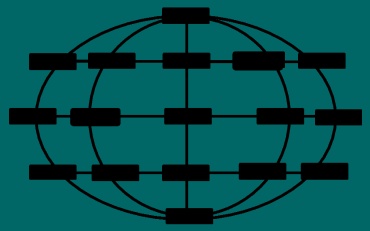
Методы построения и анализа алгоритмов



Малышкин Виктор Эммануилович

Кафедра Параллельных Вычислительных Технологий

Новосибирский государственный технический университет

E_mail: malysh@ssd.sscc.ru

Новосибирск

Об алгоритмах на примерах

Процесс решения задачи может быть условно разбит на несколько этапов:

Формулировка_задачи → анализ_задачи → модель →

функциональное_представление_решения_задачи → разработка алгоритмов →

разработка_отладка _тестирование_программы → решение задачи

Об анализе задачи

- Половина дела сделана, если знать, что исходная задача имеет решение. В первом приближении большинство задач, встречающихся на практике, не имеют четкого и однозначного описания.
- Определенные задачи, такие как разработка рецепта вечной молодости или сохранение мира во всем мире, вообще невозможно сформулировать в терминах, допускающих компьютерное решение. Даже если мы предполагаем, что наша задача может быть решена на компьютере, обычно для ее формального описания требуется огромное количество разнообразных параметров.

Если определенные аспекты решаемой задачи можно выразить в терминах какой-либо формальной модели, то это необходимо сделать, так как в этом случае в рамках формальной модели мы можем узнать, существуют ли методы и алгоритмы решения нашей задачи, можем построить множество всех решений.

Даже если такие методы и алгоритмы не существуют на сегодняшний день, то привлечение средств и свойств формальной модели поможет в построении "подходящего" решения исходной задачи.

Практически любую область математики или других наук можно привлечь к построению модели. Для задач, числовых по своей природе, можно построить модели на основе общих математических конструкций, таких как системы линейных алгебраических уравнений (задачи расчета электрических цепей или напряжений в закрепленных балках), Дифференциальные уравнения (расчета скорости протекания химических реакций, движение небесных тел).

Для задач с символьными или текстовыми данными можно применить модели символьных последователь ностей или формальных грамматик (компилятор).

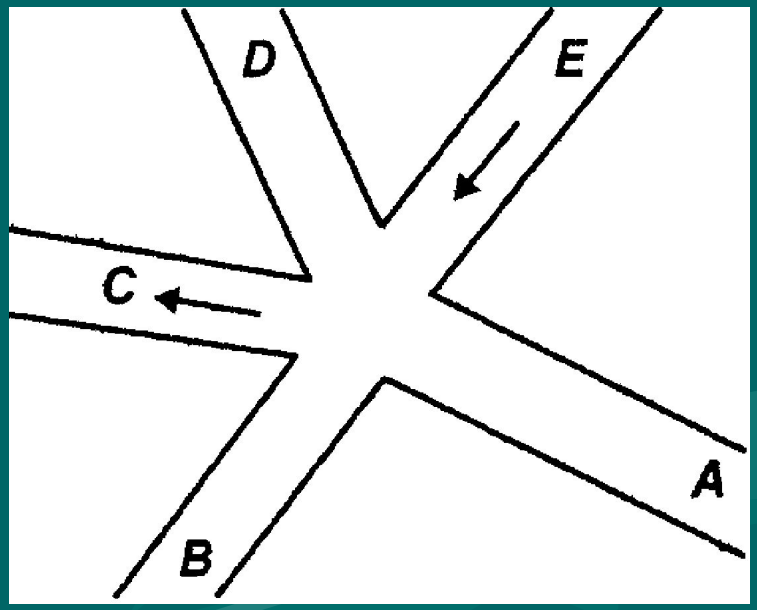
Решение таких задач содержит этапы компиляции и информационного поиска (распознавание определенных слов в списках заголовков каких-либо библиотек и т.п.).

Когда построена (подобрана) подходящая модель исходной задачи, то естественно искать решение в терминах этой модели. На этом этапе основная цель заключается в построении решения в форме алгоритма, состоящего из конечной последовательности инструкций, каждая из которых имеет четкий смысл и может быть выполнена с конечными вычислительными затратами за конечное время. Целочисленный оператор присваивания x := y + z — пример инструкции, которая будет выполнена с конечными вычислительными затратами.

Инструкции могут выполняться в алгоритме любое число раз, при этом они сами определяют число повторений. Однако требуется, чтобы при любых входных данных алгоритм завершился после выполнения конечного числа инструкций. Таким образом, программа, написанная на основе разработанного алгоритма, при любых начальных данных никогда не должна приводить к бесконечным циклическим вычислениям

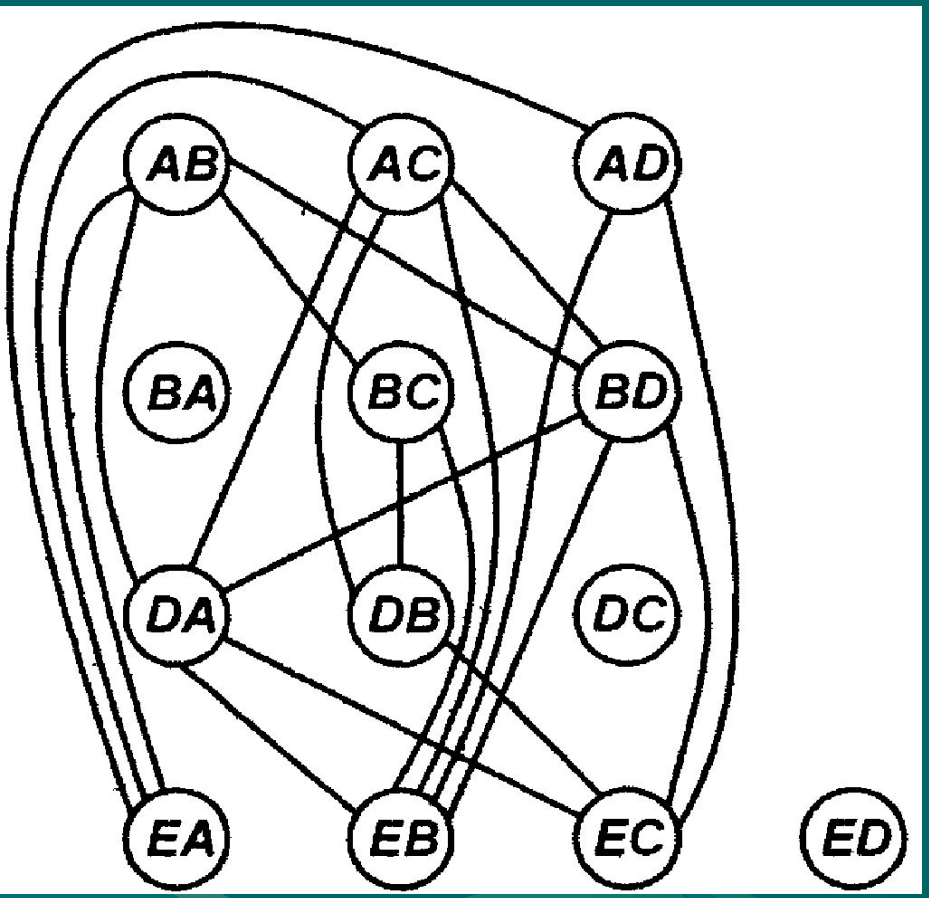
Пример модели

Модель управления светофорами на сложном перекрестке дорог



- Нужно создать программу, которая разбивает множество поворотов на несколько групп так, чтобы все повороты в группе могли выполняться одновременно
- каждой группе поворотов сопоставляется свой режим работы светофоров
- Желательно минимизировать число разбиений множества поворотов, минимизируя количество режимов работы светофоров.
- Дороги C и E односторонние, остальные — двухсторонние
- Всего на этом перекрестке возможно 13 поворотов
- Некоторые из этих поворотов, такие как AB и EC, могут выполняться одновременно
- Трассы других поворотов, например AC и DB, пересекаются, поэтому их нельзя выполнять одновременно

Графическая модель



Граф состоит из множеств вершин и ребер.

Для решения задачи можно нарисовать граф, где вершины будут представлять повороты, а ребра соединят ту часть вершин-поворотов, которые нельзя выполнить одновременно

Модель в виде графа поможет в решении задачи управления светофорами

- В рамках этой модели можно использовать решение, которое дает математическая задача раскраски графа: каждой вершине графа надо так задать цвет, чтобы никакие две соединенные ребром вершины не имели одинаковый цвет, и при этом по возможности использовать минимальное количество цветов. При такой раскраске графа несовместимым поворотам будут соответствовать вершины, окрашенные в разные цвета

Задача раскраски произвольного графа минимальным количеством цветов принадлежит к классу *переборных* задач, т.е. решается перебором всех вариантов и требует больших вычислительных затрат .

Три подхода к решению задачи.

1. Если граф небольшой, можно найти оптимальное решение, перебрав все возможные варианты раскраски. Не приемлемо для больших графов
2. Использование дополнительной информации об исходной задаче. Желательно найти какие-то особые свойства графа, которые исключали бы необходимость полного перебора всех вариантов раскраски для нахождения оптимального решения.
3. Изменяем постановку задачи и ищем не оптимальное решение, а близкое к оптимальному. Если отказаться от требования минимального количества цветов раскраски графа, то можно построить алгоритмы раскраски, которые работают значительно быстрее, чем алгоритмы полного перебора.

Алгоритмы, которые быстро находят "подходящее", но не оптимальное решение, называются ***эвристическими***.

"Жадный" алгоритм раскраски графа.

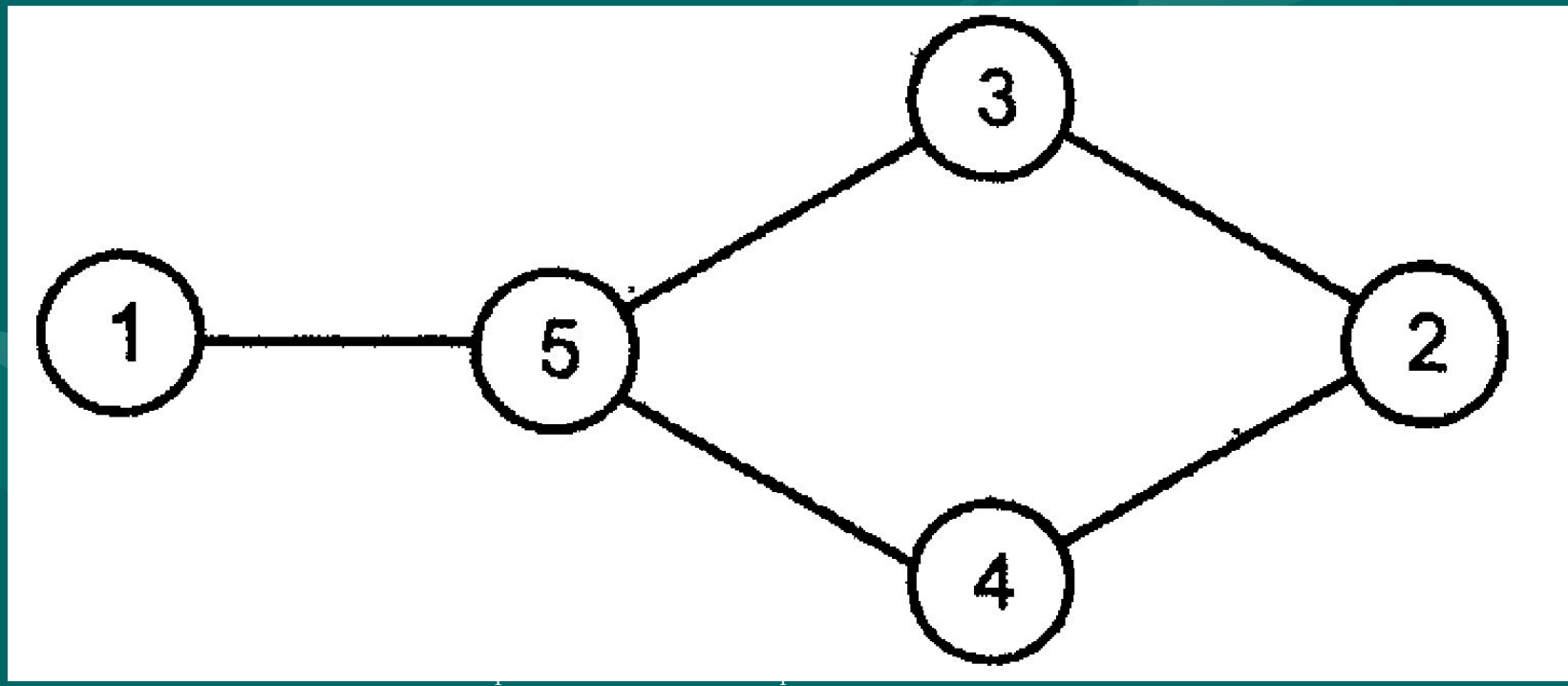
Пример рационального эвристического алгоритма

В этом алгоритме сначала раскрашивается как можно больше вершин в один цвет, затем во второй цвет закрашиваются также по возможности максимальное число оставшихся вершин, и т.д. При закраске вершин в новый цвет выполняются следующие действия.

1. Выбирается произвольная не закрашенная вершина, ей назначается новый цвет.
2. Просматривается список не закрашенных вершин и для каждой из них определяется, соединена ли она ребром с вершиной, уже закрашенной в новый цвет. Если не соединена, то к этой вершине также применяется новый цвет.

Этот алгоритм назван "жадным" из-за того, что каждый цвет применяется к максимально большому числу вершин, без возможности пропуска некоторых из них или перекраски ранее закрашенных. Возможны ситуации, когда, будь алгоритм менее "жадным" и пропустил бы некоторые вершины при закраске новым цветом, получилась бы раскраска графа меньшим количеством цветов.

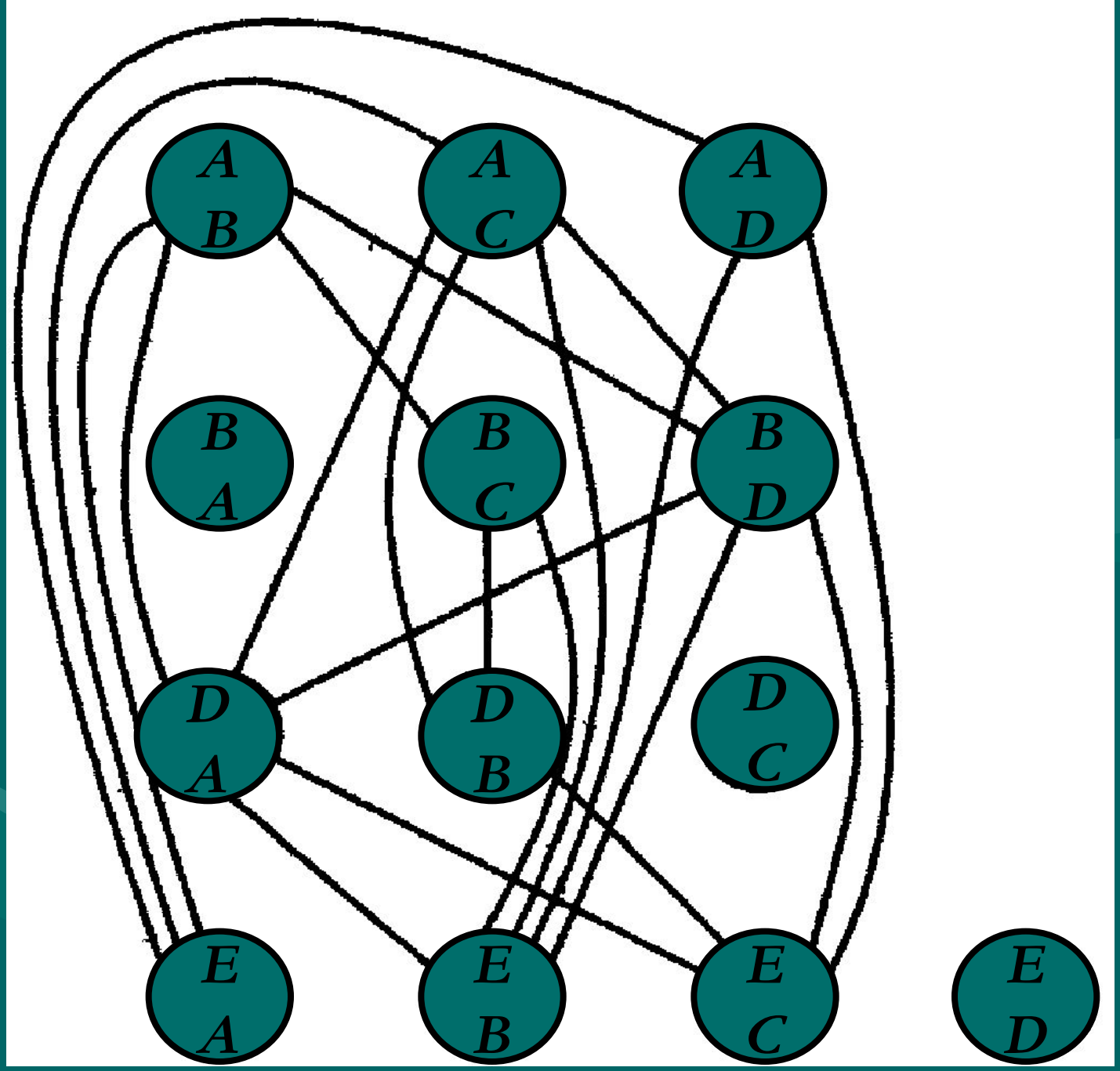
Например, для раскраски вершин 1,2,3,4,5 графа можно было бы применить два цвета, закрасив вершину 1 в красный цвет, а затем, пропустив вершину 2, закрасить в красный цвет вершины 3 и 4. Но "жадный" алгоритм, основываясь на порядковой очередности вершин, закрасит в красный цвет вершины 1 и 2, для закраски остальных вершин потребуются еще два цвета.



Применим описанный алгоритм для закраски вершин графа нашей задачи, при этом первой закрасим вершину AB в синий цвет. Также можно закрасить в синий цвет вершины AC, AD и BA, поскольку никакие из этих четырех вершин не имеют общих ребер.

Вершину BC нельзя закрасить в этот цвет, так как существует ребро между вершинами AB и BC. По этой же причине мы не можем закрасить в синий цвет вершины BD, DA и DB — все они имеют хотя бы по одному ребру, связывающему их с уже закрашенными в синий цвет вершинами. Продолжая перебор вершин, закрашиваем вершины DC и ED в синий цвет, а вершины EA, EB и EC оставляем незакрашенными.

И так далее


A B
A C
A D
B A
B C
B D
D A
D B
D C
E A
E B
E C
E D

Можно использовать результаты из общей теории графов для оценки качества полученного решения. В теории графов k-кликой называется множество из k вершин, в котором каждая пара вершин соединена ребром. Очевидно, что для закраски k-клики необходимо k цветов, поскольку в клике никакие две вершины не могут иметь одинаковый цвет.

Множество из четырех вершин AC, DA, BD и EB является 4-кликой. Поэтому не существует раскраски этого графа тремя и менее цветами, и, следовательно, решение является оптимальным в том смысле, что использует минимально возможное количество цветов для раскраски графа. В терминах исходной задачи это означает, что для управления перекрестком, необходимо не менее четырех режимов работы светофоров.

Переход от алгоритма к программе

greedy - жадный

procedure *greedy* (**var** *G*: GRAPH; **var** *newclr*: SET) ;

{ greedy присваивает переменной newclr множество вершин графа G, которые можно окрасить в один цвет}

begin

1) *newclr* := ∅;

2) **for** для каждой незакрашенной вершины *v* из *G* **do**

3) **if** *v* не соединена с вершинами из *newclr* **then**

begin

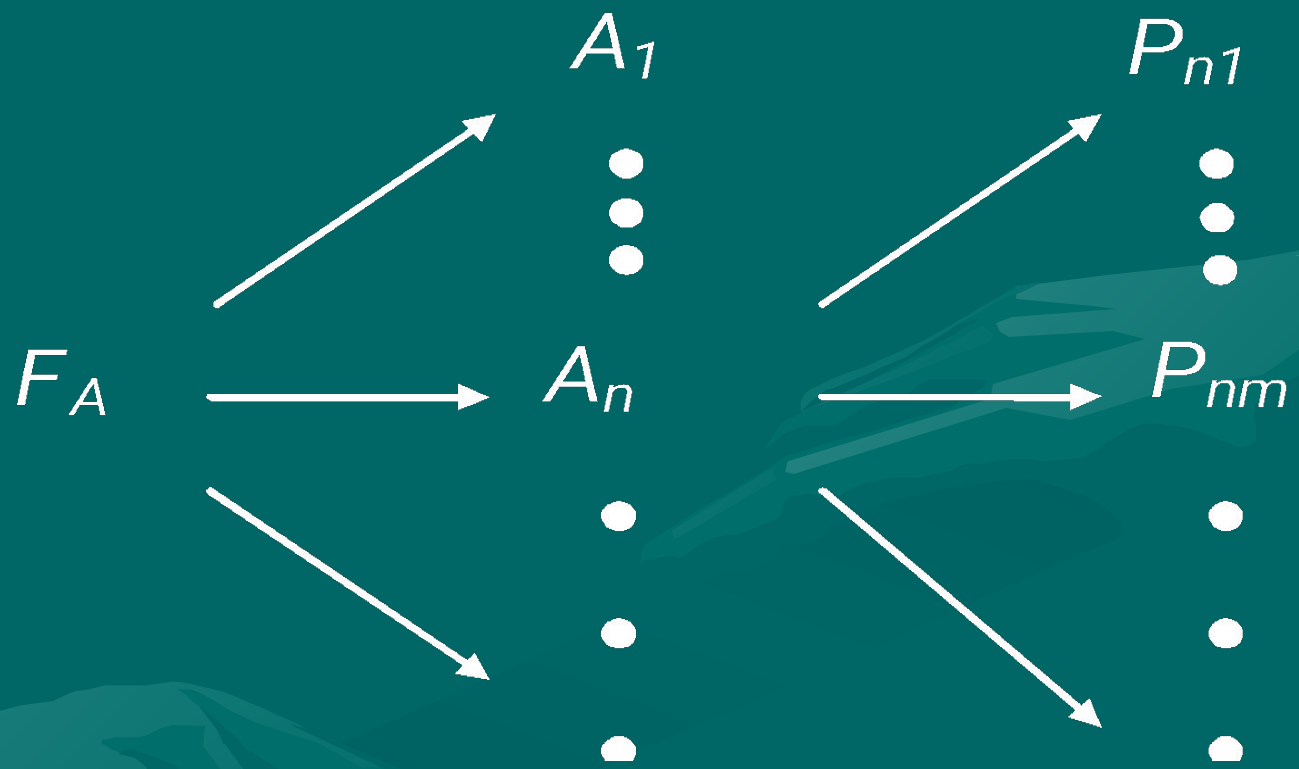
4) пометить *v* цветом;

5) добавить *v* в *newclr*

end

end;

```
procedure greedy ( var G: GRAPH; var newclr: LIST ) ;
 var   found: boolean;  v, w: integer;
 begin
    newclr:= 0;   v:= первая не закрашенная вершина из G;
 while v <> null do begin
     found:= false;   w:= первая вершина из newclr
     while w <> null do    begin
          if  существует ребро между  v  и  w  then
          found:= true;   w:= следующая вершина из newclr;
                    end;
     if found = false then   begin
          пометить v цветом;
          добавить v в newclr
                   end
     v:= следующая незакрашенная вершина из G;
                     end;
end;
```


A_1
P_{n1}
F_A
A_n
P_{nm}

Абстрактный тип данных LIST

ТИП ДАННЫХ

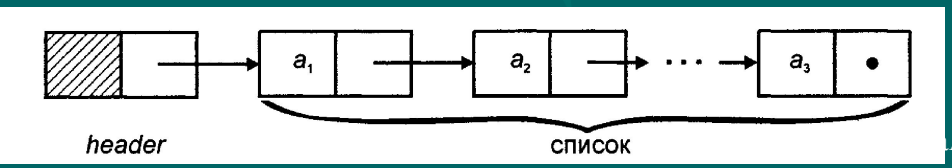
- область допустимых значений переменной
- Операции, которые могут обрабатывать эти значения

Можно реализовать тип данных любым способом, а программы, использующие объекты этого типа, не зависят от способа реализации типа — за это отвечают процедуры, реализующие операторы для этого типа данных.

ОПЕРАТОРЫ

1. Сделать список пустым (MAKENULL(*newdr*);)
2. Выбрать первый элемент списка и, если список пустой, возвратить значение *null* (*w* := FlRST(*newclr*);)
3. Выбрать следующий элемент списка и возвратить значение null, если следующего элемента нет (*w* := NEXT(*newclr*);)
4. Вставить элемент в список (INSERT (*v*, *newclr*);)

ЗНАЧЕНИЯ



Абстрактный тип данных SET

ОПЕРАТОРЫ

1. *MAKENULL(A);* Сделать множество *A* пустым
2. *UNION(A, B, C*). Эта процедура имеет два "входных" аргумента, множества *A* и *B*, и присваивает объединение этих множеств "выходному" аргументу — множеству *C*.
3. *SIZE(A)*. Эта функция имеет аргументом множество *A* и возвращает объект целого типа, равный количеству элементов множества
4. ADD (*x,A*) добавить элемент *x* в *A*

и другие

ЗНАЧЕНИЯ

Множество – файл, массив или список элементов.

Рекомендации по разработке программ

Планируйте этапы разработки программы.

- сначала черновой набросок алгоритма в неформальном стиле,
- затем псевдопрограмма,
- далее — последовательная формализация псевдопрограммы, т.е. переход к уровню исполняемого кода.

Такая стратегия организует и дисциплинирует процесс создания конечной программы, которая будет простой в отладке и в дальнейшей поддержке и сопровождении.

Применяйте инкапсуляцию.

- Все процедуры, реализующие АТД, поместите в одно место программного листинга.
- В дальнейшем, если возникнет необходимость изменить реализацию АТД, можно будет корректно и без особых затрат внести какие-либо изменения, так как все необходимые процедуры локализованы в одном месте программы.

Используйте и модифицируйте уже существующие программы.

- Один из неэффективных подходов к процессу программирования заключается в том, что каждый новый проект рассматривается "с нуля", без учета уже существующих программ. Обычно среди программ, реализованных на момент начала проекта, можно найти такие, которые если решают не всю исходную задачу, то хотя бы ее часть. После создания законченной программы полезно оглянуться вокруг и посмотреть, где еще ее можно применить (возможно, вариант ее применения окажется совсем непредвиденным).

Разрабатывайте свои инструменты.

- Инструмент (tool) — это программа с широким спектром применения.
- При создании программы подумайте, нельзя ли ее каким-либо образом обобщить, т.е. сделать более универсальной
- В частности, программу раскраски графа можно использовать не только для задания режимов работы светофоров, но для решения задач, совсем не связанных с задачей светофоров, например, для решения задачи составления расписаний

Пусть необходимо написать программу, составляющую расписание экзаменов. Вместо заказанной программы можно написать программу-инструмент, раскрашивающий вершины обычного графа таким образом, чтобы любые две вершины, соединенные ребром, были закрашены в разные цвета.

В контексте расписания экзаменов:

- вершины графа — это аудитории,
- цвета — время проведения экзаменов,
- а ребра, соединяющие две вершины-аудитории, обозначают, что в этих аудиториях экзамены принимает одна и та же экзаменационная комиссия.

Такая программа раскраски графа вместе с подпрограммой перевода списка аудиторий в множество вершин графа и цветов в заданные временные интервалы проведения экзаменов составит расписание экзаменов.

Программируйте на командном уровне.

Часто бывает, что в библиотеке программ не удается найти программу, необходимую для выполнения именно вашей задачи, но можно адаптировать для этих целей ту или иную программу. Развитые операционные системы предоставляют программам, разработанным для различных платформ, возможность совместной работы в сети вовсе без модификации их кода, за исключением списка команд операционной системы. Чтобы сделать команды компонуемыми, как правило, необходимо, чтобы каждая из них вела себя как фильтр. Отметим, что можно компоновать любое количество фильтров, и, если командный язык операционной системы достаточно интеллектуален, достаточно просто составить список команд в том порядке, в каком они будут востребованы программой.

Общность и эффективность решения задачи

Общие решения - менее эффективны.

Более узкие решения (одной задачи, например, а не целого класса) - более эффективны

Как следствие стало необходимым понятие *сложности* дискретных алгоритмов и задач, которое позволяет лучше понять причины неэффективности решений, понять, можно ли исключить полный перебор в решении задачи.

Задача об упаковке рюкзака:

Множество вещей $\{P \ni p \mid p=(v,w)\}$ Упаковать вещи так, чтобы суммарная ценность V была максимальной при суммарном весе не превосходящем W.

Другие задачи: планирование исполнения множества программ, размещение груза на корабле, и т.д. и т.п.

Полный ПЕРЕБОР Сложность = $O(n!)$

Метод ветвей и границ. Известно решение $V = k$, надо построить решение. Частично построенный вариант можно не испытывать, если он уже хуже оптимального.

Класс задач NP – множество всех переборных задач,

Класс P – переборные задачи, разрешимые за полиномиальное время

Построить оптимальную упаковку можно полным перебором, а иного, не переборного, алгоритма может и в принципе не существовать. Это и есть труднорешаемая задача.

СВОДИМОСТЬ

Переборная задача П1 сводится к переборной задаче П2, если метод решения П1 можно преобразовапть к методу решения П2.

Сводимость полиномиальная, если это преобразование можно сделать за полиномиальное время.

NP-полные задачи те, к которым полиномиально сводятся любая задача из NP, это класс универсальных в некотором смысле задач.

Если бы P= NP, тогда можно было бы надеяться создать эффективные полиномиальные алгоритмы решения переборных задач. Но к сожалению, это видимо не так и для решения каждой переборной задачи придется разрабатывать свои эффективные алгоритмы решения

Полиномиальная и экспоненциальная сложность

Функция $f(n)$ есть $O(g(n))$, если $|f(n)| \le c|g(n)|$.

Полиномиальный (полиномиальной временно́й сложностью) называется алгоритм, сложность которого $O(p(n))$, где $p(n)$ полином от n. Если алгоритм невозможно так оценить, то он имеет *экспоненциальную* сложность.

Скорость роста	n = 10	20	30	40
• линейная сложность	0,001	0,002	0,003	0,004
• квадратичная сложность	0,001	0,004	0,009	0,016 (n^2)
• показательная функция (экспоненциальная сложность)				
• 2^n	0.1	10		

Полнопереборные задачи:

- Найти в графе кратчайшее расстояние между вершинами A и B
- Раскраска графа. Задан граф G=(V,E), K≤|V|. Существует ли К-раскраска, при которой любые две связанные вершины раскрашиваются в разные цвета?

О вычислительной сложности алгоритма

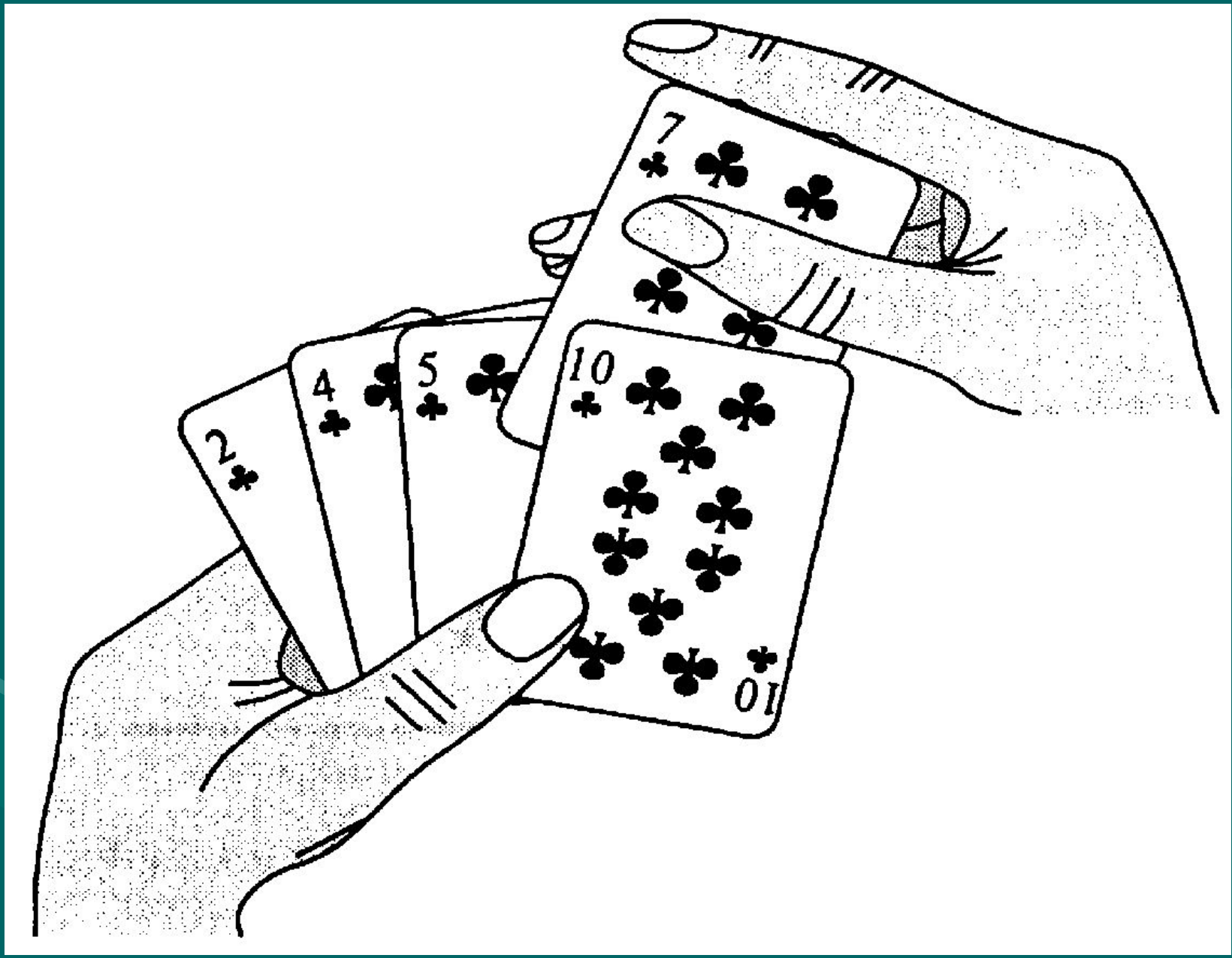
Сортировка вставкой

- Вход: последовательность из n чисел $(a_1, a_2,\ldots,a_n)$.
- Выход: перестановка $(a'_1, a'_2,\ldots, a'_n)$ входной последовательности таким образом, что для ее членов выполняется соотношение

$$a'_1 \le a'_2 \le \ldots, \le a'_n.$$

Карта представляется целым числом

- Массив $A\,[1..n]$ представляет $a_1, a_2,\ldots,a_n$
- Число элементов в A обозначено как $length\,[A].$)


7
2
4
5
10
10

```
INSERTION_SORT(A)
1   for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Вставка элемента A[j] в отсортированную
           ▷ последовательность A[1..j − 1]
4          i ← j − 1
5          while i > 0 и A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i − 1  end
8          A[i + 1] ← key

end
```

Инварианты цикла (loop invariant)

- В начале каждой итерации цикла **for** подмассив

A [1..j-1] содержит те же элементы, которые были в нем с самого начала, но расположенные в отсортированном порядке.

Инварианты циклов обладают следующими тремя свойствами.

1. Инициализация. Они справедливы перед первой инициализацией цикла.
2. Сохранение. Если они истинны перед очередной итерацией цикла, то остаются истинны и после нее.
3. Завершение. По завершении цикла инварианты позволяют убедиться в правильности алгоритма.

Анализ алгоритма

Время работы процедуры *Insertion__Sort* зависит от набора входных значений: для сортировки тысячи чисел требуется больше времени, чем для сортировки трех чисел.

Кроме того, время сортировки может быть разным для последовательностей, состоящих из одного и того же количества элементов, в зависимости от степени упорядоченности этих последовательностей до начала сортировки.

В общем случае время работы алгоритма увеличивается с увеличением количества входных данных, поэтому общепринятая практика — представлять время работы программы как функцию, зависящую от количества входных элементов.

Понятие *размер задачи* уточняется для каждой задачи. Это может быть число от числа элементов в массиве, от количества битов для представления входных данных, от количества вершин и дуг графа и т.д.

- Время работы алгоритма для того или иного ввода измеряется в количестве элементарных операций, или "шагов", которые необходимо выполнить. Каждый элементарный шаг выполняется за константное время.
- введем для процедуры *Insertion_Sort* время выполнения каждой инструкции и количество их повторений. Для каждого $j = 2,3,\ldots, n$, где $n = length\ [A]$, обозначим через t_j количество проверок условия в цикле **while**
- Время работы алгоритма (эффективность алгоритма) — это сумма времён, необходимых для выполнения каждой входящей в его состав исполняемой инструкции.
- Если выполнение инструкции длится в течение времени c_i и она повторяется в алгоритме n раз, то ее вклад в полное время работы алгоритма равно $c_i n$. Чтобы вычислить время работы алгоритма *Insertion_Sort* (обозначим его через $T(n)$), нужно просуммировать произведения значений, стоящих в столбцах время и количество раз, в результате чего получим:

Insertion_Sort(A) время количество раз

- 1 **for** $j \leftarrow 2$ **to** *length[A]* c_1 n
- 2 **do** *key* $\leftarrow$ *A[j]* c_2 $n - 1$
- 3 - Вставка элемента *A[j]* в отсортированную последовательность *A[1,...,j — 1]*. 0 $n - 1$
- 4 $i \leftarrow j - 1$ c_4 $n - 1$
- 5 **while** $i > 0$ and $A[i] > key$ c_5 $\sum_{j=2}^{n} t_j$
- 6 **do** $A[i + 1] \leftarrow A[i]$ c_6 $\sum_{j=2}^{n} (t_j - 1)$
- 7 $i \leftarrow i - 1$ c_7 $\sum_{j=2}^{n} (t_j - 1)$
- 8 $A[i + 1] \leftarrow key$ c_8 $n - 1$

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) +$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n-1).$$

величиной, время работы алгоритма может зависеть от степени упорядоченности сортируемых величин, которой они обладали до ввода. Например, самый благоприятный случай для алгоритма *Insertion__Sort* — это когда все элементы массива уже отсортированы. Тогда для каждого $j = 2,3,...., n$ получается, что $A[i] \leq key$ в строке 5, еще когда i равно своему начальному значению $j — 1$. Таким образом, при $j = 2,3,...., n$ $t_j = 1$, и время работы алгоритма в самом благоприятном случае вычисляется так:

$$T(n) = C_1 n + C_2 (n - 1) + C_4 (n - 1) + C_5 (n - 1) + C_8 (n - 1) =$$
$$= (C_1 + C_2 + C_4 + C_5 + C_8)\, n — (C_2 + C_4 + C_5 + C_8)$$

Это время работы можно записать как линейная функция $an + b$, где a и b — константы,

Если элементы массива отсортированы в порядке, обратном требуемому (в порядке убывания), то это наихудший случай. Каждый элемент *A [j]* необходимо сравнивать со всеми элементами уже отсортированного подмножества *A [l,...,j — 1],* так что для $j = 2,3,...., n$ значения $t_j = j$. С учетом того, что

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

получаем, что время работы алгоритма *Insertion_Sort* в худшем случае определяется соотношением

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) +$$

$$+ c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8 (n-1) =$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n -$$

$$- (c_2 + c_4 + c_5 + c_8) .$$

Это время работы можно записать как $an^2 + bn + c$, где константы a, b и c зависят от c_i. Таким образом, это квадратичная функция от n.

ПОРЯДОК роста функции

Огрубляем оценку роста $an^2 + bn + c$, отбрасывая члены меньших порядков и опуская коэффициент при n^2 (это константа) получаем порядок роста n^2. Алгоритм с меньшим порядком роста предпочтительнее, так как при росте n такой полином растет медленнее.

Рассматриваются как наилучший, так и наихудший случаи. Наихудший случай наиболее интересен по трем причинам:

Время работы алгоритма в наихудшем случае — это верхний предел этой величины для любых входных данных.

В некоторых алгоритмах наихудший случай встречается достаточно часто. Например, если в базе данных происходит поиск информации, то наихудшему случаю соответствует отсутствие нужной информация

Обычно один алгоритм считается эффективнее другого, если время его работы в наихудшем случае имеет более низкий порядок роста. Из-за наличия постоянных множителей и второстепенных членов эта оценка может быть ошибочной, если входные данные невелики. Однако, если объем входных данных значительный, то, например, алгоритм $O(n^2)$ в наихудшем случае работает быстрее, чем алгоритм $O(n^3)$ (*асимптотическая* оценка).

Построение алгоритма методом разделяй и властвуй

Рекурсивные алгоритмы. Задача разбивается на подзадачи. Затем эти задачи меньшего размера решаются рекурсивным вызовом разрабатываемой процедуры и так пока размер массива дойдет до единицы, а это уже упорядоченный массив.

Соединение двух упорядоченных массивов в один делается процедурой MERGE объединения двух упорядоченных массивов A[p..q] и A[q..n].

Динамическое программирование

Динамическое программирование — способ решения сложных задач путём разбиения их на более простые подзадачи, сложность которых меньше исходной

Чтобы решить поставленную задачу, требуется решить подзадачи, после чего объединить решения подзадач в одно общее решение. Часто многие из этих подзадач одинаковы. Подход динамического программирования состоит в том, чтобы решить каждую подзадачу только один раз, сократив тем самым количество вычислений.

В общем случае задача решается в три шага

- Разбиение задачи на подзадачи меньшего размера.
- Нахождение оптимального решения подзадач рекурсивно, проделывая такой же трехшаговый алгоритм
- Использование полученного решения подзадач для конструирования решения исходной задачи

Пример динамического программирования — алгоритм, позволяющий решить задачу о перемножении цепочки матриц. Пусть имеется последовательность (цепочка), состоящая из n матриц, и нужно вычислить их произведение $A_1A_2 \ldots A_n$. Это произведение можно вычислить, используя в качестве подпрограммы стандартный алгоритм умножения пар матриц. Однако сначала нужно расставить скобки, чтобы устранить все неоднозначности в порядке перемножения (зафиксировать порядок перемножения).

$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)\ A_4))$,
$((A_1(A_2A_3))\ A_4)$,
$(((A_1A_2)\ A_3)\ A_4)$.

то способ вычисления их произведения можно полностью определить с помощью скобок разными способами:

(A1(A2(A3A4)))
(A1((A2A3)A4)) ,
((A1(A2A3))A4) ,
(((A1A2)A3)AA) .

Матричное умножение обладает свойством ассоциативности, поэтому результат не зависит от расстановки скобок.

От того как расставлены скобки при умножении последовательности матриц, может сильно зависеть время, затраченное на вычисление произведения. Матрицы A и B можно перемножать, только если они совместимы: количество столбцов матрицы A должно совпадать с количеством строк матрицы B. Если A — это матрица размера $p \times q$, а B - матрица размера $q \times r$, то в результате их перемножения получится матрица C размера $p \times r$.

Время вычисления матрицы C преимущественно определяется количеством произведений скаляров. Это количество равно pqr. Это и есть стоимость умножения матриц

ПРИМЕР.

- Перемножаются три матрицы (A_1, A_2, A_3). Предположим, что размеры этих матриц равны

10 x 100, 100 x 5 и 5 x 50 соответственно. Перемножая матрицы в порядке, заданном выражением ((A_1 A_2) A_3), необходимо выполнить 10 *100 *5 = 5 000 скалярных умножений, чтобы найти результат произведения A_1, A_2 (при этом получится матрица размером 10 x 5), а затем — еще 10 * 5 * 50 = 2 500 скалярных умножений, чтобы умножить эту матрицу на матрицу A_3. Всего получается 7 500 скалярных умножений.

Если вычислять результат в порядке, заданном выражением (A_1 (A_2 A_3)), то сначала понадобится выполнить 100 *5 *50 = 25 000 скалярных умножений (при этом будет найдена матрица A_2 A_3 размером 100 x 50), а затем еще 10 *100*50 = 50 000 скалярных умножений, чтобы умножить A_1 на эту матрицу. Всего получается 75 000 скалярных умножений. Таким образом, для вычисления результата первым способом понадобится в 10 раз меньше времени. Следует полностью определить порядок умножений в матричном произведении (A_1 A_2 …, A_n), при котором количество скалярных умножений сведется к минимуму.

Первый этап применения парадигмы динамического программирования — найти оптимальную вспомогательную подструктуру, а затем с ее помощью сконструировать оптимальное решение задачи по оптимальным решениям вспомогательных задач. В рассматриваемой задаче этот этап можно осуществить следующим образом.

Обозначим для удобства результат перемножения матриц $A_i A_{(i+1)} \dots A_j$ через $A_{i.j}$, где $i \leq j$. Заметим, что если задача нетривиальна, т.е. $i < j$, то любой способ расстановки скобок в произведении $A_i \dots A_j$ разбивает это произведение на произведение матриц $A_{i..k}$ и $A_{(k+i).j}$, где к — целое, удовлетворяющее условию $i \leq к < j$. Таким образом, при некотором k сначала вычисляются матрицы, а затем они умножаются друг на друга, в результате чего получается произведение $A_{i..j}$. Стоимость, соответствующая этому способу расстановки скобок, равна сумме стоимости вычисления матрицы $A_{i,k}$, стоимости вычисления матрицы $A_{k+i..j}$ и стоимости вычисления их произведения.

Предположим, что в результате оптимальной расстановки скобок последовательность $A_i \ldots A_j$ разбивается на подпоследовательности $A_{i,k}$, и $A_{k+i..j}$ Тогда расстановка скобок в "префиксной" подпоследовательности $A_{i,k}$ тоже должна быть оптимальной. Иначе, если бы существовал более экономный способ расстановки скобок в последовательности $A_{i,k}$, то его применение позволило бы перемножить матрицы $A_i \ldots A_j$ еще эффективнее, что противоречит предположению об оптимальности первоначальной расстановки скобок. Аналогично можно прийти к выводу, что расстановка скобок в подпоследовательности матриц $A_{(k+i).j}$, возникающей в результате оптимальной расстановки скобок в последовательности $A_{(k+i).j}$, также должна быть оптимальной.

Для решения любой нетривиальной задачи об оптимальном произведении последовательности матриц всю последовательность необходимо разбить на подпоследовательности и при этом каждое оптимальное решение содержит в себе оптимальные решения подзадач. Другими словами, решение полной задачи об оптимальном перемножении последовательности матриц можно построить путем разбиения этой задачи на две подзадачи — оптимальную расстановку скобок в подпоследовательностях $A_i A_{i+1} \ldots A_k$ и $A_{k+i} A_{k+2} \bullet \bullet - A_j$. После этого находятся оптимальные решения подзадач, из которых затем составляется оптимальное решение полной задачи. Необходимо убедиться, что при поиске способа перемножения матриц учитываются все возможные разбиения — только в этом случае можно быть уверенным, что найденное решение будет глобально оптимальным.

Организация больших массивов

Алгоритмы размещения и поиска данных.

Определить алгоритмы вычисления функций РАЗМЕСТИТЬ, НАЙТИ и УДАЛИТЬ элемент.

- Массив помещается в оперативной памяти .

Прямая адресация, как в массиве программы: Ключ элемента – целочисленный номер.

Лексикографическое упорядочивание. Ключ элемента – текст имени элемента

Hash-таблицы

- Массив большой и, например, не помещается весь в оперативной памяти, но в каждый момент времени обрабатывается ограниченное количество элементов. В этом случае используются hash- функции.

Какой должна быть хорошая hash-функция?

Хорошая хеш-функция должна (приближенно) удовлетворять предположениям равномерного хеширования: для очередного ключа все m хеш-значений должны быть равновероятны. Чтобы

На практике при выборе хеш-функций пользуются различными эвристиками, основанными на специфике задачи. Например, компилятор языка программирования хранит таблицу символов, в которой ключами являются идентификаторы программы. Часто в программе используется несколько похожих идентификаторов (например, `pt` и `pts`). Хорошая хеш-функция будет стараться, чтобы хеш-значения у таких похожих идентификаторов были различны.

Построение хеш-функции методом **деления с остатком** (division method) состоит в том, что ключу k ставится в соответствие остаток от деления k на m, где m — число возможных хеш-значений:

$$h(k) = k \bmod m.$$

Например, если размер хеш-таблицы равен $m = 12$ и ключ равен 100, то хеш-значение равно 4.

Хорошие результаты обычно получаются, если выбрать в качестве m простое число, далеко отстоящее от степеней двойки.

Построение хеш-функции методом **умножения** (multiplication method) состоит в следующем. Пусть количество хеш-значений равно m. Зафиксируем константу A в интервале $0 < A < 1$, и положим

$$h(k) = \lfloor m(kA \bmod 1) \rfloor,$$

где $kA \bmod 1$ — дробная часть kA.

Программа удаления совпадающих элементов

```
procedure PURGE ( var L: LIST );
var   p, q: position;   { p - "текущая" позиция в списке L,
                          q - перемещается вперед от позиции p }
begin
      p:= FIRST(L);
          while p <> END(L) do begin
              q:= NEXT(p, L);
            while q <> END(L) do
              if same (RETRIEVE(p, L), RETRIEVE(q, L))
                   then DELETE(q, L)
                  else  q:= NEXT(q, L);
                p:= NEXT(p, L)
            end
 end;
```