АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Лекция 2

Мацкевич С.Е.

План лекции 2 «Элементарные структуры данных»

МГТУ им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

- Однонаправленные, двунаправленные списки.
- Абстрактные типы данных «Стек», «Очередь», «Дек». Способы реализации.
- Динамическое программирование.
- Жадные алгоритмы.

Абстрактные типы данных и структуры данных

Напоминание:

Абстрактный тип данных = набор методов, интерфейс.

Структура данных = программная единица для обработки связанных данных, зачастую является реализацией некоторого АТД.

Связные списки

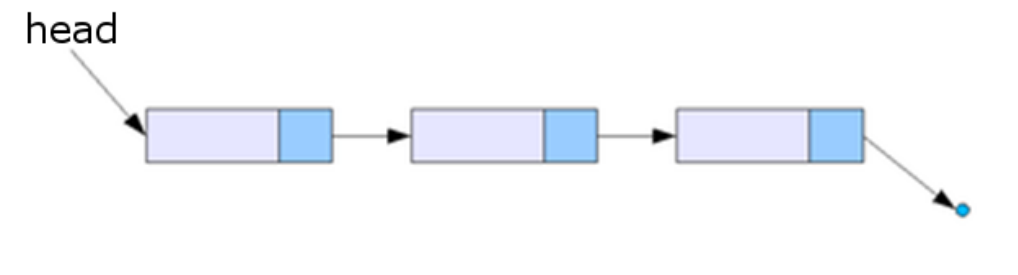
<u>Определение.</u> Связный список — динамическая структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка.

Преимущество перед массивом:

- Порядок элементов списка может не совпадать с порядком расположения элементов данных в памяти, а порядок обхода списка всегда явно задаётся его внутренними связями.

Связные списки

Односвязный список (однонаправленный связный список)



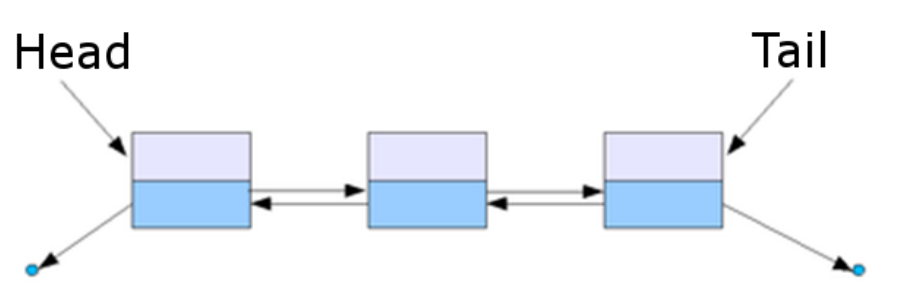
Ссылка в каждом узле одна.

Указывает на следующий узел в списке.

Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

Связные списки

Двусвязный список (Двунаправленный связный список)



Ссылки в каждом узле указывают на предыдущий и на последующий узел в списке.

Связные списки

Операции со списками:

- Поиск элемента,
- Вставка элемента,
- Удаление элемента,
- Объединение списков,
- …

Связные списки. Узел.


```
// Элемент двусвязного списка с целочисленными
// значениями.
struct CNode {
    int Data;
    CNode* Next;
    CNode* Prev;

    CNode() : Data( 0 ), Next( 0 ), Prev( 0 ) {}
};
```

Связные списки. Поиск.


```
// Линейный поиск элемента «a» в списке.
// Возвращает 0, если элемент не найден.
CNode* Search( CNode* head, int a )
{
    CNode* current = head;
    while( current != 0 ) {
        if( current->Data == a )
            return current;
        current = current->Next;
    }
    return 0;
}
```

Время работы в худшем случае = O(n), где n – длина списка.

Связные списки. Вставка.


```
// Вставка элемента «a» после текущего.
CNode* InsertAfter( CNode* node, int a )
{
    assert( node != 0 );
    // Создаем новый элемент.
    CNode* newNode = new CNode();
    newNode->Data = a;
    newNode->Next = node->Next;
    newNode->Prev = node;
    // Обновляем Prev следующего элемента, если он есть.
    if( node->Next != 0 ) {
        node->Next->Prev = newNode;
    }
    // Обновляем Next текущего элемента.
    node->Next = newNode;
    return newNode;
}
```

Время работы = O(1).

Связные списки. Удаление.


```
// Удаление элемента.
void DeleteAt( CNode* node )
{
    assert( node != 0 );
    // Обновляем Prev следующего элемента, если он есть.
    if( node->Next != 0 ) {
        node->Next->Prev = node->Prev;
    }
    // Обновляем Next предыдущего элемента, если он есть.
    if( node->Prev != 0 ) {
        node->Prev->Next = node->Next;
    }
    delete node;
}
```

Время работы = O(1).

Связные списки. Объединение.


```
// Объединение односвязных списков. К списку 1 подцепляем
список 2.
// Возвращает указатель на начало объединенного списка.
CNode* Union( CNode* head1, CNode* head2 ) {
    if( head1 == 0 ) {
        return head2;
    }
    if( head2 == 0 ) {
        return head1;
    }
    // Идем в хвост списка 1.
    CNode* tail1 = head1;
    for( ; tail1->Next != 0; tail1 = tail1->Next );
    // Обновляем Next хвоста.
    tail1->Next = head2;
    return head1;
}
```

Время работы = O(n), где n – длина первого списка.

Сравнение списков с массивами.

Недостатки списков:

- Нет быстрого доступа по индексу.
- Расходуется доп. память.
- Узлы могут располагаться в памяти разреженно, что не позволяет использовать кэширование процессора.

Преимущества списков перед массивом:

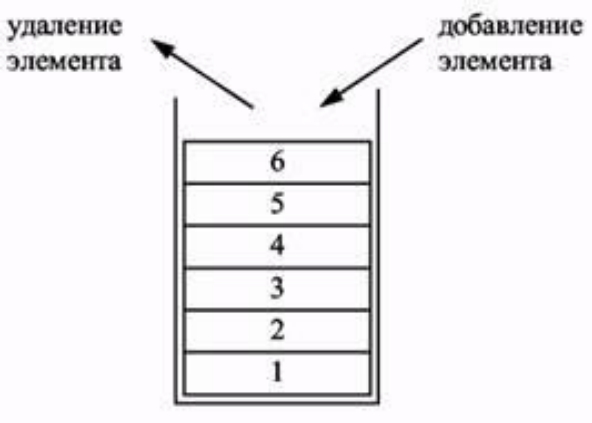
- Быстрая вставка узла.
- Быстрое удаление узла.

АТД «Стек»

Определение. Стек — абстрактный тип данных (или структура данных), представляющий из себя список элементов, организованный по принципу LIFO = Last In First Out, «последним пришел, первым вышел».

Операции:

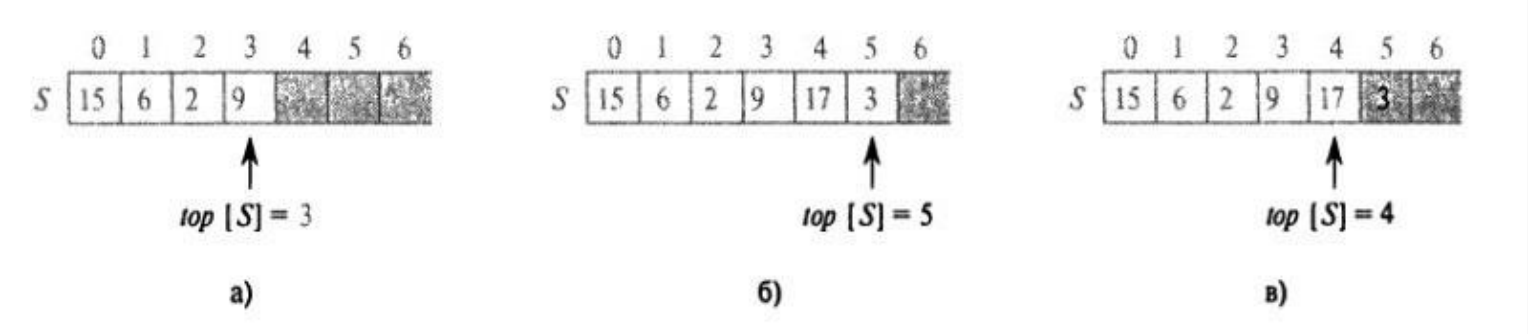
1. Вставка (Push)
2. Извлечение (Pop) — извлечение элемента, добавленного последним.



СД «Стек»

Стек можно реализовать с помощью массива или с помощью списка.

Реализация с помощью массива.



Храним указатель на массив и текущее количество элементов в стеке.

Можно использовать динамический массив.

СД «Стек»


```
// Стек целых чисел, реализованный с помощью массива.
class CStack {
public:
    explicit CStack( int _bufferSize );
    ~CStack();

    // Добавление и извлечение элемента из стека.
    void Push( int a );
    int Pop();

    // Проверка на пустоту.
    bool IsEmpty() const { return top == -1; }

private:
    int* buffer;
    int bufferSize;
    int top; // Индекс верхнего элемента.
};
```

СД «Стек»

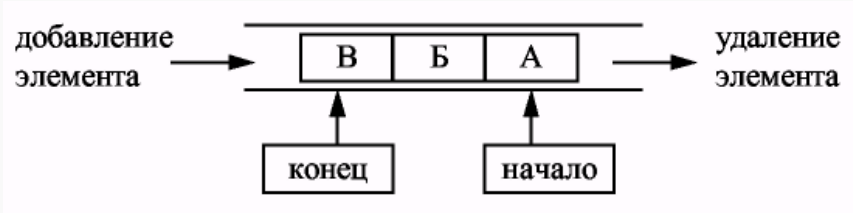

```
CStack::CStack( int _bufferSize ) :
    bufferSize( _bufferSize ),
    top( -1 )
{
    buffer = new int[bufferSize]; // Создаем буфер.
}
CStack::~CStack()
{
    delete[] buffer; // Удаляем буфер.
}
// Добавление элемента.
void CStack::Push( int a )
{
    assert( top + 1 < bufferSize );
    buffer[++top] = a;
}
// Извлечение элемента.
int CStack::Pop()
{
    assert( top != -1 );
    return buffer[top--];
}
```

АТД «Очередь»

Определение. Очередь — абстрактный тип данных (или структура данных), представляющий из себя список элементов, организованный по принципу FIFO = First In First Out, «первым пришел, первым вышел».

Операции:

1. Вставка (Enqueue)
2. Извлечение (Dequeue) – извлечение элемента, добавленного первым.



СД «Очередь»

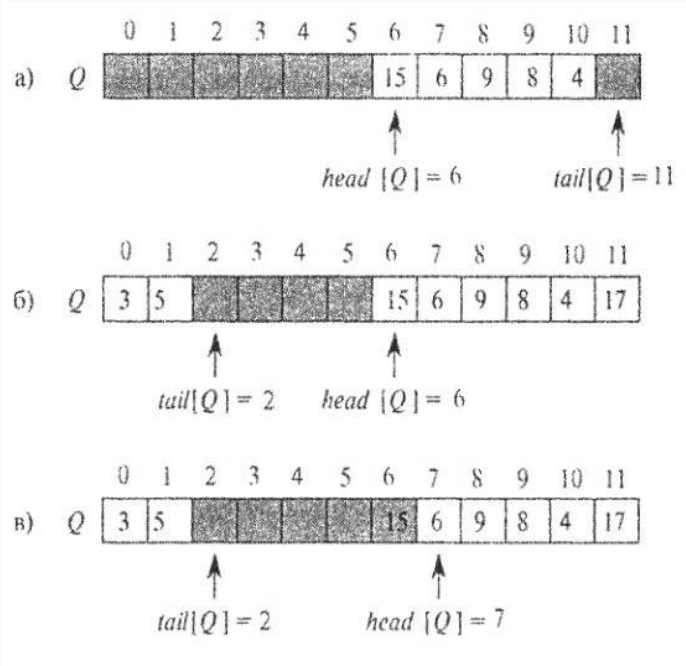
Очередь также как и стек можно реализовать с помощью массива или с помощью списка.

Реализация с помощью массива.

Храним указатель на массив, текущее начало и конец очереди.

Считаем массив зацикленным.

Можно использовать динамически растущий буфер.



СД «Очередь»


```
// Очередь целых чисел, реализованная с помощью массива.
class CQueue {
public:
    explicit CQueue( int size );
    ~CQueue() { delete[] buffer; }

    // Добавление и извлечение элемента из очереди.
    void Enqueue( int a );
    int Dequeue();

    // Проверка на пустоту.
    bool IsEmpty() const { return head == tail; }

private:
    int* buffer;
    int bufferSize;
    int head; // Указывает на первый элемент очереди.
    int tail; // Указывает на следующий после последнего.
};
```

СД «Очередь»


```
CQueue::CQueue( int size ) :
    bufferSize( size ),
    head( 0 ),
    tail( 0 )
{
    buffer = new int[bufferSize]; // Создаем буфер.
}

// Добавление элемента.
void CQueue::Enqueue( int a )
{
    assert( ( tail + 1 ) % bufferSize != head );
    buffer[tail] = a;
    tail = ( tail + 1 ) % bufferSize;
}
// Извлечение элемента.
int CQueue::Dequeue()
{
    assert( head != tail );
    int result = buffer[head];
    head = ( head + 1 ) % bufferSize;
    return result;
}
```

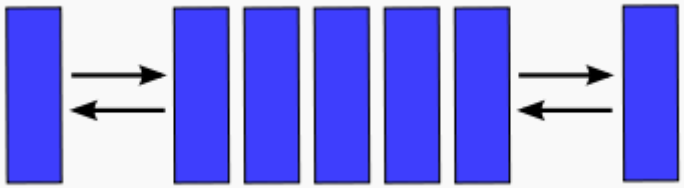
АТД «Дэк»

Определение. Двусвязная очередь — абстрактный тип данных (структура данных), в которой элементы можно добавлять и удалять как в начало, так и в конец, то есть принципами обслуживания являются одновременно FIFO и LIFO.

Операции:

1. Вставка в конец (PushBack),
2. Вставка в начало (PushFront),
3. Извлечение из конца (PopBack),
4. Извлечение из начала (PopFront).

Дек, также как стек или очередь, можно реализовать с помощью массива или с помощью списка.



Динамическое программирование

Динамическое программирование (ДП) — способ решения сложных задач путём разбиения их на более простые подзадачи.

Подход динамического программирования состоит в том, чтобы решить каждую подзадачу только один раз, сократив тем самым количество вычислений.

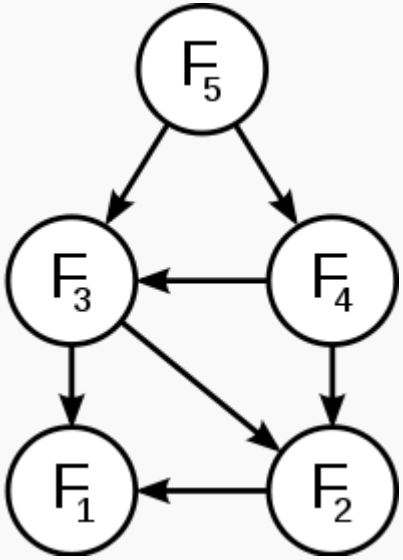
Термин «динамическое программирование» также встречается в теории управления в смысле «динамической оптимизации». Основным методом ДП является сведение общей задачи к ряду более простых экстремальных задач.

Лит.: Беллман Р., Динамическое программирование, пер. с англ., М., 1960.

Динамическое программирование

<u>Пример.</u> Числа Фибоначчи. $F_n = F_{n-1} + F_{n-2}$. $F_1 = 1, F_0 = 1$.

Можно вычислять рекурсивно:
$F_5 = F_4 + F_3$,
$F_4 = F_3 + F_2$,
$F_3 = F_2 + F_1$,



Многие значения могут вычисляться несколько раз.

Решение — сохранять результаты решения подзадач.
Этот подход называется кэшированием.

Динамическое программирование

Пример. Вычисление рекуррентных функций нескольких аргументов.

$$F(x,y) = 3 \cdot F(x-1,y) - 2 \cdot F^2(x,y-1),$$
$$F(x,0) = x, F(0,y) = 0.$$

Вычисление $F(x,y)$ сводится к вычислению двух $F(\cdot,\cdot)$ от меньших аргументов.

Есть перекрывающиеся подзадачи.

$F(x-1,y-1)$ в рекурсивном решении вычисляется дважды.

$F(x-2,y-1)$ в рекурсивном решении вычисляется три раза.

$F(x-n,y-m)$ в рекурсивном решении вычисляется C_{n+m}^{n} раз.

Снова будем использовать кэширование — сохранять результаты.

Вычисления будем выполнять от меньших аргументов к большим.

Динамическое программирование


```
// Вычисление рекуррентного выражения от двух переменных.
int F( int x, int y )
{
    vector<vector<int>> values( x + 1 );
    for( int i = 0; i <= x; ++i ) {
        values[i].resize( y + 1 );
        values[i][0] = i; // F( x, 0 ) = x;
    }
    for( int i = 1; i <= y; ++i ) {
        values[0][i] = 0; // F( 0, y ) = 0;
    }
    // Вычисляем по столбцам для каждого x.
    for( int i = 0; i <= x; ++i ) {
        for( int j = 0; j <= y; ++j ) {
            values[i][j] = 3 * values[i - 1][j] -
                2 * values[i][j - 1] * values[i][j - 1];
        }
    }
    return value[x][y];
}
```

Динамическое программирование

При вычисление рекуррентной функции $F(x, y)$ можно было не хранить значения на всех рядах.
Для вычисления очередного ряда достаточно иметь значения предыдущего ряда.

Важная оптимизация ДП: Запоминать только те значения, которые будут использоваться для последующих вычислений.

Для вычисления числа Фибоначчи F_i также достаточно хранить лишь два предыдущих значения: F_{i-1} и F_{i-2}.

Динамическое программирование

Принципы ДП:

1. Разбить задачу на подзадачи.
2. Кэшировать результаты решения подзадач.
3. Удалять более неиспользуемые результаты решения подзадач (опционально).

Динамическое программирование

Два подхода динамического программирования:

1. *Нисходящее динамическое программирование*: задача разбивается на подзадачи меньшего размера, они решаются и затем комбинируются для решения исходной задачи. Используется запоминание для решений часто встречающихся подзадач.
2. *Восходящее динамическое программирование*: все подзадачи, которые впоследствии понадобятся для решения исходной задачи просчитываются заранее и затем используются для построения решения исходной задачи.

Второй способ лучше нисходящего программирования в смысле размера необходимого стека и количества вызова функций, но иногда бывает нелегко заранее выяснить, решение каких подзадач нам потребуется в дальнейшем.

Динамическое программирование

Иногда бывает нелегко заранее выяснить, решение каких подзадач нам потребуется в дальнейшем.

Пример. $F(n) = F\left(\frac{n}{2}\right) + F\left(\frac{2 \cdot n}{3}\right)$, $F(0) = F(1) = 1$, деление целочисленное.

Восходящее динамическое программирование применить можно, но будут вычислены все значения F от 1 до n. Сложно определить, для каких k требуется вычислять $F(k)$.

Нисходящее динамическое программирование позволит вычислить только те $F(k)$, которые требуются, но рекурсивно. Максимальная глубина рекурсии $= \log_{3/2}(n)$.

Динамическое программирование


```
// Вычисление рекуррентного выражения методом нисходящего ДП.
int F( int n )
{
    // Разумнее использовать std::map, а не std::vector.
    std::vector<int> values( n, -1 );
    values[0] = 1;
    values[1] = 1;
    return calcF( n, values );
}

// Рекурсивное вычисление с запоминанием.
int calcF( int n, std::vector<int>& values )
{
    if( values[n] != -1 )
        return values[n];
    values[n] = calcF( n / 2, values ) + calcF( 2 * n / 3, values );
    return values[n];
}
```

Наибольшая общая подпоследовательность

Последовательность X является подпоследовательностью Y, если из Y можно удалить несколько элементов так, что получится последовательность X.

Задача. Наибольшая общая подпоследовательность (англ. longest common subsequence, LCS) – задача поиска последовательности, которая является подпоследовательностью нескольких последовательностей (обычно двух).

Элементами подпоследовательности могут быть числа, символы…

X = ABCAB,
Y = DCBA,
LCS(X, Y) = BA, CA, CB.

Наибольшая общая подпоследовательность

Будем решать задачу нахождения наибольшей общей подпоследовательности с помощью ДП.

Сведем задачу к подзадачам меньшего размера:

$f(n_1, n_2)$ – длина наибольшей общей подпоследовательности строк $s_1[0..n_1]$, $s_2[0..n_2]$.

$$f(n_1, n_2) = \begin{cases} 0, & n_1 = 0 \vee n_2 = 0 \\ f(n_1 - 1, n_2 - 1) + 1, & s[n_1] = s[n_2] \\ max(f(n_1 - 1, n_2), f(n_1, n_2 - 1)), & s[n_1] \neq s[n_2] \end{cases}$$

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	0	0	0	0	0
C	0	0	0	1	1	1
B	0	0	1	1	1	2
A	0	1	1	1	2	2

В каждой ячейке – длина наибольшей общей подстроки соответствующих начал строк.

Наибольшая общая подпоследовательность

$f(n_1, n_2)$ — длина НОП.

Как восстановить саму подпоследовательность?

Можно хранить в каждой ячейке таблицы «направление» перехода. Переход по диагонали означает, что очередной символ присутствует в обоих последовательностях.

Начинаем проход от правого нижнего угла.
Идем по стрелкам, на каждый переход по диагонали добавляем символ в начало строящейся подпоследовательности.

«Направления» можно не хранить, а вычислять по значениям в таблице.

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	←↑ 0	←↑ 0	←↑ 0	←↑ 0	←↑ 0
C	0	←↑ 0	←↑ 0	↖ 1	← 1	← 1
B	0	←↑ 0	↖ 1	←↑ 1	←↑ 1	↖ 2
A	0	↖ 1	←↑ 1	←↑ 1	↖ 2	←↑ 2

Желтым выделены ячейки, лежащие на лучшем пути.
Переход по диагонали соответствует совпадению символа.
Возможно существованию несколько лучших путей – по одному на каждую Наибольшую Общую Подпоследовательность.

Расстояние Левенштейна

Расстояние Левенштейна (также **редакторское расстояние** или **дистанция редактирования**) между двумя строками — это минимальное количество следующих операций:

- вставки одного символа,
- удаления одного символа,
- замены одного символа на другой,

необходимых для превращения одной строки в другую.

Впервые задачу упомянул в 1965 году советский математик Владимир Иосифович Левенштейн при изучении последовательностей 0-1.

Расстояние Левенштейна

Расстояние Левенштейна и его обобщения активно применяется:

- для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи).
- для сравнения текстовых файлов утилитой diff и ей подобными. Здесь роль «символов» играют строки, а роль «строк» — файлы.
- в биоинформатике для сравнения генов, хромосом и белков.

Расстояние Левенштейна

Будем вычислять расстояние Левенштейна с помощью Динамического программирования.

D(i, j)=количество операций, необходимых для приведения одной подстроки S[0..i] к другой T[0..j].

Краевые значения:

D(i, 0) = i для всех i в диапазоне [0, |S|].

D(0, j) = j для всех j в диапазоне [0, |T|].

Расстояние Левенштейна

$$D(i,j) = min\begin{cases} D(i-1,j-1) + m(S[i],T[j]) \\ 1 + D(i-1,j) \\ D(i,j-1) + 1 \end{cases}$$

Где m(S[i], T[j]) – 0, если символы S[i] и T[j] совпадают,
– 1, иначе.

- Первое выражение соответствует замене i-го символа первой строки на j-ый символ второй строки.
- Второе выражение соответствует удалению i-го символа первой строки и получению из S[0..i-1] строки T[0..j].
- Третье выражение соответствует получению из строки S[0..i] строки T[0..j-1] и добавлению T[j].

Расстояние Левенштейна

Пример.

		М	Е	Т	Р	О
	0	1	2	3	4	5
Л	1	1	2	3	4	5
Е	2	2	1	2	3	4
Т	3	3	2	1	2	3
О	4	4	3	2	2	2

Расстояние Левенштейна

Пример.

		А	Р	Е	С	Т	А	Н	Т
	0	1	2	3	4	5	6	7	8
Д	1	1	2	3	4	5	6	7	8
А	2	1	2	3	4	5	5	6	7
Г	3	2	2	3	4	5	6	6	7
Е	4	3	3	2	3	4	5	6	7
С	5	4	4	3	2	3	4	5	6
Т	6	5	5	4	3	2	3	4	5
А	7	6	6	5	4	3	2	3	4
Н	8	7	7	6	5	4	3	2	3

Расстояние Левенштейна


```
// Вычисление расстояния Левенштейна.
int LevenshteinDistance( const string& s, int i, const string& t, int j )
{
    // Крайние случаи.
    if( i == 0 ) return j;
    if( j == 0 ) return i;

    if( cache.Has( i, j ) )
        return cache[i, j];

    // Проверим совпадение последних символов.
    int cost = s[i - 1] == t[j - 1] ? 0 : 1;

    // Вычисляем минимум. Рекурсия глубока, надо бы восходящее решение.
    return cache[i, j] =
        min( LevenshteinDistance( s, i - 1, t, j ) + 1,
        LevenshteinDistance( s, i, t, j - 1 ) + 1,
        LevenshteinDistance( s, i - 1, t, j - 1 ) + cost );
}
```

Жадные алгоритмы

Жадный алгоритм (англ. Greedy algorithm) – алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Доказательство оптимальности часто следует такой схеме:

1. Доказывается, что жадный выбор на первом шаге не закрывает пути к оптимальному решению: для всякого решения есть другое, согласованное с жадным выбором и не хуже первого.
2. Показывается, что подзадача, возникающая после жадного выбора на первом шаге, аналогична исходной.
3. Рассуждение завершается по индукции.

Жадные алгоритмы. Размен монет.

Монетная система некоторого государства состоит из монет достоинством $a_1 = 1 < a_2 < \cdots < a_n$. Требуется выдать сумму S наименьшим возможным количеством монет.

<u>Жадный алгоритм:</u>
Берём наибольшее возможное количество монет наибольшего достоинства a_n: $x_n = \lfloor S/a_n \rfloor$.

Далее аналогично получаем, сколько нужно монет меньшего номинала для выдачи остатка.

- Не всегда даёт оптимальное решение.
  Например. Монеты в 1, 5 и 7коп. Сумма 24.
  Жадный алгоритм разменивает так:
  7к. — 3 шт., 1к. — 3 шт.
  Правильное решение:
  7к. — 2 шт., 5к. — 2 шт.
- Тем не менее, на всех *реальных* монетных системах жадный алгоритм даёт правильный ответ.

Жадные алгоритмы. Покрытие отрезками.

Задача. Дано множество отрезков $[a_i, b_i]$, покрывающее отрезок $[0, X]$. Найти наименьшее подпокрытие, т.е. минимальный набор отрезков, по-прежнему покрывающий отрезок $[0, X]$.

Жадное решение.

Упорядочим набор отрезков по возрастанию левого конца a_i.

- Шаг 1. Среди отрезков, содержащих 0, найдем такой, у которого наибольший правый конец b_i. Обозначим этот отрезок $[\widetilde{a_1}, \widetilde{b_1}]$.
- Шаг 2. Среди отрезков, содержащих $\widetilde{b_1}$, найдем такой, у которого наибольший правый конец b_i. Обозначим этот отрезок $[\widetilde{a_2}, \widetilde{b_2}]$.
- И так далее.

Жадные алгоритмы. Покрытие отрезками.

Утверждение. Жадное решение является оптимальным.

Доказательство. От противного. Пусть жадное решение состоит из K отрезков $[\widetilde{a_i}, \widetilde{b_i}]$. И пусть существует более оптимальное решение $[\widehat{a_i}, \widehat{b_i}]$, состоящие из менее чем K отрезков.

Упорядочим отрезки в решениях по правому краю. Заметим, что отрезки также будут упорядочены и по левому краю в оптимальном решении и в жадном решении.

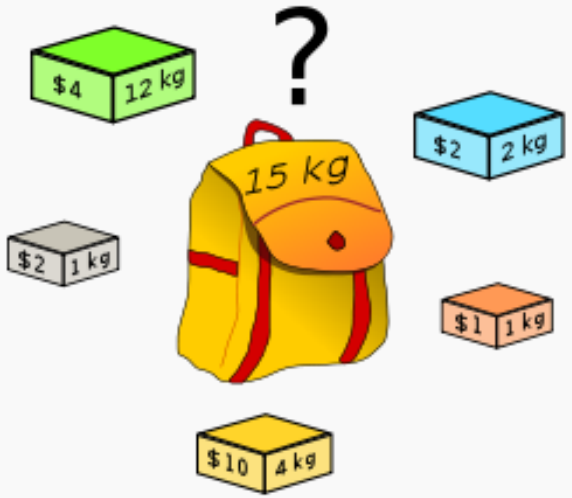
Найдем первый отрезок, отличающийся в жадном решении от оптимального. У такого отрезка оптимального решения меньше правый край и он по-прежнему покрывает правый край предыдущего общего отрезка. Значит, его можно заменить на отрезок из жадного решения.

Так, заменяя отрезки оптимального решения на отрезки жадного, получим, что менее K первых отрезков жадного решения являются оптимальным решением, что невозможно по построению жадного решения.

Жадные алгоритмы. Задача о рюкзаке.

Задача о рюкзаке (англ. Knapsack problem) — одна из NP-полных задач комбинаторной оптимизации. Название своё получила от максимизационной задачи укладки как можно большего числа нужных вещей в рюкзак при условии, что общий объём (или вес) всех предметов, способных поместиться в рюкзак, ограничен.

Имеется N грузов. Для каждого i-го груза определён вес w_i и ценность c_i. Нужно упаковать в рюкзак ограниченной грузоподъёмности G те грузы, при которых суммарная ценность упакованного была бы максимальной.



Жадные алгоритмы. Задача о рюкзаке.

Жадный алгоритм.

Предметы сортируются по убыванию стоимости единицы каждого (по отношению цены к весу).

Шаг 1. Помещаем в рюкзак первый предмет из отсортированного массива, который поместится в рюкзак.

Шаг 2. Помещаем в рюкзак первый из оставшихся предметов отсортированного массива, который поместится в рюкзак.

И т.д., пока в рюкзаке остается место или все оставшиеся предметы оказались тяжелее.

Жадные алгоритмы. Задача о рюкзаке.

Пример, когда жадный алгоритм не работает.

Пусть вместимость рюкзака 90. Предметы уже отсортированы. Применяем к ним жадный алгоритм.

	вес	цена	цена/вес
1	20	60	3
2	30	90	3
3	50	100	2

Кладём в рюкзак первый, а за ним второй предметы. Третий предмет в рюкзак не влезет. Суммарная ценность поместившегося равна 150. Если бы были взяты второй и третий предметы, то суммарная ценность составила бы 190. Видно, что жадный алгоритм не обеспечивает оптимального решения, поэтому относится к приближенным.

Задача о рюкзаке. Точное решение. ДП.

С помощью Динамического программирования можно решить задачу о рюкзаке, если веса целочисленные или дискретные.

Пусть все веса целочисленные. Определим

$D(k)$ – максимальная стоимость рюкзака с грузами общим весом $\leq k$.

Тогда $D(k+1)$ можно вычислить, пытаясь положить в рюкзак каждый предмет с весом w_i, используя $D(k+1-w_i)$.

$$D(k+1) = \max(D(k+1-w_i) + c_i).$$

Ответом в задаче о рюкзаке является $D(G)$.

Каждое вычисление $D(k+1)$ выполняется за $O(N)$, где N – количество грузов.

Общее время работы $= O(G \cdot N)$.

Вопросы???

Спасибо за внимание!