Лабораторная работа №6
ИЗУЧЕНИЕ МЕТОДОВ ОЦЕНКИ АЛГОРИТМОВ

Цель работы. Изучение методов оценки алгоритмов и программ и определение временной и емкостной сложности типовых алгоритмов и программ.

Алгоритм – это точное предписание о выполнении в определенном порядке некоторых операций, приводящих к решению всех задач данного класса.

Важнейшей характеристикой алгоритма и соответствующей ему программы является их сложность, которая может оцениваться:

a) временем решения задачи (трудоемкостью алгоритма);

b) требуемой емкостью памяти.

В общем случае сложность алгоритма можно оценить по порядку величины. Этот метод применим как к временной, так и к емкостной сложности.

Оценка порядка (временная оценка)

Эта оценка считается наиболее существенной. Для определения трудоемкости алгоритма обычно используют следующий набор «элементарных» операций, характерных для языков высокого уровня.

1. Простое присваивание: *a*<-- *b*, будем считать ее сложность, равной 1;
2. Одномерная индексация *a[i*] : (адрес (*a*)+*i**длина элемента),будем считать ее сложность, равной 2;
3. Арифметические операции: (*, /, -, +), будем считать ее сложность, равной 1;
4. Операции сравнения: *a* < *b*, будем считать ее сложность, равной 1;

5. Логические операции ($l1$) {or, and, not} ($l2$), будем считать ее сложность, равной 1;

С их помощью трудоемкость основных алгоритмических конструкций можно представить так.

1. *Линейная конструкция* из k последовательных блоков

Трудоемкость конструкции есть сумма трудоемкостей блоков, следующих друг за другом.

$$\Theta = f_1 + f_2 + \ldots + f_k,$$

где f_k – трудоемкость k-го блока.

2. Конструкция «Ветвление»

```
if(Условие) {
    Операторы
} else {
    Операторы
}
```

Общая трудоемкость конструкции «Ветвление» требует анализа вероятности выполнения операторов, стоящих после «Then» (p) и «Else» ($1-p$) и определяется как:

$$\Theta = f_{then}p + f_{else}(1-p),$$

где f_{then} и f_{else} – трудоемкости соответствующих ветвей.

3. Конструкция «Цикл»

```
for(i=1;i<=n;i++){
    Тело_цикла
}
```

После сведения этой конструкции к элементарным операциям ее трудоемкость определяется как:

$$\Theta = 1 + 3*n + n*f_{\text{цикла}},$$

где $f_{\text{цикла}}$ – сложность тела цикла,

n– число его повторений,

$3*n$ – трудоемкость изменения параметра и проверки условия выполнения цикла.

Сложность линейного участка и ветвления, как правило, не зависит от объема данных и считается равной некоторой константе. При этом алгоритмы, содержащие циклы и рекурсии, являются более сложными. Величина Θ возрастает с увеличением вложенности циклов, а также при многократном вызове методов обработки данных (процедур).

Рассмотрим примеры анализа простых алгоритмов.

Пример 1.Задача суммирования элементов квадратной матрицы размерностью *n*n.*

```
SumM (A, n; Sum)
Sum<-- 0
For i<-- 1 to n
For j<-- 1 to n
Sum <-- Sum + A[i,j]
endfor
Return (Sum)
End
```

Алгоритм выполняет одинаковое количество операций при фиксированном значении *n* и, следовательно, является количественно-зависимым. Применение методики анализа конструкции «Цикл» дает:

$$\Theta(n)=1+f_{\text{цикла по i}}= 1 + 1+ 3*n+ n *f_{\text{цикла по j}} =$$

$$1 + 1+ 3*n + n *(1+ 3*n + n *(2 + 2 + 1 + 1))$$

$$= 2 + 3n + n *(1 + 3n + 6\, n) = 9n^2+4\, n +2. \qquad (1.1)$$

Пример 2.Задачапоискамаксимумаводномерном массиве.

```
MaxS (S,n; Max)
Max <-- S[1]
For i<-- 2 ton
if Max <S[i]
Max <-- S[i]
end for
```

```
return Max
End
```

Приведенный алгоритм является количественно-параметрическим, т.е. зависит от значений исходных данных. Поэтому для фиксированной размерности исходных данных необходимо проводить анализ для худшего, лучшего и среднего случая.

1) Худший случай

Максимальное количество переприсваиваний максимума (на каждом проходе цикла) будет в том случае, если элементы массива отсортированы по возрастанию. При этом трудоемкость алгоритма будет равна:

$$\Theta_x(n)=1+ 2 +f_{\text{цикла по i}}= 1+2+1+ 3(n-1) +(n-1)\,(3+2+1+1) =$$
$$3 +1 + 7(n–1) = 7n–3. \qquad (1.2)$$

Здесь 1+2 – задание начального значения максимума (`Max<--S[1]`),

3(n-1) – на выполнение цикла,

(2 +1)(n-1) – на сравнение,

(n-1) – на запись S[i] на место Max.

2) Лучший случай

Минимальное количество переприсваиваний максимума (ни одного на каждом проходе цикла) будет в том случае, если максимальный элемент расположен на первом месте в массиве. При этом трудоемкость алгоритма будет равна:

$$\Theta_{\text{л}}(n)=1+ 2 +1+ 3(n-1) + (n-1)\,(2 +1) = 6n-2. \qquad (1.3)$$

3) Средний случай

Алгоритм поиска максимума последовательно перебирает элементы массива, сравнивая текущий элемент с текущим значением максимума. На очередном шаге, когда просматривается k-тый элемент массива, переприсваивание максимума произойдет, если в подмассиве из первых k элементов максимальным является последний. Очевидно, что в случае равномерного распределения

исходных данных, вероятность того, что максимальный из k элементов расположен в определенной (последней) позиции равна $1/k$. Тогда в массиве из n элементов общее количество операций переприсваивания максимума определяется как:

$$\sum_{k=1}^{n} 1/k = H_n \approx \ln(n) + \gamma, \gamma \approx 0.57 . \qquad (1.4)$$

Величина H_n называется n-ым гармоническим числом. Таким образом, при бесконечном количестве испытаний точное значение среднего числа операций присваивания в алгоритме поиска максимума в массиве из n элементов определяется величиной H_n, а средняя трудоемкость:

$$\Theta_{ср}(n)=1 + 2 + (n\text{-}1)\,(3+2) + 2\,(Ln(n) + \gamma)=$$
$$5n +2\,Ln(n) - 1 +2\,\gamma . \qquad (1.5)$$

Общий случай определения временной сложности

Выражения (1.1) – (1.5) представляют собой ***асимптотические*** оценки сложности соответствующих алгоритмов. В теории алгоритмов они являются семейством функций, дающих множество значений, в том числе верхнее и нижнее. Основным свойством $\Theta(n)$является то, что при увеличении размерности входных данных n время выполнения алгоритма возрастает с той же скоростью, что и функция $f(n)$.

Верхняя оценка сложности алгоритма$f(n)$обозначается $O(n)$ – *греческая буква «омикрон».* Считается, что она пропорциональна максимальному элементу в формуле $\Theta(n)$(см. (1.1) – (1.5)). Такая оценка легче вычисляется и дает предельное значение. Именно она получила наибольшее распространение.

Так, в примере 1 (нахождение суммы элементов квадратной матрицы) она определяется величиной $O(n^2)$, а в примере 2 (нахождение максимума в одномерном массиве) для всех случаев– $O(n)$. Вообще для циклов вида:

```
for(i=0;i<n;i++)
     for(j=0;j<n;j++)
          for(k=0;k<n;k++){
               Телоцикла
          }
```

сложность равна O(n^3), т.е. пропорциональна числу повторений самого внутреннего цикла.

Сложность рекурсивных алгоритмов зависит не только от сложности внутренних циклов, но и от количества итераций рекурсии. Такая процедура может выглядеть достаточно простой, но она может серьѐзно усложнить программу, многократно вызывая себя.

Рассмотрим рекурсивную реализацию вычисления факториалав среде java.

```
intfactorial(intn){
     if(n > 1)
          returnn*factorial(n-1);
     else
          return1;
};
```

Она выполняется *n* раз. Таким образом, вычислительная сложность этого алгоритма равна O*(n)*.

O-функции выражают относительную скорость алгоритма в зависимости от некоторой переменной или переменных. При этом используются три ***правила оценки сложности***.

1. O*(k*f)* = O*(f)*,
2. O*(f*g)* = O*(f)**O*(g)*,
3. O*(f+g)* = *Max (*O*(f),* O*(g)*).

Здесь *k* обозначает константу, а *f* и *g* - функции.

Первое правило означает, что постоянные множители не имеют значения для определения порядка сложности, например, O*(2*n)* = O*(n)*. В соответствии со вторым порядок сложности произведения двух функций равен произведению их сложностей. Например,

$O((10*n)*n) = O(n)*O(n) = O(n^2)$. На основании третьего правила порядок сложности суммы двух функций равен максимальному (доминирующему) значению из их сложностей. Например, $O(n^4 + n^2) = O(n^4)$.

На практике применяются следующие виды О-функций:

a) O(1) – константная сложность, для алгоритмов, сложность которых не зависит от размера данных.

b) $O(n)$ – линейная сложность, для одномерных массивов и циклов, которые выполняются n раз,

c) $O(n^2)$, $O(n^3)$, … – полиномиальная сложность, для многомерных массивов и вложенных циклов, каждый из которых выполняется n раз,

d) $O(log(n))$ – логарифмическая сложность, для программ, в которых большая задача делится на мелкие, которые решаются по-отдельности,

e) $O(2^n)$ – экспоненциальная сложность, для очень сложных алгоритмов, использующих полный перебор или так называемый «метод грубой силы».

Функции перечислены в порядке возрастания сложности. Чем выше в этом списке находится функция, тем быстрее будет выполняться алгоритм с такой оценкой.

Обычно определение сложности алгоритма, в основном, сводится к анализу циклов и рекурсивных вызовов.

Экспериментальный метод оценки трудоемкости (сложности) алгоритма

Метод основан на измерении времени выполнения алгоритма на некотором наборе данных (тесте). При этом используются стандартные средства языка программирования, позволяющие определить системное время компьютера. Например, для среды java это могут быть методы System.CurrentTimeMillis (),позволяющий фиксировать время с точность до миллисекунд и System.nanoTime() –

до наносекунд. Последний имеет ограничения и работает не на всех платформах.

Для оценки трудоемкости некоторого алгоритма достаточно запомнить время перед его выполнением и зафиксировать в конце, например, с помощью следующего фрагмента:

```
longstart = System.nanoTime();
// Фрагмент программы, реализующий
// исследуемыйалгоритм
long end = System.nanoTime();
longtraceTime = end-start;
```

Искомое время в наносекундах будет находиться в переменной *traceTime*.

Быстродействие современных процессоров с тактовой частотой в несколько Гигагерц имеет порядок нескольких десятков миллиардов операций в секунду. При этом время выполнения простых алгоритмов может быть очень малым, а измерение его предлагаемым методом будет иметь большую погрешность. Для ее уменьшения осуществляют многократное выполнение исследуемого фрагмента внутри цикла(например, повторяют его несколько десятков или сотен тысяч раз), а величину *traceTime* делят на число повторений цикла.

Оценка пространственной (емкостной) сложности

Такая оценка выполняется аналогично временной. При этом если для реализации алгоритма требуется дополнительная память для одной, двух или трех переменных, то емкостная сложность алгоритма будет константной, т.е. O(1). Такую сложность имеет, например, рассмотренный выше алгоритм поиска максимума. Если дополнительная память пропорциональна *n*, то пространственная сложность равна O(*n*) и т.д.

Для всех рекурсивных алгоритмов также важно понятие емкостной сложности. При каждом вызове метод запрашивает небольшой объѐм памяти, но этот объем может значительно

увеличиваться в процессе рекурсивных вызовов. Так, при вычислении факториала на первом этапе (разворачивания рекурсии) значения сомножителей помещаются в стек. Их количество равно n. При этом пространственная сложность алгоритма будет равна $O(n)$. В других случаях эта характеристика может быть еще больше. Таким образом,всегда необходимо проводить анализ емкостной сложности рекурсивных процедур.

Временная и емкостная сложность взаимосвязаны. Это обусловлено тем, что память современных ЭВМ имеет иерархическую структуру. При выполнении программы процессор, прежде всего, работает с данными, находящимися в быстрой кэш- памяти. Если массив данных имеет большую размерность, то они могут не поместиться в кэше. При их обработке будут происходить кэш-промахи, которые сильно замедляют выполнение программы.

Оценка временной сложности в таком случае должна осуществляться экспериментально.

Порядок выполнения лабораторной работы

Работа предполагает выполнение следующих этапов.

1. Знакомство со всеми разделами руководства.
2. Получение у преподавателя задания на асимптотическую и верхнюю оценку сложности алгоритма (см. прил. 1) и выполнение этой оценки.
3. Оценка экспериментальным способом времени выполнения того же алгоритма. Значения исходных данных необходимо задавать в начале работы программы с помощью генератора случайных чисел, причем делать это до начала измерения времени работы алгоритма. Сам алгоритм в ходе измерений должен выполняться в цикле несколько миллионов раз, чтобы он не заканчивал работу слишком быстро, а выполнялся хотя бы несколько секунд.

4. Измерения необходимо повторить пять раз для различного объема исходных данных. Количество повторений алгоритма в каждом измерении должно быть одинаковым.
5. Построить график зависимости времени выполнения от объема входных данных.

Содержание отчета о выполненной работе

Отчет о выполненной работе должен содержать.

1. Название и цель работы.
2. Формулы асимптотической и верхней оценки сложности заданного алгоритма.
3. Исходный код программы экспериментальной оценки временной сложности заданного алгоритма для массива большой размерности.
4. Значения временной сложности алгоритма, полученные экспериментальным способом, а также количество повторений алгоритма и объем исходных данных, при котором были получены эти значения.
5. График зависимости времени выполнения алгоритма от объема исходных данных.

Контрольные вопросы

1. Чем характеризуется сложность алгоритма?
2. Как оценивается асимптотическая сложность алгоритма?
3. Как получается верхняя оценка сложности алгоритма?
4. Отличаются ли и на сколько асимптотическая и верхняя оценка сложности алгоритма?
5. Какие функции используются для представления верхней оценки сложности алгоритма?
6. У каких известных вам алгоритмов сложность является константной, а у каких - линейной?
7. Как оценивается сложность экспериментальным методом?

8. Совпадают ли результаты экспериментальной и верхней оценок и, если нет, то на сколько они отличаются?

Практические задания к лабораторной работе №6

1. Составить программу, которая формирует одномерный массив из *n*случайных чисел. Определить среднее арифметическое этих чисел. Значение *n* меняется в пределах от 10 до 50 миллионов.
2. Составить программу, которая формирует одномерный массив из*n*случайных чисел. Получить из него два новых массива: один из четных чисел, а другой из нечетных. Значение *n* меняется в пределах от 10 до 50 миллионов.
3. Составить программу, которая формирует одномерный массив из*n*случайных чисел. Определить сумму отрицательных чисел и отдельно сумму остальных. Значение *n* меняется в пределах от 10 до 50 миллионов.
4. Составить программу, которая формирует одномерный массив из*n*случайных чисел. Определить количество четных чисел и количество нечетных чисел. Значение *n* меняется в пределах от 10 до 50 миллионов.
5. Составить программу, которая формирует одномерный массив из*n*случайных чисел. Отдельно определить произведение четных чисел, и произведение нечетных чисел. Значение *n*меняется в пределах от 10 до 50 миллионов.
6. Составить программу, которая формирует матрицу из *n***n*случайных чисел. Определить произведение чисел, лежащих на главной диагонали матрицы. Значение *n* меняется в пределах от 5 до 10 тысяч.
7. Составить программу, которая формирует матрицу из *n***n*случайных чисел. Определить произведение чисел, лежащих на побочной диагонали матрицы. Значение *n* меняется в пределах от 5 до 10 тысяч.

8. Составить программу, которая формирует матрицу из *n***n*случайных чисел. Определить сумму чисел, лежащих выше главной диагонали матрицы. Значение *n* меняется в пределах от 5 до 10 тысяч.

9. Составить программу, которая формирует матрицу из n*n случайных чисел. Определить произведение всех чисел в матрице. Значение *n*меняется в пределах от 5 до 10 тысяч.

10. Составить программу, которая формирует матрицу из n*n случайных чисел. Определить сумму отрицательных чисел и отдельно сумму остальных. Значение *n* меняется в пределах от 5 до 10 тысяч.

11. Составить программу, которая формирует матрицу из n*n случайных чисел. Определить количество четных чисел и количество нечетных. Значение *n* меняется в пределах от 5 до 10 тысяч.