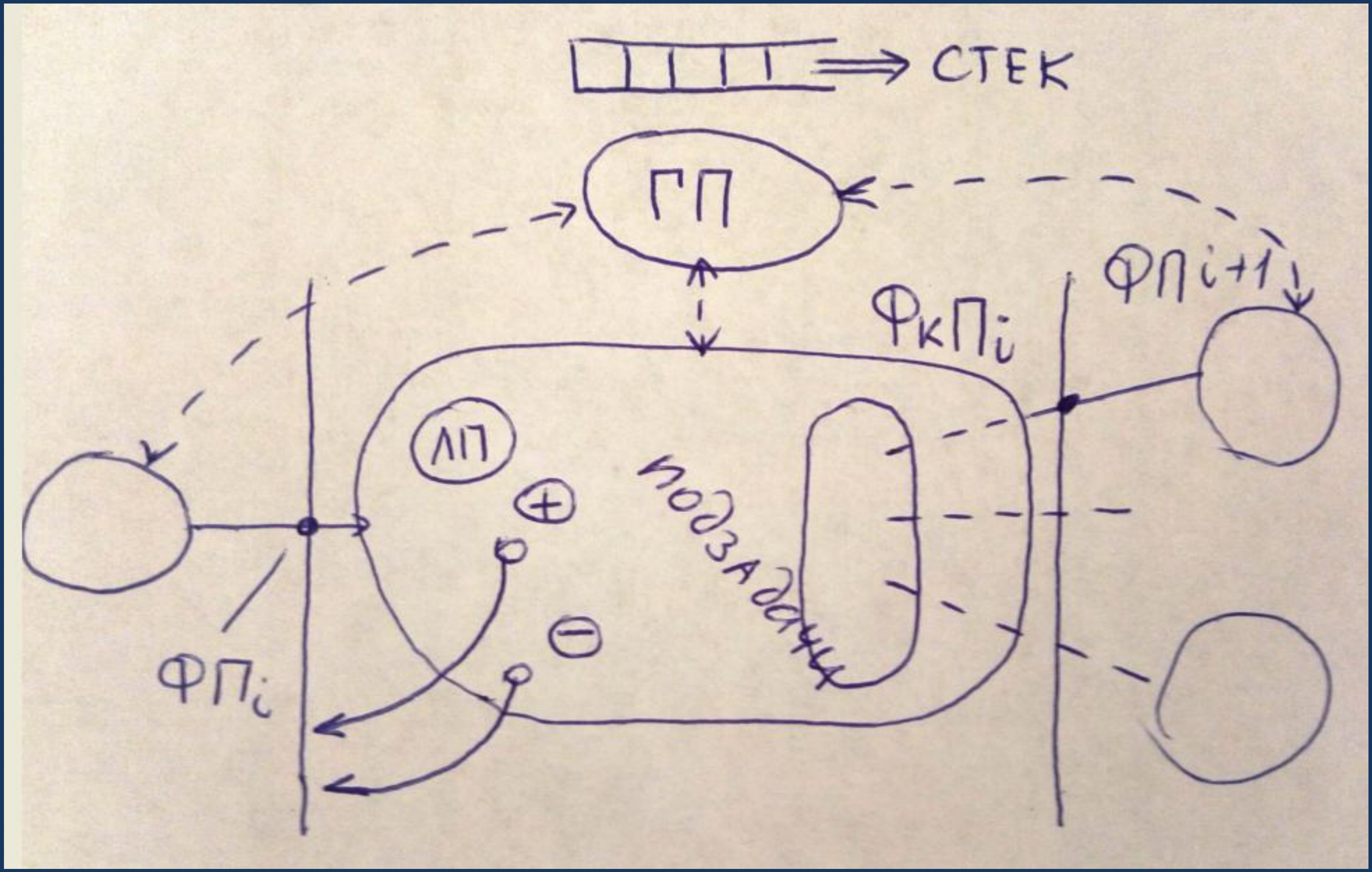
Парадоксы рекурсии

Алгоритм должен разрабатываться, не выходя за рамки текущего рекурсивного вызова:

- Шагом раньше и шагом позже будет то же самое, «исторически» отследить рекурсивный алгоритм невозможно
- Вместо этого - инвариант рекурсии: входные условия (соотношения, правила) для текущего шага (задачи) должны конвертироваться в аналогичные для следующих шагов
- Доказательство правильности рекурсивного алгоритма индуктивное (метод математической индукции по отношению к инварианту)
- **Не путь к решению задачи, а направление движения к нему:**
    - «Итак не заботьтесь о завтрашнем дне, ибо завтрашний сам будет заботиться о своем: довольно для каждого дня своей заботы». Евангелие от Матфея, 6.34
    - «Делай, что должно, и будь, что будет»
    - «Конечная цель — ничто, движение — всё» Эдуард Бернштейн (1850— 1932 гг.)

Рекурсия как процесс

Алгоритм должен разрабатываться, не выходя за рамки текущего рекурсивного вызова

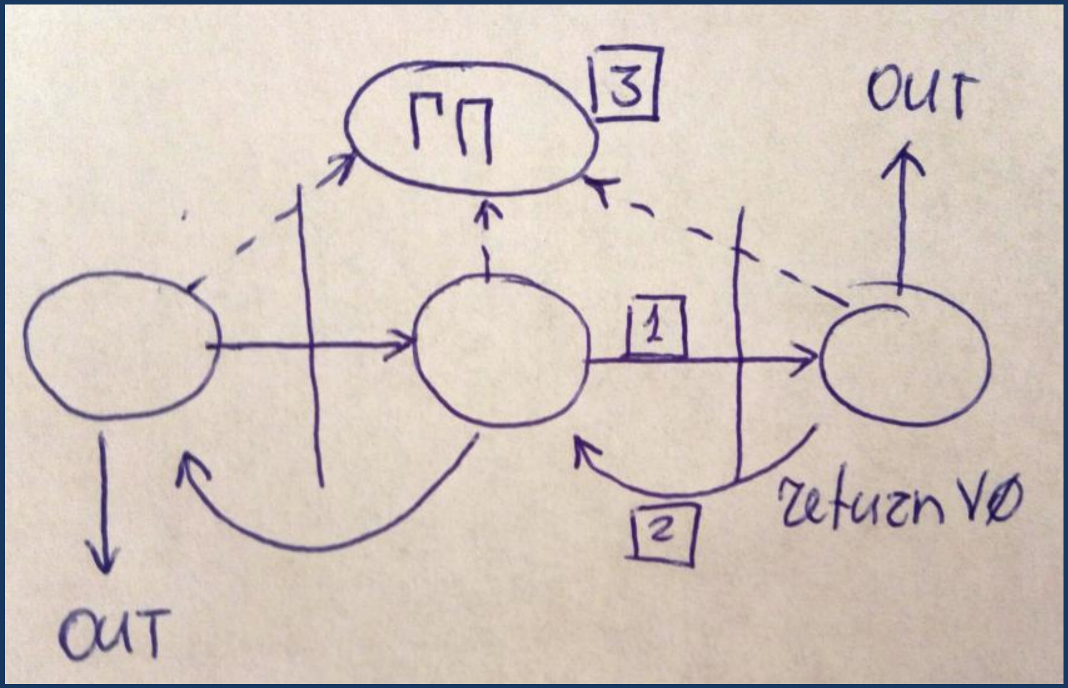


Составляющие рекурсии

1. «Зацепить рекурсию» - определить, что составляет шаг рекурсивного алгоритма;
2. Инварианты рекурсивного алгоритма. Основные свойства, соотношения, которые присутствуют на входе рекурсивной функции и которые сохраняются до следующего рекурсивного вызова, но уже в состоянии, более близком к цели;
3. Глобальные переменные – общие данные процесса в целом;
4. Начальное состояние шага рекурсивного алгоритма – формальные параметры рекурсивной функции;
5. Ограничения рекурсии – обнаружение «успеха» - достижения цели на текущем шаге рекурсии и отсечение «неудач» - заведомо неприемлемых вариантов;
6. Правила перебора возможных вариантов – способы формирования рекурсивного вызова;
7. Начальное состояние следующего шага – фактические параметры рекурсивного вызова;
8. Содержание и способ обработки результата – полный перебор с сохранением всех допустимых вариантов, первый возможный, оптимальный;
9. Условия первоначального вызова рекурсивной функции в main.

Способы формирования результата

- Прямой – ФП вызова
- Обратный – данные return (объединение от нескольких подзадач)
- Через глобальные данные (стек, откат данных при возврате)



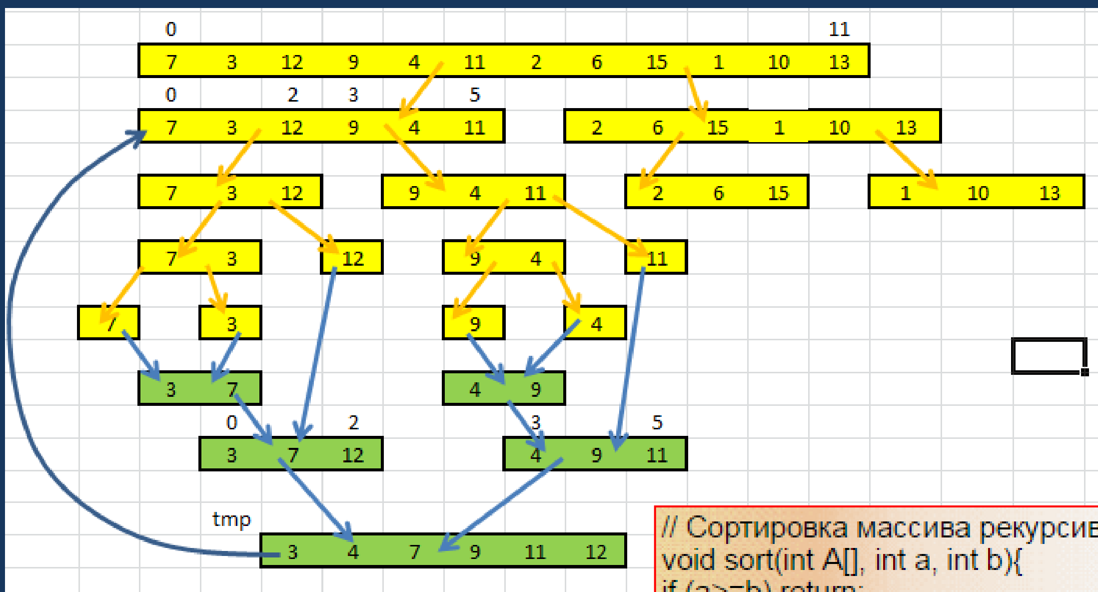
Рекурсивные сортировки

Идеи:

- Соединение упорядоченных частей – слияние (линейный процесс)
- Разделение на части относительно медианы (частичное упорядочение)
- Трудоемкость при идеальном разделении - $N \log_2 N$
- Трудоемкость при делении 1 и N-1 – N^2/2

Разделение/слияние

Рекурсивное слияние – рекурсивное разделение на 2 части, пока не станут <=1, обратное попарное слияние



$T(N) = O(N^*\log_2 N)$ – $\log_2 N$ раз слияние из N элементов, нечувствительна к данным, требуется памяти для слияния

```
// Сортировка массива рекурсивным слиянием
void sort(int A[], int a, int b){
if (a>=b) return;                        // Разделение закончилось
int m=(a+b+1)/2,i,j,k;                   // Нет - взять середину интервала
sort(A,a,m-1);                           // Рекурсивный вызов для частей
sort(A,m,b);
int *tmp=new int[b-a+1];                 // Слияние частей во временный массив
for (i=a,j=m,k=0; k<=b-a; k++)
        if (i==m || j!=b+1 && A[j]<A[i])
                tmp[k]=A[j++];           // слить из второй части, если
        else                             // первая кончилась или во второй меньше
                tmp[k]=A[i++];           // слить из первой части
for (i=a,j=0; i<=b; i++,j++)
        A[i]=tmp[j];                     // вернуть слитые части обратно в A
delete tmp;       }                      // удалить временный массив
```

Рекурсивное разделение

Быстрая сортировка

```
void sort(int in[], int a, int b){
int i,j,mode;
if (a>=b) return;
for (i=a, j=b, mode=1; i < j; mode >0 ? j-- : i++)
                if (in[i] > in[j]){
        int c = in[i]; in[i] = in[j]; in[j]=c;
        mode = -mode;
        }
sort(in,a,i-1); sort(in,i+1,b);}
```

a				b	mode до	
7	4	9	2	6	1	обмен
6	4	9	2	7	-1	i++
6	4	9	2	7	-1	i++
2	4	9	6	7	-1	обмен
2	4	7	6	9	1	j--
2	4	7	6	9	1	обмен
2	4	6	7	9	-1	i++
			a,b			
2	4	6	7	9	Рекурсия	

медина – самое левое значение
количество шагов = длина диапазона -1
разделение обменом на две части, больших и меньших медианы
медиана – между частями
рекурсивный вызов для частей

Рекурсивное разделение

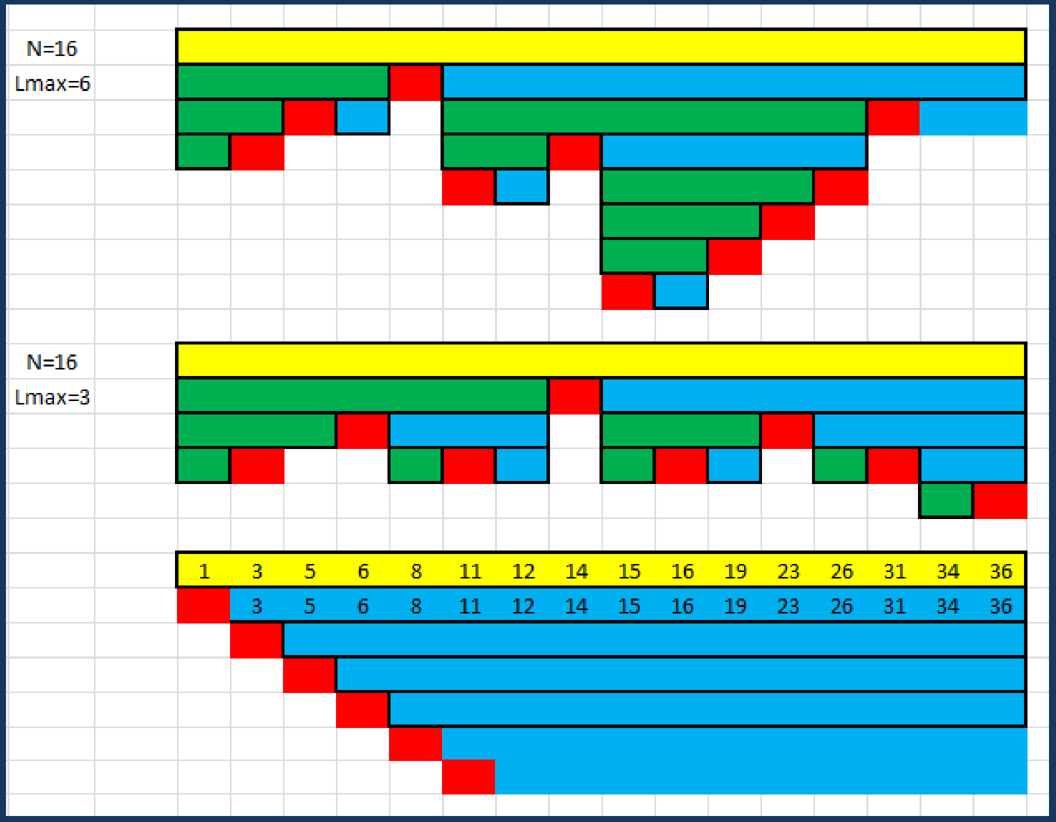
Трудоемкость

$T_{step,min}$ = $N\log_2(N)$ – идеальное разделение на равные части (без учета медианы), сумма шагов на уровне = N, уровней разделения $\log_2(N)$

$T_{step,mid}$ = …

$T_{step,max}$ = $N^2/2$ – вырожденный случай, медиана слева, сортировка упорядоченного массива

$T_{swap,min}$ = 0 – обменов нет



Рекурсивное разделение списка

- Первый элемент - медиана
- Создается 2 списка разрезанием исходного
- Рекурсивно сортируются
- Склеиваются «первый-медиана-второй»

```
void sort(list **pp){
    list *m,*p,**p1,**p2,*q;
    if (pp[0]==NULL || pp[0]->next==NULL)
        return;             // не более одного - конец разделения
    p = pp[0];
    m=p; p=p->next;         // m-медиана - первый элемент
    m->next = NULL;
    p1 = new list*[2];
    p1[0] = p1[1] = NULL;
    p2 = new list*[2];
    p2[0] = p2[1] = NULL;               // p1,p2 - разделенные списки
    while(p!=NULL){
        q=p; p=p->next;         // извлечь очередной
        if (q->val < m->val)
            add(p1,q);
        else
            add(p2,q);
            }
    printf("m=%d\n",m->val);
    printf("s1= "); show(p1);
    printf("s2= "); show(p2);
    sort(p1);               // рекурсивная сортировка частей
    sort(p2);
    printf("s1= "); show(p1);
    printf("s2= "); show(p2);
    m->next=p2[0];                  // "склеить" медиану и списки
    if (p1[0]!=NULL)
        p1[1]->next = m;
    pp[0] = (p1[0]==NULL ? m : p1[0]);  // первый - пустой = первая - м
    pp[1] = (p2[0]==NULL ? m : p2[1]);  // второй - пустой = последняя
    }
```

```
void add(list *pp[], list *q){
    q->next = NULL;
    if (pp[0]==NULL)
        pp[0]=pp[1]=q;
    else{                       // Вставка последним
        pp[1]->next = q;        // Новый вслед за последним
        pp[1]=q;                // Послений = новый
        }
    }
```

```
41 67 34 0 69 24 78 58 62 64 5 45
m=41
s1= 34 0 24 5
s2= 67 69 78 58 62 64 45
m=34
s1= 0 24 5
s2=
m=0
s1=
s2= 24 5
m=24
s1= 5
s2=
s1= 5
s2=
s1=
s2= 5 24
s1= 0 5 24
s2=
m=67
s1= 58 62 64 45
s2= 69 78
m=58
s2= 78
s1=
s2= 78
s1= 45 58 62 64
s2= 69 78
s1= 0 5 24 34
s2= 45 58 62 64 67 69 78
0 5 24 34 41 45 58 62 64 67 69 78
```

Рекурсия: комбинаторный перебор

Идеи:

- Простое решение – полный перебор
- Линейный перебор (цикл), перебор пар (2 цикла), комбинаторный
- Рекурсия – моделирование процесса генерации вариантов: выбор одного элемента порождает несколько комбинаций

Вид последовательности	**Алгоритм n-го шага**	**Ветвление**	**T**
Все элементы	Цикл **for(i=0;i<n;i++)**	---	n
Все пары	Цикл в цикле **for(i=0;i<n-1;i++) for(j=i-1;j<n;j++)**	---	n(n-1)/2
Все подмножества	Очередной (n-ый) элемент из R может быть «добавлен» и «не добавлен» в W	2	2^n
Подмножества из n по m	Очередной (n-ый) элемент из R может быть «добавлен» и «не добавлен» в W при ограничении их количества в W	2(1,0)	N! m!(n-m!)
Последовательность без повторений	В W может быть добавлен любой из R, который в нем остался	n...1	N!
Последовательность с повторениями	В W может быть добавлен любой из R	n	n^n

Генерация подмножеств

Идеи:

- Выбрано подмножество из k элементов
- Очередной может быть включен и не включен (2 варианта)
- Для каждого – рекурсия
- Трудоемкость – 2^N

```
//----------------------------------------------------------74-01
// Множество всех подмножеств
// R - исходное множество
// W - результирующее множество (последовательность)
// n - номер шага в глубину - индекс выбираемого элементв из R
// k - количество выбранных элементов
// N - размерность задачи
// cnt - счетчик вариантов
void F1(int W[],int R[],int n,int k,int N, int &cnt){
int i;
if (n==N){
    cnt++; printf("\n");          // достигнута требуемая размерность
    for (i=0;i<k;i++) printf("%d ",W[i]);
    return; }                     // подсчет и вывод полученного вариаета
F1(W,R,n+1,k,N,cnt);              // очередной не включается
W[k]=R[n];
F1(W,R,n+1,k+1,N,cnt);            // очередной включается
}
```

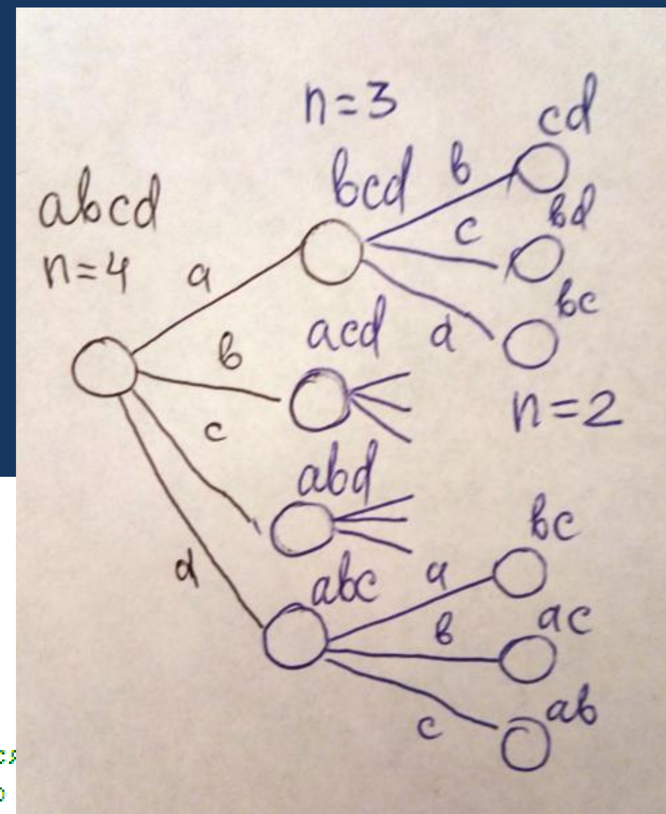
Альтернатива – битовая карта:

- Машинное слово – N разрядов, каждый разряд 0-не включен, 1-включен
- Варианты – все значения машинного слова
- Генератор – цикл int sz=1<<N; for(v=0;v<sz;v++){…}

Генерация последовательностей

Идеи:

- Выбирается 1 из N – N вариантов последовательностей
- Для каждого варианта выбирается по 1 из N-1 (оставшихся)
- Трудоемкость – N*(N-1)*(N-2)…1 = N!
- В каждом вызове – цикл с пропуском уже выбранных




```
// Последовательноть без повторений
void F3(int W[],int R[],int n, int N, int &cnt){
int i;
if (n==N){
    cnt++; printf("\n");
    for (i=0;i<n;i++) printf("%d ",W[i]);
    return; }
for (i=0;i<N;i++){              // цикл по всем оставшимс
    if (R[i]==0) continue;      // пропуск уже выбранного
    W[n]=R[i];                  // выбор оставшегося
    R[i]=0;                     // исключение из исходного
    F3(W,R,n+1,N,cnt);
    R[i]=W[n];                  // возвращение выбранного
  }}
```

Домино – генератор последовательностей

Идеи:

- Кость – двузначное число
- Генератор последовательностей для построения цепочек
- Доп. условие – совпадение второй цифры последней кости с одной из цифр очередной
- Первый шаг – нет предыдущей, берется любая

```
27 74 44 46 63
27 74 44 46 63
36 64 44 47 72
36 64 44 47 72 cnt=55
```

```
void F3(int W[],int R[],int n, int N, int &cnt){
int i;
cnt++;
if (n==N){
    printf("\n");
    for (i=0;i<n;i++) printf("%d ",W[i]);
    return; }
for (i=0;i<N;i++){
    if (R[i]==-1) continue;
    W[n]=R[i];
    R[i]=-1;
    if (n==0 || W[n]/10==W[n-1]%10)
        F3(W,R,n+1,N,cnt);
    invert(W[n]);
    if (n==0 || W[n]/10==W[n-1]%10)
        F3(W,R,n+1,N,cnt);
    invert(W[n]);
    R[i]=W[n];
    }
  }
```

```
int main(){
    int in[]={27,74,63,64,44};
    int out[100];
    int cnt=0;
    F3(out,in,0,sizeof(in)/sizeof(int),cnt);
    printf("\ncnt=%d\n",cnt);
    return 0;
```

```
void invert(int &a){
    a = (a%10)*10 + a/10;
    }
```

Раскраска карты

Шаг 0. Данные

```
#include <stdio.h>
struct coloredcard{
    //-------------Общие данные алгоритма (псевдо-глобальные)
    int *D;             // Массив раскраски D[i]-цвет i-ой страны
    int **M;            // Матрица смежности M[i][j]==1 - общая граница
    int N;              // КОличество стран
    int *DMIN;          // Оптимальная раскраска
    int ncMin;          // Найденное мин. кол-во цветов
    void calc(int n){
        N=n;
        D=new int[n];   // Инициализировать не нужно
        DMIN=new int[n];
        ncMin=-1;       // Защелка- нет решения
        }

    void step(){}
    };

    void main(){
        coloredcard CC;
        CC.calc(10);
        }
```

Шаг 1. Зацепить рекурсию

```
//---Зацепить рекурсию - шаг - возможные варианты раскраски i-ой страны
// nc - количество используемых цветов
// D[0..i-1] - раскрашенные страны
void step(int i, int nc){
    int k;
    D[i]=k;         // Какой-то цвет k=0..nc-1
    step(i+1,nc);
    ...
    D[i]=nc;        // Новый цвет
    step(i+1,nc+1);
    }
};
```

Раскраска карты

Шаг 2. Логика перебора

```
//---Зацепить рекурсию - шаг - возможные варианты раскраски i-ой страны
// nc - количество используемых цветов
// D[0..i-1] - раскрашенные страны
void step(int i, int nc){
    int k;
    D[i]=k;         // Какой-то цвет k=0..nc-1
    step(i+1,nc);
    ...
    D[i]=nc;        // Новый цвет
    step(i+1,nc+1);
    }
};
```

Шаг 3. Цикл перебора – проверка раскрашенных ранее соседей

```
//---Зацепить рекурсию - шаг - возможные варианты раскраски i-ой страны
// nc - количество используемых цветов
// D[0..i-1] - раскрашенные страны
void step(int i, int nc){
    for(int k=0;k<nc;k++){
        if (1/* k-цвета нет у раскрашенных соседей 0..i-1 */){
            D[i]=k;         // Какой-то цвет k=0..nc-1
            step(i+1,nc);
            }
        }
    D[i]=nc;        // Новый цвет
    step(i+1,nc+1);
    }
};
```

Раскраска карты

Шаг 4. Цикл перебора – признак совпадения цветов

```
void step(int i, int nc){
    for(int k=0;k<nc;k++){
        int disable=0;
        D[i]=k;                         // Взять цвет
        for (int j=0;j<i;j++){              // Все предыдущие
            if (M[i][j]!=0 && D[j]==D[i]){
                disable=1;              // Есть граница и цвета совпадают
                break;                  // - НЕЛЬЗЯ
                }
            }
        if (disable==0){
            step(i+1,nc);
            }
        }
    D[i]=nc;         // Новый цвет
    step(i+1,nc+1);
    }
```

```
void main(){
    coloredcard CC;
    CC.calc(10);
    CC.step(0,0);
    }
```

Шаг 7. Начальный вызов - main

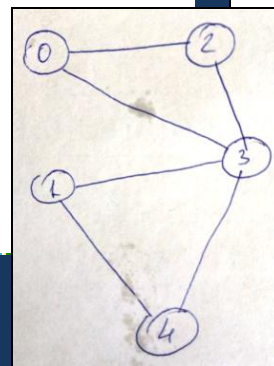
Шаг 5. Завершение рекурсии – очередная раскраска N стран

```
void step(int i, int nc){
    if (i==N){          // Перебрали всех - очередной вариант
        if (Lmin==-1 || nc <= Lmin){
            Lmin=nc;    // Очередной ЛУЧШЕ
            for (j=0;j<MM.N;j++) Dmin[j]=D[j];
            }
        return;         // Завершить рекурсию
        }
```

Раскраска карты

Шаг 8. Тестирование. Оживление

```
void calc(int **A,int n){
    N=n;
    M=A;
    D=new int[n];    // Инициализировать не нужно
    DMIN=new int[n];
    ncMin=-1;        // Защелка- нет решения
    }
void show(){
    for(int i=0;i<N;i++)
        printf("%d ",DMIN[i]);
    printf("\nnc=%d\n",ncMin);
    }
```



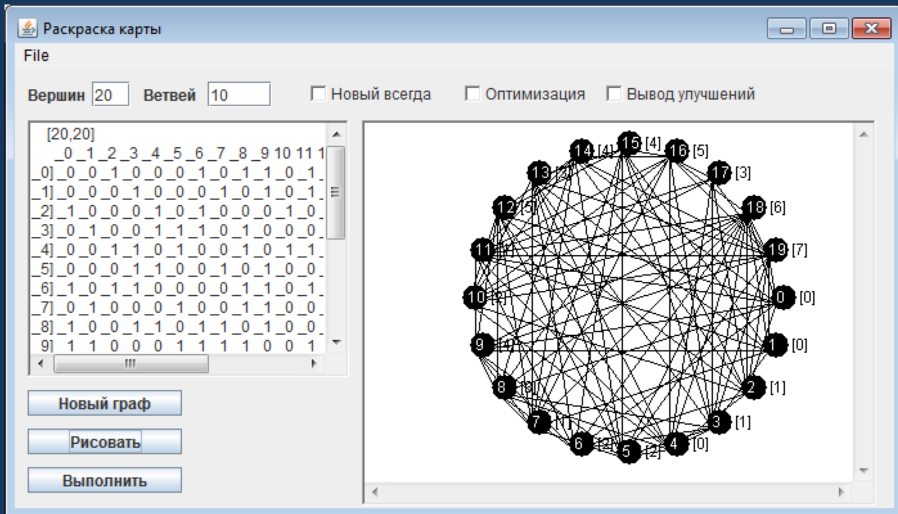

```
int a0[]={0,0,1,1,0};
int a1[]={0,0,0,1,1};
int a2[]={1,0,0,1,0};
int a3[]={1,1,1,0,1};
int a4[]={0,1,0,1,0};
int *A[]={a0,a1,a2,a3,a4};
    void main(){
        coloredcard CC;
        CC.calc(A,5);
        CC.step(0,0);
        CC.show();
        getchar();
        }
```

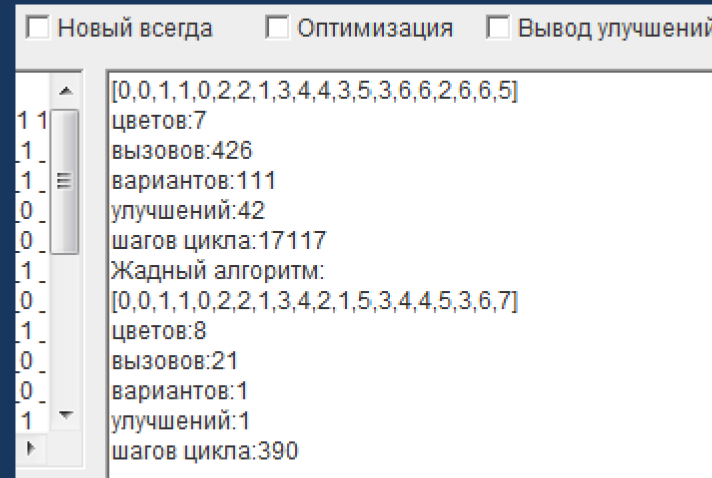
```
D:\temp\fufu\Debug\fufu.exe
0 1 1 2 0
nc=3
```

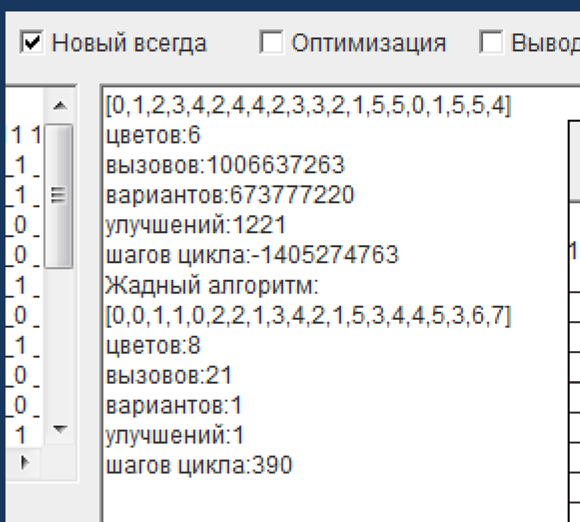
Шаг 9. Оптимизация

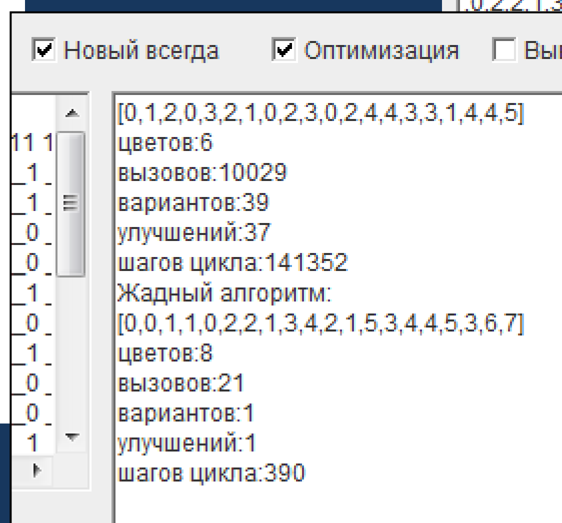
- **Динамическое программирование** – использование ранее накопленных данных, чтобы не делать лишнюю работу: если nc >= ncMin на промежуточном шаге, то смысла продолжать нет
- **Жадный алгоритм** – выбирается один из (первый попавшийся) вариант продолжения, остальные не рассматриваются = линейный алгоритм. Доказательство результативности алгоритма
    - дает оптимальное решение (всегда)
    - дает субоптимальное решение
    - не находит решения при его наличии

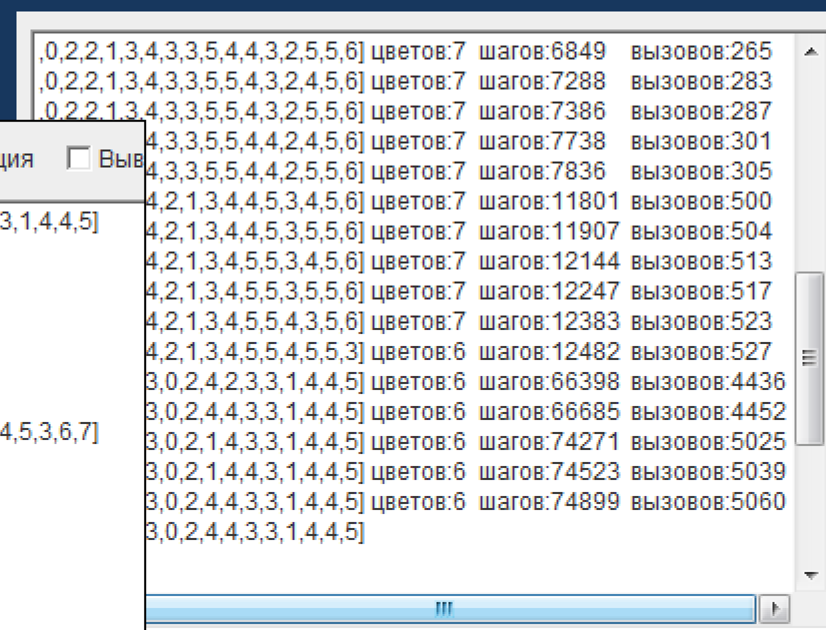
Раскраска карты (Java)











Раскраска карты (модель, Java)

```
// Раскраска карты - СУБОПТИМАЛЬНЫЙ ЖАДНЫЙ АЛГОРИТМ и ПОЛНЫЙ РЕКУРСИВНЫЙ ПЕРЕБОР
// mode0=1 - проверка на превышение количества цветов (учет предыдущего решения)
// mode1=1 - проверка нового цвета ВСЕГДА (даже если есть другие цвета)
    void step(int i,int nc,boolean only){
        int j,k;
        ncall++;
        if (i==MM.N){
            nvar++;
            if (Lmin==-1 || nc <= Lmin){
                nplus++;
                Lmin=nc;
                for (j=0;j<MM.N;j++) MM.Dmin[j]=MM.D[j];
                if (mode2){
                    String s="[";
                    for (j=0;j<MM.N-1;j++) s=s+MM.Dmin[j]+",";
                    s=s+MM.Dmin[j]+"] цветов:"+Lmin+"  шагов:"+nmatr+"\tвызовов:"+ncall+"\n";
                    OUT.append(s);
                    }}
            return;
            }
        if (mode0 && Lmin!=-1 && nc >=Lmin) return;
        int v=0;
        for (k=0;k<nc;k++){
            MM.D[i]=k;
            for (j=0;j<i;j++,nmatr++)
                if (MM.M[i][j]!=0 && MM.D[j]==k) break;
            if (j==i) { v++; step(i+1,nc,only); if (only) return;  }
            }
        MM.D[i]=nc;
        if (v==0 || mode1) step(i+1,nc+1,only);
        }
```

жадный

кол-во вариантов

оптимизация

новый всегда

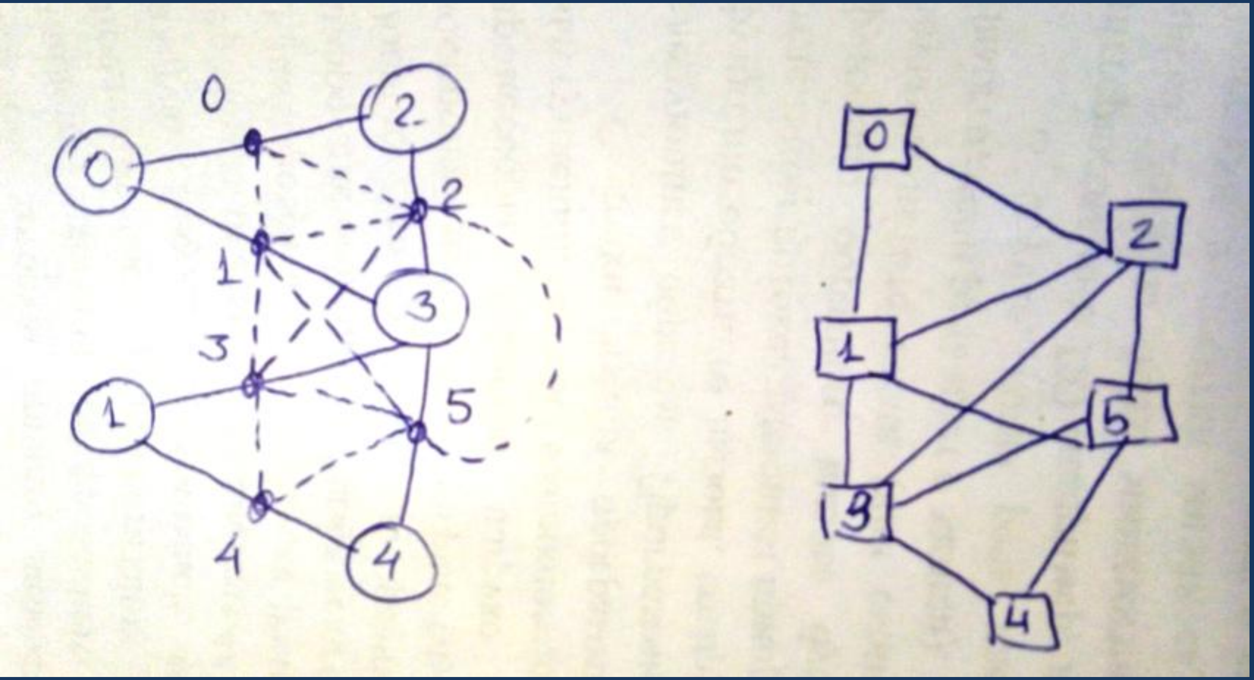
жадный

Выбор проводов

Задача. В каждом узле входящие соединения с соседними должны быть разного цвета (чтобы электрики не спутали)

Решение. Задача раскраски карты для **рёберного** графа

1. Дуги графа – вершины нового
2. Вершины нового связаны дугой, если дуги старого имеют общую вершину



Трудоемкость рекурсии

Зависит от количества подзадач (рекурсивных вызовов) и размерностей данных для них, а также от количества операций на текущем уровне .

	Размерность	Изменение размерности задачи	Трудоемкость
1	$T_N=NT_{N-1}+N$	не каждом шаге рекурсии возникает N задач размерности, меньшей на 1, на каждом шаге число выполняемых операций пропорционально размерности задачи.	$T=N!$
1	$T_N=T_{N-1}+N$	с каждым шагом рекурсии размерность задачи уменьшается на 1, на каждом шаге число выполняемых операций пропорционально размерности задачи.	$T=N^2/2$
2	$T_N=T_{N/2}+1$	с каждым шагом рекурсии размерность задачи уменьшается в два раза при выполнении единственной на этом шаге операции	$T=\log_2 N$
3	$T_N=T_{N/2}+N$	с каждым шагом рекурсии размерность задачи уменьшается в два раза, число операций на каждом шаге пропорционально размерности задачи. Общая трудоемкость	$T=2N$
4	$T_N=2T_{N/2}+N$	с каждым шагом рекурсии задача разбивается на две, размерность которых в два раза меньше исходной, число операций на каждом шаге пропорционально размерности задачи	$T=N\log_2 N$
5	$T_N=2T_{N/2}+1$	с каждым шагом рекурсии задача разбивается на две, размерность которых в два раза меньше исходной, при выполнении единственной на этом шаге операции.	$T=2N$

Трудоемкость рекурсии

Примеры рекурсивной реализации алгоритмов

```
int F12(int A[],int n){
    if (n<=0) return 0;
    if (n==1) return A[0];
    int m=n/2;
    int v1=F12(A,m);
    int v2=F12(&A[m],n-m);
    return v1>v2 ? v1: v2;
    }
```

Поиск максимального: Tn = 1 + 2Tn/2   T=2N

Двоичный поиск:
Tn = 1 + Tn/2
T=$\log_2$N

```
int F15(int A[], int a, int b, int val){
    int i,j,mode;
    if (a>=b) return -1;
    int m=(a+b)/2;
    if (val==A[m]) return m;
    if (val<A[m])  return F15(A,a,m-1,val);
    return F15(A,m+1,b,val); }
```

«Тупая» сортировка:
Tn = n + Tn-1   T= N^2/2

```
void F16(int A[],int n){
    if (n==1) return;
    int i,k;
    for(i=k=0;i<n;i++)
        if (A[i]<A[k]) k=i;
    int c=A[k]; A[k]=A[n-1];A[n-1]=c;
    F16(A,n-1);
    }
```

«Умная» сортировка: Tn = n + 2Tn/2
T= N $\log_2$N (при равном делении)

```
void F21(int A[], int a, int b){
    int i,j,mode;
    if (a>=b) return;
    for (i=a, j=b, mode=1; i != j; mode >0 ? i++ : j--)
    if (A[i] > A[j]){
        int c;
        c=A[i]; A[i]=A[j]; A[j]=c; mode = -mode;
        }
    F21(A,a,i-1); F21(A,i+1,b); }
```

Генератор подмножеств: Tn = 1 + 2Tn-1
T= 2^N

```
void F1(int W[],int R[],int n,int k,int N,
int i;
if (n==N){
    cnt++; printf("\n");          // достигн
    for (i=0;i<k;i++) printf("%d ",W[i]);
    return; }                     // подсчет
F1(W,R,n+1,k,N,cnt);              // очередн
W[k]=R[n];
F1(W,R,n+1,k+1,N,cnt);            // очередн
}
```

Эффективность алгоритмов

Основные идеи, связанные с эффективностью (трудоемкостью)

Тип алгоритма	Идея алгоритма и «природа» его эффективности	Диапазон трудоемкостей
Рекурсивное или обычное разделение	«Разделяй и властвуй»: задача разбивается на идентичные подзадачи, результаты которых объединяются в общее решение	$N \ldots N \log N \ldots N^2$
Полный перебор	«Хуже не бывает» (без комментариев)	$2^N \ldots N^N \ldots N!$
Динамическое программирование	«Де жа вю»: запоминание результатов повторяющихся подзадач, увеличение производительности за счет дополнительной памяти.	
Жадный алгоритм	«Рыцарь на распутье»: локальный выбор единственной из подзадач на каждом шаге дает глобальное оптимальное решение	$\log N \ldots N$

Рекурсия, стек, очередь

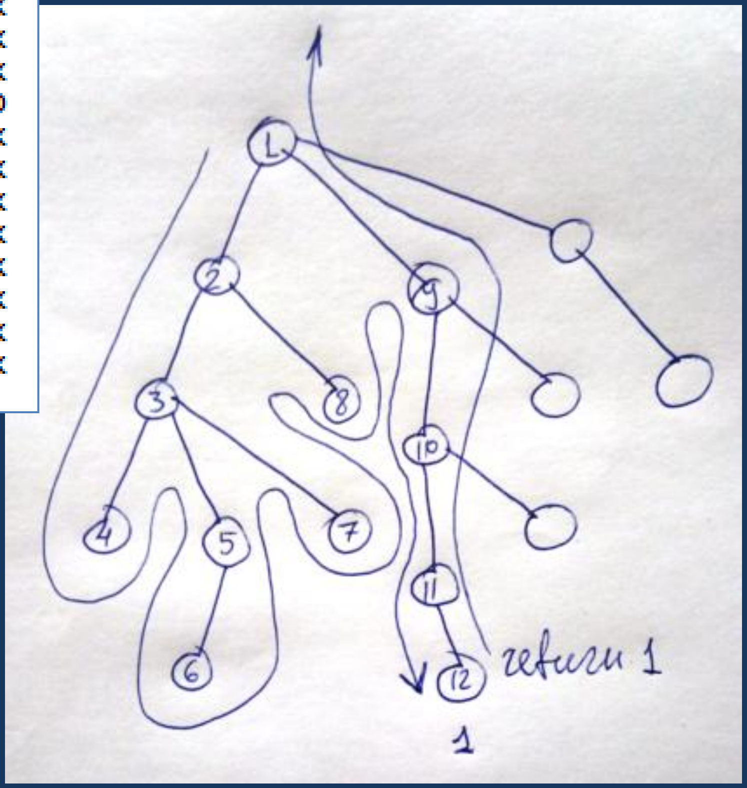
- Рекурсивный обход (перебор вариантов) – стек вызовов
- Обход на основе стека – «в глубину»
- Явное использование стека вместо рекурсии
- Очередь для перебора вариантов – обход «в ширину» (по слоям)
- Явное использование очереди

Пример: поиск выхода в лабиринте (73-03.cpp)

Файл Правка Формат

```
xxxxxxxxxxxxxxxxxx
x x              x
x x xxxxxxxxxxx  x
x x   x          x
x     x  xxx x xxx
x xxxxxx x   x   O
x          x   x x
x xxxxxxxxxx xxx x
x   x   x    x   x
xxx xxx xxx xxx xx
x    x  x   x x xx
xxx xxx x xxx x xx
x    I  x   x   xx
xxxxxxxxxxxxxxxxxx
```

```
int step(int y,int x){
if (LB[y][x]=='O') return 1;          // выход найден
if (LB[y][x]!=' ') return 0;          // стенки и циклы
LB[y][x]='.';                         // отметить точку
if (step(y+1,x)) return 1;
if (step(y,x+1)) return 1;
if (step(y-1,x)) return 1;
if (step(y,x-1)) return 1;
LB[y][x]=' ';                         // снять отметку
return 0;}
```



Рекурсия, стек, очередь

- Явное использование стека вместо рекурсии
- Очередь для перебора вариантов – обход «в ширину» (по слоям)
- Явное использование очереди

Пример: поиск выхода в лабиринте (75-01.cpp)

```
int LB[10][10]={
{1,1,1,1,1,1,1,1,1,1},
{1,1,0,1,0,1,1,0,1,1},
{1,1,0,0,0,0,1,0,0,1},
{1,1,0,1,1,0,1,0,1,1},
{1,0,0,1,1,0,1,0,0,0},
{1,1,0,1,1,0,1,0,1,1},
{1,0,0,1,1,0,1,0,1,1},
{1,1,0,0,1,0,0,0,0,1},
{1,1,1,1,1,1,1,1,1,1},
{1,1,1,1,1,1,1,1,1,1}};
```

```
void main(){
S.push(A);                      // Поместить в стек
int found=0;                    // Признак завершения
while(S.size()!=0){             // Цикл извлечения шагов из стека
    XY D=S.pop();               // Извлечь из стека данные нового шага
    if (D.last==1){
        LB[D.x][D.y]=0;         // "последний" вариант - снять отметку
        continue;
        }
    if (D.x<0 || D.x>9 || D.y<0 || D.y>9)
        {found=1; break; }      // Достигли края - выйти из цикла
    if (LB[D.x][D.y]!=0)
        continue;               // стенки и повторы - завершить шаг
    LB[D.x][D.y]=2;             // отметить точку
    S.push(XY(D.x,D.y,1));      // Вариант для "завершения" шага рекурсии
    S.push(XY(D.x+1,D.y));      // варианты для соседних точек
    S.push(XY(D.x,D.y+1));
    S.push(XY(D.x-1,D.y));
    S.push(XY(D.x,D.y-1));
    }
if (found)
    for (int i=0; i<10; i++,puts(""))
    for (int j=0; j<10; j++) printf("%d",LB[i][j]);
```

```
struct XY{
    int x,y,last;
    XY(int x0,int y0,int la
    XY(){ x=y=last=0; }
    };
stack<XY,100> S;                // Явный стек
XY  A(5,5);
```

Рекурсия, стек, очередь

- Очередь для перебора вариантов – обход «в ширину» (по слоям), волновой алгоритм
- Явное использование очереди

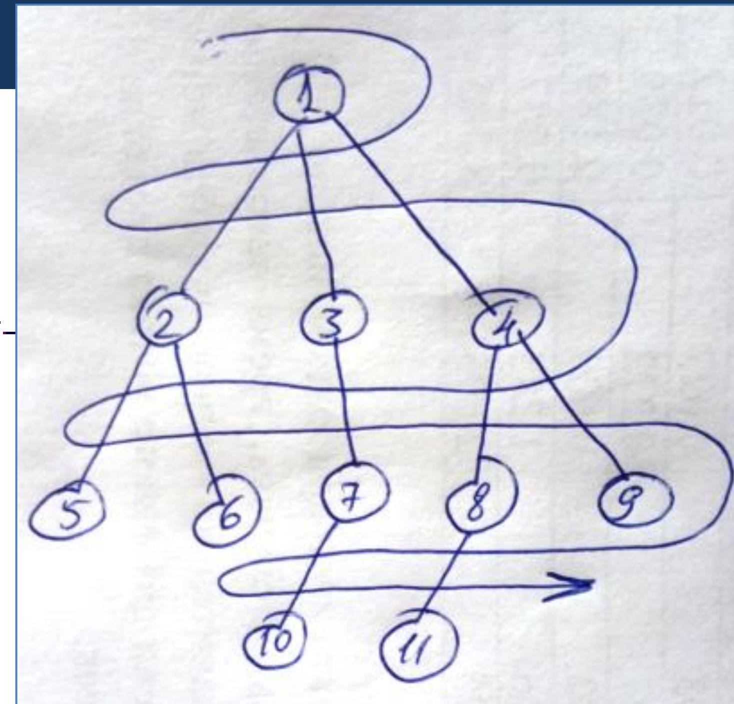
Пример: поиск кратчайшего пути

- прохождение волны через вершину моделируется помещением ее в очередь
- волновой алгоритм извлекает вершины из очереди и помещает в нее некоторую часть «соседей», в которые эта волна распространяется
- при распространении волны в соседнюю вершину в нее помещается длина пути из начальной = длина пути до текущей + расстояние до «соседа»
- волна распространяется в не пройденные волной вершины
- волна распространяется в пройденные вершины, если новое расстояние меньше старого, в этом случае она вызывает «повторную волну»
- зацикливание алгоритма и повторное прохождение волны в обратном направлении исключается предыдущим условием
- **Алгоритм Дейкстры - более эффективный (без повторной волны)**

Рекурсия, стек, очередь

Циклическая очередь

```
template <class T, int sz>
class queue{
    T area[sz];
    int fst,lst;
public:
    int size(){ return lst>=fst ? lst-fst : fst+N-
    queue(){ fst=lst=0; }
    T out(){
        static T null;
        if (fst==lst) return null;
        null=area[fst++];
        fst%=sz;
        return null;
        }
    void in(T v){ area[lst++]=v; lst%=sz; }
    };
```




```
queue<int,100> Q;                // Очередь номеров вершин
```

```
#define N 100
int A[N][N];                     // матрица расстояний до соседей
int W[N];                        // матрица расстояний от начального
```

Рекурсия, стек, очередь

```
void main(){
int nc=0,ncmp=0,i;
for (i=0;i<N;i++) W[i]=-1;
create(0.05);
int n0=0;                      // Начальная вершина и расстояние до самой себя =0
W[n0]=0;
queue<int,100> Q;              // Очередь номеров вершин
Q.in(n0);                      // Поместить исходную в очередь
while(Q.size()!=0){            // Пока очередь не пуста
    int ni=Q.out();            // Извлечь номер очередной вершины
    if (W[ni]==-1) continue;   // ошибка - она еще не пройдена
    nc++;                      // подсчет трудоемкости алгоритма
    for (i=0;i<N;i++)          // проверка всех соседей
    if (A[ni][i]!=0){          // Это неотмеченный сосед
        if (W[i]==-1 || W[i]>W[ni]+A[ni][i])
            {                  //или сосед с более длинным путем
            printf("city %2d => %2d way %2d => %2d \n",ni,i,W[i],W[ni]+A[ni][i]);
            W[i]=W[ni]+A[ni][i]; // Уменьшить длину пути до него
            Q.in(i);           // Поместить в очередь (вторая волна)
            }
        ncmp++;                // подсчет трудоемкости алгоритма
        }
    }
for (i=0;i<N;i++) printf("%d ",W[i]);
printf("\nnc=%d ncmp=%d\n",nc,ncmp);
}
```