Рекурсивные алгоритмы
Реализация и оценка сложности

Определение рекурсивных процессов

Рекурсивным называется объект, частично состоящий из самого себя или определяемый с помощью себя.

Например:

1. Узел однонаправленного списка:

```
struct node
{       Titem info;
        node *next;// ссылка на тип, который создается
}
```

2. Определение натурального числа:

- 1 есть натуральное число
- Целое число, следующее за натуральным, есть натуральное число.

3. Алгоритм вычисления n!:

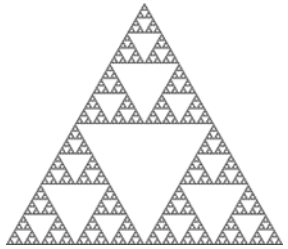
$$n! = \begin{cases} n * (n-1)! \text{ при } n > 1 \\ 1 \text{ при } n = 1 \end{cases}$$

4. Вычисление наибольшего общего делителя двух натуральных чисел по алгоритму Евклида: если a≠0 и b ≠0 то НОД(a,b) =НОД(b,r) где r – остаток от деления a на b.

НОД(18,4)=НОД(4,2)=НОД(2,0)

5. Метод Гаусса — Жордана для решения систем линейных алгебраических уравнений является рекурсивным.

6. Геометрические фракталы задаются в форме бесконечной рекурсии (например, треугольник Серпинского).



одно из определений **фрактала** - **это геометрическая** фигура, состоящая из частей и которая может быть поделена на части, каждая из которых будет представлять уменьшенную копию целого (по крайней мере, приблизительно).

7. Стандартный пример вычислимой рекурсивной функции, не являющейся примитивно рекурсивной — функция Аккермана: для неотрицательных целых чисел.

$$A(n,m) = \begin{cases} n+1 \text{ при } m = 0 \\ A(m-1,1) \text{ при } m > 0, n = 0 \\ A\big(m-1, A(m,n-1)\big) \text{ при } m > 0, n > 0 \end{cases}$$

Метод декомпозиции – Разделяй и властвуй

Многие полезные алгоритмы имеют рекурсивную структуру.

Такие алгоритмы часто разрабатываются с помощью метода декомпозиции, или разбиения:

- сложная задача разбивается на несколько более простых, которые подобны исходной задаче, но имеют меньший объем;
- далее эти вспомогательные задачи решаются рекурсивным методом,
- после чего полученные решения комбинируются для получения решения исходной задачи.

Парадигма, лежащая в основе метода декомпозиции "разделяй и властвуй", на каждом уровне рекурсии включает в себя три этапа.

1. Разделение задачи на несколько подзадач.
2. Рекурсивное решение этих подзадач, Когда объем подзадачи достаточно мал, выделенные подзадачи решаются непосредственно (получаем простейшее решение).
3. Комбинирование решения исходной задачи из решений вспомогательных задач.

Рекуррентное соотношение

Рекуррентное соотношение (recurrence) — это ***уравнение или неравенство, описывающее функцию с использованием ее самой***.

Рекуррентное соотношение – определяет рекурсивную природу алгоритма в зависимости от значений.

Рекурсивный алгоритм для вычисления значения реализуемой задачи, используя на каждом шаге ранее вычисленные значения для подсчета текущего значения, начиная с наименьшего, .

Примеры представления рекурсивного процесса через рекуррентное соотношение

1. Вычисление x^n сводится к умножению x*x*x*x*x*x*x раз, т. е. $x^n = x*x^{n-1}$

$$x^n = \begin{cases} 1, n = 0 \\ x \cdot x^{n-1}, n > 0 \end{cases}$$

2. Вычисление суммы s=1+2+3+4+5+6+7+8+9+10 можно представить

$$S = \sum_{i=1}^{10} i \qquad S(10) = \sum_{i=1}^{10} i \qquad S(11) = S(10) + 11$$

$$S(n) = \begin{cases} 1, n = 1 \\ n + s(n-1), n > 1 \end{cases}$$

Примеры представления рекурсивного процесса через рекуррентное соотношение (продолжение)

3. Целочисленное деление a на b (где b≠0)

$$div(a,b) = \begin{cases} 0 & при \quad a < b \quad // условие \\ div(a-b,b)+1 & иначе \quad // глубина \end{cases}$$

4. Числа Фибоначчи. Описание процесса вычисления n-ого числа Фибоначчи

$f_1=1,\ f_2=1,\ f_n=f_{n-1}+f_{n-2}$

$$f(n) = \begin{cases} 1 \text{ если } n < 3 \\ f(n-1)+f(n-2) \text{ иначе} \end{cases}$$

Рекурсивный алгоритм

Рекурсивный алгоритм – это алгоритм, в описании которого прямо или косвенно содержится обращение к самому себе. В рекурсивном алгоритме задача делится на несколько простых, каждая реализуется этим же алгоритмом. Из полученных решений формируется решение задачи.

Виды рекурсии по организации обращения к подзадачам: **прямая и косвенная.**

Косвенная рекурсия имеет место, если алгоритм А вызывает алгоритм В, и алгоритм В вновь вызывает алгоритм А.

Прямая рекурсия имеет место, если решение задачи сводится к разделению ее на меньшие подзадачи, выполняемые с помощью одного и того же алгоритма.

Процесс разбиения завершается, когда достигается простейшее возможное решение.

1.Рекурсия имеет место, если решение задачи сводится к разделению ее на меньшие подзадачи, выполняемые с помощью одного и того же алгоритма.

Процесс разбиения завершается, когда достигается простейшее возможное решение.

2. Рекурсивные алгоритмы действуют по принципу «разделяй и властвуй».

3. Во всех формулах (рекур. соотн.) мы писали условие выхода и вход в рекурсию.

Условие выхода из рекурсии - определяет завершение рекурсии и формирование конкретного значения вычислительного процесса.

В этой лекции рассматриваем прямую рекурсию.

Характеристики рекурсивного алгоритма

Условие выхода из рекурсии - определяет завершение рекурсии и формирование конкретного значения вычислительного процесса.

Шаг рекурсии – это выполнение одного действия.

Глубина рекурсивных вызовов – наибольшее одновременное количество рекурсивных вызовов функции, определяющее максимальное количество слоев рекурсивного стека, в котором осуществляется хранение отложенных вычислений.

Глубину рекурсии можно изобразить в виде графа – дерева рекурсии.

Дерево рекурсии – это **полное дерево**, т.е. на каждом уровне, кроме последнего, располагается максимально допустимое количество узлов,

Количество узлов на уровне зависит от степени дерева, так если степень дерева 2 (дерево бинарное), то количество узлов на i-ом уровне равно 2^i.

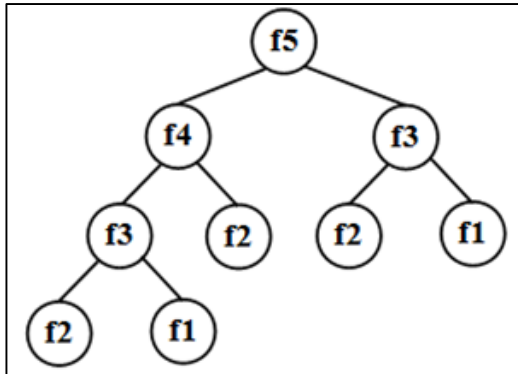
На последнем уровне размещаются вершины – листья, показывающие вызовы, приводящие к завершению вычислений. Уровни нумеруются с 0.

Дерево рекурсии

Для построения дерева рекурсии будем использовать следующие обозначения для
конкретного входного параметра x:
R(x) – общее число вершин дерева рекурсии,
$R_V(x)$ – объем рекурсии без листьев (только внутренние вершины),
$R_L(x)$ – количество листьев дерева рекурсии,
$H_R(x)$ – глубина рекурсии, определяется как количество незавершенных входов в рекурсию, т.е. в дереве рекурсии
- это количество внутренних вершин (т.е. без учета листьев).

Пример 1. Дерева рекурсии алгоритма вычисления 5-ого числа Фибоначчи и определение глубины рекурсии

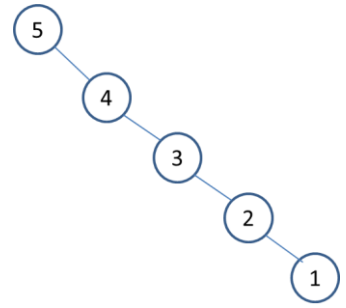
$$f(n) = \begin{cases} 1 \text{ если } n < 3 \\ f(n-1) + f(n-2) \text{ иначе} \end{cases}$$



Характеристиками рассматриваемого метода оценки алгоритма будут следующие величины.

	D = 5	D = n
R(D)	9	2 n-1
$R_V(D)$	4	n-1
$R_L(D)$	5	n
$H_R(D)$	***4***	***n-1***

Пример 1. Дерево рекурсивных вызовов для 5! При n=5 Глубина рекурсии 4, т.е. n-1 или 4 внутренних узла
4, т.е. n-1 или 4 внутренних узла



Основные правила при создании рекурсивных алгоритмов

1. Определить рекурсивную зависимость и оформить рекуррентное соотношение
2. Определить условие завершения рекурсии

$$P = W[S_i, p]$$

$$f(n) = \begin{cases} 1 & \text{при } n = 1 \\ n \cdot f(n-1) & \text{иначе} \end{cases}$$

Варианты схем организации рекурсивного алгоритма

Схема 1. Без указания явного выхода из рекурсии

P≡ if (B) then

W[Si,P] Шаг в рекурсию, выход не указан но выполниться по завершении входов

или

Схема 2. С указанием выхода из рекурсии при достижении простейшего решения

P≡ if (B) then

W[Si,P] Шаг в рекурсию

else

return нач. знач. Явный выход из рекурсии

или

Схема 3. Явный выход из рекурсии при достижении определенного значения параметра x (здесь рассматривалось уменьшение значения параметра в каждом вызове)

P (x) ≡ if x=нач.знач.

return x завершение рекурсии

else

return P(x-1)] Шаг в рекурсию

Как завершить рекурсию?

Наиболее надежный способ обеспечить завершение конечной рекурсии - это связать с подпрограммой P некоторое значение – **параметр значение** и рекурсивно вызвать P с параметром, измененным на некоторое значение и приводящим к начальному значению .

Пример: Вычисление n!. Параметр – число (n), которое при каждом вызове понижается, приводя к выполнению условия выхода из рекурсии.

$n! = \begin{cases} n * (n-1)! \text{ при } n > 1 \\ 1 \text{ при } n = 1 \end{cases}$	int fact (int n)
	{ if (n= =1)
	return 1;
	return n*fact(n-1);
	}

Примечание. Рекурсия по механизму выполнения похожа на цикл: цикл имеет условие завершения и рекурсивный алгоритм тоже. Если в цикле условие продолжения записано неверно, то цикл может стать бесконечным. То же может произойти и рекурсивным процессом.

Стек времени выполнения рекурсии

При каждом новом рекурсивном вызове, для функции создается новое множество локальных переменных и форменных параметров, их имена одинаковы, но они имеют различные значения.

Пример представления стека для вычисления 5! (тв – точка возврата после вызова функции)

	n		n		n		n		n
тв1	5	тв2	4	тв3	3	тв4	2	тв5	1

Идентификаторы всегда ссылаются на множество переменных, созданных в последнем вызове.

```
int fact (int n)
{       if (n= =1)
          return 1;
        return n*fact(n-1);
   }
```

Рекурсивные функции

Повторное выполнение алгоритма при новых значениях переменных, можно оформить только в виде подпрограммы, которую можно вызвать при новых значениях параметров.

В языках программирования это процедуры или функции.

Общий формат рекурсивной подпрограммы можно представить как композицию некоторых базовых операторов и операторов вызова подпрограммы.

$$P = W[S_i, P]$$

P – рекурсивная подпрограмма, W – композиция операторов тела подпрограммы, среди которых вызов подпрограммы.

Н. Вирт так описывает рекурсивные подпрограммы.

Подпрограмму (функцию) называют рекурсивной, если хотя бы одна конструкция содержит имя этой подпрограммы.

Рекурсивные функции (пример 1)

Задача: Дана последовательность из n целых чисел, вывести сначала отрицательные числа, а затем положительные.

Тест

Входные данные

n =8,

последовательность чисел:

1 2 -3 -4 5 6 -7 -8 (enter нажимаем только здесь)

Результат

-3 -4 -7 -8 6 5 2 1

$$print(n) = \begin{cases} \text{Выход, если } n = 0 \\ \text{Если } n > 0, \text{то}\{ \\ \text{ввод x} \\ \text{вывод } x \text{ и } print(n-1) \text{если x} < 0 \\ print(n-1) \text{ если x} \geq 0\} \end{cases}$$

```
void poslnumbers1(int n) {
   int x;
    if (n > 0) {
         std::cin >> x;

        if (x<0){
            std::cout << x << " ";
                poslnumbers1(n - 1);
                 }
         else {
              poslnumbers1(n - 1);
              std::cout << x << " ";
                 }
     }
}
```

Рекурсивные функции (продолжение примера)

Описание стека рекурсивных вызовов, рассмотренного примера

Дана последовательности из n=5 элементов: 1 -2 -3 4 -5.

Вызов 1. в стеке формируются следующие значения: точка возврата 1(в основную программу), n=5, x=1.

Так как n>0, тогда проверяется значение x. Так как x>0 то происходит новый вызов.

Вызов 2. в стеке формируются следующие значения: точка возврата 2 (на оператор, следующий за вызовом), n=4, x=-2. Так как n>0, тогда проверяется значение x. Так как x<0 то: осуществляется вывод x и вновь вход в рекурсию (происходит новый вызов).

Вызов 3. в стеке формируются следующие значения: точка возврата 2 (на оператор, следующий за вызовом), n=3, x=-3. Так как n>0, тогда проверяется значение x. Так как x<0 то: осуществляется вывод x и вновь вход в рекурсию (происходит новый вызов).

Вызов 4. в стеке формируются следующие значения: точка возврата 2 (на оператор, следующий за вызовом), n=2, x=4. Так как n>0, тогда проверяется значение x. Так как x>0 то: осуществляется вновь вход в рекурсию (происходит новый вызов).

Вызов 5. в стеке формируются следующие значения: точка возврата 2 (на оператор, следующий за вызовом), n=1, x=-2. Так как n>0, тогда проверяется значение x.

Так как x<0 то: осуществляется вывод x и вновь вход в рекурсию при n=0(происходит новый вызов).

Этот вызов завершает рекурсию: т.е. освобождается стек, в котором сохранились значения переменных и последовательно управление передается в точку возврата 2, и когда освобождается стек от параметров первого вызова, то выбирается точка возврата в основную программу.

Рекурсивные функции (пример 2)

Задача. Разработать рекурсивную функцию, которая выводит введенную последовательность символов, заканчивающуюся точкой, в обратном порядке.

Для хранения последовательности массив не использовать.

1) Воспользуемся стеком функции и сохраним в ней всю введенную последовательность (втолкнем в стек).

2) Когда вся последовательность будет введена, завершим рекурсию, стек будет освобождаться автоматически и в этот момент применим операцию вывода элементов.

Описание процесса

print()=$\begin{cases} print(\quad)\ \text{x!} = '.' \\ \text{вывод x при x} = '.' \end{cases}$,

Реализация алгоритма

Тест: *Входные данные текст*:
asdfghjkl.

Результат: lkjhgfdsa

```
void poslnumbers() {
    char x;
   std::cin >> x;
   if (x !='.') {
     poslnumbers();
     std::cout << x << " ";
}}
```

Рекурсивные функции (описание выполнения функции примера 2)

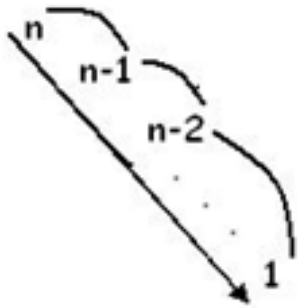
Пусть введена последовательность: абвгде.

Серия вызовов приведет к тому, что в стеке будут сохранены значения абвгде (в такой последовательности как вводились).

Как только будет прочитана из последовательности точка, вызовы прекратятся и стек будет освобождаться, от вершины, к первому элементу и будет работать оператор, не выполненный в каждом вызове процедуры оператор - ***cout<<x***, так как до него очередь не доходила, но в стеке формировалась точка возврата при каждом вызове (вернуть управление на следующий за вызовом оператор).

Виды рекурсий и графы рекурсивных вызовов

1. **Линейная рекурсия**: каждый вызов порождает ровно один новый вызов. Пример графа вызовов n!

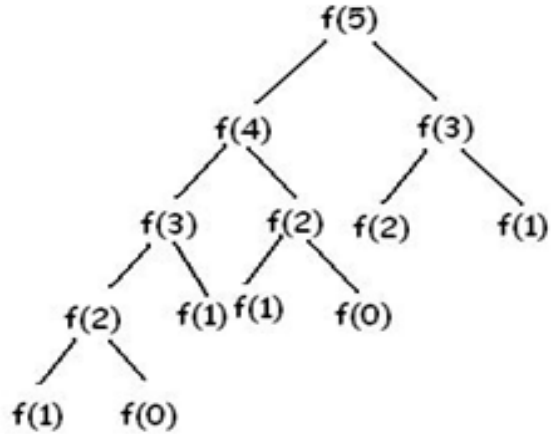


Алгоритм бинарного поиска

```
int binfinde(int l,int r, int *x,int key){
        if (l>r) return -1;
        int i=(l+r)/2;
        if(x[i]==key) return i;
        if(x[i]>key)
                binfinde(l,r-1,x,key);
        else
                binfinde(l+1,r,x,key);
}
```

2. **Каскадная рекурсия**: каждый вызов порождает несколько новых вызовов. Числа Фибоначчи

$$f(n)=\begin{cases}1 & при\ \ n=0\ \ или\ \ n=1\\ f(n-2)+f(n-1) & при\ \ n>0\end{cases}$$



Виды рекурсий и графы рекурсивных вызовов (продолжение)

3. **Повторная рекурсия**: *если в линейно-рекурсивном определении функции, во всех рекурсивных вызовах на ветвях различия случаев рекурсивный вызов является самым внешним.*

Пример, деления a на b нацело при b>0.

```
int Div1(int a, int b) {
  if (a >= b)
      return 1+Div1(a - b, b);
}
```

4. **Удаленная рекурсия**

Если в определении рекурсивной функции вызов функции в списке фактических параметров содержит вызов самой функции.

Например, сумма цифр десятичного натурального числа.

```
int sum(int n) {
    if (n<9)
        return n;
    else
        return sum(sum(n / 10)) + n % 10;
    }
```

Ханойская башня

Это классический пример построения рекурсивного алгоритма.

Одна из древних легенд гласит: «В непроходимых джунглях недалеко от города Ханоя есть храм бога Брамы. В нем находится бронзовая плита с тремя алмазными стержнями. На один из стержней бог при сотворении мира нанизал 64 диска разных диаметров из чистого золота. Наибольший диск лежит на бронзовой плите, а остальные образуют пирамиду, сужающуюся кверху. Это башня Брамы. Работая день и ночь, жрецы храма переносят диски с одного стержня на другой, следуя законам Брамы:

1) диски можно перемещать с одного стержня на другой только по одному;

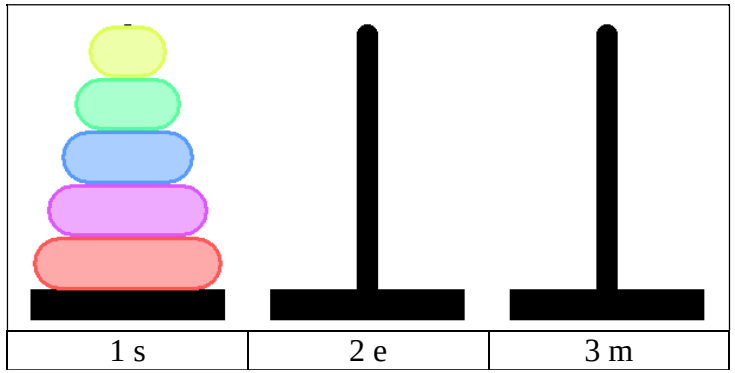
2) нельзя класть больший диск на меньший;

3) нельзя откладывать диски в сторону, при переносе дисков с одного стержня на другой можно использовать промежуточный третий стержень, на котором диски должны находиться тоже только в виде пирамиды, сужающейся кверху.

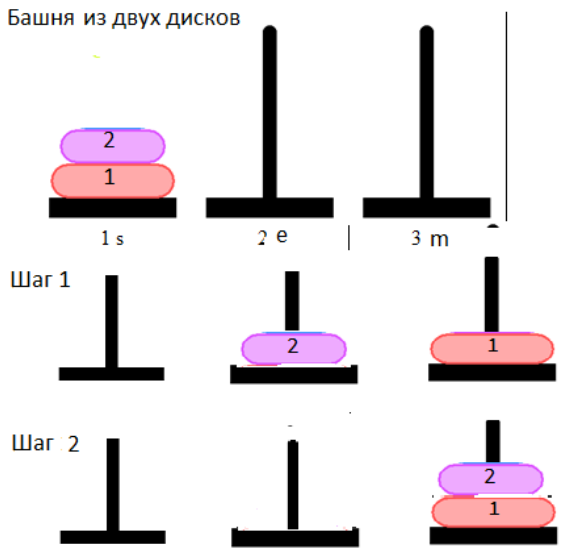
Когда все 64 диска будут перенесены с одного стержня на другой, наступит конец света».

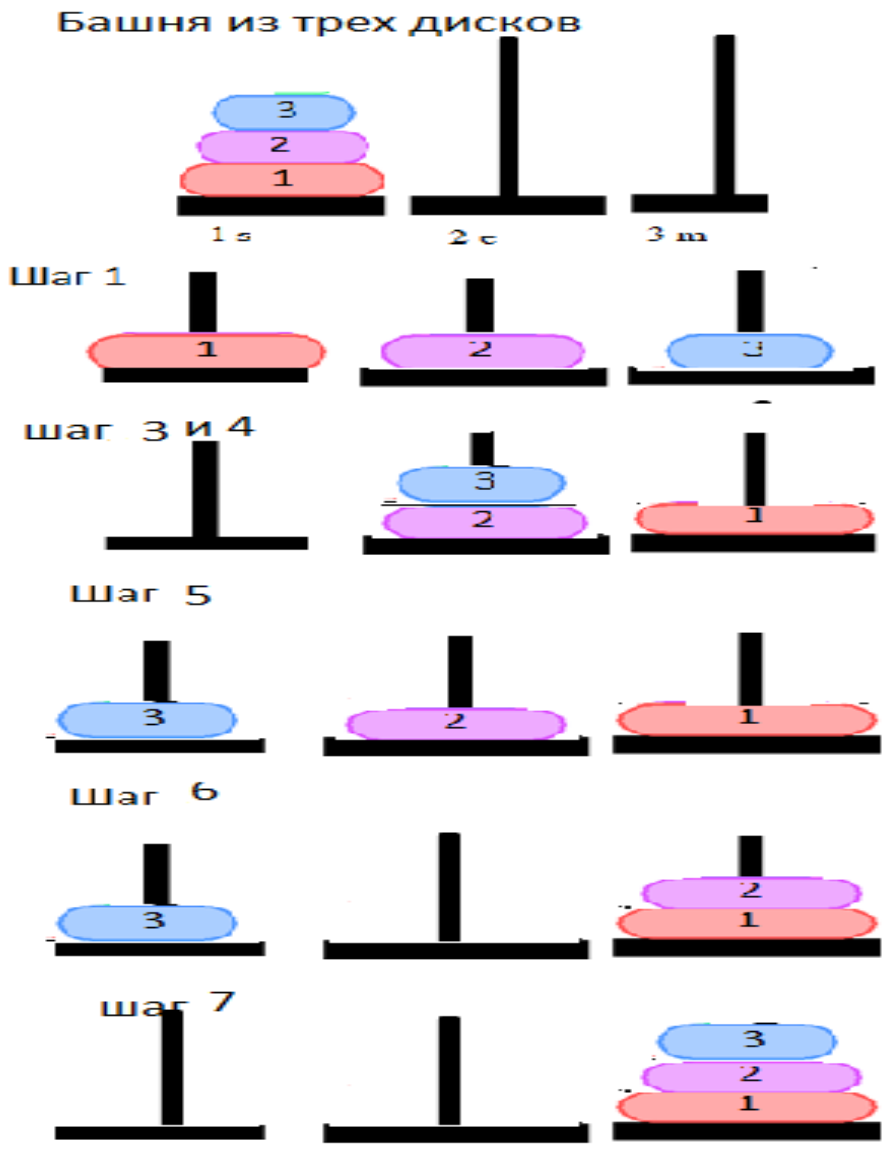
Задача ханойской башни

переместить n дисков со стержня 1 на стержень 3, используя промежуточный стержень 2 и соблюдая законы Брамы.



Алгоритм задачи о ханойской башне





Для башни, состоящей из n колец, мы используем алгоритм для чуть более простой ситуации:

перенос башни, состоящей из n - 1 кольца.

В свою очередь, в алгоритме для башни из n - 1 кольца используется этот же алгоритм для n - 2 колец и т. д.

Описание выполнения алгоритма задачи о ханойской башне

1. Башня из двух дисков переносится за три хода: 1 → 2, 1 → 3, 2 → 3.

2. Для переноса башни из трех дисков потребуется уже семь ходов: 1 → 3, 1 → 2, 3 → 2, 1 → 3, 2 → 1, 2 → 3, 1 → 3. *Обратите внимание, за первые три хода мы переносим башню из двух верхних дисков на второй промежуточный стержень*.

Затем переносим самый большой диск с первого стержня на третий и еще раз проделываем хорошо знакомую нам операцию: переносим башню из двух дисков на третий диск.

3. Следовательно, чтобы перенести башню из четырех дисков с первого стержня на третий, необходимо действовать по плану:

1) перенести башню из трех верхних дисков с первого стержня на второй (7 ходов);
2) самый большой диск перенести с первого стержня на третий (1 ход);
3) перенести башню из трех дисков со второго стержня на третий (7 ходов).

Всего на перенос потребуется 15 ходов. Рассуждая аналогичным образом, сосчитаем число ходов, необходимых для переноса башни из пяти дисков: 15 + 1 + 15 = 2 • 15 + 1 = 31.

Для башни из 6 дисков получаем: 2 • 31 + 1 = 63 и т. д.

Рассмотренный нами алгоритм решения задачи «Ханойская башня» обладает одним удивительным свойством: в ходе его выполнения для башни, состоящей из n колец, мы используем алгоритм для чуть более простой ситуации — переноса башни, состоящей из n - 1 кольца. В свою очередь, в алгоритме для башни из n - 1 кольца используется этот же алгоритм для n - 2 колец и т. д.

Как видим правила рекурсии налицо

Реализация алгоритма задачи о ханойской башне

```
void hanoi(int n, int s, int m, int e)
{   if(n==1)
    {
       cout<<"disk"<<s<<"na"<<e<<endl;
    }
   else
    {
          hanoi(n-1,s,e,m);
          cout<<"disk"<<s<<"na"<<e<<endl;
          hanoi(n-1,e,s,m);
          }
}
int main()
{    hanoi(4,1,2,3);
     return 0;
     }
```

С диска s перемещаем на e n-1 диск и один самый большой на m. s – пустой.

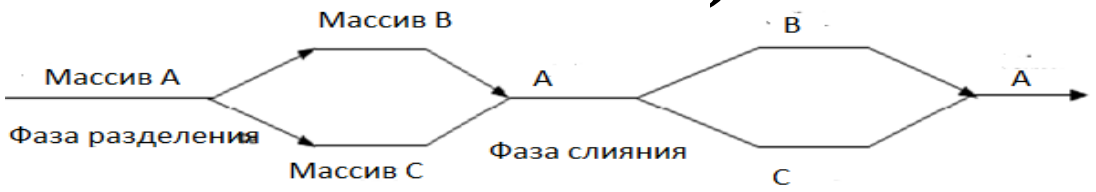
С диска e перемещаем n-1 диск на s и самый большой на m

Быстрые сортировки, реализованные рекурсивно

Сортировка прямого слияния
Сортировка естественного слияния
Сортировка многофазного слияния слияния
Быстрая сортировка (Хоара)

Сортировки методом слияния

Алгоритм *сортировки прямого слиянием* (merge sort)



Исходный массив A: 8 7 6 5 4 3 2 1 сортировать по возрастанию

Не рекурсивное решение

Разбиение на два подмассива из n/2 элементов:
B(8 7 6 5) и C(4 3 2 1)
Сливаем в A упорядоченные пары, выбирая по одному элементу из B и C
A(4 8 3 7 | 2 6 1 5)
Разливаем
B(4 8 3 7) C(2 6 1 5)
Сливаем в A упорядоченные четверки, выбирая упорядоченные пары из B и C
A(2 4 6 8| 1 3 5 7)
Разливаем
B(2 4 6 8)
C(1 3 5 7)
Сливаем в A упорядоченные восьмерки выбирая упорядоченные восьмерки
A(1 3 4 5 6 7 8)

Рекурсивное решение

Разбиения: A: 8 7 6 5 4 3 2 1
B: (8 7 6 5) (8 7) (6 5) (8) (7) (6) (5)
C: (4 3 2 1) (4 3) (2 1) (4)(3)(2)(1)
Сливаем в A упорядоченные пары, выбирая по одному элементу из B и C
A:(1 5) (2 6) (3 7) (4 8) – в упорядоченные пары
B: (1 5) (2 6)
C (3 7) (4 8)
A: (1 3 5 7) (2 4 6 8)- упорядоченные четверки
B : (1 3 5 7)
C : (2 4 6 8)-
A: 1 2 3 4 5 6 7 8 – упорядоченные восьмерки

Сортировка прямого слиянием

```
Merge_Sort(A,p,r )
1   if p < r
2       then q ← ⌊ (p + r)/2 ⌋
3            Merge_Sort(A,p, q)
4            Merge_Sort(A,q+1, r)
5            Merge (A,p, q, r)
6    fi
```

Описание алгоритма

1. Сначала запускаются вызовы сортировки подзадач (операторы 3 и 4) длины $\frac{n}{2^k}$ пока длина подзадачи не будет равна 1.
2. Затем в ходе работы алгоритма происходит попарное объединение сначала одноэлементных последовательностей в отсортированные последовательности длины 2, затем – попарное объединение двухэлементных последовательностей в отсортированные последовательности длины 4 и т.д., пока не будут получены две последовательности, состоящие из n/2 элементов, которые объединяются в конечную отсортированную последовательность длины n.

Примечание. Выражение ⌊ x⌋ обозначает наибольшее целое число, которое меньше или равно x.

Алгоритм прямого слияния на основе метода декомпозиции

Разделение. Сортируемая последовательность, состоящая из n элементов, разбивается на две меньшие последовательности, каждая из которых содержит n/2 элементов.

Рекурсивное решение. Сортировка обеих вспомогательных последовательностей методом слияния.

Комбинирование. Слияние двух отсортированных последовательностей для получения окончательного результата.

Рекурсия достигает своего нижнего предела, когда длина сортируемой последовательности становится равной 1.

Основная операция, которая производится в процессе сортировки по методу слияний, - это объединение двух отсортированных последовательностей в ходе комбинирования.

Это делается с помощью вспомогательной процедуры **Merge**(A,p,q,r), где A - массив, а p,q,r - индексы, нумерующие элементы массива, такие, что $p \le q < r$. В этой процедуре предполагается, что элементы подмассивов A [p..q] и A [q + l..r] упорядочены.

Она ***сливает*** эти два подмассива в один отсортированный, элементы которого заменяют текущие элементы подмассива A[p..r].

Алгоритм Merge

```
Merge(A,p,q,r)
1   n1 ← q - p + 1
2   n2 ← r - q
3   Создаем массивы L[1. .n1 +1] и R[1.. n2 +1]
4    for i ← 1 to n1
5         do L[i] ← A[p+i-1] od
6    for j ← 1 to n2
7         do R[j] ← A[q+j] od
8    L[n1 +1] ← ∞
9    R[n2 +1] ← ∞
10    i ← 1
11    j ← 1
12  for k ← p to r
13  do  if  L[i] ≤ R[j]
14            then A[k] ← L[i]
15                    i ← i + 1
16             else   A[k] ← R[j]
17   j ← j + l  fi   od
```

Чтобы показать, что время работы процедуры **Merge** равно $\theta(n)$,

где $n = r - p + 1$,

каждая из строк 1-3 и 8-11 выполняется в течение фиксированного времени; длительность циклов **for** в строках 4-7 равна $\theta(n_1 + n_2) = \theta(n)$,

а в цикле **for** в строках 12-17 выполняется n итераций, на каждую из которых затрачивается фиксированное время.

Недостатки сортировки прямого слияния

При выполнении слияния в сортировке прямого слияния на каждом проходе ***длина полученной подпоследовательности увеличивалась вдвое,*** т.е на к-ом проходе длина подпоследовательности $<=2^k$.

Сортировка прямого слияния ***не учитывает упорядоченность сливаемых последовательностей,*** хотя при слиянии двух упорядоченных подпоследовательностей длины n и m можно получить упорядоченную подпоследовательность длины n+m.

Асимптотическая сложность сортировки прямого слияния

Количество сравнений C=**nlog(n),**

Количество перемещений: M еще меньше.

Порядок роста для всех случаев состояния массива (наилучший, наихудший, средний) **O(n*log(n))**

Сортировка естественного слияния

Сортировка *естественного* слияния, рассматривает две сливаемые подпоследовательности как упорядоченные.

Упорядоченные подпоследовательности принято называть ***сериями.***

Пусть исходный массив разделен на два массив, каждый из которых содержит по n – серий (один может содержать n-1 серию). Тогда при слиянии этих файлов будет получен массив из n серий.

При каждом проходе число серий **уменьшается вдвое**, и **общее число обменов в худшем случае равно n $\log_2 n$,** а **в среднем меньше.**

Процесс сортировки заканчивается, если при очередном проходе в файл будет перелита только одна серия.

Пример выполнения сортировки естественного слияния

Пусть есть массив A, содержащий записи с ключами:

17 31 5 59 13 41 43 67 11 23 29 47 3 7 71 2 19 57 37 61

Выделим серии, завершая запятой, чтобы было нагляднее:

17 31’ 5 59’ 13 41 43 67’ 11 23 29 47’ 3 7 71’ 2 19 57’ 37 61

Получилось 7 серий.

Разделим массив на два массива B и C, переписывая в них поочередно по серии:

B: 17 31’ 13 41 43 67’ 3 7 71’ 37 61

C: 5 59’ 11 23 29 47’ 2 19 57

Сольем массивы B и C в массив A, сливая серии в упорядоченные серии

A:5 17 31 59’ 11 13 23 29 41 43 47 67’2 3 7 19 57 71’37 61

Опять разольем в B и C поочередно переписывая серии

B: 5 17 31 59’2 3 7 19 57 71

C: 11 13 23 29 41 43 47 67’37 61

Сливаем в массив A по сериям

A: 5 11 13 17 23 29 31 41 43 47 59 67’2 3 7 19 37 57 61 71

Разливаем

B: 5 11 13 17 23 29 31 41 43 47 59 67

C: 2 3 7 19 37 57 61 71

Сливаем

2 3 5 7 11 13 17 19 29 31 37 41 43 47 57 59 61 67 71

Т.е. понадобилось три прохода для такой сортировки.

ДЗ: Составить схему выполнения алгоритма естественного слияния

Быстрая сортировка

Быстрая сортировка (общее описание)

Быстрая сортировка, подобно сортировке слиянием, основана на парадигме "разделяй и властвуй". Процесс сортировки массива

A [$p..r$], состоит из трех этапов.

Разделение. Массив A [$p..r$] разбивается (путем переупорядочения его элементов) на два (возможно, пустых) подмассива ***A*** [$p..q$ - 1] и ***A*** [q + 1..r].

Каждый элемент подмассива ***A*** [$p..q$ - 1] не превышает элемент ***A***[q], а каждый элемент подмассива ***A*** [q + 1..r] не меньше элемента ***A***[q]**.** Индекс q вычисляется в ходе процедуры разбиения.

Рекурсивный шаг. Подмассивы ***A*** [$p..q$ - 1] и ***A*** [q + 1..r] сортируются путем рекурсивного вызова процедуры быстрой сортировки.

Комбинирование. Поскольку подмассивы сортируются на месте, для их объединения не нужны никакие действия: весь массив A [$p..r$] оказывается отсортирован.

Алгоритм быстрой сортировки:

Quicksort (A, p, r)

if $p < r$

then $q \leftarrow$ Partition (A, p, r) // разбиение на подмассивы относительно A[q]

Quicksort (A, p, q - 1)

Quicksort (A, q + 1, r)

Вызов процедуры для сортировки всего массива A должен иметь вид

Quicksort (A, 1, *length*[A]).

Алгоритм быстрой сортировки (общий случай)

Пример. Пусть задан следующий массив чисел: 2, 8, 7, 1, 3, 5, 6, 4.
Тогда работа алгоритма может быть проиллюстрирована следующим образом:
Пусть $q=n$, т.е. $x= 4$ (опорный элемент).
После 1-го прохождения цикла:
2, 8, 7, 1, 3, 5, 6, 4 (элемент со значением 2 "переставлен сам с собой" и помещен в ту часть, элементы которой не превышают x).
После 2-го и 3-го прохождения цикла:
2, 8, 7, 1, 3, 5, 6, 4 (элементы со значениями 8 и 7 добавлены во вторую часть массива (которая до этого момента была пустой).
После 4-го прохождения цикла:
2, 1, 7, 8, 3, 5, 6, 4 (элементы 1 и 8 поменялись местами, в результате чего количество элементов в первой части возросло).
После 5-го прохождения цикла:
2, 1, 3, 8, 7, 5, 6, 4 (элементы 3 и 7 поменялись местами, в результате чего количество элементов в первой части возросло).
После 6-го и 7-го прохождения цикла:
2, 1, 3, 8, 7, 5, 6, 4 (элементы со значениями 5 и 6 добавлены во вторую часть массива).
После 8-го прохождения цикла:
2, 1, 3, 4, 7, 5, 6, 8 (опорный элемент меняется местами с тем, который находится на границе раздела двух областей).
Время обработки процедурой Partition подмассива $A\ [p..r]$ равно $\theta(n)$, где $n = r - p + 1$.

Алгоритм шага Разбиение массива

Ключевой частью рассматриваемого алгоритма сортировки является процедура Partition, изменяющая порядок элементов подмассива ***A*** [*p..r*] без привлечения дополнительной памяти:

Partition (*A, p, r*)

1 $x \leftarrow A[r]$

2 $i \leftarrow p - 1$

3 **for** $j \leftarrow p$ to $r - 1$

4 **do if** $A[j] \leq x$

5 **then** $i \leftarrow i + 1$

6 Обменять $A[i] \leftrightarrow A[j]$ **fi od**

7 Обменять $A[i + 1] \leftrightarrow A[r]$

8 return $i + 1$

Эта процедура всегда выбирает элемент $x = A[r]$ в качестве ***опорного*** элемента (в принципе можно было бы выбрать первый элемент и начинать цикл со второго элемента массива). Разбиение подмассива A [p..r] будет выполняться относительно этого элемента. В начале выполнения процедуры массив разделяется на четыре области (они могут быть пустыми): все элементы подмассива $A[p..i]$ меньше либо равны x, все элементы подмассива $A[i+1..j-1]$ больше x и $A[r] = x$, а все элементы подмассива $A[j..r-1]$ могут иметь любые значения (см.пример).

Оценка сложности рекурсивных алгоритмов

Рекурсивные алгоритмы относятся к классу алгоритмов с высокой ресурсоемкостью, так как при большом количестве самовызовов рекурсивных функций происходит быстрое **заполнение стековой области**.

Кроме того, **организация хранения и закрытия очередного слоя рекурсивного стека являются дополнительными операциями, требующими временных затрат.**

На трудоемкость рекурсивных алгоритмов влияет и количество передаваемых функцией параметров.

Методика анализа алгоритмов, основанных на принципе «разделяй и властвуй»

Если алгоритм рекурсивно обращается к самому себе, время его работы часто описывается с помощью *рекуррентного уравнения*, или *рекуррентного соотношения*, **в котором полное время, требуемое для решения всей задачи с объемом ввода n, выражается через время решения вспомогательных подзадач.**

Получение рекуррентного соотношения для времени работы алгоритма, основанного на принципе "разделяй и властвуй", базируется на трех этапах, соответствующих парадигме этого принципа.

Рекуррентное соотношение описывает время работы алгоритма, в котором задача размером **n** разбивается на **a** вспомогательных задач, размером **n/b** каждая, где **a** и **b** — положительные константы.

Полученные в результате разбиения **a** подзадач решаются рекурсивным методом, причем время их решения равно T(**n/b**).

Время, требуемое для разбиения задачи и объединения результатов, полученных при решении вспомогательных задач, описывается функцией f (**n**).

Например, в рекуррентном соотношении, возникающем при анализе процедуры QSORT,

a = 2, **b** = 2, a f(**n**) = **θ** (**n**).

Хотя, в общем случае, число **n/b** может не быть целым, замена каждого из **a** слагаемых T(**n/b**) выражением T(⌊ **n/b** ⌋) или T(⌈ **n/b**⌉) не влияет на асимптотическое поведение решения.

Поэтому обычно при составлении рекуррентных соотношений подобного вида, полученных методом "разделяй и властвуй", обычно мы будем игнорировать функции "пол" и "потолок", с помощью которых аргумент преобразуется к целому числу.

Рекуррентные уравнения (соотношения)и их решение

Если алгоритм рекурсивно вызывает сам себя, время его работы часто можно описать с помощью рекуррентного соотношения.

Рекуррентное соотношение (recurrence) — это уравнение или неравенство, описывающее функцию с использованием ее самой, но только с меньшими аргументами.

Решение рекуррентного соотношения: *функция определяющая порядок роста времени по обозначению O, Θ.*

Например, время T(n) работы процедуры MERGESORT в наихудшем случае описывается с помощью следующего соотношения:

$$T(n) = \begin{cases} \Theta(1) & \text{при n=1} \\ 2T(n/2) + \Theta(n) & \text{при n>1} \end{cases}$$

Рекуррентное соотношение **T(n)=2T(n/2)+Θ(n)** где T(n)= Θ(n) означает c*n и отведено на выполнение алгоритма слияния подмассивов, 2T(n/2) время выполнения двух вызовов.

Решением этого соотношения является функция ***T(n)* = θ(*n* ln *n*).**

Рекуррентные соотношения (создание рекуррентного соотношения)

Рекурсивный алгоритм может делить задачу на подзадачи разного размера, например, разбивая на части, представляющие собой 2/3 и 1/3 исходной задачи.

Если при этом разделение и комбинирование выполняются за линейное время, время работы такого алгоритма определяется рекуррентным соотношением

$T(n) = T(2n/3) + T(n/3) + \Theta(n)$.

Примечание. Подзадачи не обязаны быть ограниченными некоторыми постоянными долями размера исходной задачи.

Например, **рекурсивная версия линейного поиска** создает только одну подзадачу, содержащую только на один элемент меньше, чем исходная задача.

Каждый рекурсивный вызов требует константного времени плюс время на рекурсивный вызов, в свою очередь осуществляемый им, что приводит к рекуррентному соотношению

$T(n) = T(n - 1) + \Theta(1)$.

Создание рекуррентного соотношения

Рекуррентное соотношение, описывающее время работы процедуры M e r g e S o r t **в наихудшем случае**, в действительности равно

$$T(n) = \begin{cases} \Theta(1), \text{если } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n), \text{если } n > 1 \end{cases} \quad (3)$$

Для определения времени работы алгоритмов, **для достаточно малых n,** используется соотношение $T(n) = \Theta(1)$.

Поэтому для удобства граничные условия рекуррентных соотношений, как правило, опускаются и предполагается, что для малых n время работы алгоритма T (n) является константным $\Theta(1)$. .

Например, рекуррентное уравнение (3) обычно записывается так

$$T(n) = 2T(n/2) + \Theta(n)$$

Причина состоит в том, что, хотя изменение значения T(1) и приводит к изменению решения рекуррентного соотношения, это решение обычно изменяется не более чем на постоянный множитель, так что порядок роста остается неизменным.

Создание рекуррентного соотношения QSort

void qs(int* s_arr, int first, int last) {	
int i = first, j = last, x = s_arr[(first + last) / 2];	Θ(1)

do {
while (s_arr[i] < x) i++;
while (s_arr[j] > x) j--;
if(i <= j) {
if (s_arr[i] > s_arr[j]) swap(&s_arr[i], &s_arr[j]);
i++;
j--;
}
} while (i <= j);
	Θ(n)

if (i < last) qs(s_arr, i, last);	T(n/2)
if (first < j) qs(s_arr, first, j);	T(n/2)

Полним подсчет времени выполнения алгоритма, используя быстрый подход к определению сложности алгоритма на основе асимптотических обозначений, получим

$T(n)= Θ(1)+ Θ(n)+ T(n/2)+ T(n/2)=2T(n/2)+ Θ(n)$; при $n>1$

$$T(n) = \begin{cases} \Theta(1) \text{ при } = 1 \\ 2T(n/2) + \Theta(n); \text{ при } n > 1 \end{cases}$$

Методы решения рекуррентных уравнений

Рассмотрим три метода получения асимптотических θ и O-оценок решения рекуррентных уравнений.

Метод подстановок (substitution method). Требует догадки, какой вид имеют граничные функции, а затем с помощью метода математической индукции доказать, что догадка правильная.

Метод деревьев рекурсии (recursion-tree method). Рекуррентное соотношение преобразуется в дерево, узлы которого представляют время выполнения каждого уровня рекурсии; затем для решения соотношения используется метод оценки сумм.

О***сновной метод*** (master method). Граничные оценки решений рекуррентных соотношений представляются в таком виде:

$T(n) = aT(n/b) + f(n)$

где $a \geq 1$, $b > 1$, а функция $f(n)$ — это заданная функция; для применения этого метода необходимо запомнить три различных случая, после чего определение асимптотических границ во многих простых рекуррентных соотношениях становится очень простым.

Решение рекуррентных соотношений. Метод подстановки

Метод подстановки, применяющийся для решения рекуррентных уравнений, состоит из двух этапов:

1. делается догадка о виде решения;
2. с помощью метода математической индукции определяются константы и доказывается, что решение правильное.

Происхождение этого названия объясняется тем, что предполагаемое решение подставляется в рекуррентное уравнение. Этот метод применим только в тех случаях, когда легко сделать догадку о виде решения.

Метод подстановки можно применять для определения либо верхней, либо нижней границ рекуррентного соотношения.

Решение рекуррентных соотношений. Метод подстановки (пример1)

Пример 1. Определим верхнюю границу рекуррентного соотношения

$T(n) = 2T(\lfloor n/2 \rfloor) + n$,

Мы ***предполагаем***, что решение имеет вид $T(n) =$ **$O(n \log n)$.**

Докажем, что при подходящем выборе константы $c > 0$ выполняется неравенство $\mathbf{T(n) \le cn \lg n.}$

1. Предположим справедливость этого неравенства для величины $\lfloor n/2 \rfloor$, т.е. что выполняется соотношение

$$T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor).$$

2. После подстановки данного выражения в рекуррентное соотношение получаем:

$T(n) \le 2(c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)) + n \le cn \lg (n/2) + n = cn \lg n - cn \lg 2 + n = cn \lg n - cn + n \le c n \lg n$,

это неравенство выполняется при $c \ge 1$.

Для завершения доказательства требуется док-во справедливости этого утверждения для указанных граничных условий: найти C и n0. Получено что $c \ge 1$.

В рассматриваем примере можно всегда указать для большого C такое n_0, для которого это условие выполняется.

Тогда для всех $n \ge n_0$, это неравенство справедливо и предполагаемое **$O(n \log n)$** является решением рекуррентного уравнения

Метод подстановки (пример 2)

Пример 2. Покажем, что $O(\log_2 n)$ является решением рекуррентного соотношения

$$T(n) = T(\lceil n/2 \rceil)+1$$

Решение

Т.к. дано предполагаемое решение **$O(\log_2 n)$, т.е. ограничение сверху** и что $\log_2 n$ является решением соотношения. Надо доказать, что справедливо

$T(n) <= C^* \log_2 n$ для $C>0$ и $n0>n$ (XX)

Предположим, что оно справедливо для всех положительных $m < n$, в частности для $m = [n/2]$, тогда получим

$$T(\lceil n/2 \rceil) \leq C\ log2(\lceil n/2 \rceil \quad (X)$$

Подставим (X) в рекуррентное соотношение (XX)

$T(n) \leq T(\lceil n/2 \rceil)+1= C\ log(\lceil n/2 \rceil) +1=C \log_2(n) –C \log_2 2+1= C \log_2(n) –C+1$ Отсюда $C>=1$

Проверим решение $O(\log_2 n)$, для граничных условий

$n=1$, т.к. $T(n) <= C \log_2 n$ имеем $C \log_2 1=C^*0$, т.е. $C>=1$ и $n>1$ т.е. функция $\log_2 n$ является решением рекуррентного соотношения.

Решения рекуррентного соотношения методом подстановки

Если мы не знаем вида оценочной функции или не уверены в том, что выбранная оценочная функция будет наилучшей границей для T(n), то можно применить подход, который в принципе всегда позволяет получить точное решение для T(n), хотя на практике он часто приводит к решению в виде достаточно сложных сумм, от которых не всегда удается освободиться.

Рассмотрим этот подход на соотношении

$$T(n) = \begin{cases} c_1 & \text{если } n = 1 \\ 2T\left(\frac{n}{2}\right) + c_2 n & \text{если } n > 1 \end{cases} \quad (1)$$

1) Заменив n на n/2

$T(n/2)<=2T(n/4)+c_2n/2$, подставим это выражение в (1) вместо T(n/2), получим

$T(n)<=2(2T(n/4)+c_2n/2)+c_2n=4T(n/4)+2c_2n$ (3)

2) Аналогично заменяем в (1) n на n/4 получаем оценку для T(n/4):

$T(n/4)<=2T(n/8)+c_2n/4$ подставляем его в (3), получаем

$T(n)<=8T(n/8)+3c_2n$

3) Видна закономерность выводов метода подстановки.

Используем индукцию по i : *для любого i можно получить соотношение*

$T(n)<=2^iT(n/2^i)+ic_2n$ (4)

4) Предположим, что n является степенью 2, например $n=2^k$. Тогда при i=k, получаем

$T(n)<=2^kT(1)+kc_2n$, (5)

так как $n=2^k$, то k=log(n), а так как $T(1)<=c_1$ согласно (1), следует

$T(n)<=2^kT(1)+kc_2n=n\ c_1 + \log(n)\ c_2n$ это неравенство показывает верхнюю границу для T(n), а это и доказывает, что T(n) имеет порядок роста не более O(nlogn)

Решения рекуррентного соотношения. Метод деревьев рекурсии

В ***дереве рекурсии*** (recursion tree) каждый узел представляет время, необходимое для выполнения отдельно взятой подзадачи, которая решается при одном из многочисленных рекурсивных вызовов функций.

Далее значения времени работы отдельных этапов суммируются в пределах каждого уровня, а затем - по всем уровням дерева, в результате чего получаем полное время работы алгоритма. Деревья рекурсии находят практическое применение при рассмотрении рекуррентных соотношений, описывающих время работы алгоритмов, построенных по принципу "разделяй и властвуй".

Деревья рекурсии лучше всего подходят для того, чтобы помочь сделать догадку о виде решения, которая затем проверяется методом подстановок. При этом в догадке часто допускается наличие небольших неточностей, поскольку впоследствии она все равно проверяется. Если же построение дерева рекурсии и суммирование времени работы по всем его составляющим производится достаточно тщательно, то само дерево рекурсии может стать средством доказательства корректности решения.

Метод дерева рекурсии

Пример. Решение рекуррентного уравнения методом дерева рекурсии

Построим дерево рекурсии для определения решения уравнения

T(n) = 2T(⌊ n/2 ⌋) + cn где cn время на вызов функции

Пусть глубина рекурсии - k.

Значение каждого из узлов на уровне k: $cn/2^k$.

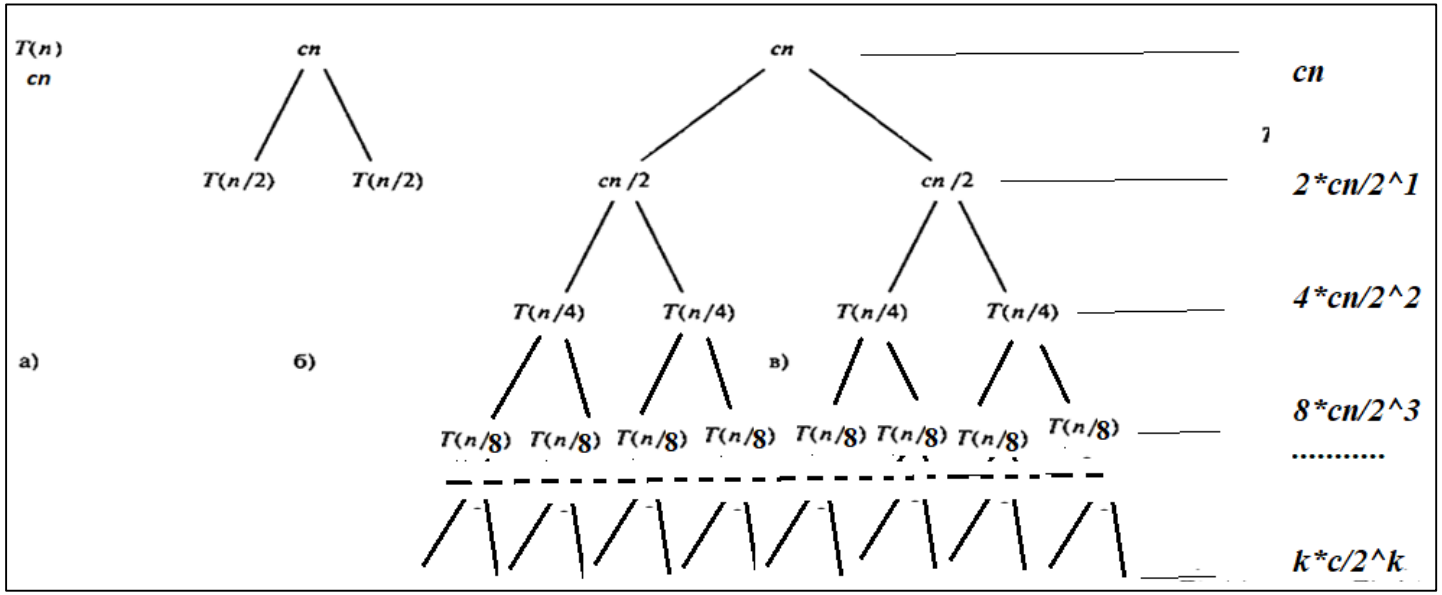
Время выполнения на каждом из уровней: c*n.

Общее время c*n*k.

Найдем k: $n/2^k=1 \rightarrow 2^k=n \rightarrow k=\log n$.

Общее времяT(n)= c*n*k=c*n*logn.

T(n)=O(nlogn)



Решение рекуррентных уравнений. Основной метод

Основной метод является своего рода "сборником рецептов", по которым строятся решения рекуррентных соотношений вида: T (**n**) = **a**T (**n/b**) + f (**n**), где **a**≥1и **b**>1 - константы, а f(**n**) — асимптотически положительная функция.

Для использования этого метода необходимо различать три случая, которые определяет теорема. Основной метод базируется на теореме.

Теорема. (Основная теорема). Пусть a≥1 и b>1 - константы, а f(n) - произвольная функция а T (n) - функция, определенная на множестве неотрицательных целых чисел с помощью рекуррентного соотношения **T (n) = aT (n/b) + f (n),** где выражение n/b интерпретируется либо как ⌊ n/b ⌋, либо как ⌈n/b⌉.

Тогда асимптотическое поведение функции T (n) можно выразить следующим образом.

Случай 1. Если $\mathbf{f(n) = O(n^{\log_b a - \varepsilon})}$ для некоторой константы $\varepsilon > 0$, то $\mathbf{T(n) = \theta(n^{\log_b a})}$.

Случай 2. Если $\mathbf{f(n) = \theta(n^{\log_b a})}$, то $\mathbf{T(n) = \theta(n^{\log_b a} \lg n)}$.

Случай 3. Если $\mathbf{f(n) = \Omega(n^{\log_b a + \varepsilon})}$ для некоторой константы $\varepsilon > 0$, и если $\mathbf{af(n/b) \le cf(n)}$ для некоторой константы $\mathbf{c < 1}$ и всех достаточно больших **n**, то $\mathbf{T(n) = \theta(f(n))}$.

Суть основной теоремы. В каждом из трех выделенных в теореме случаев функция **f(n)** сравнивается с функцией $\mathbf{n^{\log_b a}}$. Интуитивно понятно, что асимптотическое поведение решения рекуррентного соотношения определяется большей из двух функций. Если большей является функция $\mathbf{n^{\log_b a}}$, как в случае 1, то решение - $\mathbf{T(n) = \theta(n^{\log_b a})}$. Если быстрее возрастает функция **f(n)**, как в случае 3, то решение – $\mathbf{T(n) = \theta(f(n))}$. Если же обе функции сравнимы, как в случае 2, то происходит умножение на логарифмический множитель и решение - $\mathbf{T(n) = \theta(n^{\log_b a} \lg n) = \theta(f(n) \lg n)}$.

Основной метод. Примеры решения рекуррентного уравнения

Пример 1. Пусть дано рекуррентное соотношение:

$T(n) = 9T(n/3) + n$.

В этом случае $a = 9$, $b = 3$, $f(n) = n$, так что $n^{\log_b a} = n^{\log_3 9} = \theta(n^2)$. Поскольку

$f(n) = O(n^{\log_3 9 - \varepsilon})$, где $\varepsilon = 1$, можно применить случай 1 основной теоремы и сделать вывод, что решение - $T(n) = \theta(n^2)$.

Пример 2. Пусть дано рекуррентное соотношение:

$T(n) = T(2n/3) + 1$,

в котором $a = 1$, $b = 3/2$, $f(n) = 1$, а $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Здесь применим случай 2, поскольку $f(n) = \theta(n^{\log_b a}) = \theta(1)$, поэтому решение - $T(n) = \theta(\lg n)$.

Пример 3. Пусть дано рекуррентное соотношение:

$T(n) = 3T(n/4) + n\lg n$, в котором $a = 3$, $b = 4$, $f(n) = n\lg n$, и $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Поскольку $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$ где $\varepsilon = 0.2$, применяется случай 3 (если удастся показать выполнение условия регулярности для функции f(n)). При достаточно больших n условие $af(n/b) = 3(n/4)\lg(n/4) < (3/4)\, n\lg n = cf(n)$ выполняется при $c = 3/4$. Следовательно, согласно случаю 3, решение этого рекуррентного соотношения - $T(n) = \theta(n \lg n)$.

Методы быстрой сортировки и ее анализ

Быстрая сортировка, подобно сортировке слиянием, основана на парадигме "разделяй и властвуй". Процесс сортировки подмассива A [$p..r$], состоит из трех этапов.

Разделение. Массив A [$p..r$] разбивается (путем переупорядочения его элементов) на два (возможно, пустых) подмассива $\boldsymbol{A}$ [$p..q$ - 1] и $\boldsymbol{A}$ [q + 1..r]. Каждый элемент подмассива $\boldsymbol{A}$ [$p..q$ - 1] не превышает элемент $\boldsymbol{A}$[q], а каждый элемент подмассива $\boldsymbol{A}$ [q + 1..r] не меньше элемента $\boldsymbol{A}$[q]**.** Индекс q вычисляется в ходе процедуры разбиения.

Рекурсивный шаг. Подмассивы $\boldsymbol{A}$ [$p..q$ - 1] и $\boldsymbol{A}$ [q + 1..r] сортируются путем рекурсивного вызова процедуры быстрой сортировки.

Комбинирование. Поскольку подмассивы сортируются на месте, для их объединения не нужны никакие действия: весь массив A [$p..r$] оказывается отсортирован.

Алгоритм быстрой сортировки:

```
Quicksort (A, p, r)
1  if p < r
2  then q ← Partition (A, p, r)
3  Quicksort (A, p, q - 1)
4  Quicksort (A, q + 1, r)
```

Вызов процедуры для сортировки всего массива A должен иметь вид Quicksort (A, 1, $length[A]$).

Производительность быстрой сортировки

Время работы алгоритма быстрой сортировки **зависит от степени сбалансированности,** которой характеризуется разбиение.

Сбалансированность, в свою очередь, зависит от того, какой элемент выбран в качестве опорного.

Если разбиение сбалансированное, асимптотически алгоритм работает так же быстро, как и сортировка слиянием.

В противном случае асимптотическое поведение этого алгоритма столь же медленное, как и у сортировки вставкой.

1) Наихудшее разбиение не сбалансированное разбиение

Когда подпрограмма, выполняющая разбиение, порождает одну подзадачу с n-1 элементов, а вторую - с 0 элементов. Разбиение выполняет процедура Partition (*A*, *p*, *r*)

Предположим, что такое ***несбалансированное разбиение*** возникает при каждом рекурсивном вызове.

Для выполнения разбиения требуется время $\theta(n)$. Поскольку рекурсивный вызов процедуры разбиения, на вход которой подается массив размера 0, приводит к возврату из этой процедуры без выполнения каких-либо операций, $T(0) = \theta(1)$.

Итак, рекуррентное соотношение, описывающее время работы этой процедуры, записывается следующим образом:

$$T(n) = T(n - 1) + T(0) + \theta(n) = T(n - 1) + \theta(n).$$

При суммировании промежутков времени, затрачиваемых на каждый уровень рекурсии($T(n-1)+T(n-2)+\ldots+T(1)$, получается арифметическая прогрессия , что дает в результате $\theta(n^2)$.

С помощью метода подстановок легко доказать, что $T(n) = \theta(n^2)$ является решением рекуррентного соотношения $T(n) = T(n - 1) + \theta(n)$.

Доказательство. Покажем, что $T(n) = cn^2$ для некоторого c.

Предположим, что $T(n - 1) = c(n-1)^2$.

Тогда $T(n) = c(n-1)^2 + \theta(n) = c n^2 - 2cn + c + \theta(n) = c n^2 + \theta(n) = \theta(n^2)$, ч.т.д.

2) Наилучшее разбиение –(метод Хоара)

Процедура Partition делит задачу размером n на две подзадачи, размер каждой из которых не превышает n/2

Размер одной из них равен [n/2], а второй – ([n/2] -1).

В такой ситуации быстрая сортировка работает намного производительнее, и время ее работы описывается следующим рекуррентным соотношением:

$T(n) \leq 2T(n/2) + \theta(n)$.

Это рекуррентное соотношение подпадает под случай 2 основной теоремы, так что его асимптотическая сложность $T(n) = O(n\lg n)$.

Таким образом, разбиение на равные части приводит к асимптотически более быстрому алгоритму.

3) Средний случай разбиения

Предположим, что разбиение происходит в соотношении один к девяти, что на первый взгляд весьма далеко от сбалансированности.

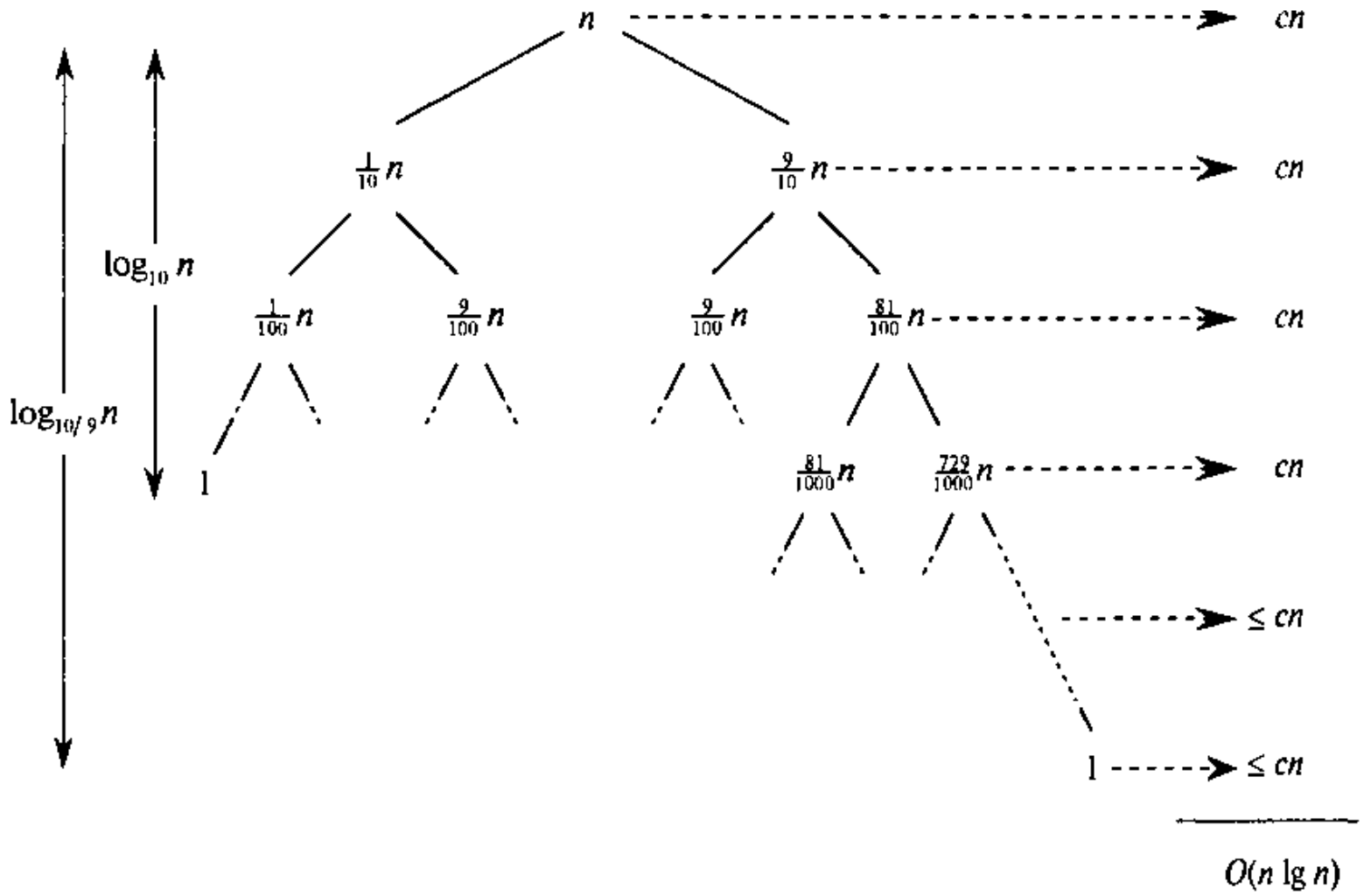
В этом случае для времени работы алгоритма быстрой сортировки мы получим следующее рекуррентное соотношение:

$T(n) \leq T(9n/10) + T(n/10) + \theta(n) = T(9n/10) + T(n/10) + cn.$

Время работы на каждом уровне рекурсивного дерева, соответствующего этому рекуррентному соотношению. содержит константу c, скрытую в члене $\theta(n)$, и равно cn. Так происходит до тех пор, пока не будет достигнута глубина $\log_{10} n = \theta(\lg n)$. Время работы более глубоких уровней не превышает величину cn. Рекурсия прекращается на глубине $\log_{10/9} n = \theta(\lg n)$ (см. рис. на следующем слайде).

Таким образом, полное время работы алгоритма быстрой сортировки равно $O(n \lg n)$.

В асимптотическом пределе поведение алгоритма быстрой сортировки в среднем случае намного ближе к его поведению в наилучшем случае, чем в наихудшем.



Рекурсивное дерево, соответствующее делению задачи процедурой Partition в соотношении 1:9.

Рандомизированная версия быстрой сортировки (случайное разбиение)

В алгоритме быстрой сортировки можно применить метод рандомизации, получивший название ***случайной выборки*** (random sampling).

Вместо того чтобы в качестве опорного элемента всегда использовать $A[r]$, такой элемент будет выбираться в массиве $A[p..r]$ случайным образом.

Подобная модификация, при которой опорный элемент выбирается случайным образом среди элементов с индексами от p до r, обеспечивает равную вероятность оказаться опорным любому из $r - p+1$ элементов подмассива.

Благодаря случайному выбору опорного элемента можно ожидать, что разбиение входного массива в среднем окажется довольно хорошо сбалансированным.

Изменения, которые нужно внести в процедуры Partition и Quicksort, незначительны. В новой версии процедуры Partition непосредственно перед разбиением достаточно реализовать перестановку (см. строки 1 и 2):

Randomized_Partition (A, p, r)

1 i ← Random (p, r)

2 Обменять $A[r] \leftrightarrow A[i]$

3 r**eturn** Partition (A, p, r)

В новой процедуре быстрой сортировки вместо процедуры Partition вызывается процедура Randomized_Partition.

Randomized Quicksort (A, p, r)

1 if $p < r$

2 then $q \leftarrow$ Randomized Partition (A, p, r)

3 Randomized Quicksort (A, p, q - 1)

4 Randomized Quicksort (A, q + *1*, r)