Практика 1 -2 - 3 Анализ алгоритмов.

1. Напишите алгоритм , подсчитывающий количество каких –либо букв в текстовом файле. Подсчитайте, сколько сравнений требуются этому алгоритму? Каково максимальное возможное значение числа операций увеличения счетчика? Минимальное такое число? Выразите ответ через число N символов во входном файле.(N – общее число символов в файле.)

2. Напишите алгоритм, который получает на входе три целых числа, и находит наибольшее из них. Каковы возможные классы входных данных? На каком из них Ваш алгоритм делает наибольшее число сравнений? На каком меньше всего? Если разницы между наилучшим и наихудшим классами нет, то перепишите свой алгоритм с простыми сравнениями так, чтобы он не использовал временных переменных, и чтобы в наилучшем случае он работал быстрее, чем в наихудшем.

3. Напишите алгоритм, не использующий сложных условий, который по трем введенным целым числам определяет, различны ли они все между собой? Сколько сравнений в среднем делает ваш алгоритм? Обязательно исследуйте все классы входных сигналов.

4. Составить алгоритм нахождения максимального или минимального из нескольких(3-4) чисел двумя способами. На основе составленных алгоритмов написать программы. Провести сравнения этих алгоритмов по эффективности.

Мерой эффективности выполнения алгоритма считается асимптотическая временная сложность. Это время выполнения алгоритма в самом худшем случае .

Для начала посчитайте, сколько в вашей программе различных операций: есть ли в вашей программе присваивания, сравнения, приращения и сколько их.

Анализируя свой алгоритм, проверьте, сколько у вас операций сравнения и арифметических операций. Операторы сравнения проверяют равенство или неравенство величин между собой.

Арифметические операции делятся на две группы – первая группа – сложение, вычитание, увеличение и уменьшение счетчика.

Вторая группа – умножение деление, взятие остатка по модулю.
Есть еще третья группа – логарифмы, тригонометрические функции. Они еще более времяемкие.

Замечание Целочисленное умножение или деление на степень двойки приводит к обычному сдвигу, что эквивалентно сложению.

Классы входных сигналов.

Роль входных данных очень велика, так как последовательность действий алгоритма определяется не в последнюю очередь входными данными.
Например, ищем наибольший элемент в списке из N элементов.:

```
largest=list[1]
for i=2 to N do
    if(list[i]>largest) then
        largest=list[i]
    end if
end for
```

Какие могут быть варианты?

1) Список упорядочен в порядке убывания. Тогда перед началом цикла будет сделано одно присваивание, а в теле цикла присваиваний не будет.

2) Ну а если список упорядочен по возрастанию? Тогда будет сделано N присваиваний (одно перед началом цикла и N-1 в цикле. Если мы проводим анализ, то не должны ограничиваться одним множеством, вдруг окажется, что это самое быстрое, или наоборот самое медленное множество? Мы должны рассматривать все возможные типы множеств. Для этого разбивают различные входные множества на классы. В зависимости от поведения алгоритма на каждом множестве.

Пример. Число различных перестановок 10 различных чисел в списке 10!=3 628 800. А мы хотим найти наибольший элемент. У нас есть 362880 входных множеств, у которых первое число является наибольшим. Их все можно поместить в один класс.

Если наибольшее число стоит на 2 месте, то алгоритм сделает ровно 2 присваивания. Множеств, в которых наибольшее число стоит на втором месте, 362880. Их можно отнести к другому классу. Мы видим, как будет меняться число присваиваний при изменении положения наибольшего числа от 1 до N. А это значит, что мы должны разбить все входные множества на N различных классов по числу сделанных присваиваний. Нет необходимости выписывать или описывать детально все множества, помещенные в каждый класс. Надо знать лишь количество классов и объем работы на каждом множестве класса.

Число возможных наборов входных данных может стать очень большим при увеличении N. 10 разных чисел можно расположить в списке 3628800 способами. Все эти способы рассмотреть невозможно. Но мы разбили списки на классы в зависимости от того, что будет делать алгоритм. У нашего алгоритма разбиение основывается на местоположении наибольшего значения. В результате получили 10 разных классов.

Для алгоритма поиска, например, наименьшего и наибольшего значения разбиение можно основывать на том, где располагается наибольшее и наименьшее значения. В таком разбиении 90 классов. Когда классы выделены, то можно посмотреть на поведение алгоритма на одном множестве из каждого класса.

Если классы выделены правильно, то на всех множествах входных данных одного класса алгоритм производит одинаковое количество операций, а на множестве из другого класса это количество операций, скорее всего будет другим.

Кроме сложности алгоритмов по времени есть еще и сложность по памяти. У карманных компьютеров, у которых до 10 мегабайт памяти на все – и на данные и на программу становится актуальной экономия памяти

Итак, выбор входных данных может существенно повлиять на выполнение алгоритма с точки зрения времени его выполнения.

Наилучший случай.

Это такой набор входных данных, на котором алгоритм выполняется за минимальное время.

Что значит за минимальное время? Это значит, что мы выбираем алгоритм , который выполняет меньше всего действий.

Если исследуется алгоритм поиска, то набор исходных данных является наилучшим, если искомое значение (целевое значение) записано в первой проверяемой алгоритмом ячейке. Этому алгоритму вне зависимости от его сложности, потребуется всего одно сравнение. При поиске в списке, каким бы длинным он ни был, наилучший случай требует постоянного времени.

Наихудший случай.

Позволяет узнать максимальное время работы алгоритма. Для анализа наихудшего случая необходимо найти входные данные, на которых алгоритму придется выполнять больше всего работы. Для алгоритма поиска - это список, в котором искомый элемент окажется последним в списке. Или вообще будет отсутствовать. В результате может потребоваться N сравнений. Анализ наихудшего случая дает верхние оценки для работы программы в зависимости от выбранных алгоритмов.

Средний случай

Самый сложный, так как он требует учета множества разнообразных деталей. В основе анализа – разбиение возможных входных наборов данных на группы. Потом следует определить вероятность, с которой входной набор данных принадлежит каждой группе. Затем подсчитывается время работы алгоритма на данных из каждой группы. Время работы на всех входных данных одной группы должно быть одинаковым, иначе группу следует разбить на подгруппы. Среднее время работы вычисляется по формуле

$$A(n)= \sum_{i=1}^{m} p_i t_i$$

Где через n обозначен размер входных данных, через m- число групп, через p- вероятность того, что входные данные принадлежат группе с номером I, а через t – время, необходимое алгоритму для обработки данных из группы с номером i.

В некоторых случаях мы будем предполагать, что вероятность попадания входных данных в каждую из групп одинакова. (Если, например, групп 5, то вероятность попасть в первую группу такая же как вероятность попасть во 2 и т. д., то есть вероятность попасть в каждую группу 0.2.

Тогда можно воспользоваться упрощенной формулой

$$A(n)= \frac{1}{m}\sum_{i=1}^{m} t_i$$

Задача

Напишите алгоритм, который по данному списку чисел и среднему значению этих чисел определяет, превышает ли число элементов списка, больших среднего значения, число элементов , меньших этого значения, или наоборот. Опишите группы, на которые распадаются возможные наборы входных данных. Какой случай для алгоритма является наилучшим? Наихудшим? Средним?

Если наилучший и наихудший случаи совпадают, то перепишите алгоритм так, чтобы он останавливался, как только ответ на поставленный вопрос становится известным, делая наилучший случай лучше наихудшего.

(решение этой задачи зависит от того , знаем ли мы заранее длину входных данных или она выясняется в процессе работы алгоритма.)

Скорости роста и их классификация.

Наиболее важным при анализе алгоритмов является скорость роста количества операций при возрастании объема входных данных. Это называется скоростью роста алгоритма. Интересно изучить, как поведет себя алгоритм при увеличении объема данных. Скорость роста разных функций при увеличении объема входных данных различна. При малых объемах входных данных разница между функциями практически мало заметна, но при больших объемах входных данных мы можем выделить быстро растущие функции и функции с медленным ростом.

Составьте таблицу классов роста различных функций

$\log_2 n$, n, $n\log n$, n^2 , n^3 на некотором диапазоне значений аргумента (1,2,5,10,20,30,40,50,60,70,80,90,100)

Скорость роста определяется старшим членом формулы. Так как младшие растут медленно, ими можно пренебречь. Отбросив младшие члены, получаем то, что принято называть порядком функции или алгоритма. Алгоритмы таким образом можно группировать по скорости роста их сложности.

Алгоритмы сгруппированы в три категории:

- Алгоритмы, сложность которых растет по крайней мере так же быстро, как и данная функция.
- Алгоритмы, сложность которых растет с той же скоростью
- Алгоритмы, сложность которых растет медленнее, чем эта функция.

Класс функций $\Omega(f)$. (омега большое) (функция, принадлежащая этому классу, ограничена снизу функцией f)

Функция g принадлежит этому классу, если при всех значениях аргумента n , больших некоторого порога n_o значение g(n) > c*f(n) для некоторого положительного числа c. Класс задается указанием своей нижней границы. А это значит, что в класс $\Omega(n^2)$ входят все функции, растущие быстрее, чем n^2 (n^3 или 2^n).Этот класс мало интересен, если говорить об эффективности алгоритма.

Если есть функция f(n) и функция cg(n) , то

$\exists c > 0, n0 > 0 : 0 =< c * g(n) =< f(n)$ (то есть функция g(n) лежит под кривой f(n))

Класс функций O(f) (O большое) (функция, принадлежащая этому классу, ограничена сверху функией f)

Этот класс состоит из функций, растущих не быстрее f .Функция f образует *верхнюю границу для класса*. Функции g принадлежит классу O(f), если g(n) <= c*f(n) для всех n, больших некоторого порога n_0 , и для некоторой положительной константы c. Это важно для нас. У нас есть два алгоритма. Принадлежит ли сложность первого из них классу O большое от сложности второго. Если окажется, что это так, значит, второй алгоритм не лучше первого решает поставленную задачу.

(оценка O требует только, чтобы функция f(n) не превышала g(n), начиная с n>n0, с точностью до постоянного множителя:

$\exists c > 0, n0 > 0 : 0 =< f(n) =< c * g(n), \forall n > n0$ (то есть функция f(n) не должна превышать g(n) , начиная с n>n0,с точностью до постоянного множителя.)

Класс θ(f) (тета большое) (функция, принадлежащая этому классу, ограничена снизу и сверху функцией f)

Это класс функций, растущих с той же скоростью, что и f. Этот класс представляет собой пересечение двух предыдущих классов,

θ (f) = $\Omega(f)$ $\cap$ O(f)

При сравнении алгоритмов нас будут интересовать такие, которые решают задачу быстрее, чем уже изученные. Алгоритмы этого класса не очень интересны.

($f(n)=\theta(g(n))$, если существуют положительные c1,c2,n0, такие, что c1*g(n)=<f(n)=<c2*g(n), при n>n0.) Обычно говорят, что при этом функция g(n) является асимптотически точной оценкой функции f(n), так как по определению функция f(n) не отличается от функции g(n) с точностью до постоянного множителя. При этом из $f(n)=\theta(g(n))$ следует , что $g(n)=\theta(f(n))$

Самый для нас интересный класс O. Проверить принадлежность функции классу O(f) можно либо с помощью данного выше определения, либо так:

$g \in O(f)$, если $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ для некоторой константы c.

Это означает, что если предел отношения g(n)/f(n) существует и он меньше бесконечности, то $g \in O(f)$. Это не просто проверить для некоторых функций. В таких случаях по правилу Лопиталя (раскрывающего неопределенности вида ноль на ноль или бесконечность на бесконечность) можно заменить предел самих функций пределом их производных.

Каждый из классов является множеством, поэтому имеет смысл выражение “g – элемент этого множества”. И когда мы пишем g=O(f) , то это значит $g \in O(f)$.Примеры

Примеры

$f(n) = 4n^2 + n\ln N + 174 - f(n) = \theta(n^2)$;

Расположите следующие функции в порядке возрастания:

2^n , $\log_2 \log_2 n$, $n^3 + \log_2 n$, $n^2, n^3, n\log_2 n, 6, n, n - n^2 + 5n^3$

Для приведенных пар функций f и g выполняется одно из равенств: либо f=O(g), либо g=O(f), но не оба сразу. Определите, какой из случаев имеет место.

$f(n) = (n^2 - n)/2, g(n) = 6n$

$f(n) = n + 2\sqrt{n}, g(n) = n^2$

$f(n) = n + n\log_2 n, g(n) = n\sqrt{n}$

$f(n) = n^2 + 3n + 4, g(n) = n^3$

Выводы:

Итак, общее количество операций, выполняемых алгоритмом, не играет существенной роли при анализе алгоритмов. А вот скорость роста этого числа при возрастании объема входных данных очень важна. Она - то и называется скоростью роста алгоритма.

Зная такие оценки для разных алгоритмов решения какой-либо задачи, мы получаем возможность сравнивать эти алгоритмы с точки зрения их эффективности. Однако асимптотические оценки дают нам только порядок функции, а сами результаты сравнения справедливы только при очень больших размерах входа.

В реальной работе важно знать среднее количество операций, совершаемых алгоритмом. Здесь обычно рассматриваются три случая :

- Худший - для наибольшего количества операций, совершаемых алгоритмом для решения конкретных проблем размерности n.

- Лучший - для наименьшего количества операций совершаемых алгоритмом для решения конкретных проблем размерности n.

- Средний -для среднего количества операций, совершаемых алгоритмом для решения конкретных проблем размерности n.

Конкретная проблема представляет собой упорядоченное множество элементов(слов).

При решении на ЦВМ практических задач возникает вопрос о границах практического применения данного алгоритма решения задачи в смысле ограничения на ее размерность.