Лабораторная работа №1-2

Анализ алгоритмов и их сложности

Цель работы: провести анализ алгоритма и оценить его сложность. Приобретение навыков исследования временной сложности алгоритмов и определения ее асимптотических оценок.

Вариант 11

Задачи и многообразие алгоритмов их решения

За многовековую практику человечество убедилось, что для большинства задач существует более одного способа их решения, и, следовательно, можно сформулировать несколько алгоритмов, приводящих к одному и тому же результату. В качестве примера, иллюстрирующего такое положение дел, рассмотрим задачу возведения в степень.

Задача возведения в степень. Дано число x и натуральное (целое неотрицательное) число n > 0. Вычислить значение функции .

Очевидный способ решения этой задачи следует из определения операции возведения в степень: искомый результат есть:

.

Положим f = 1, далее в цикле, повторяющемся n раз, вычислим произведение f=f\*x. По окончании цикла f будет содержать искомый результат . На рис. 1 приведена блок-схема этого алгоритма.



Рисунок 1 - Тривиальный алгоритм возведения в степень

Однако можно предложить и другие способы решения.

Проведем анализ формулы:

Здесь и далее символ div обозначают операцию целочисленного деления.

Подсчет временной сложности: для вычисления Si требуется 1 умножение. Всего такая итерация осуществляется n раз. Таким образом, временная сложность этого алгоритма: Т(п) = n умножений = n операций.

То есть таким образом мы видим то что алгоритм линейный (О(n)) и время выполнения такого алгоритма 1 секунда

Задание 2

Это pyton

#Вариант 1

def caboom(a, b):

return a\*\*b

#Вариант 2

def caboom2(a,b):

f = 1

for i in range(b):

f = f\*a

return f

print(caboom(2,4))

print(caboom2(2,4))

Контрольные вопросы

1.Временная сложность алгоритма - это функция от размера входных данных, равная количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера.

2. Теоретическое нахождение времени выполнения программ (даже без определения констант пропорциональности) — сложная математическая задача. Однако на практике определение времени выполнения (также без нахождения значения констант) является вполне разрешимой задачей – для этого нужно знать только несколько базовых принципов.

3.

1. При оценке за функцию берется количество операций, возрастающее быстрее всего. То есть, если в программе одна функция, например, умножение, выполняется O(n) раз, а сложение – O(n2) раз, то общая сложность программы – O(n2), так как в конце концов при увеличении n более быстрые ( в определенное, константное число раз) сложения станут выполнятся настолько часто, что будут влиять на быстродействие куда больше, нежели медленные, но редкие умножения. Это принцип обуславливает правило сумм асимптотических соотношений. Пусть T1(n) и Т2(п) — время выполнения двух программных фрагментов Р1 и Р2, T1(n) имеет степень роста O(f(n)), а Т2(п) — O(g(n)). Тогда Т1(п) + Т2(п), т.е. время последовательного выполнения фрагментов Р1 и Р2, имеет степень роста O(max(f(n), g(n))).

2. При оценке O( ) константы не учитываются. Пусть один алгоритм делает 2500n + 1000 операций, а другой – 2n+1. Оба они имеют оценку O(n), так как их время выполнения растет линейно. Этот принцип является следствием правила произведений, которое заключается в следующем. Если T1(n) и Т2(п) имеют степени роста O(f(n)) и O(g(n)) соответственно, то произведение T1(n)T2(n) имеет степень роста O(f(n)g(n)).

3. Другое следствие опускания константы – алгоритм со временем O(n2) может работать значительно быстрее алгоритма O(n) при малых n. За счет того, что реальное количество операций первого алгоритма может быть n2 + 10n + 6, а второго – 1000000n + 5. Впрочем, второй алгоритм рано или поздно обгонит первый. n2 растет куда быстрее 1000000n.

4. Основание логарифма внутри символа O( ) не пишется. Причина этого весьма проста. Пусть у нас есть O(log2n). Но log2n=log3n/log32, а log32, как и любую константу, асимптотика – символ О( ) не учитывает. Таким образом, O(log2n) = O(log3n). К любому основанию мы можем перейти аналогично, а значит и писать его не имеет смысла.

4.

1.Время выполнения оператора присваивания складывается из времени вычисления выражения и времени выполнения операции присваивания. Если выражение не содержит вызовов функций (в том числе перегруженных операций), то на его вычисление тратится некоторое постоянное время. Операция присваивания так же требует некоторого постоянного времени. Таким образом, время выполнения операторов присваивания обычно имеет порядок О(1).

2. Время выполнения операторов ввода – вывода обычно имеет порядок О(1).

3. Время выполнения последовательности операторов определяется с помощью правила сумм. Поэтому степень роста времени выполнения последовательности операторов без определения констант пропорциональности совпадает с наибольшим временем выполнения оператора в данной последовательности.

4. Время выполнения условных операторов состоит из времени выполнения условно исполняемых операторов и времени вычисления самого логического выражения. Время вычисления логического выражения обычно имеет порядок O(1). Время для всей конструкции if-then-else состоит из времени вычисления логического выражения и наибольшего из времени, необходимого для выполнения операторов, исполняемых при значении логического выражения true (истина) и при значении false (ложь).

5. Время выполнения цикла является суммой времени всех исполняемых итераций цикла, в свою очередь состоящих из времени выполнения операторов тела цикла и времени вычисления условия прекращения цикла (обычно последнее имеет порядок O(1)). Часто время выполнения цикла вычисляется, пренебрегая определением констант пропорциональности, как произведение количества выполненных итераций цикла на наибольшее возможное время выполнения операторов тела цикла. Время выполнения каждого цикла, если в программе их несколько, должно определяться отдельно.

6. Для программ, содержащих несколько процедур или функций (среди которых нет рекурсивных), можно подсчитать общее время выполнения программы путем последовательного нахождения времени выполнения этих подпрограмм, начиная с той, которая не имеет вызовов других процедур или функций. Так как мы предположили, что все подпрограммы нерекурсивные, то должна существовать хотя бы одна подпрограмма, не имеющая вызовов других процедур или функций. Затем можно определить время выполнения подпрограмм, вызывающих эту подпрограмму, используя уже вычисленное время выполнения вызываемой подпрограммы. Продолжая этот процесс, найдем время выполнения всех подпрограмм и, наконец, время выполнения всей программы.

7. Если есть рекурсивные процедуры или функции, то нельзя упорядочить все подпрограммы таким образом, чтобы каждая подпрограмма вызывала только процедуры или функции, время выполнения которых подсчитано на предыдущем шаге. В этом случае мы должны с каждой рекурсивной процедурой (функцией) связать временною функцию Т(n), где n определяет объем ее аргументов. Затем мы должны получить рекуррентное соотношение для T(n), т.е. уравнение (или неравенство) для Т(п), где участвуют значения T(k) для различных значений k<n.

Общий метод решения таких рекуррентных соотношений состоит в последовательном раскрытии выражений Т(к) в правой части уравнения (путем подстановки в исходное соотношение k вместо n) до тех пор, пока не получится формула, у которой в правой части отсутствует Т. При этом часто приходится находить суммы различных последовательностей; если значения таких сумм нельзя вычислить точно, то для сумм находятся верхние границы, что позволяет, в свою очередь, получить верхние границы для Т(n).