

Рекурсия

- **Рекурсивным** называют **объект, частично состоящий или определяемый с помощью самого себя.**
- Наиболее известные примеры рекурсии:
 1. Натуральные числа:
 - а) число 0 – натуральное;
 - б) число, следующее за натуральным есть натуральное.
 2. Деревья:
 - а) 0 есть нулевое (пустое) дерево;
 - б) если t_1 и t_2 – деревья, то построение, содержащее вершину с двумя ниже расположенными деревьями опять же дерево;
 3. Функция факториала $n!$ (для неотрицат. целых чисел)
 - а) $0! = 1$;
 - б) $n > 0 \quad n! = n * (n - 1)!$
- Рекурсию можно представить как некоторую композицию P из самой себя и множества операторов S , не содержащих P , то есть

$$P \equiv P [S, P]$$

- Рекурсия может быть ***прямой (или явной)*** и ***косвенной (или неявной)***.
- **Прямая** рекурсия заключается в **прямом** вызове подпрограммой самой себя.
- **Косвенная** рекурсия может вызывать себя не прямо, а **через другую функцию**, то есть, одна функция вызывает вторую функцию, а вторая, в свою очередь, вызывает первую.
- При использовании рекурсии:
- **каждое** обращение к рекурсивной подпрограмме при **каждом** вызове порождает «поколение» локальных переменных и параметров, что при глубокой рекурсии может привести к переполнению стека.

Фрейм активации

- Совокупность данных для одной активации подпрограммы- **фрейм активации**
 - ***Фрейм активации*** включает:
 - копию всех локальных переменных;
 - копии параметров, переданных по значению;
 - адреса параметров-переменных (по ссылке) и параметров-констант;
 - служебную информацию (около 12 байт, точный размер этой области зависит от способа вызова (ближний, дальний) и внутренней организации подпрограммы).

- рекурсивные подпрограммы м. приводить к не заканчивающимся вычислениям, поэтому необходимо следить тем, чтобы в них обязательно был нерекурсивный выход.
- Например, при рекурсивном обращении к подпрограмме с параметром n в качестве параметра задавать $n - 1$.

Запишем это так:

- $P(n) \equiv \text{IF } n > 0 \text{ THEN } P[S, P(n - 1)] \text{ End,}$
 - или так:
- $P(n) \equiv P[S, \text{IF } n > 0 \text{ THEN } P(n - 1) \text{ End}]$
- При создании рекурсивных подпрограмм также важно убедиться, что максимальная глубина рекурсии не **только конечна**, но и **достаточно мала**.

Аналогии операторов и данных:

- **оператор присваивания**
 - - соответствует **простым данным**
- **составные операторы**
 - - соответствуют **записям (структурам)**
- **повторение (цикл *for*)**
 - - соответствует **массиву**
- **выбор *одного* (оператор *if*) или *нескольких* (оператор *case*)**
 - - соответствует **записи с вариантами**
- **повторение с неизвестным числом раз (циклы *while u repeat*)**
 - - соответствует структуре **файл**.

Структура, соответствующая оператору ***процедура (рекурсивная процедура)***

Эта структура должна содержать обращение к самой себе.

Пример рекурсивного определения типа — **арифметическое выражение** из языка программирования: оно отражает возможность вложенности, т. е., использование в выражениях в качестве операндов самих выражений.

Т.о., **определение выражения** формулируется так:

Выражение — это комбинация составных частей выражения: **операнда (терма)**, за которым следует **знак операции**, за которым идет опять же **операнд**. **Операнд** или, **терм**, **м.** **б.** либо **идентификатором**, либо **выражением**.

Представление выражения при размещении в памяти:

$$x + y$$

+	
T	x
T	y

$$x - (y * z)$$

-		
T	x	
F	*	
	T	y
	T	z


$$(x + y) * (z - w)$$

*		
F	+	
	T	x
	T	y
F	-	
	T	z
	T	w

$$(x / (y + z)) * w$$

*			
F	/		
	T	x	
	F	+	
		T	y
		T	z
T	w		

Например:

- Type
 - EXPRESSION {выражение} = Record
 - OP : operation;
 - PRD1, OPRD2 : term;
 - End;
 - Term = Record
 - Case T : Boolean Of
 - True : id : type_id;
 - False : subex : EXPRESSION;
 - End;
 - End;
- 
подвыражение

пример рекурсивной СД – генеалогическое дерево:

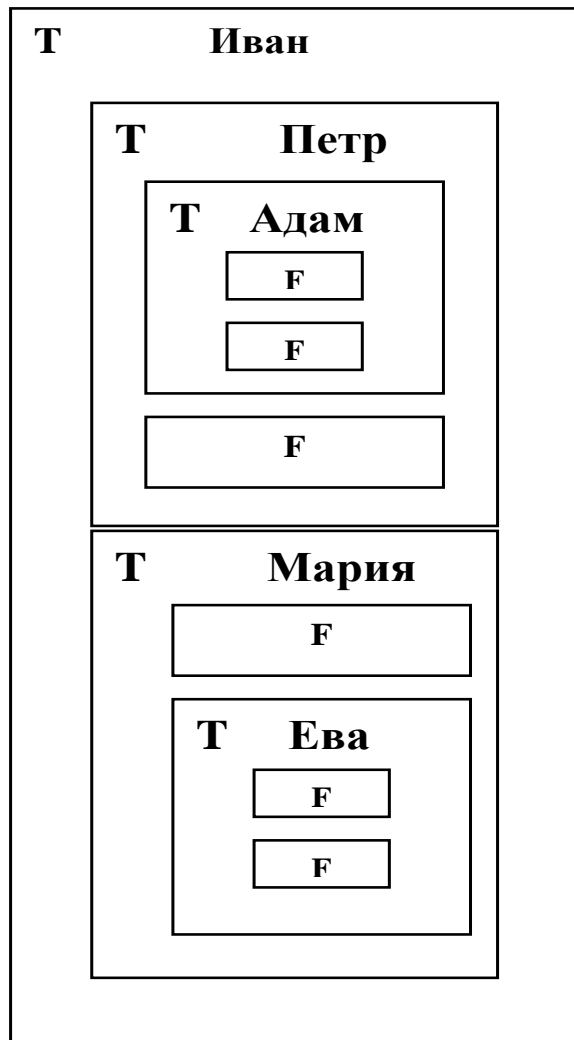
- определяемое как **имя человека + два дерева его родителей**. (конечна, т. к. на каком-то уровне сведений о предках не окажется).
- Эту структуру можно представить :
- Type PRED = Record

Case KNOW {известен} : Boolean Of

- True : Begin
- NAME : ALFA;
- FATHER, MOTHER : PRED;
- End;
- False : // пусто
- End;
- End;

- память под структуру выделяется динамически, можно представить ее списком.
 - Например:
- если у Ивана папа- **Петр**, а мама-**Мария**,
- а у **Петра**
- папа – **Адам**, мама -?,
- а у **Марии** папа -? Мама- **Ева**,
- то : (True, имя (папа,мама))
- В списковом представлении эта рекурсивная структура:
- (Т, Иван (Т, Петр (Т, Адам, (F), (F)), (F)), (Т, Мария, (F), (Т, Ева, (F), (F)))).
- В таком случае, возможное представление в памяти таково:

ВОЗМОЖНОЕ представление в памяти:



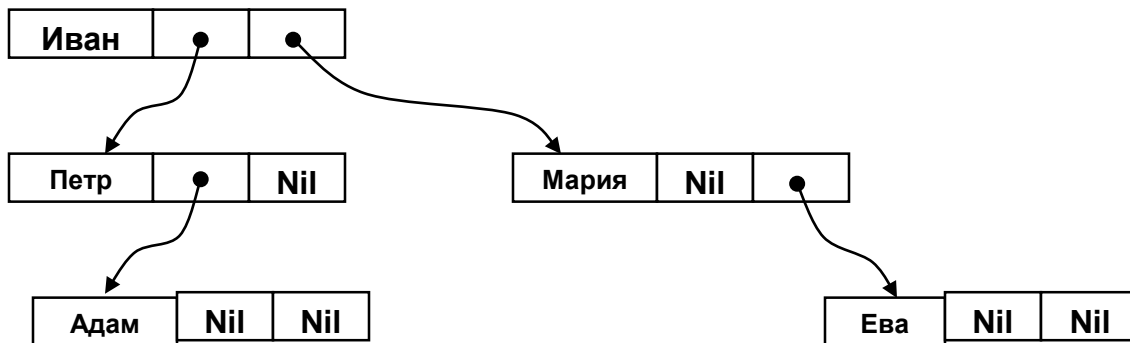
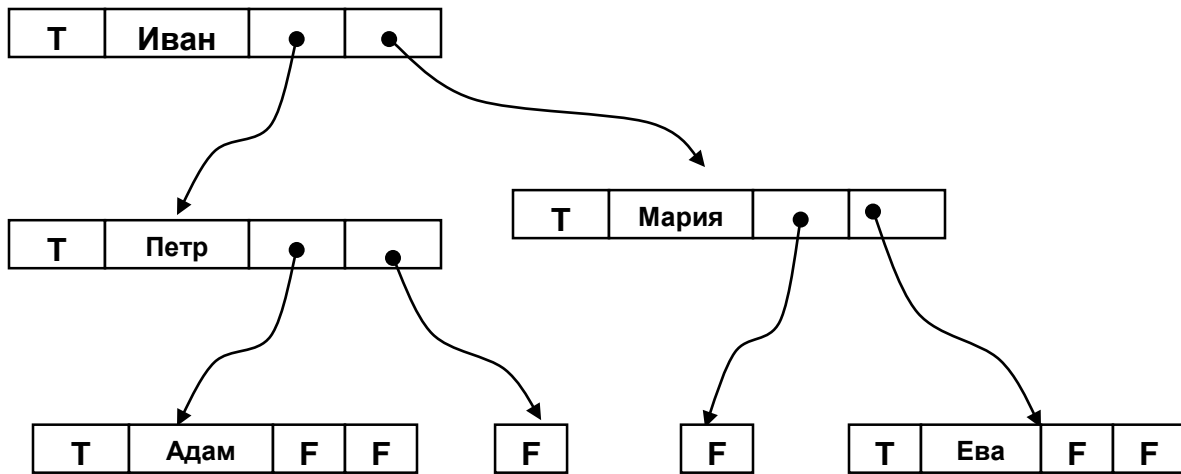
Отсюда **важное свойство записей с вариантами** – это есть единственное средство, позволяющее ограничить рекурсивную структуру.

Характерной особенностью рекурсивных структур является способность изменять размер, а отсюда — невозможность выделения памяти фиксированного размера. Т. о., для рекурсивных структур требуется динамическое распределение памяти, которое удобнее всего графически изображать стрелками или ссылками:

- Если вести явную работу со ссылками, то можно строить структуры более общего вида, чем те, которые следуют из рекурсивного определения данных. Можно вводить **потенциально бесконечные** или **циклические** структуры (но **ввод таких структур требует тщательности и осторожности, т. к. может привести к бесконечным вычислениям!!!**) и указывать, что некоторая подструктура принадлежит **разным** структурам.

-

- Если заменить поле False (* пусто *) на Nil, то можно получить экономию памяти и прийти к уже известной структуре:
- Type
- PredPtr = Pointer to Pred;
- Type
- Pred = Record
- NAME : Alfa;
- FATHER, MOTHER : PredPtr;
- end;
- struct pred { *alfa* name
- pred *father
- pred *mother
- }
- Ну, а если Петр и Мария – брат и сестра, то получим еще экономию памяти за счет одноразового хранения данных об Адаме и Еве:



- Рекурсивные подпрограммы следует применять для задач, где данные определяются в терминах рекурсий.
- Программы, в которых следует избегать алгоритмической рекурсии можно охарактеризовать некоторой схемой, отражающей их строение:
- $P \equiv \text{If } \mathbf{B} \text{ Then } S; P \text{ End,}$
- $P \equiv S; \text{If } \mathbf{B} \text{ Then } P \text{ End}$
- где \mathbf{B} – условие вызова рекурсии.
- (Специфично, что здесь имеется единственное обращение к P в конце или в начале всей конструкции).
- Такие схемы естественны в ситуации, где вычисляемые значения определяются с помощью рекуррентных соотношений.


- Например, вычисление факториала можно представить так:
- Function Fact (N : Integer) : Longint;
- Begin
- If N > 0 Then Fact \leftarrow N * Fact(N - 1)
- Else Fact \leftarrow 1;
- End;
- И теперь ясно, что рекурсия заменяется итерацией:
- i \leftarrow 0;
- Fact \leftarrow 1;
- While i < N Do
- Begin
- i \leftarrow i + 1;
- Fact \leftarrow i * Fact;
- End;

Рекурсивно:

- **long double fact(int N)**
- **{ if(N < 0) return 0;**
- **if (N == 0) return 1;**
- **else return N * fact(N - 1);**
- **}**

Итеративное вычисление:

- **long double fact(int N)**
- **{ if (N < 0) return 0;**
- **if (N == 0) return 1;**
- **long double result = 1;**
- **for (int i = 1; i <= N; i++)**
- **{ result *= i; }**
- **return result;**
- **}**

- В общем-то, программы, построенные по схеме , следует переписывать, руководствуясь схемой:
- $P \equiv [X := X_0; \text{While } B \text{ Do } S \text{ End}]$.

Например, вычисление чисел Фибоначчи,

- $FIB_{n+1} = FIB_n + FIB_{n-1}$ для $n > 0$
- и
- $FIB_1 = 1, FIB_0 = 0$
- приводит к следующей рекурсивной подпрограмме:
- Function FIB (N : Integer) : Longint;
- Begin
- If N = 0
- Then FIB \leftarrow 0
- Else
- If N = 1
- Then FIB \leftarrow 1
- Else FIB \leftarrow FIB(N - 1) + FIB(N - 2);
- End;

- Вычисление по этой схеме приводит при каждом обращении к рекурсии еще к двум обращениям, то есть, число вызовов растет экспоненциально.

- Лучше сделать так:

- $i \leftarrow 1;$
- $X \leftarrow 1;$
- $Y \leftarrow 0;$
- While $i < N$ do
- Begin
- $Z \leftarrow X;$
- $X \leftarrow X + Y;$
- $Y \leftarrow Z;$
- $i \leftarrow i + 1;$
- End;

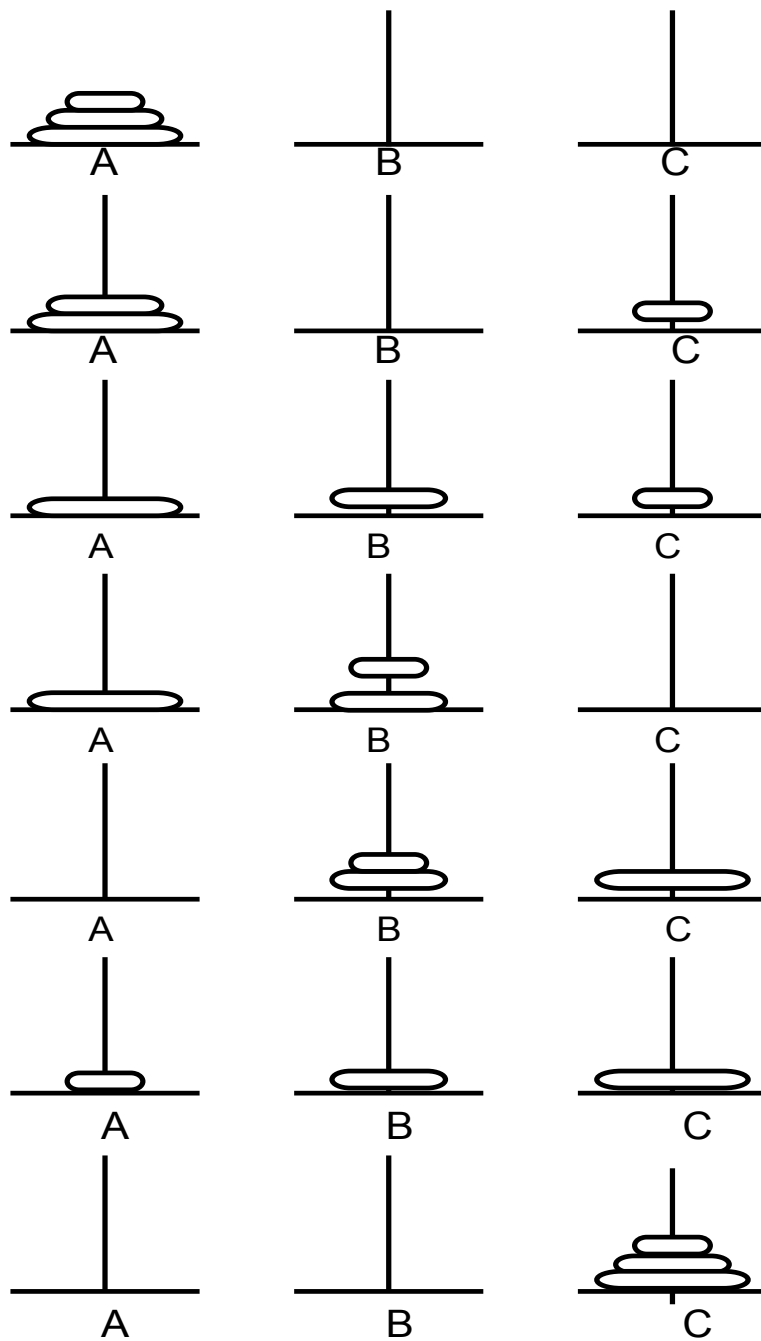
Быстрая сортировка (Хоара)

- Для сравнения можно взять алгоритмы рекурсивный и не рекурсивный:
- Выбрать любой (лучше средний) элемент массива, затем переставить все элементы массива так, чтобы слева располагались все элементы меньше выбранного, а справа – все элементы больше выбранного. Для этого массив просматривается с двух сторон и меняются местами элементы, стоящие не в своей части массива. Тем самым выбранный элемент оказывается на своем месте. Далее этот алгоритм применяют к левой и правой частям массива, пока очередной подмассив не станет равен одному элементу, который всегда упорядочен. Сложность этого алгоритма равна $O(n \log_2 n)$.

Алгоритмы **«разделяй и властвуй»**

- Алгоритмы, основанные на разделении больших множеств на более мелкие, пока не останется один элемент, удовлетворяющий условию, имеют общее название **«разделяй и властвуй»**.
- Например – сортировка Хоара, сортировка слиянием.
- Алгоритмы, основанные на принципе «разделяй и властвуй» (еще называются алгоритмы, использующие **метод декомпозиции** или **разбиения**) являются по своей природе **рекурсивными** и их переводить в итеративную форму **не следует**.

- Сравним еще два алгоритма для решения задачи о Ханойских башнях:
- В центре мира в вершинах равностороннего треугольника в землю вбиты 3 алмазных шпиля. На одном из них надето 64 золотых диска убывающих радиусов. Трудолюбивые буддийские монахи день и ночь переносят их с одного шпиля на другой. При этом диски надо переносить по одному и нельзя класть больший по радиусу диск на меньший. Когда все диски будут перенесены, наступит конец света.
- (Автором этой задачи принято считать французского математика Эдуарда Люка, создавшего ее в 1883 году на основе древней легенды).



- То есть, при $n = 3$ получим 7 перемещений:
- $A \rightarrow C, A \rightarrow B, C \rightarrow B, A \rightarrow C, B \rightarrow A, B \rightarrow C, A \rightarrow C$.
- Очевидно, что при $n = 1$ будет одно перемещение $A \rightarrow C$,
- а при $n = 2$ будет 3 перемещения $A \rightarrow B, A \rightarrow C$ и $B \rightarrow C$.
- При любом n можно использовать рекурсивный алгоритм:
 - Переместить верхние $n - 1$ дисков с A на B ;
 - Переместить верхний (n -ый) диск с A на C ;
 - Переместить башню из $n - 1$ дисков с B на C .
- Т.о. кол-во шагов (T_n) при n дисках: $T_n = 2T_{n-1} + 1$,
при $n > 0$ $T_n = 2^n - 1$

- Нерекursивный алгоритм состоит из чередования перемещений двух видов:
 1. Перенести наименьший диск с того стержня, на котором он находится в данный момент, на стержень, следующий в порядке движения часовой стрелки;
 2. Перенести любой диск, кроме наименьшего. Вторым шагом не произвольный, так как всегда найдется лишь одно перемещение.
- Рекурсивный алгоритм задачи о Ханойских башнях (5 строк кода) и нерекursивный (15 строк кода + вложенный цикл). Если на практике сравнить коды программ и время их выполнения, то в данном случае вы увидите, что рекурсивный алгоритм будет лучше нерекursивного и по простоте, и по времени выполнения.
- По времени выполнения рекурсивный алгоритм работает в 7-10 раз быстрее.

Еще один из примеров правильности использования рекурсии – это задача так называемого искусственного интеллекта, использующая некоторые **эвристики** (т. е., эмпирические (опытные) правила, упрощающие или ограничивающие поиск решения в предметной области), например – **алгоритм поиска с возвратом**. **Эвристические** алгоритмы обычно быстро находят «подходящее» решение, но оно не всегда бывает оптимальным.

Например. Есть монеты 10, 9, 5 и 1 копейки, набрать из них нужную сумму, напр. 18.

«Жадность» заключается в том, что берется макс. из возможных, < суммы. Т.е. возьмем $10+5+1+1+1$, тогда как оптимально будет $9+9$. Но, тогда, возможно, надо сделать полный перебор, а здесь на каждом шаге выбирается локально оптимальный вариант (поиск максимального).

Алгоритм закрашки графа

- Необходимо закрасить граф (т. е., множество вершин и ребер) так, чтобы никакие 2 смежные (то есть, соединенные ребром) вершины не были одного цвета, при этом по возможности стараться использовать **минимальное** количество цветов.
- Эта математическая задача принадлежит к классу **NP** – полных задач, для которых решение найдется, если перебрать все возможные варианты, то есть, сначала закрасить все вершины одним цветом, затем вторым, третьим и т.д., пока не получим верный результат. Т.О. решение задачи требует очень больших вычислительных затрат.

- **Возможны 3 варианта:**

1. Полный перебор (для небольших графов);
2. Поиск каких-либо свойств графа, позволяющих исключить полный перебор для нахождения оптимального решения;
3. Отказаться от минимальности цветов и от оптимальности раскраски, что даст возможность уменьшить время, затраченное на решение. НО это решение не будет оптимальным. Т. е., использовать эвристический алгоритм.

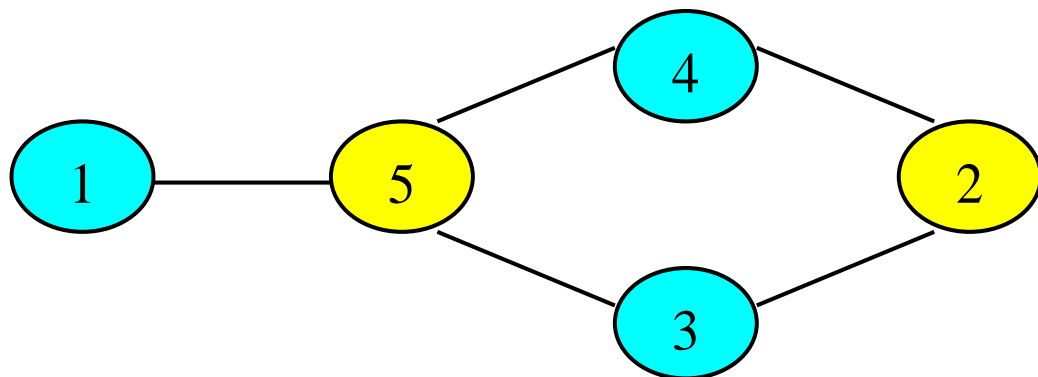
- Применительно к задаче о раскраске графа идея этого алгоритма состоит в том, чтобы раскрасить как можно больше вершин в один цвет, а затем раскрасить во второй цвет максимально возможное количество вершин, оставшихся не закрашенными первым цветом и т.д.

- При этом делаем:

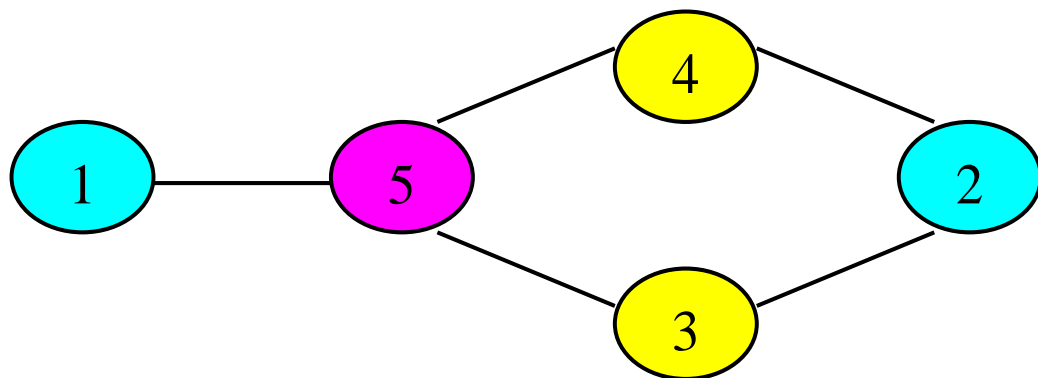
1. Выбираем произвольную незакрашенную вершину и назначаем ей новый цвет;
2. Просматриваем (последовательно, по номеру) список незакрашенных вершин и каждую из них проверяем на смежность с выбранной и уже закрашенной вершиной. Если вершина не является смежной с выбранной, то закрашиваем ее в тот же цвет.

- «Жадный» алгоритм не всегда обеспечивает в целом оптимальное решение, но работает за приемлемое время.

раскраска графа, указанного ниже, может быть такой:

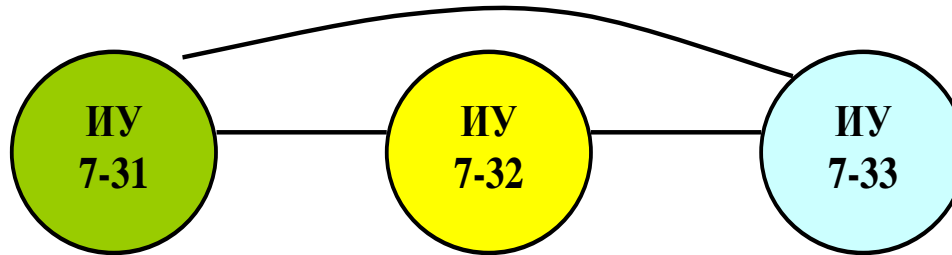


«Жадный» же алгоритм раскрасит его так:

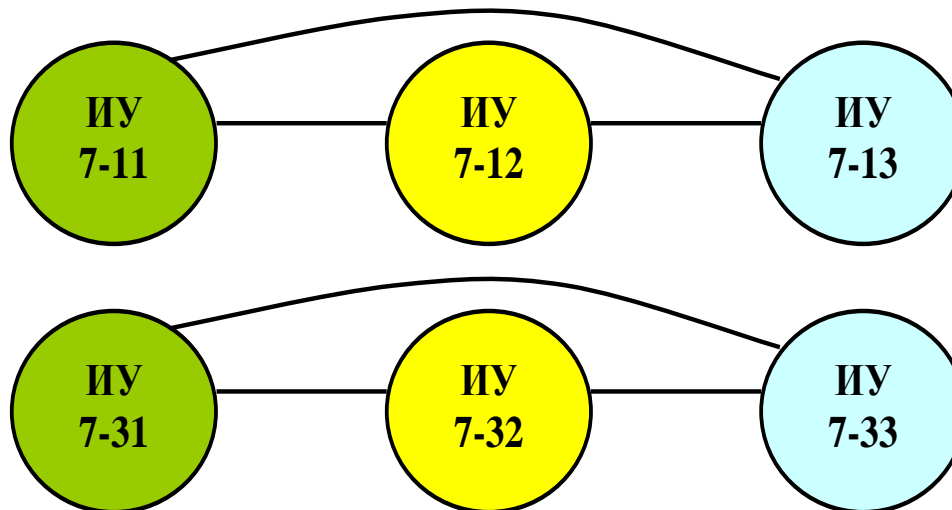


использование алгоритма раскраски графов:

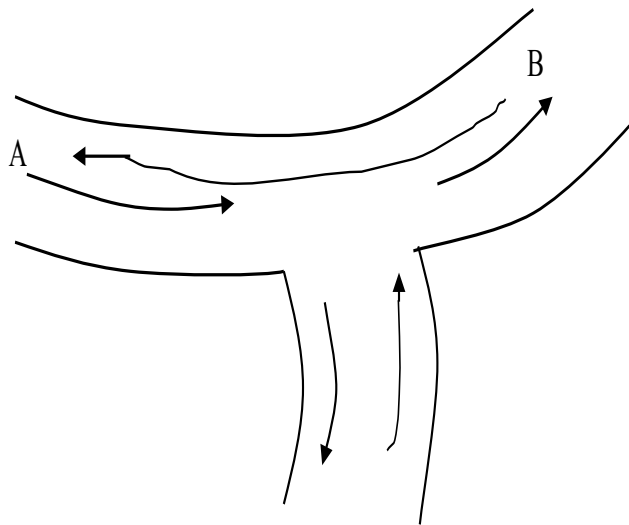
один преподаватель:



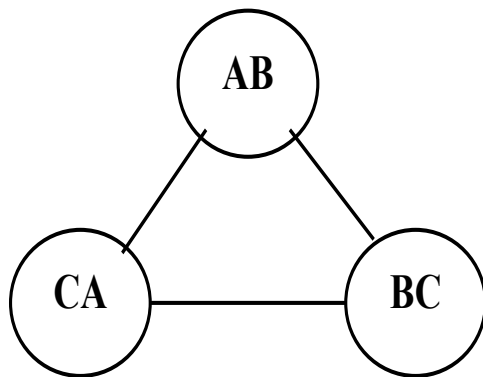
два преподавателя:



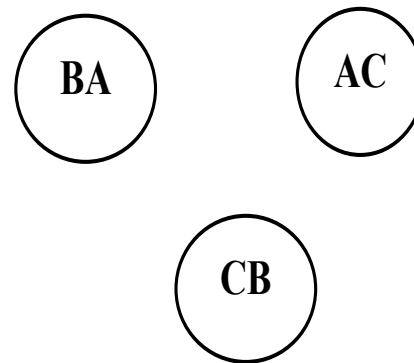
Эту же программу можно использовать для задания режимов работы светофоров на сложном перекрестке:



Для решения задачи управления перекрестком вершины в графе представляют собой повороты, а ребра соединяют ту часть вершин-поворотов, которые нельзя выполнить одновременно. Для нашего перекрестка граф будет такой:



несовместимые повороты



совместимые

Некоторые критерии для выбора алгоритмов

- Если программа будет использована немного раз, то стоимость написания и отладки программы будет основной в общей стоимости программы, т. е., фактическое время выполнения не сильно скажется на общей стоимости, и в этом случае следует предпочесть наиболее простой для реализации алгоритм, т.е. сократить время его создания.
- Если программа работает только с «малыми» входными данными, то м. б. разумнее использовать алгоритмы, менее эффективные, но работающие в этой области данных быстрее.
- Нежелательны сложные, хотя и эффективные, алгоритмы, если готовые программы будут поддерживать лица, не участвующие в написании программ. В противном случае надо очень тщательно комментировать программы для того, чтобы сопровождающему эти программы можно было легко в них разобраться.
- Известны примеры, когда эффективные по сложности и времени алгоритмы требуют таких больших объемов машинной памяти (без возможности использования более медленных внешних устройств хранения), что этот фактор сводит на нет преимущество «эффективности» таких алгоритмов.
- В численных алгоритмах **точность и устойчивость** алгоритмов не менее важны, чем их временная эффективность.

- **Псевдоязык программирования** – это комбинация обычных конструкций языков программирования с выражениями на человеческом языке.
- Procedure Greedy (var G : *GRAPH*, var NEWCLR : *SET*);
- {Greedy присваивает переменной NEWCLR (новый цвет) множество вершин графа G, кот. можно окрасить в один цвет}
- Begin
- NEWCLR := 0;
- For каждой незакрашенной вершины V из G do
- If V не соединена с вершинами из NEWCLR Then
- Begin
- пометить V цветом;
- добавить V в NEWCLR;
- End;
- End;

Абстрактные типы данных

- В данном примере используются *абстрактные типы данных (АТД)* *GRAPH* (граф) и *SET* (множество). Поскольку любой тип данных, определяет множество значений и множество операций над ними, то:
- **АТД** определяется как математическая модель с совокупностью операторов (подпрограмм), определенных в рамках этой модели, т. о. **АТД** – это тип, для которого описание значений и операций, выполняемых над ними, отделено от представления значений и реализации операций

- Для нашей задачи раскраски графа, АТД – это граф (*GRAPH*), для которого необходимы операторы, выполняющие следующие действия:
 1. Выбрать первую незакрашенную вершину;
 2. Проверить существование ребра между двумя вершинами;
 3. Понетить вершину цветом;
 4. Выбрать следующую незакрашенную вершину.

Процесс программирования:

- Создать модель исходной задачи, привлекая необходимые математические модели (например, теорию графов) и разработать неформальный алгоритм.
- Записать алгоритм на псевдоязыке, создать АТД для каждого зафиксированного типа данных (кроме простых) и задать имена подпрограмм (процедур и функций) обработки этих данных.
- Реализовать АТД конкретной структурой данных, написать процедуры их обработки и преобразовать псевдокод в программу на языке программирования.

Этапы разработки алгоритмов с использованием АТД



рекомендации по практике программирования

- Планировать этапы разработки программ. Это организует и дисциплинирует процесс создания конечной программы, которая будет проще в отладке и в дальнейшей поддержке и сопровождении.
- Применять инкапсуляцию, то есть помещать все подпрограммы, реализующие АТД, в одно место программы.
- Использовать и модифицировать уже существующие программы. Не начинать «с нуля», пользоваться своими наработками и думать, где и как можно применить созданные вами программы (возможны непредвиденные варианты).
- Стараться создавать программы-инструменты, то есть, такие программы, которые будут универсальными, с широким спектром применения.

Выбор представления данных

- Данные делятся на **входные, выходные и внутренние.**
- **Входные и выходные данные должны быть максимально удобны и понятны при их вводе и выводе**, т. е., ввод д. б. максимально прост и комментирован, а вывод — в **наглядной и понятной** форме, не требующей к.-л. усилий для соотношения с условием задачи. Во многих случаях достаточно удобно хранить наборы **входных** данных в файлах. Например, при тестировании программы.

- Тест – это набор входных данных и точное описание результатов теста на данном наборе.
 - При тестировании необходимо:
 - Реализовать полный тест: т.е. подобрать такие входные данные, чтобы *каждый* оператор, особенно стоящий в цикле или ветви, был выполнен хотя бы один раз.
 - Проверять граничные значения для индексов массивов.
 - Отслеживать варианты неверных (ошибочных) данных в математических функциях (корень из отрицательного числа, логарифм отрицательного числа, деление на 0 и т.п.).
 - Проверять граничные значения данных (минимумов и максимумов), пустые данные, при этом проверяется очистка памяти (зануление).
 - Проверять правильности ввода-вывода.
- Внутренние данные – это конкретная реализация АТД во внутреннюю структуру данных. Здесь надо следить за правильностью представления и адекватностью данных условию задачи.