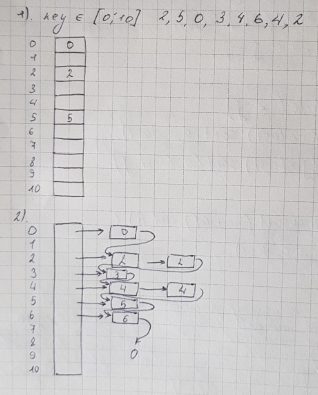
**9. Карманная сортировка.**

Если сортируемые ключами являются целые числа о которых заранее известно (к какому интервалу все эти ключи принадлежат) то сортировку можно провести со сложностью О(n).

Для этого заводят массив, индексы которого соответствуют всем возможным значениям ключей. Каждый элемент массива карман, в карман помещается соответствующий элемент последовательности.

За 1 проход, после чего вся последовательность становится отсортированной

Пример:



**10. Задача поиска Прямой поиск**

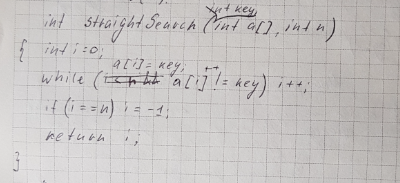
Не менее важной чем сортировка является задача поиска эл последовательности по некоторому ключу

**Прямой поиск**

Самый простой способ поиска по ключу является перебор всей последовательности элементов пока требуемый ключ не найдет или не будет просмотренная вся последовательность

Сложность O(n)

Код:



В данной функции непосредственно за поиск отвечает лишь 2 часть условия a[i]!=key

А первое i<n, на каждом шаге проверяется.

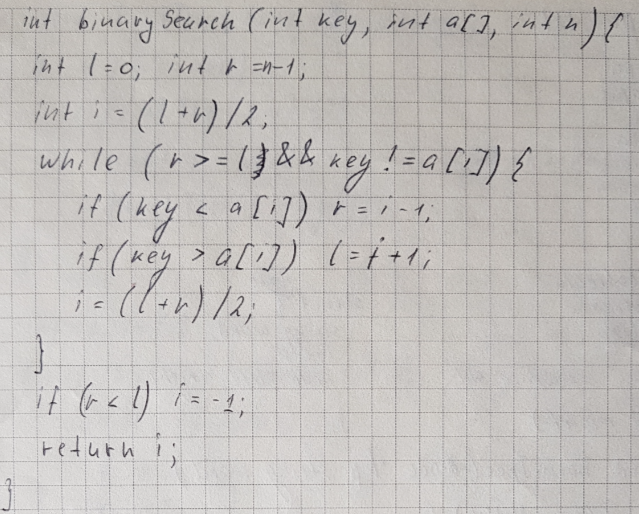
Чтобы избавится от первой проверки добавим в массиве еще информац. эл n перед началом поиска куда будем помещать ключа

**11. Поиск в упорядоченной таблице. Бинарный поиск.**

Производится на отсортированной последовательности в которой выбирается средний элемент и ключ сравнивается с ним. **Если ключ <среднего поиск продолжается. В левой половине иначе в правой**

Сложность метода O(log2n)

Код:

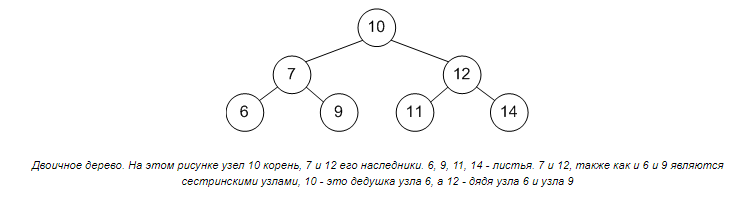


**12. Поиск по бинарному дереву. Дерево поиска.**

Дерево поиска формируется по след принципу: пусть имеется некоторая последовательность ключей, которые нужно поместить в дерево (1й ключ становится корнем) **элементы которые больше корня размещаются справа, которые меньше слева**

И так далее пока рекурсивно до тех пор, пока не будет достигнут пустой элемент куда и вставляется очередной ключ. Т.О. дерево поиска – бинарное дерево, для каждого узла которого выполняется правило.

**Все элементы левого поддерева узла имеют меньшее значение чем данный узел, а правого поддерева- большее**



Недостатком дерева поиска ялвяет то что в худшем случае оно вырождается в линейный список. И сложность в таком дереве O(n) хотя в лучшем случае O(log2n). Для того чтобы сложность поиска была лучшей необходимо иметь сбалансированное дерево поиска.

Код:

**void insert(Node \*\*head, int value) {**

**Node \*tmp = NULL;**

**Node \*ins = NULL;**

**if (\*head == NULL) {**

**\*head = getFreeNode(value, NULL);**

**return;**

**}**

**tmp = \*head;**

**while (tmp) {**

**if (CMP\_GT(value, tmp->data)) {**

**if (tmp->right) {**

**tmp = tmp->right;**

**continue;**

**} else {**

**tmp->right = getFreeNode(value, tmp);**

**return;**

**}**

**} else if (CMP\_LT(value, tmp->data)) {**

**if (tmp->left) {**

**tmp = tmp->left;**

**continue;**

**} else {**

**tmp->left = getFreeNode(value, tmp);**

**return;**

**}**

**} else {**

**exit(2);**

**}**

**}**

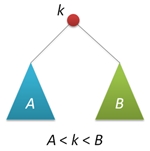
**}**

13. АВЛ-деревья.

АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм. Для простоты дальнейшего изложения будем считать, что все ключи в дереве целочисленны и не повторяются.

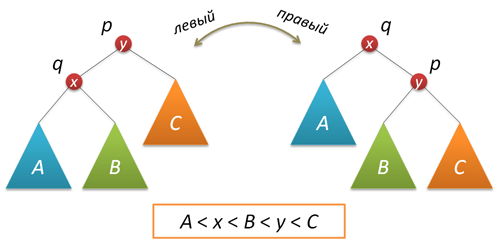
Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле:**для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу**.

Этот метод предложили Адельсон Вельский, для поддержания баланса необходимо добавить 2 дополнительных бита на узел что гарантирует сложность в O(log2n)



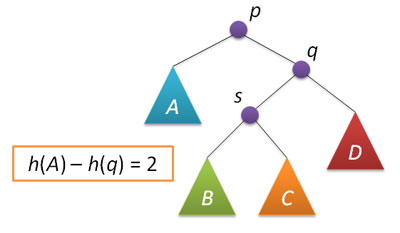
**Балансировка узлов**

В процессе добавления или удаления узлов в АВЛ-дереве возможно возникновение ситуации, когда balance factor некоторых узлов оказывается равными 2 или -2, т.е. возникает расбалансировка поддерева. Для выправления ситуации применяются хорошо нам известные повороты вокруг тех или иных узлов дерева. Напомню, что простой поворот вправо (влево) производит следующую трансформацию дерева:

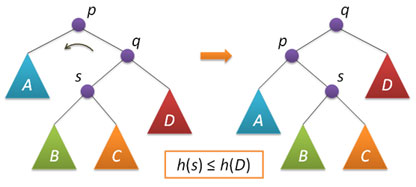


**Дизбаланс узлов**

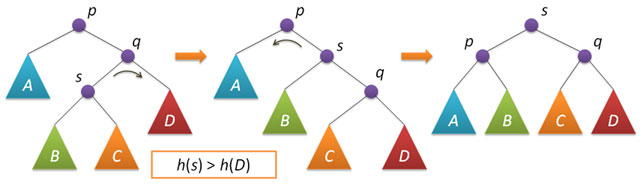
Рассмотрим теперь ситуацию дисбаланса, когда высота правого поддерева узла p на 2 больше высоты левого поддерева (обратный случай является симметричным и реализуется аналогично). Пусть q — правый дочерний узел узла p, а s — левый дочерний узел узла q.



Анализ возможных случаев в рамках данной ситуации показывает, что для исправления расбалансировки в узле p достаточно выполнить либо простой поворот влево вокруг p, либо так называемый большой поворот влево вокруг того же p. Простой поворот выполняется при условии, что высота левого поддерева узла q больше высоты его правого поддерева: h(s)≤h(D)



Большой поворот применяется при условии h(s)>h(D) и сводится в данном случае к двум простым — сначала правый поворот вокруг q и затем левый вокруг p.



Вставка ключей

Вставка нового ключа в АВЛ-дерево выполняется, по большому счету, так же, как это делается в простых деревьях поиска: спускаемся вниз по дереву, выбирая правое или левое направление движения в зависимости от результата сравнения ключа в текущем узле и вставляемого ключа. Единственное отличие заключается в том, что при возвращении из рекурсии (т.е. после того, как ключ вставлен либо в правое, либо в левое поддерево, и это дерево сбалансировано) выполняется балансировка текущего узла. Строго доказывается, что возникающий при такой вставке дисбаланс в любом узле по пути движения не превышает двух, а значит применение вышеописанной функции балансировки является корректным.

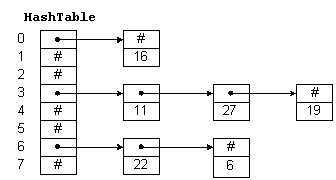
**14. Хеш-таблицы. Открытое хеширование.**

*Хеш-таблица* - это обычный массив с необычной адресацией, задаваемой хеш-функцией.

Например, на **hashTable** рис. 3.1 - это массив из 8 элементов. Каждый элемент представляет собой указатель на линейный список, хранящий числа. Хеш-функция в этом примере просто делит ключ на 8 и использует остаток как индекс в таблице. Это дает нам числа от 0 до 7. Поскольку для адресации в **hashTable** нам и нужны числа от 0 до 7, алгоритм гарантирует допустимые значения индексов.

Хеширование полезно, когда широкий диапазон возможных значений должен быть сохранен в малом объеме памяти, и нужен способ быстрого, практически произвольного доступа. Хэш-таблицы часто применяются в базах данных, и, особенно, в языковых процессорах типа компиляторов и ассемблеров, где они изящно обслуживают таблицы идентификаторов. В таких приложениях, таблица - наилучшая структура данных.

Так как всякий доступ к таблице должен быть произведен через хэш-функцию, функция должна удовлетворять двум требованиям: Она должна быть быстрой, и она должна порождать хорошие ключи для распределения элементов по таблице. Последнее требование минимизирует коллизии (случаи, когда два разных элемента имеют одинаковое значение хеш-функции) и предотвращает случай, когда элементы данных с близкими значениями попадают только в одну часть таблицы.



Чтобы вставить в таблицу новый элемент, мы хешируем ключ, чтобы определить список, в который его нужно добавить, затем вставляем элемент в начало этого списка. Например, чтобы добавить 11, мы делим 11 на 8 и получаем остаток 3. Таким образом, 11 следует разместить в списке, на начало которого указывает hashTable[3]. Чтобы найти число, мы его хешируем и проходим по соответствующему списку. Чтобы удалить число, мы находим его и удаляем элемент списка, его содержащий.

При добавлении элементов в хеш-таблицу выделяются куски динамической памяти, которые организуются в виде связанных списков, каждый из которых соответствует входу хеш-таблицы. Этот метод называется связыванием.

**15. Хеш-таблицы. Прямое закрытое хеширование. Методы разрешения коллизий.**

При закрытом (внутреннем) *хешировании* в хеш-таблице хранятся непосредственно сами элементы, а не заголовки списков элементов. Поэтому в каждой записи (сегменте) может храниться только один элемент. При закрытом *хешировании* применяется методика *повторного хеширования*. Если осуществляется попытка поместить элемент х в сегмент с номером h(х), который уже занят другим элементом (коллизия), то в соответствии с методикой повторного *хеширования* выбирается последовательность других номеров сегментов h1(х),h2(х),..., куда можно поместить элемент х. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то, следовательно, таблица заполнена, и элемент х добавить нельзя.

При поиске элемента х необходимо просмотреть все местоположения h(x),h1(х),h2(х),..., пока не будет найден х или пока не встретится пустой сегмент. Чтобы объяснить, почему можно остановить поиск при достижении пустого сегмента, предположим, что в хеш-таблице не допускается удаление элементов. Пусть h3(х) – первый пустой сегмент. В такой ситуации невозможно нахождение элемента х в сегментах h4(х),h5(х) и далее, так как при вставке элемент х вставляется в первый пустой сегмент, следовательно, он находится где-то до сегмента h3(х). Но если в хеш-таблице допускается удаление элементов, то при достижении пустого сегмента, не найдя элемента х, нельзя быть уверенным в том, что его вообще нет в таблице, так как сегмент может стать пустым уже после вставки элемента х. Поэтому, чтобы увеличить эффективность данной реализации, необходимо в сегмент, который освободился после *операции удаления элемента*, поместить специальную константу, которую назовем, например, *DEL*. В качестве альтернативы специальной константе можно использовать дополнительное поле таблицы, которое показывает состояние элемента. Важно различать константы *DEL* и NULL – последняя находится в сегментах, которые никогда не содержали элементов. При таком подходе выполнение поиска элемента не требует просмотра всей хеш-таблицы. Кроме того, при вставке элементов сегменты, помеченные константой *DEL*, можно трактовать как свободные, таким образом, пространство, освобожденное после удаления элементов, можно рано или поздно использовать повторно. Но если невозможно непосредственно сразу после удаления элементов пометить освободившиеся сегменты, то следует предпочесть закрытому *хешированию* схему открытого *хеширования*.

Существует несколько методов повторного *хеширования*, то есть определения местоположений h(x),h1(х),h2(х),...:

* линейное опробование;
* квадратичное опробование;
* двойное *хеширование*.

*Линейное опробование* сводится к *последовательному перебору* сегментов таблицы с некоторым фиксированным шагом:

адрес=h(x)+ci,

где i – номер попытки разрешить коллизию;

c – константа, определяющая шаг перебора.

При шаге, равном единице, происходит *последовательный перебор* всех сегментов после текущего. *Квадратичное опробование*отличается от линейного тем, что шаг перебора сегментов нелинейно зависит от номера попытки найти свободный сегмент:

адрес=h(x)+ci+di2,

где i – номер попытки разрешить коллизию,

c и d – *константы*.

Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов. Однако даже относительно небольшое число проб может быстро привести к выходу за *адресное пространство* небольшой таблицы вследствие квадратичной зависимости адреса от номера попытки.

Еще одна разновидность метода открытой адресации, которая называется *двойным хешированием*, основана на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций:

адрес=h(x)+ih2(x).

Очевидно, что по мере заполнения хеш-таблицы будут происходить *коллизии*, и в результате их разрешения очередной *адрес* может выйти за пределы *адресного пространства* таблицы. Чтобы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым *хеш-функцией*. С одной стороны, это приведет к сокращению числа *коллизий* и ускорению работы с хеш-таблицей, а с другой – к нерациональному расходованию памяти. Даже при увеличении длины таблицы в два раза по сравнению с областью значений *хеш-функции* нет гарантии того, что в результате *коллизий* *адрес* не превысит длину таблицы. При этом в начальной части таблицы может оставаться достаточно свободных сегментов. Поэтому на практике используют циклический переход к началу таблицы.

Однако в случае многократного превышения *адресного пространства* и, соответственно, многократного циклического перехода к началу будет происходить просмотр одних и тех же ранее занятых сегментов, тогда как между ними могут быть еще свободные *сегменты*. Более корректным будет использование сдвига адреса на 1 в случае каждого циклического перехода к началу таблицы. Это повышает *вероятность* нахождения свободных сегментов.

В случае применения схемы закрытого *хеширования* скорость выполнения вставки и других операций зависит не только от равномерности распределения элементов по сегментам *хеш-функцией*, но и от выбранной методики повторного *хеширования*(опробования) для разрешения *коллизий*, связанных с попытками вставки элементов в уже заполненные *сегменты*. Например, методика линейного опробования для разрешения *коллизий* – не самый лучший выбор.

Как только несколько последовательных сегментов будут заполнены, образуя группу, любой новый элемент при попытке вставки в эти *сегменты* будет вставлен в конец этой группы, увеличивая тем самым длину группы последовательно заполненных сегментов. Другими словами, для поиска пустого сегмента в случае непрерывного расположения заполненных сегментов необходимо просмотреть больше сегментов, чем при случайном распределении заполненных сегментов. Отсюда также следует очевидный *вывод*, что при непрерывном расположении заполненных сегментов увеличивается *время выполнения* вставки нового элемента и других операций.

### Ключевые термины

**Вторичные ключи** – это ключи, не позволяющие однозначно идентифицировать *запись* в таблице.

**Закрытое хеширование или Метод открытой адресации** – это технология разрешения *коллизий*, которая предполагает хранение записей в самой хеш-таблице.

**Коллизия** – это ситуация, когда разным ключам соответствует одно *значение* *хеш-функции*.

**Коэффициент заполнения хеш-таблицы** – это количество хранимых элементов массива, деленное на число возможных значений *хеш-функции*.

**Открытое хеширование или Метод цепочек** – это технология разрешения *коллизий*, которая состоит в том, что *элементы множества* с равными хеш-значениями связываются в цепочку-*список*.

**Первичные ключи** – это ключи, позволяющие однозначно идентифицировать *запись*.

**Повторное хеширование** – это *поиск* местоположения для очередного элемента таблицы с учетом шага перемещения.

**Пространство записей** – это множество тех ячеек памяти, которые выделяются для хранения таблицы.

**Пространство ключей** – это множество всех теоретически возможных значений ключей записи.

**Синонимы** – это совпадающие ключи в хеш-таблице.

**Хеширование** – это преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины.

**Хеш-таблица** – это *структура данных*, реализующая *интерфейс* ассоциативного массива, то есть она позволяет хранить пары вида "*ключ*- *значение*" и выполнять три *операции*: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

**Хеш-таблицы с прямой адресацией** – это хеш-таблицы, использующие инъективные *хеш-функции* и не нуждающиеся в механизме разрешения *коллизий*.

**16. Эффективность алгоритмов. NP-полные и труднорешаемые задачи.**

**Полиномиальные алгоритмы и труднорешаемые задачи**

Существует целый класс задач, которые попадают в категорию теоретически разрешимых, но практически неосуществимых. Для нас нет особой разницы между программой, не заканчивающей своей работы никогда, и программой, работающей 1000 лет. Это класс - класс "переборных задач". Прежде чем дать его описание, приведем пример одной задачи такого рода.

Имеется набор грузов известного веса и машина известной грузоподъемности (вес грузов и грузоподъемность машины - целые числа). Мы должны определить, можно ли полностью загрузить машину, поместив в нее некоторые из имеющихся грузов, т.е. существует ли набор грузов с суммарным весом, равным грузоподъемности машины.

Теоретическая разрешимость этой задачи очевидна. Если у нас имеется n грузов, то возможных комбинаций этих грузов будет 2n. Перепробовав их все, можно узнать, есть ли среди них комбинация, полностью загружающая машину. Оценим число операций, необходимых для выполнения этого алгоритма. Если n=100, то 2n = 2100 = (210)10 ≈ 100010 = 1030, так что нужно перебрать около 1030вариантов. Сколько же времени будет работать машина? Пусть мы располагаем компьютером, который выполняет миллиард (109) операций в секунду. На проверку одного варианта уходит не меньше одной операции. Значит, понадобится не менее 1030/109 = 1021 секунд, что больше 1013 лет. Видно, что с любой практической точки зрения этот алгоритм не пригоден.

Не спасет дело и рост скорости вычислительной техники: если быстродействие компьютера возрастет в 10 раз, то за такое же время мы сможем решать задачу с тремя - четырьмя лишними грузами. Возрастание 2n в 10 раз соответствует увеличению n на три - четыре единицы.

Единственный выход - это изменить алгоритм, избавившись от необходимости перебирать огромное число вариантов. В настоящее время этого никто делать не умеет для описанной задачи. Более того, есть причины полагать, что это в принципе невозможно. Хотя доказать невозможность быстрого решения переборных задач пока никто не может.

Разные алгоритмы имеют различную временную сложность, и выявление того, какие алгоритмы "достаточно эффективны", а какие "совершенно неэффективны", всегда будет зависеть от конкретной ситуации. Однако теоретики, занимающиеся разработкой и анализом алгоритмов, предлагают для сравнения эффективности алгоритмов один простой подход, позволяющий существенно прояснить ситуацию. Речь идет о различии между полиномиальными и экспоненциальными алгоритмами.

***Полиномиальным алгоритмом или алгоритмом полиномиальной временной сложности называется алгоритм, у которого временная сложность равна O(p(n)), где p(n) - некоторая полиномиальная функция, а n - входная длина.***

***Алгоритмы, временная сложность которых не поддается подобной оценке, называются экспоненциальными.****Следует отметить, что это определение включает и такие функции, как n log n, хотя они и не являются полиномиальными, но обычно не считаются экспоненциальными. Большинство экспоненциальных алгоритмов - это просто варианты полного перебора, в то время как полиномиальные алгоритмы обычно можно построить лишь тогда, когда удается более глубоко проникнуть в суть решаемой задачи.*

Эта точка зрения, различающая, с одной стороны, полиномиальные алгоритмы, а с другой стороны, экспоненциальные, является отправным пунктом в определении труднорешаемых задач и теории NP-полных задач.

Вопрос различия между этими двумя классами впервые обсуждался в работах Кобхема и Эдмонса. Так Эдмонс отождествлял полиномиальные алгоритмы с "хорошими" алгоритмами. Согласно такой точки зрения, экспоненциальные алгоритмы не следует считать "хорошими", и чаще всего так оно и есть.

Имеется широко распространенное соглашение, согласно которому задача не считается "хорошо решаемой" до тех пор, пока для нее не получен полиномиальный алгоритм. Поэтому мы будем называть *задачу труднорешаемой, если для ее решения не существует полиномиального алгоритма.*

Но это формальное определение следует рассматривать только как одну из возможных трактовок понятия "труднорешаемая задача". Различие между "эффективными" (полиномиальными) алгоритмами и "неэффективными" (экспоненциальными) алгоритмами может принять совсем иной характер, когда размеры решаемых задач невелики. Даже функция f(n) = 2n ведет себя лучше, чем f(n) = n5 при n ≤ 20. Можно легко построить еще более яркие примеры.

Более того, известны некоторые экспоненциальные алгоритмы весьма хорошо зарекомендовавшие себя на практике. Дело в том, что *временная сложность определена как мера поведения алгоритма в наихудшем случае*, и тот факт, что какой-то алгоритм имеет временную сложность порядка 2n, означает только, что решение по крайней мере одной задачи размера n требует времени порядка 2n. На самом деле может оказаться, что большинство индивидуальных задач требует для своего решения значительно меньших затрат времени, и такого рода ситуация имеет место для нескольких хорошо известных алгоритмов.

Например, симплекс- метод для решения задач линейного программирования имеет экспоненциальную временную сложность, но в тоже время многочисленные свидетельства подтверждают, что этот метод хорошо работает на практике. Другой пример: алгоритмы ветвей и границ столь успешно решают задачу о рюкзаке, что многие исследователи считают эту задачу "хорошо решаемой", хотя алгоритмы ветвей и границ имеют экспоненциальную временную сложность.

К сожалению, подобные примеры очень редки. Хотя экспоненциальные алгоритмы известны для многих задач, немногие из них считаются приемлемыми для практических целей. Даже при наличии успешно работающих экспоненциальных алгоритмов, исследователи не отказались от попыток найти для соответствующих задач полиномиальные алгоритмы. В настоящее время пока не получено удовлетворительных объяснений, почему эти экспоненциальные алгоритмы работают успешно, и не известны методы, позволяющие заранее прогнозировать хорошую работу того или иного экспоненциального алгоритма в практической ситуации.

С другой стороны, полиномиальные алгоритмы часто позволяют делать такого рода прогнозы, поскольку полиномиальные функции более адекватно оценивают время работы алгоритмов. Хотя алгоритмы, имеющие временную сложность типа n100 или 1099n2, не могут считаться эффективными с практической точки зрения, естественно возникающие "полиномиальные задачи" обычно требуют для своего решения (в самом худшем случае) времени порядка n2или n3, причем коэффициенты при старших членах полиномов не слишком велики. Алгоритмы, обладающие такими оценками, можно считать "эффективными", и именно это весьма желательное свойство заставляет отдавать предпочтение полиномиальным алгоритмам как средству решения задач.

Под массовой задачей или просто задачей мы будем понимать некоторый вопрос, на который следует дать ответ. Обычно задача содержит несколько параметров, или свободных переменных, конкретные значения которых не определены*. Задача П определяется следующей информацией*:

1. *общим списком всех ее параметров*,
2. *формулировкой тех свойств, которым должен удовлетворять ответ или, другими словами, решение задачи*.

*Индивидуальная задача I получается из массовой задачи П, если всем параметрам задачи П присвоить конкретные значения.*

Наше определение "труднорешаемой задачи" создает базу для теории, обладающей значительной общностью и большими возможностями. Понятие "труднорешаемой задачи" оказывается независимым от конкретной схемы кодирования и модели ЭВМ, используемых при определении временной сложности.

Рассмотрим сначала схемы кодирования. Стандартные схемы кодирования, используемые на практике для любой конкретной задачи, по-видимому, всегда будут отличаться друг от друга не более чем полиномиальным образом, т.е. получаемые входы отличаются друг от друга не более чем "полиномиальным образом". Это значит, что любой алгоритм, имеющий полиномиальную временную сложность при одной из этих схем кодирования, будет также обладать полиномиальной временной сложностью при всех остальных схемах.

Было бы трудно представить себе разумную схему кодирования для какой-либо задачи, которая отличалась бы более чем полиномиальным образом от стандартных схем. Хотя мы не можем формально выразить такое понятие, как "разумная схема кодирования", следующие два условия охватывают важные требования, связанные с этим понятием:

1. *код любой индивидуальной задачи должен быть "сжатым, т.е. не содержать избыточной информации или символов;*
2. *числа, входящие в условия задачи, должны быть представлены в двоичной системе счисления (или десятичной, или восьмеричной, или иметь любое другое основание, но только не****1****).*

Если мы ограничимся рассмотрением только тех схем кодирования, которые удовлетворяют этим условиям, то выяснение вопроса, является ли данная задача труднорешаемой, не будет зависеть от выбора конкретной схемы кодирования.

Аналогичные замечания можно сделать относительно выбора модели ЭВМ. Все известные в настоящее время реалистические модели ЭВМ (одноленточные и многоленточные машины Тьюринга, машины с произвольным доступом к памяти или равнодоступные адресные машины - РАМ, машины с произвольным доступом к памяти и хранимой программой) эквивалентны относительно полиномиальной временной сложности. Эти модели эквивалентны с точки зрения принципиальной вычислительности, но не с точки зрения скорости вычислений. Можно ожидать, что и любая иная "разумная" модель ЭВМ будет эквивалентна по сложности всем перечисленным моделям.

Под словами "разумная" модель здесь главным образом имеется в виду то, что объем работы, выполняемой машиной в единицу времени, ограничен (сверху) полиномом. Так, например, модель, обладающая способностью выполнять параллельно произвольно много операций, не будет считаться "разумной", и в действительности ни одна из существующих (или проектируемых) ЭВМ не обладает подобным свойством. Во всяком случае, если ограничиться рассмотрением стандартных моделей реальных ЭВМ, то класс труднорешаемых задач не будет зависеть от выбора конкретной модели.

Первый аспект состоит в том, что для отыскания решения требуется экспоненциальное время.

Второй заключается в том, что искомое решение настолько велико, что не может быть представлено в виде выражения, длина которого была бы ограничена полиномом от длины входа. Труднорешаемостью этого вида нельзя ни в коем случае пренебрегать и очень важно ее своевременно обнаружить. В большинстве случаев ее наличие ясно из постановки задачи. Этот аспект труднорешаемости можно рассматривать как указание на то, что постановка задачи не реалистична, поскольку мы запрашиваем больше информации, чем когда либо сможем использовать. Поэтому будем изучать такие задачи, длина решения которых ограничена полиномиальной функцией от длины входной информации. Вторая ситуация возникает, например, если в задаче о коммивояжере в качестве дополнительного параметра фигурирует число В и требуется найти **все** маршруты длины, не превосходящей В. Легко построить экспоненциальное число маршрутов длины, не превосходящей В, поэтому не существует алгоритма с полиномиальной временной сложностью, который все их перечисляет.

Первые результаты о труднорешаемости задач - классические результаты о неразрешимости - были получены Тьюрингом. Около 60 лет назад Тьюринг доказал, что некоторые задачи "неразрешимы" в то смысле, что вообще не существует алгоритма их решения. Он доказал, например, что невозможно указать алгоритм, который по произвольной программе определял бы, остановится ли эта программа на произвольно заданном входе или нет. Известно большое число других неразрешимых задач. К ним относятся, в частности, задача выяснения тривиальности конечно-порожденных групп, десятая проблема Гилберта (разрешимость в целых числах полиномиальных (диофантовых ) уравнений, а также несколько задач о покрытии плоскости равными частями. Поскольку эти задачи не могут быть решены никаким алгоритмом, а тем более полиномиальным алгоритмом и, значит действительно труднорешаемы в самом сильном смысле.

Было показано, что некоторые задачи не могут быть решены за полиномиальное время даже с помощью недетерминированного вычислительного устройства, обладающего способностью параллельно выполнять неограниченное количество независимых вычислений.

Все известные в настоящее время задачи, труднорешаемость которых доказана, попадают в один из классов: они либо вовсе неразрешимы, либо труднорешаемы недетерминированной машиной. Однако большинство представляющихся труднорешаемыми практических задач в действительности разрешимы, и могут быть решены за полиномиальное время с помощью недетерминированного вычислительного устройства.

## **Теория np- полных задач. Класс np задач и np- полные задачи**

Пока теоретики продолжают поиск более мощных методов доказательства труднорешаемости задач, параллельно ведутся работы по сравнению сложности различных задач.

Основной метод, используемый для доказательства того, что две задачи близки, состоит в "сведении" их друг к другу с помощью конструктивного преобразования, которое позволяет превратить любой алгоритм решения второй задачи в соответствующий алгоритм решения первой задачи.

Фундамент теории **NP - полных задач**был заложен в работе С.Кука, опубликованной в 1971 г.

Во - первых, Кук подчеркнул важность понятия "сводимость за полиномиальное время", т.е. сводимость, которая выполняется с помощью алгоритма с полиномиальной временной сложностью. Если одна задача сводится за полиномиальное время к другой, то любой полиномиальный алгоритм решения второй задачи может быть превращен в полиномиальный алгоритм решения первой.

Во - вторых, он обратил внимание на класс задач распознавания свойств (класс NP), которые могут быть решены за полиномиальное время на недетерминированном вычислительном устройстве. *Задачей распознавания свойств называется задача, решениями которой могут быть либо "да", либо "нет".*Большинство не поддающихся решению задач, которые встречаются на практике, после переформулировки их в виде задач распознавания попадают в этот класс.

Класс NP, определяемый неформально, - это класс всех задач распознавания, которые при разумном кодировании могут быть решены недетерминированными алгоритмами за полиномиальное время, точнее недетерминированный алгоритм проверяет за полиномиальное время экспоненциальное число возможностей.

Недетерминированный алгоритм состоит из двух различных стадий - стадии угадывания и стадии проверки. По заданной индивидуальной задаче I на первой стадии происходит просто "угадывание" некоторой структуры S. Затем I и S вместе подают в качестве входа на стадию проверки, которая выполняется обычным детерминированным образом и либо заканчивается ответом "да", либо заканчивается ответом "нет", либо продолжается бесконечно без остановки (последние две возможности различать не обязательно).

Например, недетерминированный алгоритм решения задачи коммивояжера можно было бы построить, используя в качестве стадии угадывания просто выбор произвольной последовательности городов, а в качестве стадии проверки - полиномиальную процедуру "проверки доказательства", т.е. за полиномиальное время проверить, "доказывает" ли этот маршрут, что ответ на вопрос о решении задачи есть "да".

Термином "решает" следует пользоваться осторожно. Должно быть ясно, что основное назначение "полиномиального недетерминированного алгоритма" состоит в объяснении понятия "проверяемость за полиномиальное время", а не в том, чтобы служить реалистическим методом решения задач распознавания свойств. *Именно понятие полиномиальной "проверяемости" позволяет выделить задачи классаNP.*Отметим, что проверяемость за полиномиальное время не влечет разрешимости за полиномиальное время. Утверждая, что за полиномиальное время можно проверить ответ "да" для задачи **коммивояжер**, мы не учитываем время, которое может понадобиться на поиск нужного маршрута среди экспоненциального числа всех возможных маршрутов. Мы лишь утверждаем, что по любому заданному маршруту для индивидуальной задачи I можно за полиномиальное время проверить, "доказывает" ли этот маршрут, что вопрос относительно индивидуальной задачи I есть "да".

В третьих, С. Кук доказал, что одна конкретная задача из NP, называемая *задачей о выполнимости*, обладает тем свойством, что всякая другая задача из класса NP может быть сведена к ней за полиномиальное время, т.е. если задача о выполнимости может быть решена за полиномиальное время, то и любая задача из класса NP полиномиально разрешима, а если какая-то задача из NP труднорешаема, то и задача о выполнимости также должна быть труднорешаемой. Таким образом, задача о выполнимости - "самая трудная" в классе NP.

Наконец, С. Кук предположил, что и другие задачи из класса NP могут быть самыми "трудными" представителями класса NP.

Вслед за этим Р. Карп опубликовал ряд результатов, из которых следует, что многие хорошо известные комбинаторные задачи, включая задачу о коммивояжере, будучи сформулированы в виде задач распознавания, столь же "трудны", как задача о выполнимости. Позднее относительно широкого круга других задач было доказано, что они по трудности эквивалентны этим задачам, а сам класс эквивалентности, состоящий из "самых трудных" задач из NP, получил название "клас NP- полных задач" (часто его называют "класс универсальных задач").

Честь быть "первой" NP-полной задачей выпала на долю задачи распознавания из булевской логики задачи, которую обычно называют "ВЫПОЛНИМОСТЬ".

Условие. Заданы множество переменных U и набор С дизъюнкций над U.

Вопрос. Существует ли выполняющий набор значений истинности для С?

В настоящее время известно более тысячи NP-полных задач. Если бы все доказательства NP-полноты были так же сложны, как доказательство NP-полноты задачи ВЫПОЛНИМОСТЬ, то очень сомнительно, что список NP-полных задач смог бы стать столь обширным, как в настоящее время. Однако, если уже известна одна NP-полная задача, то процедура доказательства NP-полноты других задач значительно упрощается. Хотя теоретически любую из известных NP-полных задач можно наравне с другими выбрать для доказательства NP-полноты новой задачи, на практике оказывается, что некоторые задачи подходят для этой цели гораздо лучше других. Следующие шесть задач входят в число тех, которые используются наиболее часто и для начинающего они могут служить "основным ядром" списка известных NP-полных задач: **3 - ВЫПОЛНИМОСТЬ, ТРЕХМЕРНОЕ СОЧЕТАНИЕ, ВЕРШИННОЕ ПОКРЫТИЕ, КЛИКА, ГАМИЛЬТОНОВ ЦИКЛ, РАЗБИЕНИЕ.**

Зачастую задачи, разрешимые за полиномиальное время, очень незначительно отличаются от NP-полных задач. Например, **трехмерное сочетание** NP-полна, а **паросочетание**разрешима за полиномиальное время.

Среди NP-полных задач очень часто встречаются задачи из теории графов. Большинство задач из теории графов может быть решено за полиномиальное время, если ограничить максимальную степень вершин. Например, если потребовать, чтобы все вершины графа имели степень не более 2, то задачи ГАМИЛЬТОНОВ ЦИКЛ, ВЕРШИННОЕ ПОКРЫТИЕ и РАСКРАШИВАЕМОСТЬ ГРАФА В 3 ЦВЕТА, а также почти любая задача из теории графов, которую можно себе вообразить, тривиально решаются за полиномиальное время.

(Теорема Брукса утверждает, что связный граф с максимальной степенью вершин, равной 3, раскрашиваем в 3 цвета тогда и только тогда, когда он не является полным графом на четырех вершинах).

В настоящее время для большинства задач из класса NP либо известны эффективные алгоритмы решения, либо доказана NP-полнота, но есть и исключения. В списке М. Гэри и Д. Джонсона их одиннадцать. Например, задача о простоте числа.

Вопрос о том, действительно ли NP-полные задачи труднорешаемы, в настоящее время считается одним из основных открытых вопросов современной математики и теоретической кибернетики. Вопреки готовности большинства специалистов считать, что все NP-полные задачи труднорешаемы, прогресс как в доказательстве, так и в опровержении этого далеко идущего предположения весьма незначителен. Однако, несмотря на отсутствие доказательства того, что из NP-полноты следует труднорешаемость, NP-полнота задачи означает, что для ее решения полиномиальным алгоритмом требуется по крайней мере крупное открытие.

С.Кук выделил очень широкий и представительный класс задач, класс NP, к которому относится большинство возникающих на практике задач выбора. Этот класс в себя все задачи выбора, для которых известен эффективный алгоритм решения, подкласс P. Кроме того, в NP Куком выделен подкласс NP-полных задач, которые обладают тем свойством, что построение эффективного алгоритма решения хотя бы для одной из *NP*-полных задач будет означать построение эффективных алгоритмов сразу для всех задач из *NP*.

*Класс задач NP*-полноты,*т.е. NP*-полных задач - *полных для недетерминированного полиномиального времени (Nondeterministically Polinomial) - класс задач, ни для одной из которых не удалось найти алгоритма полиномиальной сложности.*

Простая диаграмма класса *NP*изображена на следующем рисунке.

Взаимоотношение классов *P* и *NP* имеет фундаментальное значение для теории *NP*-полных задач. Одно соотношение заключается в том, что *P ⊆ NP*. Всякая задача распознавания, разрешимая за полиномиальное время детерминированным алгоритмом, разрешима также за полиномиальное время недетерминированным алгоритмом.

Существует широко распространенное мнение, что *P ≠ NP*, хотя доказательство этой гипотезы отсутствует.

# Np-полные задачи и методы их решения

Найдется немного научных терминов, так быстро завоевавших широкую известность, как понятие "*NP*-полная задача". За короткий отрезок времени с момента возникновения этого понятия в начале 70-ых годов оно стало символом громадных трудностей, с которыми все чаще приходится сталкиваться разработчикам алгоритмов по мере того, как они берутся за решение задач постоянно возрастающей размерности и усложняющейся структуры. Развитие вычислительной техники и ее использование при решении практических задач из самых разных областей выявили большое количество задач выбора, для которых, несмотря на предпринимаемые в течение многих лет усилия значительным числом квалифицированных математиков, не удалось построить эффективных алгоритмов решения, т.е. таких, чтобы их трудоемкость выражалась в виде полинома от размерности задачи. В книге М.Гэри и Д.Джонсона "Вычислительные машины и труднорешаемые задачи" приведен список из более чем 300 таких задач, а в настоящее время их известно в несколько раз больше. Некоторые из этих задач, относящиеся, например, к теории чисел, комбинаторике, теории графов были поставлены еще в прошлом веке. Примерами NP-полных задач являются: *нахождение гамильтонова пути, гамильтонова цикла, задача коммивояжера, упаковка в контейнеры, расписание многопроцессорное, составление учебного расписания, задача о рюкзаке, составление кроссворда.*

Большинство дискретных и комбинаторных задач допускает решение с помощью некоторого процесса перебора. Такие задачи называются переборными. Однако число шагов переборного метода растет экспоненциально в зависимости от размера задачи. Как правило, в переборных задачах имеется конечное множество вариантов, среди которых нужно найти решение. Так, в задаче о рюкзаке решение отыскивается среди 2nбулевских векторов длины n, и перебрав это экспоненциальное множество векторов, мы обязательно решим задачу. Но с ростом n число векторов растет быстро и задача становится "труднорешаемой", т.е. практически неразрешимой. Поэтому в конечном счете аналогом алгоритмической неразрешимости является перебор экспоненциального числа вариантов, а аналогом алгоритмической разрешимости - существование алгоритма существенно более экономичного, чем перебор. Стало общепринятым считать (переборную) задачу решаемой эффективно, если имеется алгоритм, решающий ее за время, ограниченное полиномом от "размера задачи".

Из приведенных рассуждений следует, что встретившись с новой задачей, естественно сначала задать вопрос: "Можно ли рассматриваемую задачу решить полиномиальным алгоритмом?".

Если ответ на этот вопрос оказывается положительным, то с точки зрения теории NP-полноты ничего большего о задаче сказать нельзя. Дальнейшие усилия мы можем сконцентрировать на поиске как можно более эффективных полиномиальных алгоритмов.

Однако, если для решения задачи не известно полиномиального алгоритма, возникает второй вопрос: "Верно ли, что рассматриваемая задача NP-полна?" Заметим, что в некоторых случаях легко устанавливается полиномиальная разрешимость интересующей нас задачи, в других же оказывается очевидной ее NP-полнота. В большинстве случаев, однако найти ответ на каждый из поставленных вопросов довольно трудно. Обычно совсем не очевидно, что задача разрешима за полиномиальное время, или что она NP-полна, и требуется некоторая работа для выяснения, какая из этих двух возможностей в действительности реализуется, если реализуется.

Таким образом, при анализе задач лучше пользоваться двусторонним подходом. Пытаясь с одной стороны доказать NP-полноту задачи, одновременно целесообразно искать полиномиальный алгоритм ее решения. Если задача оказалась NP-полной, то это является сильным аргументом в пользу того, что ее нельзя решить за полиномиальное время.