Алгоритмы сортировок

Наиболее часто используемые алгоритмы в задачах обработки структур данных

Под ***сортировкой (sorting problem)*** понимается процесс упорядочивания элементов некоторой последовательности данных (например, массива) в возрастающем, убывающем, не возрастающем, не убывающем порядке их значений.

Цель такого упорядочения - облегчение дальнейшей обработки данных, в первую очередь, задач поиска.

Таким образом, задача сортировки - одна из наиболее важных и необходимых задач программирования.

Задача алгоритма сортировки

Имеется множество элементов $a_1, a_2, \ldots a_n$, каждый элемент имеет специальное поле, называемое ***ключом*** .

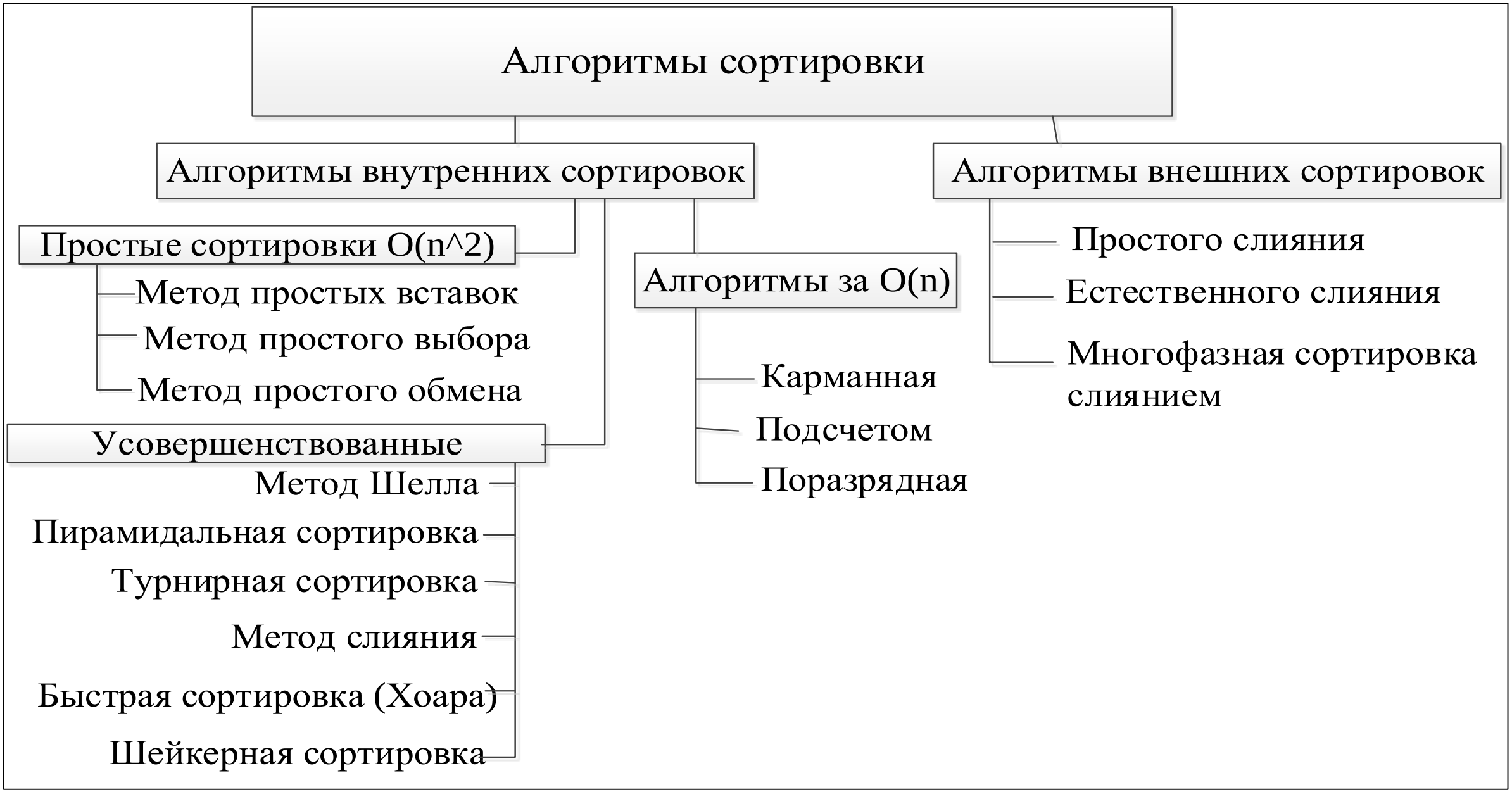
Элемент a_i имеет структуру: (***ключ, данные***).

Задача сортировки: выполнить перестановку элементов $a_1, a_2, \ldots a_n$ в соответствии со значением **ключей** $a_{k1}, a_{k2}, \ldots a_{kn}$ в соответствии с заданным отношением порядка: ключей, например, $a_{k1} \le a_{k2} \le \ldots \le a_{kn}$

Стандартные отношения порядка: по возрастанию, по убыванию, по не возрастанию, по не убыванию.

Например,

```
struct student{
        int numberBook;        //номер зачетной книжки – ключ записи
        Tdata data;            //другие данные о студенте
};
//Группа студентов – массив со сведениями о студентах
Student a[100];
```


Алгоритмы сортировки
Алгоритмы внутренних сортировок
Алгоритмы внешних сортировок
Простые сортировки O(n^2)
Метод простых вставок
Метод простого выбора
Метод простого обмена
Усовершенствованные
Метод Шелла
Пирамидальная сортировка
Турнирная сортировка
Метод слияния
Быстрая сортировка (Хоара)
Шейкерная сортировка
Алгоритмы за O(n)
Карманная
Подсчетом
Поразрядная
Простого слияния
Естественного слияния
Многофазная сортировка слиянием

Внутренние и внешние сортировки

Внутренние

ориентированы на сортировку массивов, хранящихся в оперативной (внутренней) памяти компьютера. При *внутренней сортировке* имеются более гибкие возможности для построения и выбора структур данных.

Внешние

ориентированы на сортировку данных в файлах, которые размещаются на внешних запоминающих устройствах (дисках или лентах).

Внешняя *сортировка* файлов (или *последовательностей*) характеризуется:

- очень б***о***льшими объемами обрабатываемых данных;
- более медленной внешней памятью и ограниченными методами доступа к данным.

Три категории простых сортировок on place (на месте)

В зависимости от используемого в алгоритме метода перемещения элементов, различают:

- Сортировка выбором
- Сортировка обменом
- Сортировка включениями (вставкой)

Каждая категория представлена:

- ***одним простым*** (или прямым) алгоритмом сортировки
- и, в общем случае, ***группой усовершенствованных алгоритмов***.

Приведенные простые алгоритмы сортировки – ***устойчивые***.

Усовершенствованные алгоритмы сортировки не устойчивые.

Устойчивость алгоритмов сортировки

Устойчивость сортировки бывает важной в случаях, когда элементы уже отсортированы по каким-то другим ключам.

Алгоритм сортировки *называется устойчивым*, если в результате сортировки относительное расположение элементов с равными ключами не изменилось.

Например.

struct Ts{ int key; char fam[30];char name[20];char otch[40];};

Ts x[]={{1, “Петров”, “Александр”, “Михайлович”}, {2, “Петров”, “Олег”, “Борисович”},{3, “Борисов”, “Андрей”, “Петрович”}};

При сортировке записей вида (фамилия, имя, отчество) по фамилии значения ключей для Петров Александр и Петров Олег будут одинаковы, поэтому устойчивая сортировка не переставит Александра и Олега местами.

В результате записи будут отсортированы только по фамилии, с сохранением исходного порядка среди записей с одинаковыми фамилиями:

3 Борисов Андрей Петрович

1 Петров Александр Михайлович

2 Петров Олег Борисович

Устойчивость можно обеспечить при использовании дополнительной памяти для хранения данных.

Критерии оценки сложности алгоритмов сортировки

Два типа основных (критических) операций над сортируемыми данными:

- сравнение ключей двух элементов (***x<y***)
- перемещение элементов (***x←y***).

Естественной мерой оценки временной сложности алгоритма сортировки массива, ***a[n]*** являются количество сравнений ***C(n)*** и количество перемещений элементов ***M(n).***

При этом оценку желательно проводить подсчитывая количество критических операций, выполняемых в ***лучшем, среднем и худшем случаях.***

Алгоритмы внутренних простых сортировок массивов – ***сортировки сравнением***

Предусматривают экономное использование оперативной памяти компьютера, которое устанавливает, что все перестановки элементов должны выполняться внутри сортируемого массива, как говорят **«на месте» (*on place*),** то есть, **дополнительные массивы при сортировке не используются.**

Постановка задачи сортировки массива

В общем случае на примере целочисленного массива, a[n] описывается следующим образом.

Вход: массив чисел (ключей) (a1 , a2 , … , an).

Выход: массив чисел (ключей) (a1’, a2’, … , an’) – перестановка последовательности чисел исходного массива, для которой выполняется

a1’<=a2’<= , … , <= an’

Например, получив на входе массив:

31 41 77 26 41 58 4 56

алгоритм сортировки должен выдать на выходе в качестве результата массив

4 26 31 41 41 56 58 77

Основное требование к сортировкам массивов – экономное использование памяти за счет переупорядочения элементов «on place(на том же месте)» т.е. в выделенной массиву памяти.

Методы, которые пересылают элементы из массива a в массив b не рассматриваются.

Характеристики простых методов сортировки массивов

1. Подходят для разъяснения принципов алгоритмов сортировки.
2. Программы, основанные на этих методах легко читаются.
3. Хотя **сложные** методы требуют выполнения меньшего числа операций, но эти операции более сложны.
4. При достаточно малых n простые методы работают быстрее, но их ***не*** следует использовать при больших n.

Алгоритм сортировки методом прямого обмена («Пузырек»)

Алгоритм.

1. Элементы массива перебираются «снизу вверх», демонстрируя всплытие легких элементов (обмен с более тяжелым), которые обгоняют более тяжелые и занимают место в соответствии со значением.

2. После этого активизируется новый проход по массиву в направлении «снизу вверх», но до того элемента, который уже сортирован и заняла свое место.

За один проход на место встает только один элемент.

Модель сортировки прямого обмена

J	0	1	2	3	4	5	6	7	8номер прохода
	45	6	6	6	6	6	6	6	6
	55	45	12	12	12	12	12	12	12
	12	55	45	18	18	18	18	18	18
	42	12	55	45	42	42	42	42	42
	94	42	18	55	45	45	45	45	45
	18	94	42	42	55	55	55	55	55
	6	18	94	67	67	67	67	67	67
	67	67	67	94	94	94	94	94	94

Алгоритм сортировки прямого обмена

Надо определить C(n) – количество сравнений и M(n) количество перемещений

EXCHANGE_SORT(a,n)	Количество раз
For j ←1 to n-1	n
do	
For i ←n-1 to j **do**	$\sum_{i=n}^{2} i$ +1=n+(n-1)+(n-2)…2+1 = n(n-1)/2=(n^2-n)/2

If a[i] <= a[i-1]
 	
SWAP(a[i], a[I-1]) // блок обмена(3 операции)
endIf
	C=$\sum_{i=n}^{2} i$ =n+(n-1)+(n-2)…+1 = n(n-1)/2=(n^2-n)/2
	
	M=3*$\sum_{i=n}^{2} i$ =n+(n-1)+(n-2)…+1 = n(n-1)/2=3/2*(n^2-n)/2

od od	

Инвариант: перед j-ым проходом отсортированы первые j-1 элементов и установлены на свои места, а элементы a[n..j) еще не сортированы. Доказать при j=1, при j=n.
Внутренний цикл: элемент, который должен занять позицию j находится в любой ячейке от a[i] до a[j] , при i=j элемент a[i] займет свое место.

Анализ алгоритма сортировки Прямого обмена

В наихудшем случае *(*массив сортирован и выполняется полное число сравнений и перемещений), тогда

$C(n)=(n^2-n)/2$ и $M(n)=3/2(n^2-n)$

Наилучший случай *(*массив сортирован по заданному правилу), тогда перемещений не будет, и $M(n)=0$, $C(n)==(n^2-n)/2$

Средний случай

$C(n)=(n^2-n)/2$, а перемещений в два раза меньше чем сравнений $M(n)=3/4(n^2-n)$

Можно заметить, что по мере ухудшения упорядоченности исходного массива число неупорядоченных элементов приближается к числу сравнений (каждый неупорядоченный элемент требует три операции обмена).

Сложность алгоритма сортировки *методом простого обмена* порядка $O(n^2)$ операций сравнения и порядка $O(n^2)$ операций перемещения.

Вывод: метод прямого обмена имеет общую сложность $(n^2-n)/2$ т.е. $O(n^2)$

Условие Айверсона

В рассмотренном примере есть особенность: после четвертого прохода массив оказался сортированным. Алгоритм продолжил выполнение, выполнив еще 4-е прохода. А если бы массив был достаточно большим?

Следовательно, чтобы понизить временную характеристику при сортировке массива, надо фиксировать наступление сортированности массива. Это просто сделать, введя в алгоритм флажковую переменную, отслеживающую событие обмена элементов.

Условие Айверсона: *если в очередном проходе сортировки при сравнении элементов не было сделано ни одной перестановки, то множество считается упорядоченным*

Алгоритм сортировки методом простого выбора

Основой является алгоритм поиска индекса минимального (максимального) элемента в массиве.

2 7 3 1 4 1 7 3 2 4 1 2 3 7 4 1 2 3 7 4 1 2 3 4 7

1.На первом шаге в неупорядоченном исходном массиве длиной ***n*** выбирается элемент с минимальным значением, который обменивается местами с первым элементом и исключается из дальнейшей сортировки.

2. Затем выбирается минимальный элемент среди оставшихся ***n-1*** элементов исходного массива, обменивается местами со вторым элементом и также исключается из дальнейшей сортировки.

3. Данный процесс продолжается до тех пор, пока в не сортированной части массива не окажется один единственный элемент.

Таким образом, на каждом шаге данного алгоритма текущий минимальный элемент записывается на ***i***-ое место исходного массива (i = 1, 2, …, n*)*, а элемент с ***i***-ого места записывается на его место.

При этом упорядоченные элементы исключаются из дальнейшей сортировки, поэтому длина просматриваемой части массива на каждом шаге уменьшается на единицу.

Проиллюстрируем алгоритм на примере

Псевдокод алгоритма

sortSelect(a,n)	Количество раз
for i ←1 To n-1 do	n
imin ← 1	n-1
for j ←1 To n-i do	$(n^2-n)/2$
If a[j] <= a[imin]	$C=(n^2-n)/2 -1$
then imin ← j endIf od	$(n^2-n)/2 -1$

SWAP(a[i], a[imin]) // блок обмена a[i] и a[min]
od
	M=3(n-1)

Анализ алгоритма прямого выбора

Подсчитаем сравнения

Внутренний цикл, осуществляющий поиск минимального элемента, выполняется всего *n(n-1)/2* раза. Это означает, что общее количество операций сравнения для данного алгоритма сортировки равно $1/2(n^2-n)$ или:

$C(n) = 1 + 2 + ... + n - 1 = n(n - 1)/2$

То есть, для большого числа элементов ***n*** этот алгоритм сортировки будет выполняться слишком **медленно.**

Подсчитаем обмены

В *лучшем случае* (когда исходный массив уже упорядочен) потребуется поменять местами только ***n-1*** элементов, а каждая операция обмена требует три операции пересылки. Количество операций обмена, которые производятся только во внешнем цикле равно ***3(n-1).***

Итак, $M(n) = 3(n - 1)$.

В *худшем случае,* когда каждый раз выполняются: сравнение и его оператор, и swap:

$M(n) =(n^2-n)/2 -1+3(n-1)$

В *среднем* число операций обмена равно $M(n) =n(\log n+y)$, где ***y*** является константой Эйлера, приблизительно равной 0,577216.

Алгоритм сортировки методом Прямой вставки

Идея алгоритма сортировки ***методом простых вставок*** (***Insertion sort***) заключается в добавлении элементов один за другим **к ужу отсортированной части массива с сохранением в ней критерия упорядоченности.**

j	0	1	2	3	4
1	6	7	2	4	1
2	6	7	2	4	1
3	2	6	7	4	1
4	2	4	6	7	1
5	1	2	4	6	7

Сортировка ***методом простых вставок*** удобна для сортировки коротких последовательностей данных.

- На 1-ом шаге второй элемент сравнивается с первым. Поскольку он больше первого, то остается на своем месте. Остальная часть массива остается без изменения.
- На 2-ом шаге третий элемент сравнивается уже с двумя упорядоченными элементами. Поскольку он меньше их, то начинает обмениваться с предыдущими большими его элементами.

Анализ алгоритма методом Прямой вставки

INSERTION_SORT(a,n)	
for j ← 2 To n	n
do x ←a[j] i ←j-1	n-1 n-1

// блок вставки a[j] в отсортированную часть массива a[1..j-1]
If (a[i]>x) do
n	

While i>0 And a[i]>x	n*n
do a[i+1] ←a[i] i ←i-1 Od od	n*n-1 n*n-1
a[i+1] ←x od	n-1

Лучший случай-массив отсортирован в соответствии с отношением порядка сортировки: 1 2 3 4 5.

Внутренний цикл не выполняется, а сравнение выполняется

C(n)=n

M(n)=3(n-1)+n=4n-3

Сложность O(n)

Худший случай – массив полностью не сортирован по заданному критерию: 5 4 3 2 1.

C(n)=n+n*n

M(n)=3n*n+n-3

Сложность O(n^2)

Средний случай – массив заполнен случайными числами

Достоинства сортировки методом Прямая вставка

Во-первых, она обладает естественным поведением, то есть, выполняется быстрее для упорядоченного массива и дольше всего выполняется, когда массив упорядочен в обратном направлении. Это делает данную сортировку полезной для упорядочения почти отсортированных массивов.

Во-вторых, элементы с одинаковыми ключами *не* переставляются (свойство *устойчивости*): если массив элементов сортируется с использованием двух ключей, то после завершения сортировки вставкой он по-прежнему будет упорядочен по двум ключам.

Сложность алгоритма сортировки *методом простых вставок* порядка $\boldsymbol{O(n^2)}$ операций сравнения и порядка $\boldsymbol{O(n^2)}$ операций перемещения.

Сравнение алгоритмов простых сортировки

Анализ показывает, что сортировка Прямого обмена во всех случаях **хуже**, чем сортировки вставки и выбора.

Алгоритм сортировки Прямого выбора эффективней сортировки Прямого обмена по критерию M(n), поскольку при сортировке каждый элемент перемещается не более чем один раз.

Если сортируемая последовательность состоит из записей большого размера, этот критерий может иметь решающее значение.

Метод Прямого обмена (Пузырек) вряд ли где – то будет использоваться. **Его алгоритм хорошо запоминается и только.**

Улучшенные алгоритмы сортировки

За счет уменьшения значений характеристик M и C или хотя бы одного из них

Алгоритм шейкерной (челночной) сортировки

Является как-бы улучшением обменной сортировки.

Рассмотрим два теста для сортировки прямого обмена

Первый тест: 12 18 42 44 55 67 94 6

Видно, что ***не*** сортирован только один элемент 6, и он всплывет на свое место за один проход при проходе снизу вверх.

Второй тест: 94 6 12 18 42 44 55 67

Видно, что ***не*** сортирован только один элемент 94, но алгоритм выполнялся от последнего элемента к первому. Чтобы этот элемент переместился на место понадобиться 7 проходов.

Т.е. проявляется некая асимметрия в алгоритме. Эта асимметрия подсказывает новое решение: менять направление следующих друг за другом проходов.

Идея алгоритма *шейкерной сортировки:*

- в запоминании индекса (k) элемента, участвующего в последнем обмене. Все элементы с индексом, превышающим ***k***, уже упорядочены и можно прекратить дальнейшие просмотры массива –условие Айверсона.
- чередование направлений просмотра массива.

В этом случае сильно удаленные от своего места элементы будут быстро перемещаться на свои законные места.

Например: 94 6 12 18 42 44 55 67 6 12 18 42 44 55 67 94

Модель выполнения Шейкер сортировки

1	**45**	**55**	**12**	**42**	**94**	**18**	**6**	**67**
2	45	12	42	55	18	6	67	94
3	6	45	12	42	55	18	67	94
4	6	12	42	45	18	55	67	94
5	6	12	18	42	45	55	67	94
6	6	12	18	42	45	55	67	94
7	6	12	18	42	45	55	67	94
8	6	12	18	42	45	55	67	94

Алгоритм Шейкер сортировки

```
SheikerSort(a)
l←0; //индекс левого элемента
r ← n-1 ; //индекс правого элемента
k ← 0;
while(l<r)// проходы
do
//слева направо
i ← l;
    while(i<r)
    do
      if(a[i]>a[i+1])
          swap(a[i],a[i+1]);
        k ← i;       //индекс последнего переставл.
        endIf
      i ←i+1;  // увеличение i
    od
r=r-k
//справа налево
i ← r;
    while(i>l)
     do
       if(a[i]<a[i-1])
            swap(a[i],a[i-1]) ;
            k ← i;
        endIf
      i ←i-1;
    od
     l ← l + k;
od
```

Анализ шейкерной сортировки

Хотя данная сортировка является улучшением ***«пузырькового метода»***, ее нельзя рекомендовать для использования, поскольку время выполнения по-прежнему квадратично зависит от числа входных элементов.

Количество операций сравнений не изменяется,

а количество операций перемещения уменьшается лишь на незначительную величину.

Сложность алгоритма *шейкерной сортировки* порядка $O(n^2)$ операций сравнения и порядка $O(n^2)$ операций обмена.

Шейкер сортировку можно использовать, если заранее известно, что массив уже почти упорядочен.

Сортировка методом Шелла

Это **улучшенная версия сортировки вставками.**

В алгоритме вставками за один проход на место перемещается один элемент.

Сортировка Шелла поддерживает алгоритм перемещения: за один проход нескольких элементов, отстоящих друг от друга на заданное расстояние. Это расстояние называют **смещением.**

В алгоритме Шелла используется принцип ***убывающего смещения***: т.е.

- первый проход имеет наибольшее смещение,
- на втором и последующих проходах это смещение уменьшается
- на последнем проходе **смещение равно 1.**

Модель выполнения сортировки Шелла

d=4	Исходный массив: 44 55 12 42 94 18 06 67 n=8. По i перебираем от 0 до n- n/2^k или (n-4) из-за смещения	
i=0	44 55 12 42 94 18 06 67	Исходный массив
i= 0 j=0	a[0] c a[4]	Нет обмена
i=0 j=-1		завершение
i=1	44 55 12 42 94 18 06 67	
i=1 j=1	a[1] c a[5] 44 18 12 42 94 55 06 67	a[1]>a[5] обмениваются
i=1 j=0	a[0] c a[4]	a[0]<a[4] Не обмениваются
i=1 j=-1		Завершение
i=2	44 18 12 42 94 55 06 67	
i=2 j=2	a[2] c a[6] 44 18 6 42 94 55 12 67	a[2]>a[6] обмениваются
i=2 j=1	a[1] c a[5]	Не обмениваются
i=2 j=0	a[0] c a[4]	Не обмениваются
i=3	44 18 6 42 94 55 12 67	
i=3 j=3	a[3] c a[7]	Не обмениваются
i=3 j=2	a[2] c a[6]	Не Обмениваются
i=3 j=1	a[1] c a[5]	Не Обмениваются
i=3 j=0	a[0] c a[4]	Не Обмениваются
d=2	По i перебираем от 0 до n-d-1 (5) из-за смещения	

Сортировка Шелла со смещением убывающим вдвое

```
    d← n / 2; //инициализация смещения половиной размера массива
while (d > =1)     do   // пока смещение >=1
      for i ← 0 to d   do
        j ← i;
        //как в алгоритме вставки, только сравниваем i-ый и j+d
         while(j>=0 and  a[j] > a[j + d])
            do
           h ← a[j];
           a[j] ← a[j + d];
           a[j + d] ← h;
           j ← j-1;
         od
      od
    d ← d/ 2;
od
```

Описание выполнения алгоритма Шелла

В алгоритме сортировки *методом Шелла* (Shell sort) на каждом шаге сравниваются между собой элементы массива, расположенные на расстоянии *d* друг от друга.

Первоначально значение *d* принимается равным ***n/2***, в конце выполнения каждого шага расстояние *d* уменьшается вдвое *d =d/2*. На втором шаге устанавливается ***d = d/2*** *= 2* и процесс повторяется.

На третьем шаге устанавливается *d = d/2 = 1*. То есть, на данном шаге будет иметь место процесс, аналогичный методу ***простой вставки***, с тем принципиальным отличием, что в данном случае массив уже частично отсортирован.

Внутренний цикл имеет два условия проверки.

Условие a[j] > a[j + d]) необходимо для упорядочения элементов.

Условие *j>0* необходимы для того, чтобы предотвратить выход за пределы массива *a*. Эта дополнительная проверка в некоторой степени ухудшает сортировку Шелла.

Можно использовать барьерный элемент, чтобы убрать проверку по j.

Последовательности приращений по Кнуту

В результате проведенных исследований этой сортировки Д. Кнут предложил использовать последовательности приращений, которые оптимизируют сортировку.

1) 1 4 13 40 121 …..

Где $d_{k-1}=3d_k+1$, $d_k=1$ и $k=\log_3 n - 1$

2) 1 3 7 15 31 …

Где $d_{k-1}=2d_k+1$, $d_k=1$ и $k=\log_2 n - 1$ - сложность алгоритма, использующего эту последовательность $\Theta(n)$

А вот последовательностей степени двойки следует избегать, так как проведенные исследования убедительно показывают, что такой выбор снижает эффективность алгоритма сортировки.

Пример определения количества смещений и их значения

1) n=27 смещения по формуле 1) Кнута.

Решение: $k=\log_3 27 - 1=2$; $d_2=1$ $d_1=4$

2) n=81

Анализ сортировки Шелла

Анализ сложности алгоритма Шелла требует решения непростых математических задач.

Внешний цикл будет выполняться при смещении=n/2 при: 2^k 2^{k-1} … 4 2 1 т.е. будет выполнен ***к*** раз. На к-ом шаге d=1 т.е.$n/2^k = 1$, тогда к= $\log_2 n$.

Время выполнения сортировки Шелла может быть оценено по формуле

0,3*n (log n)², что пропорционально $n^{1.2}$. Как видно, эта зависимость значительно лучше квадратичной зависимости, которой подчиняются простые алгоритмы сортировки.

Прежде чем Вы решите использовать сортировку Шелла, следует иметь в виду, что быстрая сортировка имеет более привлекательные характеристики работы.

Сложность алгоритма сортировки *методом Шелла* порядка ***O(n (log(n))²).***

Турнирная сортировка

От сортировки выбора к ее улучшенному варианту

Сортировка выбором основана на выборе наименьшего ключа среди n элементов, затем среди n-1 и т.д.

Как ее можно улучшить? Понизить показатели M и C.

Это можно сделать, если от каждого прохода получать больше информации, чем просто указание на один наименьший (наибольший) элемент.

Идея метода:

- с помощью n/2 сравнений можно определить наименьший ключ из каждой пары;
- при помощи следующих n/4 сравнений можно выбрать наименьшие из каждой пары таких наименьших и т.д.

Наконец при помощи n-1 сравнения мы можем построить дерево выбора.

Например.

35 25 50 20 15 45 10 40

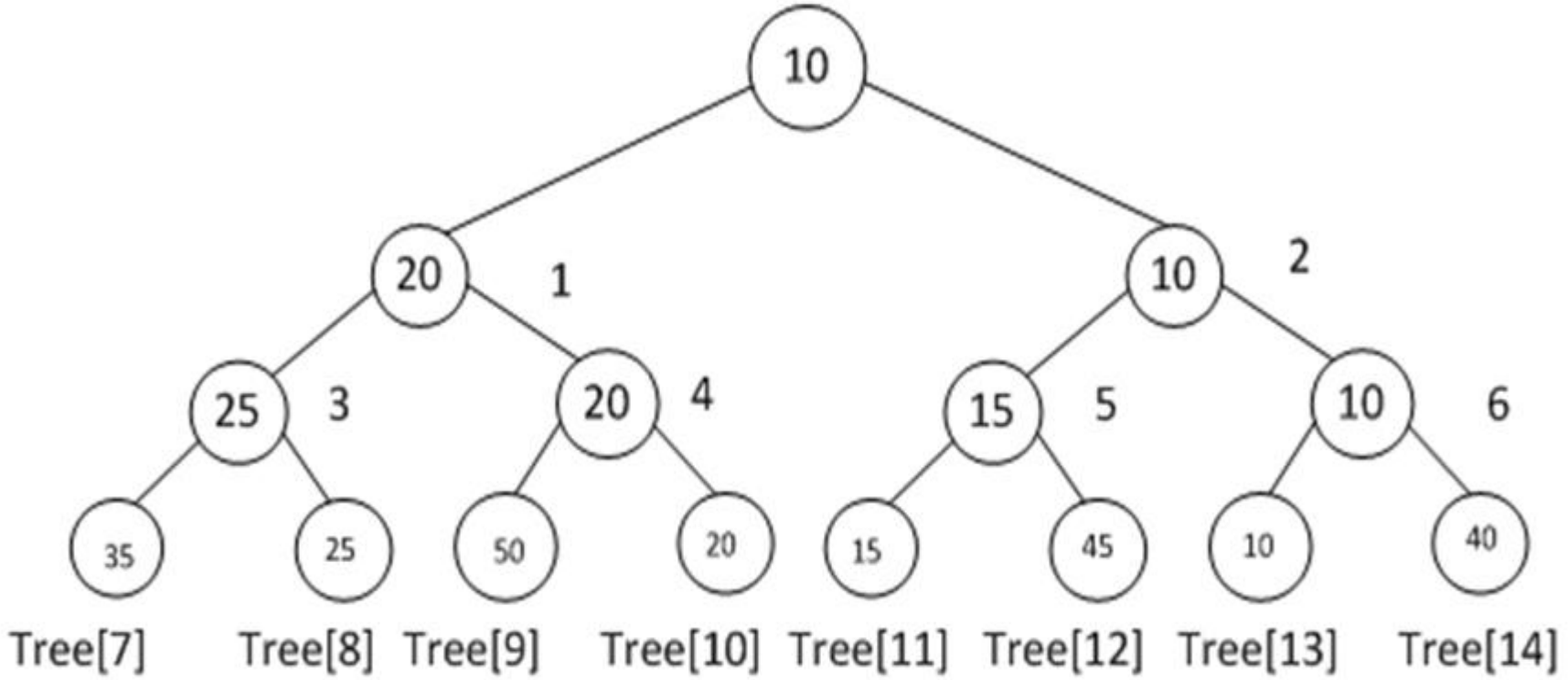
25 20 15 10

20 10

10

Модель дерева выбора

Исходный массив ключей: 35 25 50 20 15 45 10 40



Tree

							35	**25**	**50**	**20**	**15**	**45**	**10**	**40**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Турнирная сортировка – оптимизация сортировки выбором

Это первая сортировка в которой использовалось бинарное дерева для представления механизма перемещения элементов.

Алгоритм относится к, так называемым, «древесным» методам, так как в своей работе в качестве структуры данных использует бинарное дерево, размещенное в массиве.

Более упрощенным вариантом турнирной сортировки является пирамидальная сортировка.

Идея алгоритма ***турнирной сортировки*** заключается в повторяющихся поисках наименьшего элемента сначала среди ***n*** элементов, затем среди оставшихся ***n-1*** элементов, потом – среди ***n-2*** элементов и так далее.

Определение размера массива под дерево

Как мы видим дерево состоит из 15 узлов и каждый узел хранит значение. Значит для хранения дерева в памяти понадобится массив из 15 элементов.

А как рассчитать размер массива для хранения дерева сортируемого массива из n элементов?

Ответ: так как дерево полное, то на каждом уровне с индексом i размещается 2^i узлов, тогда всего узлов $2^{k+1}-1$ узлов, где к – высота дерева.

Так как n=8 и это последний уровень и $8= 2^3$, то высота k=3

Количество элементов в массиве = $2^4-1=15$

Назовем массив хранящий дерево Tree[15]

Алгоритм

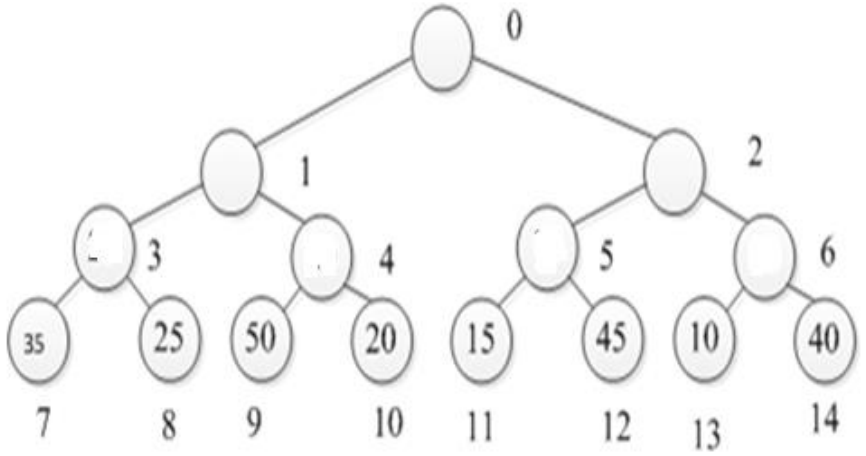
В алгоритме выделено два этапа.

Алгоритм 1. Построение бинарного дерева выбора.

Алгоритм 2. Перемещение наименьшего значения из корня в новый массив и формирование нового массива из элементов корня

Особенность дерева: это совершенное бинарное дерево – его узлы можно пронумеровать так :

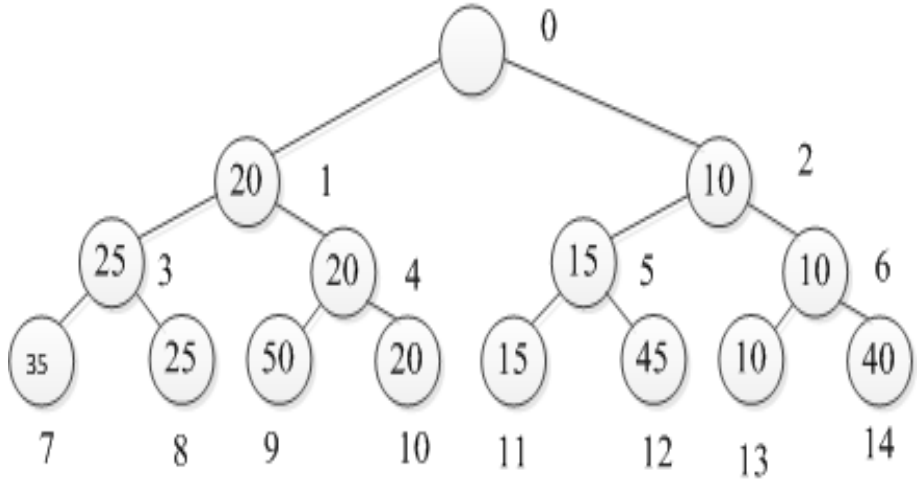
- в левом поддереве узлы с индексами 2i+1
- в правом поддереве узлы с индексами 2i+2



Модель Алгоритма 1. Построение бинарного дерева выбора

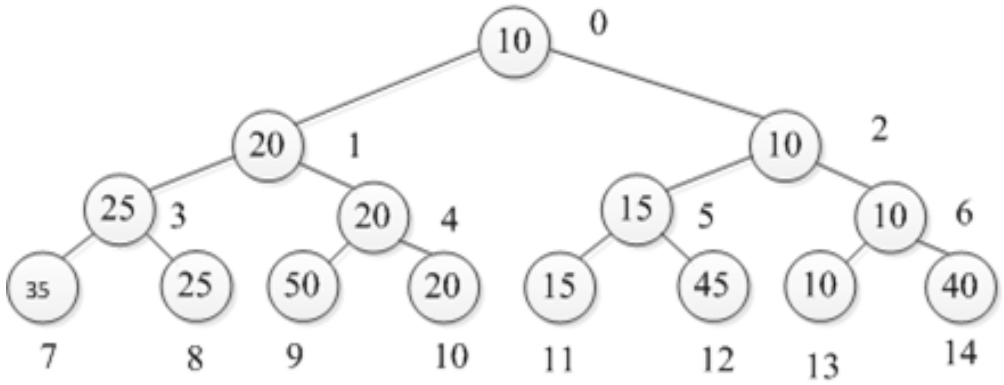
Сначала из ***n*** элементов исходного массива чисел строится нижний уровень (листья) бинарного дерева.

После проведения *n/2* сравнений в каждой паре элементов определяется элемент с наименьшим значением и размещается на уровень выше.



Затем из каждой пары уже выбранных минимальных элементов с помощью **n/4** сравнений определяется свой минимальный элемент.

И, так далее. Таким образом, проделав ***n - 1*** сравнений, будет построено, так называемое, **бинарное дерево выбора**, корнем которого является элемент массива с **наименьшим** значением.



Алгоритм 1. Построение бинарного дерева выбора(на массиве)

Шаг 1. Заполнение массива турнирного дерева элементами массива – в конец массива

							35	25	50	20	15	45	10	40
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Шаг 2. В родительские узлы помещаются наименьшие значения пар нижнего уровня.

Например, между элементами Tree[7] и Tree[8] выигрывает меньший из них, и значение 25 записывается в Tree[3].

Родитель узлов 7 и 8 это узел с номером 3. Родитель узла i рассчитывается по формулам: если i четное, то i=2p+2, то p=(i-2)/2, а если i нечетное, то p=(i-1)/2.

			25			10	35	25	50	20	15	45	10	40
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Алгоритм 2. Перемещение наименьшего значения из корня в новый массив

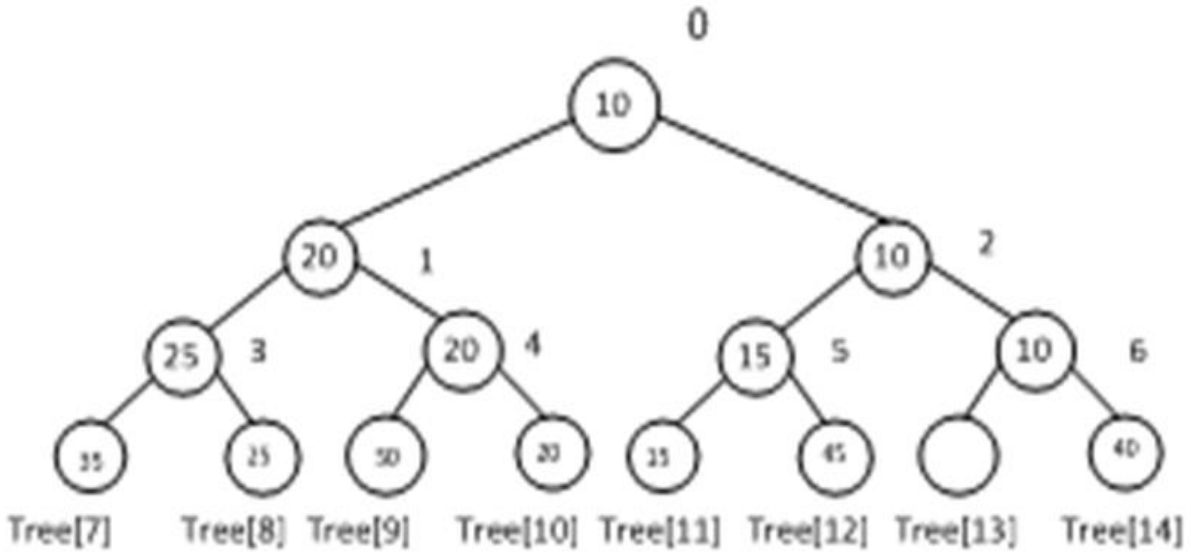
Шаг 3. Запись наименьшего значения в массив A и удаление наименьшего из листа

Как только наименьший элемент оказывается в корневом узле, он удаляется со своего старого места - листа и копируется в новый массив A в текущую свободную ячейку. Этот массив будет хранить рассортированные элементы.

Сортировка завершится, когда в листьях не останется сортируемых элементов

В результате шага 3 мы имеем

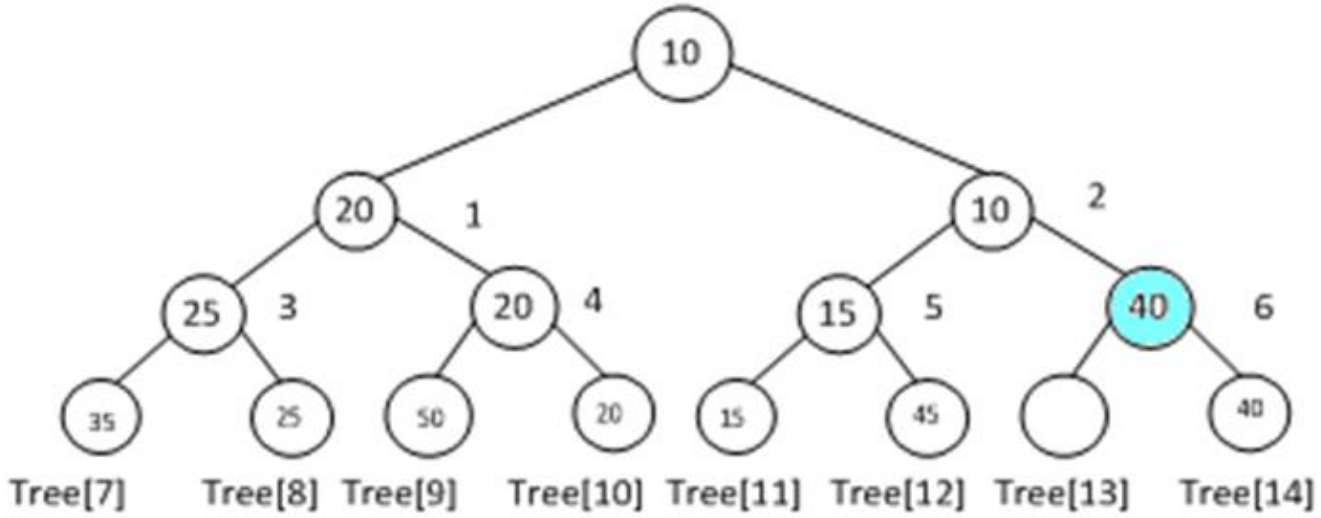
Шаг 4. Перемещение элементов в дереве выбора



Алгоритм 1. Построение бинарного дерева выбора

Продолжение **шага 4**.

Поскольку число 10 изначально было в Tree[13], проигравший в первом круге Tree[14] = 40 должен снова участвовать в турнире. Tree[14] копируется в свой родительский узел т.е. в Tree[6]

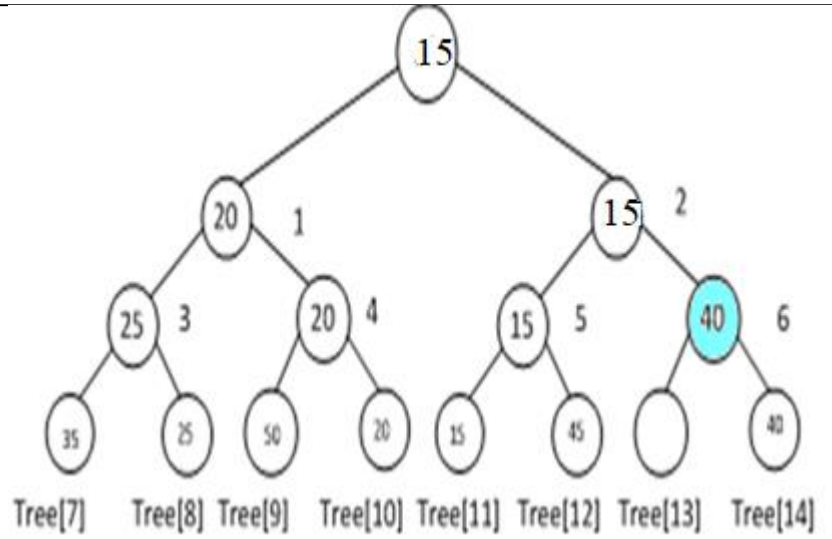


10	20	10	25	20	15	40	35	25	50	20	15	45		40
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Алгоритм 1. Построение бинарного дерева выбора (*продолжение сортировки шага 2*)

Теперь снова сравниваются элементы второго уровня:

- Tree[3] и Tree[4] изменений в дереве не происходит



- Tree[5] и Tree[6] изменение в дереве происходит:

число 15 заменяет значение Tree[2]

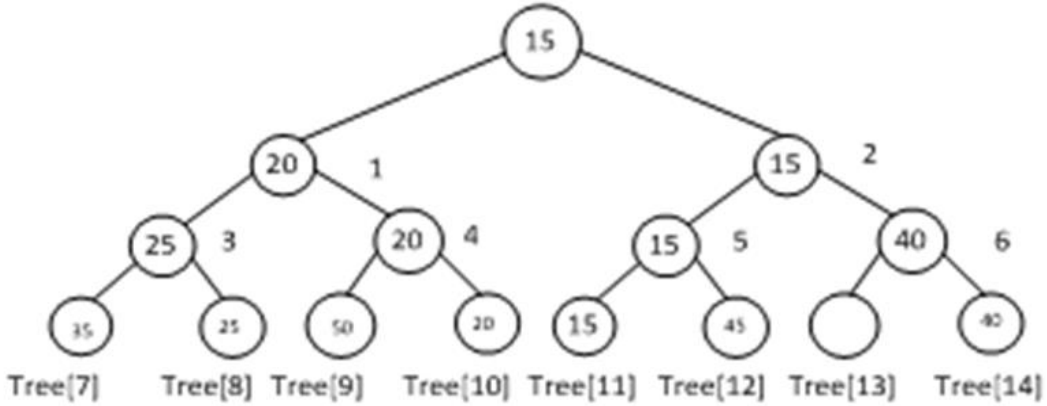
10	20	15	25	20	15	40	35	25	50	20	15	45		40
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- Tree[1] и Tree[2] в корень записывается число 15

15	20	15	25	20	15	40	35	25	50	20	15	45		40
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

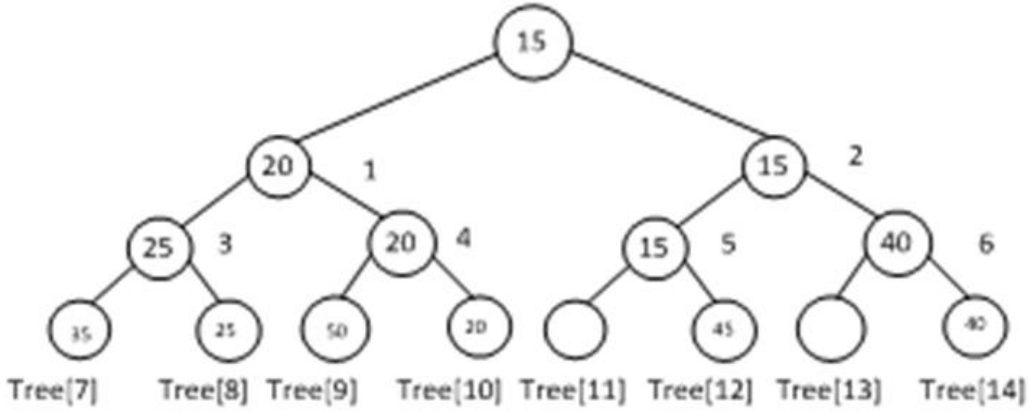
Продолжение сортировки (шаг 2)

Имеем дерево



Значение 15 из корня переписываем в массив A в следующий свободный элемент и значение 15 удаляем из листа.

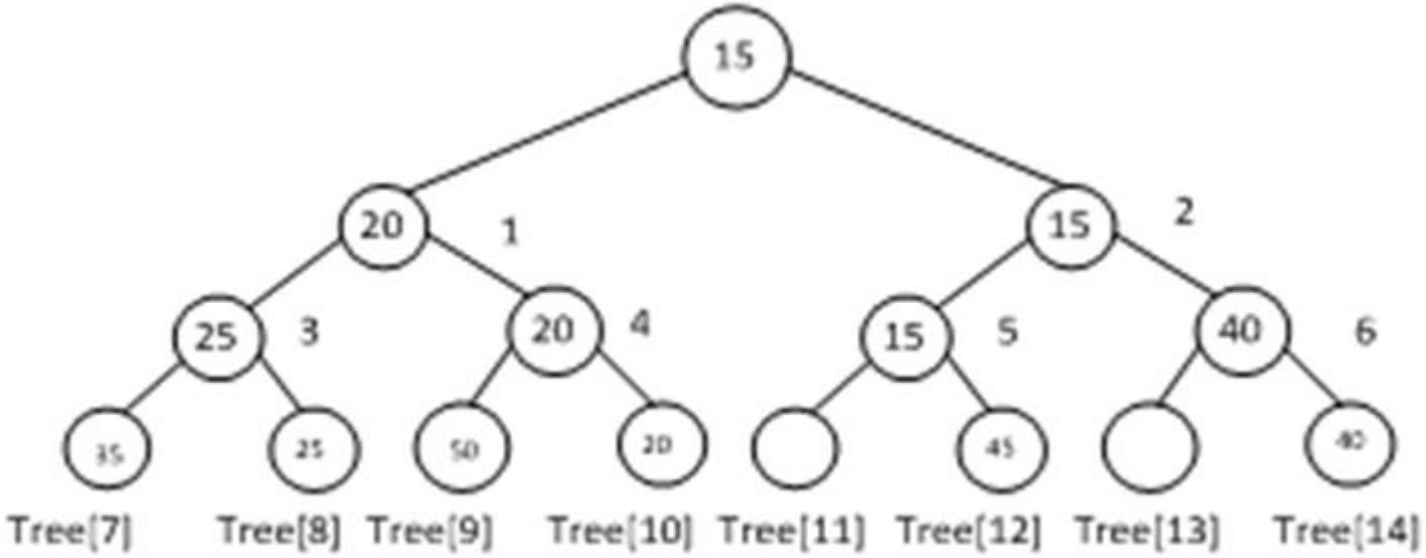
A	10	15						



15	20	15	25	20	15	40	35	25	50	20		45		40
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Продолжение сортировки шаг 3

Значение 15 из корня переписываем в массив A в следующий свободный элемент и значение 15 удаляем из листа.



A	10	15						

15	20	15	25	20	15	40	35	25	50	20		45		40
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Задание:

- Продолжить сортировку массива
- Определить задачи и схему алгоритма сортировки
- Реализовать алгоритм и протестировать

Алгоритм турнирной сортировки

Алгоритм 1. Построение турнирного дерева

1) Определить размер массива B для хранения дерева
2) Записать в массив исходные данные

$$l = 2^{\lceil \log_2(n+1) \rceil} - 1$$

Алгоритм 2. Формирование массива результатов A и переформирование дерева

Повторять n раз

1) Заполнить родительские узлы: сравнений n/2.
2) Запись элемента B[0] в массив A
3) Удалить узел из исходного дерева (записать значение null) Записать оставшийся узел в родительский узел

Анализ турнирной сортировки

Каждый из n шагов требует лишь log(n) сравнений.

А вся сортировка ***n*log(n)*** элементарных операций, не считая шагов, которые требуются для построения дерева.

Сложность Турнирной сортировки O(***n*log(n)) по времени,***

по памяти O (n+$2^{log(n+1)+1}$-1)

При использовании турнирного дерева задача хранения стала сложнее и поэтому увеличилась сложность отдельных шагов.

Опять встает задача: оптимизации по памяти – использовались два дополнительных массива.

Это решение найдено: это пирамидальная сортировка.

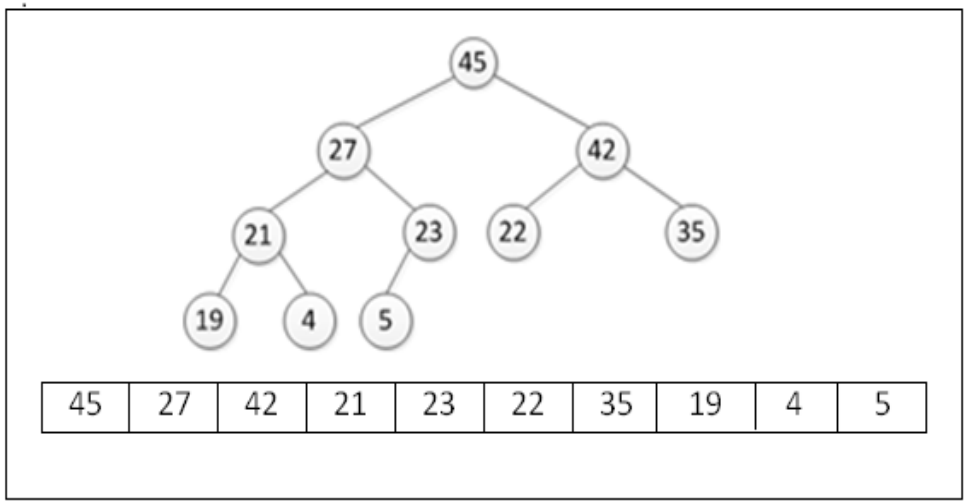
Пирамидальная сортировка «сортировка кучей»

Пирамидальная сортировка и двоичная куча

Пирамидальная сортировка - это метод сортировки сравнением, основанный на такой структуре данных как **двоичная куча**.

Двоичная куча – это законченное дерево в котором значения хранятся в особом порядке: значение в родительском узле больше (или меньше) значений двух его дочерних узлов. Т.е. различают убывающую или возрастающую кучу.

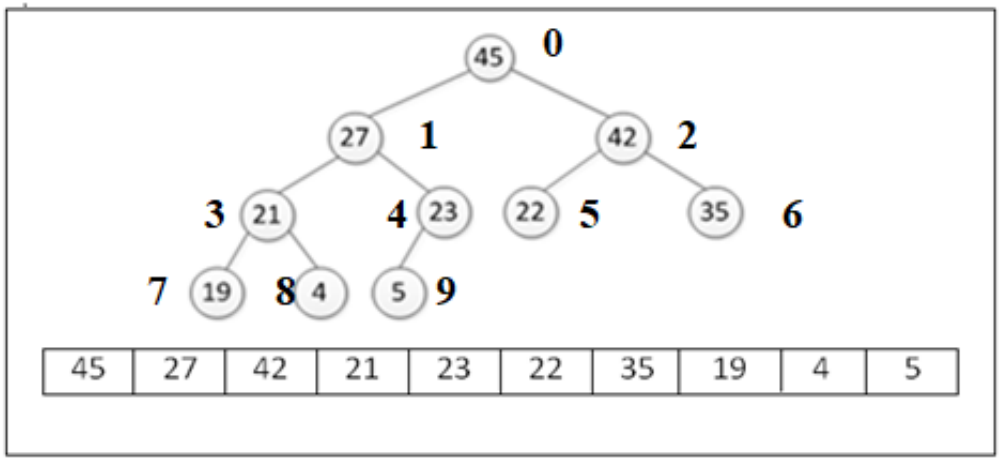
Законченное дерево*(Совершенное дерево)* - это полное бинарное дерево **до предпоследнего уровня**, в котором на каждом уровне полное число узлов ($2^{\text{номер уровня}}$), кроме нижнего, на нижнем уровне узлы *располагаются прижаты к левому краю (только слева).*



Пирамидальная сортировка. Свойство законченного дерева

Точное, местоположение данных (индекс в массиве) можно определить по следующим правилам:

- в элементе с индексом [0] находиться значение корня;
- для некорневого узла [i] индекс родителя определяется по формуле [(i-1)/2];
- если известно положение узла с индексом [i], то его дочерние узлы находятся в элементах с индексами 2*i+1 (левый) и 2*i+2 (правый).



Законченное дерево можно реализовать на массиве. Индексы мы можем использовать для поиска дочерних узлов и родителей.

1) Найти индекс левого и правого дочерних узлов узла 3.
2) Найти индекс родителя узла с индексом 5; 4.

Куча – структура данных

Рассматриваем массив a как двоичное дерево:

- Элемент a[i] является узлом дерева
- Элемент a[i/2] является родителем узла a[i]
- Элементы a[2*i+1] и a[2*i+2] являются детьми узла a[i]

Для всех элементов пирамиды выполняется соотношение (основное свойство кучи):

Для нумерации индексов от 0 до n-1:

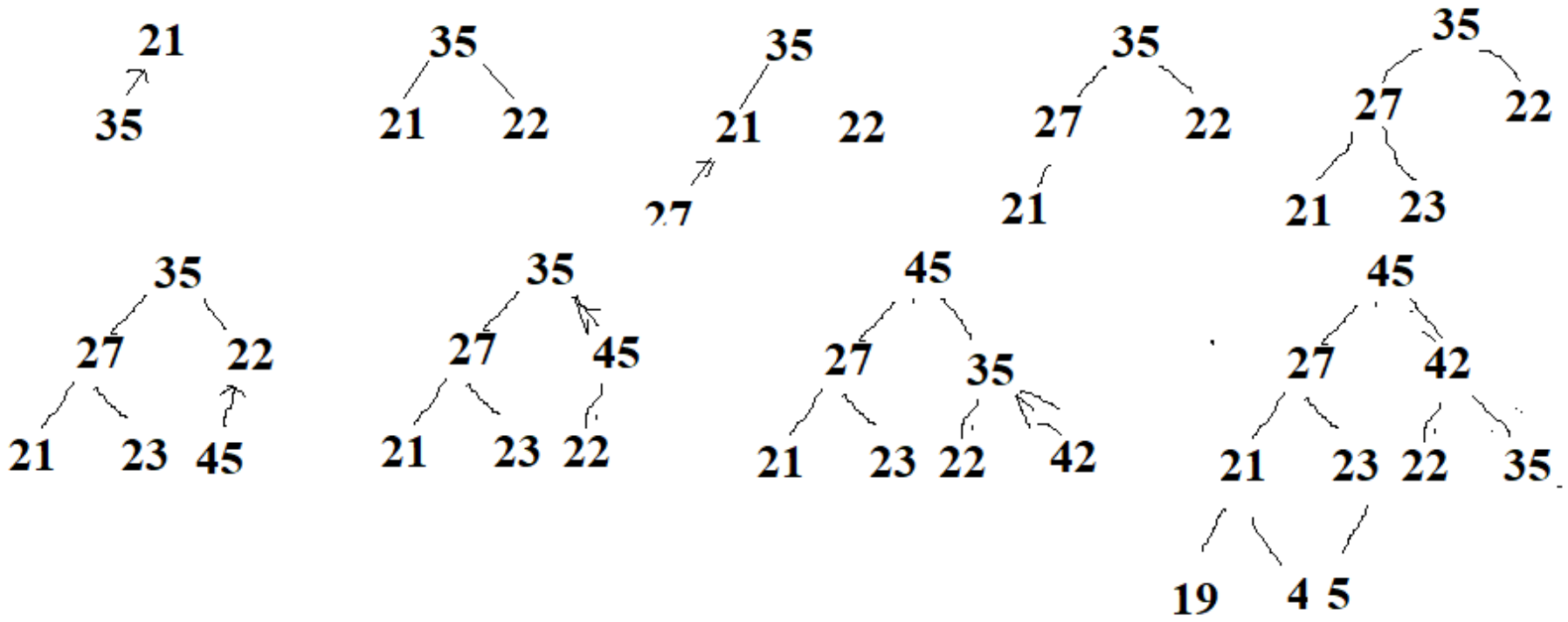
a[i] >= a[2*i+1] и a[i] >= a[2*i+2]

Сравнение может быть как в большую, так и в меньшую сторону: убывающая или возрастающая куча.

Алгоритм построения возрастающей кучи из последовательности чисел

Массив 21, 35, 22, 27, 23, 45, 42, 19, 4, 5

Алгоритм просеивания вверх



Пирамидальная сортировка. Идея алгоритма сортировки

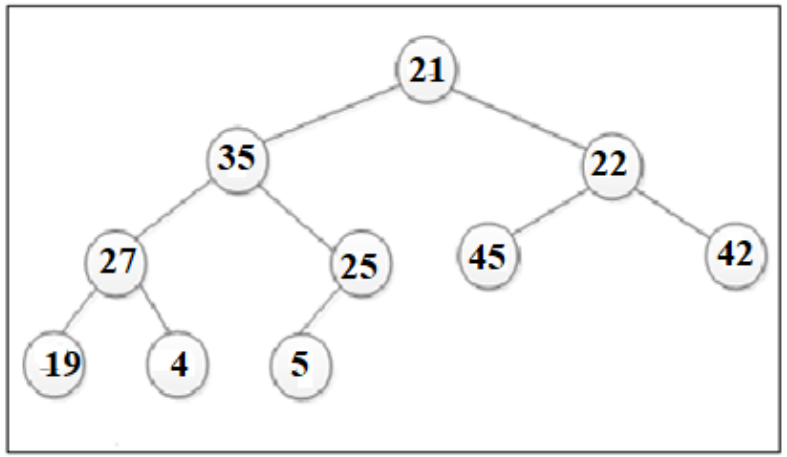
Алгоритм пирамидальной сортировки можно сформулировать из правила кучи:

переформировать массив так, чтобы соответствующее ему бинарное дерево было кучей, т.е. во внутреннем (родительском) узле должно быть значение не меньше(не больше) значений его дочерних узлов.

Возрастающей пирамидой называется почти заполненное дерево, в котором значение каждого элемента больше либо равно значения всех его потомков.

В *убывающей пирамиде* значение каждого элемента меньше либо равно значения потомков.

Пусть есть массив ключей: 21, 35, 22, 27, 23, 45, 42, 19, 4, 5, его элементы занесем в дерево, формируя дерево сверху и заполняя каждый уро направлении слева направо.

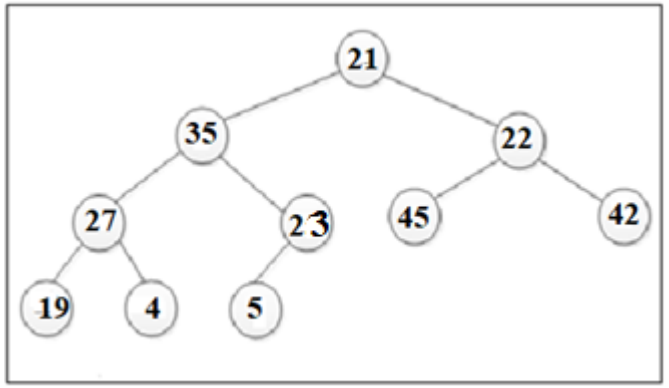


Полученное дерево **не является кучей**, так как нарушено свойство: упорядоченность по значениям.

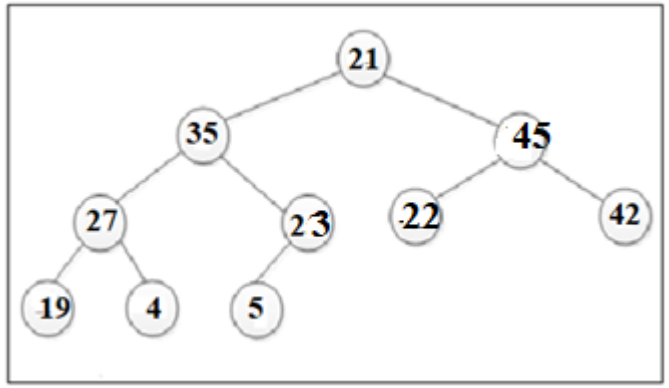
Алгоритм пирамидальной сортировки массива

1. Построить бинарное дерево из элементов массива заполняя обходом в ширину, из всех элементов массива, не заботясь о соблюдении основного свойства кучи

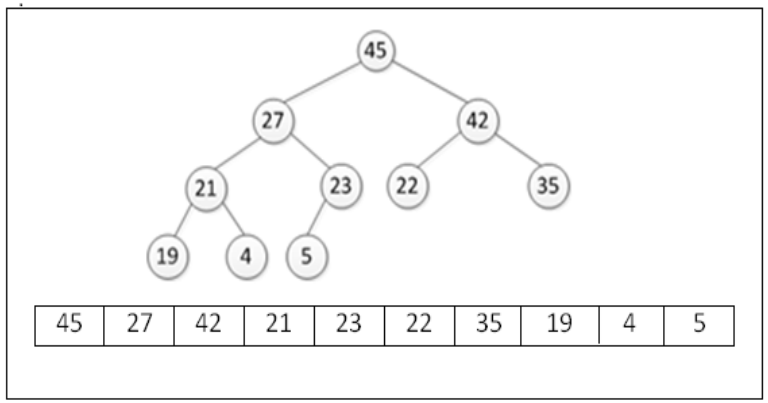
1)



2)



3)



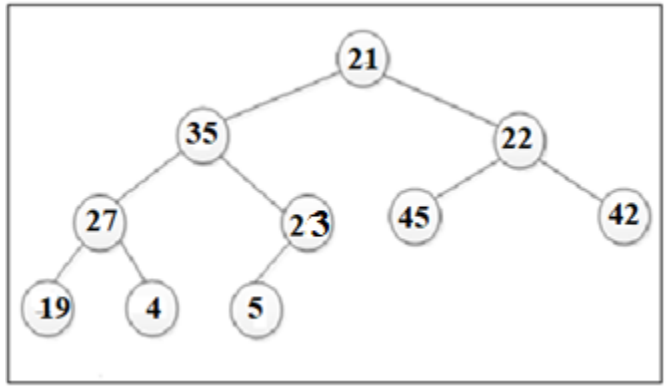
2. Упорядочение кучи *просеиванием вниз*. Применить метод упорядочения кучи для всех вершин, у которых есть хотя бы один потомок (так как поддеревья, состоящие из одной вершины без потомков, уже упорядочены). Потомки гарантированно есть у первых n/2 вершин.

3. Повторять п.2 пока дерево не станет кучей

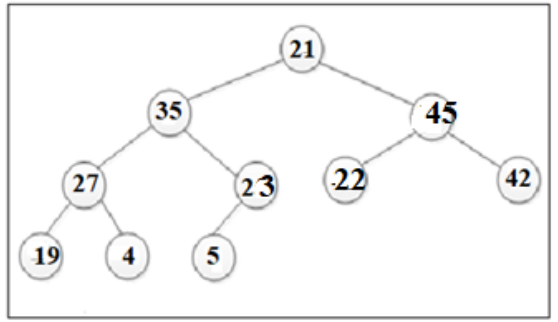
4. Поменять значение корня с последним элементом в не сортированной части массива

Алгоритм пирамидальной сортировки. Создание убывающей кучи

1)Спускаемся сверху и восстанавливаем кучу, её левое и правое поддеревья.



2)Перестраиваем поддерево с корнем 22, i=2i+2=2*0+2=2: из 45 и 22 наибольшее 45, тогда 45 поднимается на место 22, а 22 спускается на место 45.



3) Затем просеивается левое поддерево узла 21 с корнем 35, и так до листьев

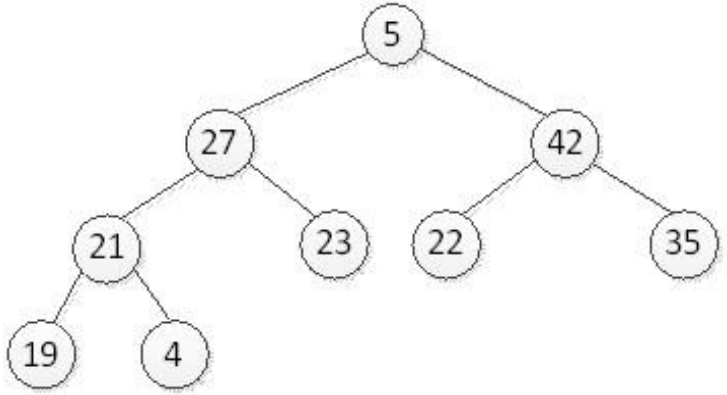
Пирамидальная сортировка (продолжение)

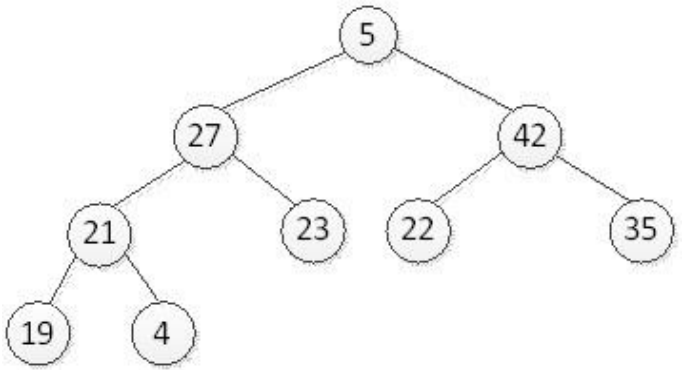
Получаем массив из двух частей: ***неупорядоченной*** и ***упорядоченной.*** Длина неупорядоченной части равна n-1.

5	27	42	21	23	22	35	19	4	45
Неупорядоченная часть – хранит неупорядоченное дерево - «почти кучу»									Упорядоченная часть

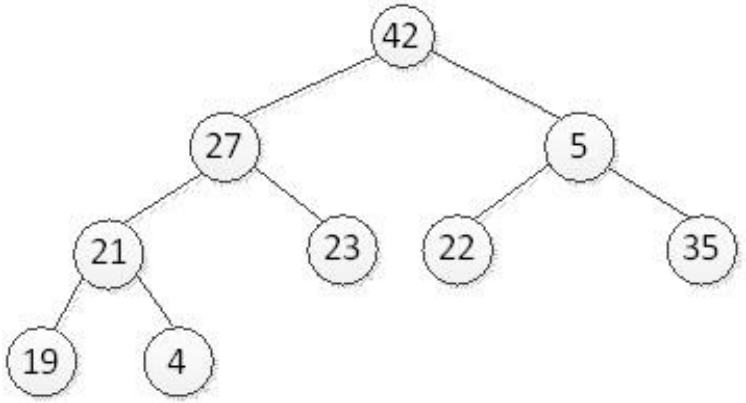
Надо снова выполнить восстановление кучи (просеивание).

Так как в корне находится значение меньше значений потомков корня, то правило кучи нарушено и кучу надо восстановить.





Значение 5 находится в корневом узле. Удаляется узел, содержащий ранее 5. После этих корректировок дерево является «почти кучей». Надо привести его к куче, для этого значение корня спускаем вниз «по дереву», обменивая с потомком, имеющим большее значение.



Спуск вниз значения 5 (шаг 1)

Обмениваются 5 и 42. Снова дерево не является кучей, так как 5 меньше своих наследников. Изменения в дереве указывают, какие элементы обмениваются: с индексом 0 (i=0) и его левым потомком с индексом 2 (j=2i+2).

42 27 5 21 23 22 35 19 4 45

Алгоритм пирамидальной сортировки

- Создаем бинарную кучу размером n на месте исходного массива, перемещая элементы
- На i-ом шаге обмениваем приоритетный элемент кучи из позиции 0 с элементом в позиции n-i-1
- Размер кучи уменьшается на единицу и приоритетный элемент занял свое место в соответствии с правилом упорядочения.

Сложность алгоритма

Создание кучи O(1)

Вставка n элементов O(nlogn)

Извлечение n приоритетных элементов O(nlogn)

Общая сложность алгоритма O(nlogn)

Пирамидальная сортировка (продолжение)

Анализ пирамид

При первоначальном превращении списка в пирамиду осуществляется создание множества пирамид меньшего размера. **Для каждого внутреннего узла дерева строится пирамида с корнем в этом узле**. Если дерево содержит N элементов, то в дереве O(N) внутренних узлов, и в итоге **приходится создать O(N) пирамид**.

При создании каждой пирамиды может потребоваться продвигать элемент вниз по пирамиде, возможно до тех пор, пока он не достигнет концевого узла. Самые высокие из построенных пирамид будут иметь высоту порядка O(log(N)). Так как создается O(N) пирамид, и для построения самой высокой из них требуется O(log(n)) шагов, то **все пирамиды можно построить за время порядка O(N * log(N)).**

На самом деле времени потребуется еще меньше — порядка O(N).

Быстрые сортировки, реализованные по методу «Разделяй и властвуй»

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

Сортировки методом слияния

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.
3. **Комбинирование** решения исходной задачи из решений вспомогательных задач.

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.

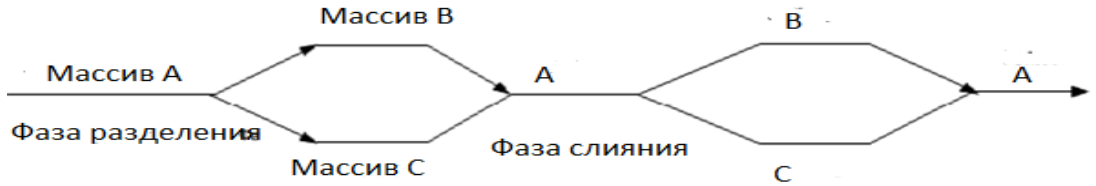
Рекурсия достигает своего нижнего предела, когда длина сортируемой последовательности становится равной 1.

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 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 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: $cn/2^k$.

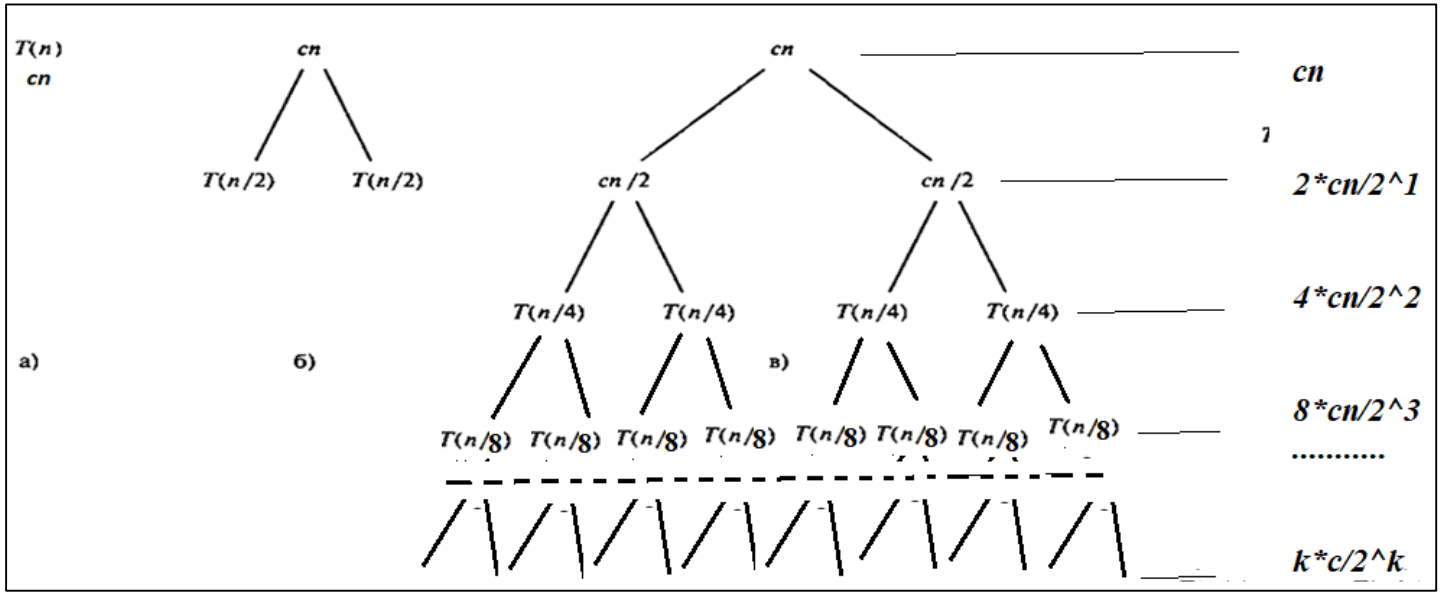
Время выполнения на каждом из уровней: c*n.

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.

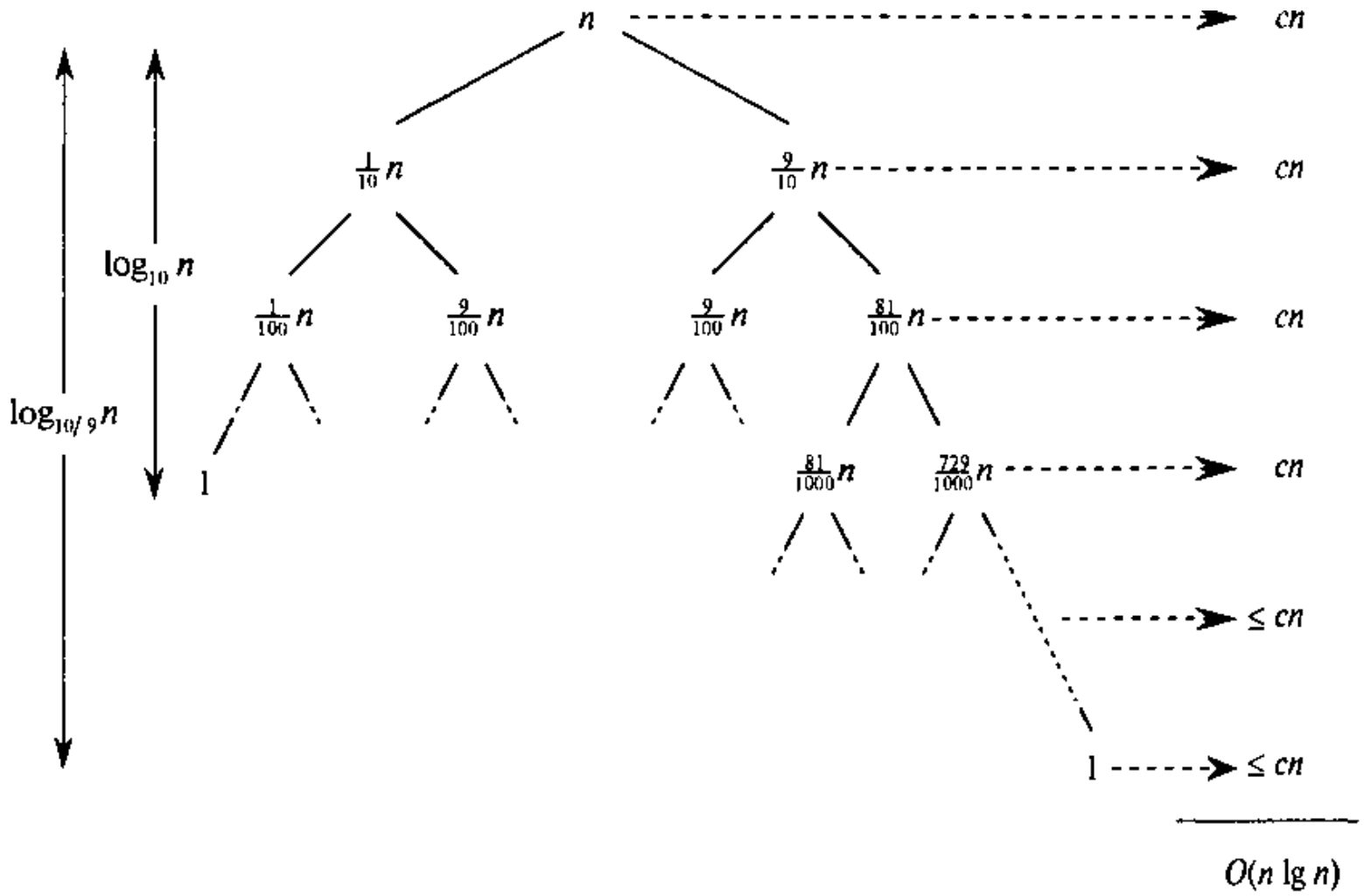
В этом случае для времени работы алгоритма быстрой сортировки мы получим следующее рекуррентное соотношение:

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

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):

Алгоритм рандомизированной быстрой сортировки

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)

Сравнение алгоритмов быстрых сортировок

Сортировки за линейное время

- Подсчета
- Карманная
- Поразрядная

Виды сортировок

- Сортировки сравнением, использующие сравнение входных данных
- Сортировки, выполняемые за линейное время, не используют сравнение входных элементов

Сортировка подсчетом(counting sort)

Постановка задачи

Дано. Массив A из n элементов. Элементы – целые числа, принадлежащие интервалу от 0 до k, где k – некоторая целая константа и 0<=A[i]< =k.

Если k= O(n), т.е. k <=n то время работы алгоритма сортировки подсчетом равно θ(n).

Результат. Массив отсортированный по заданному правилу f.

Ограничения на данные. 0<=A[i]< =k.

Основная идея сортировки подсчетом заключается в том, чтобы для каждого входного элемента x **определить количество элементов, которые меньше *x*.**

На основании этого: элемент x можно разместить на той позиции выходного массива, где он должен находиться.

Например, если всего имеется 17 элементов, которые меньше x, то в выходной последовательности элемент x должен занимать 18-ю позицию. Если допускается ситуация, когда несколько элементов имеют одно и то же значение, эту схему нужно модифицировать, поскольку невозможно разместить все такие элементы в одной и той же позиции.

Алгоритм сортировки подсчетом

На вход подается массив $A[1..n]$, так что $length\ [A] = n$.
Требуются два дополнительных массива: $B[1..n]$ будет содержать отсортированную выходную последовательность, массив *и* $C\ [0..k]$ служит временным рабочим хранилищем – массивом счетчиков – элемент массива C хранит *количество вхождений элемента A[i] в массив.*

Модель выполнения алгоритма сортировки подсчетом

A	2	5	3	0	2	3	0	3
	1	2	3	4	5	6	7	8

C	0	0	0	0	0	0
	0	1	2	3	4	5

После выполнения строки 4 алгоритма В C[i] количество значений числа i в массиве A

C	2	0	2	3	0	1
	0	1	2	3	4	5

После выполнения строки 7: В $C[i]$ – количество значений в массиве A, не превышающих i.

C	2	2	4	7	7	8
	0	1	2	3	4	5

После выполнения строки 11: В массиве В результат сортировки

B	0	0	2	2	3	3	3	5
	1	2	3	4	5	6	7	8

Counting_Sort (A, B, k)

1. for i ← 0 to k
2. do C[i] ← 0
3. for j ← 1 to length[A]
4. do C[A[j]] ← C[A[j]] + 1
5. { В C[i] хранится количество элементов, равных i}
6. for i ← 1 to k
7. do C[i] ← C [i] + C[i - 1]
8. { В C[i]] ? количество элементов, не превышающих i}
9. for j ← length[A] downto 1
10. do B[C[A[j]]] ← A[j]
11. C[A[j]] ← C[A[j]] – 1

Описание выполнения алгоритма

Если все n элементов различны, то при первом переходе к строке 9 для каждого элемента $A[j]$ в переменной $C[A[j]]$ хранится корректный индекс конечного положения этого элемента в выходном массиве, поскольку имеется $C[A[j]]$ элементов, меньших или равных $A[j]$.

Если разные элементы могут иметь одни и те же значения, помещая значение $A[j]$ в массив B, значение $C[A[j]]$ каждый раз уменьшается на единицу. Благодаря этому следующий входной элемент, значение которого равно $A[j]$ (если таковой имеется), в выходном массиве размещается непосредственно перед элементом A $[j]$.

Анализ алгоритма сортировки Подсчетом

На выполнение цикла for в строках 1-2 затрачивается время $\theta(k)$.

На выполнение цикла for в строках 3-4 время $\theta(n)$.

Цикл в строках 6-7 требует $\theta(k)$ времени.

Цикл в строках 9-11 - $\theta(k)$.

Таким образом, полное время можно записать как $\theta(k+n)$.

На практике сортировка подсчетом применяется, когда k = O(n), а в этом случае время работы алгоритма равно $\theta(n)$.

Временная сложность $\theta(k+n)$.

Сложность по памяти (дополнительная память) $\theta(k+2n)$.

Карманная сортировка

Если при сортировке методом подсчета предполагается, что входные данные состоят из целых чисел, принадлежащих небольшому интервалу,

То при карманной сортировке предполагается, **что входные числа генерируются случайным процессом и равномерно распределены в интервале [0,1) .**

Карманная сортировка, как и сортировка подсчетом, работает быстрее, чем алгоритмы сортировки сравнением.

Это происходит благодаря определенным предположениям о входных данных.

Идея карманной сортировки:

разбить интервал [0,1) на ***n*** *одинаковых интервалов (или* ***карманов*** *(buckets)), а затем распределить по этим карманам* ***n*** *входных величин.*

Поскольку входные числа равномерно распределены в интервале [0,1), мы предполагаем, что в каждый из карманов попадет ***не*** много элементов.

Чтобы получить выходную последовательность, нужно просто выполнить сортировку чисел в каждом кармане, а затем последовательно перечислить элементы каждого кармана.

Карманная сортировка

Постановка задачи

Дано. Массив A из n чисел, принадлежащих интервалу [0..1)

Результат. Массив A, отсортированный по правилу f.

Ограничения на данные. 0<=A[i]<1

Идея алгоритма

Используется дополнительный массив связанных списков (карманов) *B* [0..*9*].

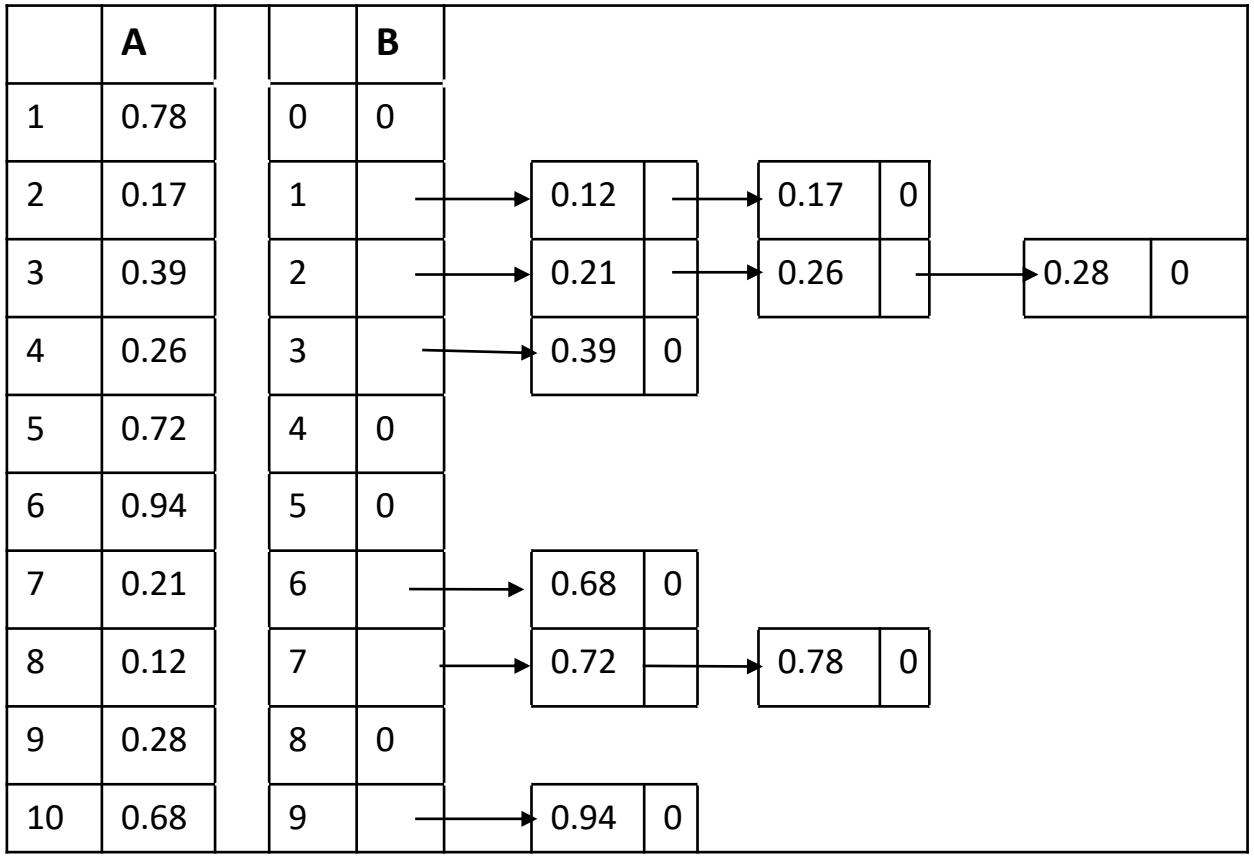
В *i*-м кармане содержатся величины, принадлежащие полуоткрытому интервалу [*i*/10, (*i*+1)/10).

i-ый элемент массива содержит ссылку на вершину списка который хранит числа массива A, начинающиеся с цифры i.

После размещения значений массива в списках, список будет отсортирован методом Простой вставки (что в лучшем случае дает сложность O(n) в худшем O(n^2)).

Отсортированная выходная последовательность представляет собой объединение списков *B*[0], *B*[1],..., *B*[9].

Карманная сортировка. Пример выполнения карманной сортировки



Алгоритм **Bucket_Sort (A)** n <- length[A] for i <-1 to n do Вставить элемент A[i] в список B[⌊nA[i]⌋] od for i ⍰ 0 to n – 1 do Сортировка вставкой списка B[i] Объединение списков B[0], B[1],..., B[n -1] od	 **n+1** **n** **n+1** **n или n^2** **b**

Анализ Карманная сортировка

Так как используется метод сортировки вставками для сортировки списков, то это может привести в худшем случае к $O(n^2)$ или к лучшему O(n).

При вставке элементов в упорядоченный список. Вставка элемента в список осуществляется за время O(1).

Тогда в лучшем случае временная сложность: O(n)/

Емкостная сложность: O(2n+n)

Упражнения

1. Проиллюстрируйте обработку алгоритмом Bucket_Sort массива ***A=***(0.79,0.13,0.16,0.64,0.39,0.20,0.89,0.53,0.71,0.42).
2. Чему равно время работы алгоритма карманной сортировки в наихудшем случае?
3. Какое простое изменение следует внести в этот алгоритм, чтобы его ожидаемое время работы осталось линейным, а время работы в наихудшем случае стало равным ***O (nlgn)?***

Поразрядная сортировка

Особенности

Во-первых, он совсем не использует сравнений сортируемых элементов.

Во-вторых, ключ, по которому происходит сортировка, необходимо разделить на части, *разряды* ключа.

Например, слово можно разделить по буквам, а число - по цифрам...

До выполнения сортировки необходимо знать два параметра: *k* и *m*, где

k - количество разрядов в самом длинном ключе

m - разрядность данных: количество возможных значений разряда ключа

При сортировке русских слов m = 33, так как буква может принимать не более 33 значений. Если в самом длинном слове 10 букв, k = 10.

Аналогично, например, для шестнадцатеричных чисел m=16, если в качестве разряда брать цифру, и m=256, если использовать побайтовое деление.

Эти параметры нельзя изменять в процессе работы алгоритма. В этом - еще одно отличие метода от вышеописанных.

Модель поразрядной сортировки

Дан массив двузначных чисел: 27 13 44 25 10 2

Сортировать по возрастанию.

Создаем 9 очередей (по количеству цифр и их номеру)

1) Сортировка по младшей цифре. 0: 10 1 2 2 3 13 4 44 5 25 6 27 7 8 9	2) Сливаем в один список обходя очереди от 0 к 9 2 10 13 44 25 27	3) По старшей цифре 2 10 13 25 27 44	4) Сливаем 2 10 13 15 27 44

Поразрядная сортировка на списках

Пусть элементы линейного списка L есть k-разрядные десятичные числа, разрядность максимального числа известна заранее.

Обозначим d(j,n) - j-ю справа цифру числа n, которую можно выразить как

$d(j,n) = [\ n / 10^{j-1}\ ] \% 10$

Пусть L0, L1,..., L9 - вспомогательные списки (карманы), вначале пустые. Поразрядная сортировка состоит из двух процессов, выполняемых для j=1,2,...,k:

- распределение
- сборка

Фаза распределения разносит элементы L по карманам: элементы li списка L последовательно добавляются в списки Lm, где m = d(j, li). Таким образом получаем десять списков, в каждом из которых j-тые разряды чисел одинаковы и равны m.

Дано. Список, содержащий одно и двухразрядные числа.

Результат. Список узлы которого отсортированный по возрастанию чисел

Фаза распределения разносит элементы L по карманам: элементы li списка L последовательно добавляются в списки Lm, где m = d(j, li). Таким образом получаем десять списков, в каждом из которых j-тые разряды чисел одинаковы и равны m.

Рассмотрим пример работы алгоритма на входном списке

0 => 8 => 12 => 56 => 7 => 26 => 44 => 97 => 2 => 37 => 4 => 3 => 3 => 45 => 10.

Максимальное число содержит две цифры, значит, разрядность данных k=2.

Первый проход j=1 (по младшей цифре)

L0	0**=>**10
L1	пусто
L2	12=>2
L3	3=>3
L4	44=>4
L5	45
L6	56=>26
L7	7=>97=>37
L8	8
L9	пусто

Фаза сборки состоит в объединении списков L0, L1,..., L9 в общий список

L = L0 => L1 => L2 => ... => L9

Сборка:соединяем списки Li один за другим

L: 0 => 10 => 12 => 2 => 3 => 3 => 44 => 4 => 45 => 56 => 26 => 7 => 97 => 37 => 8

Сборка: соединяем списки Li один за другим
L: 0 => 10 => 12 => 2 => 3 => 3 => 44 => 4 => 45 => 56 => 26 => 7 => 97 => 37 => 8
Второй проход, j=2. Распределение по второй справа цифре

L0	0**=>2=>3=>3=>4=>7=>8**
L1	10=> 12
L2	26
L3	37
L4	44=>45
L5	56
L6	
L7	
L8	
L9	97

Сборка: 0=>2=>3=>3=>4=>7=>8 => 10=> 12=>26 =>37 =>44 =>45 =>56 =>97
Сортировку можно организовать так, чтобы не использовать дополнительной памяти для карманов, т.е элементы списка не перемещать, а с помощью перестановки указателей присоединять их к тому или иному карману.

```
typedef struct slist_ {
  long val;
  struct slist_ *next;
} slist;
// функция сортировки возвращает указатель на начало отсортированного списка
slist *radix_list(slist *l, int t) {
  //  t - разрядность (максимальная длина числа)
  int i, j, d, m=1;
  slist *temp, *out, *head[10], *tail[10];
  out=l;
```

Параметры head[i], tail[i] указывают соответственно на первый и на последний элементы кармана Li.

При этом сами карманы являются "виртуальными", память под них не выделяется.

```
  for (j=1; j<=t; j++) {
    for (i=0; i<=9; i++)
      head[i] = (tail[i]=NULL);
  while ( l != NULL ) {
      d = ((int)(l->val/m))%(int)10;
      temp = tail[d];
      if ( head[d]==NULL ) head[d] = l;
      else temp->next = l;
      temp = tail[d] = l;
      l = l->next;
      temp->next = NULL;
    }
    for (i=0; i<=9; i++)
      if ( head[i] != NULL ) break;
    l = head[i];
    temp = tail[i];
    for (d=i+1; d<=9; d++) {
      if ( head[d] != NULL) {
        temp->next = head[d];
        temp = tail[d];
      }
    }
    m*=10;
  }
  return (out);
}
```

Оценка сложности времени выполнения

Лучшая	Средняя	Худшая
n * k/m	n * k/m	n * k/m

Емкостная сложность: 10+n=n