Введение в алгоритмы

Определение и свойства алгоритма

Алгоритм - любая корректно определенная вычислительная процедура, на вход (input) которой подается любая величина или набор величин и результатом выполнения которой является выходная (output) величина или набор величин. Т.о., алгоритм - конечная последовательность вычислительных шагов, преобразующая входные величины в выходные.

Свойства алгоритма:

- *дискретность*: алгоритм - процесс последовательного посроения величин, идущий в дискретном времени таким образом, что в начальный момент задается исходная конечная система величин, а в каждый следующий момент система величин получается по определенному закону (программе) из системы величин, имевшихся в предыдущий момент времени
- *детерминированность (определенность)*: система величин, получаемая в любой (не начальный) момент времени, однозначно определяется системой величин, полученных в предшествующие моменты времени
- *элементарность шагов*: закон получения последующей системы величин из предшествующей должен быть простым и локальным (не требовать дополнительной информации; быть реализуемым на компьютере)
- *результативность (направленность)*: если способ получения какой-либо величины из предыдущей величины не дает результата, должно быть указано, чтО надо считать результатом алгоритма; алгоритм не может завершаться безрезультатно
- *конечность*: итоговая система величин (результат) должна получаться за конечное число шагов
- *массовость*: начальная система величин может выбираться из некоторого потенциально бесконечного множества

Алгоритм корректен, если для любых входных данных результатом его работы являются корректные выходные данные.

Алгоритм может быть задан на естественном языке, в виде программы, псевдокода, блок-схемы и т.д. Главное требование - его спецификация должна предоставлять точное описание вычислительной процедуры, которую требуется выполнить.

Анализ алгоритмов

Анализ алгоритма заключается в том, чтобы предсказать требуемые для его выполнения ресурсы. Основные ресурсы, потребляемые алгоритмом (реализующей алгоритм

программой) - процессорное время и память. Т.к. ресурсы компьютера не бесконечны, желательны разработка и использование алгоритмов, эффективных в плане расхода времени и/или памяти.

Для анализа алгоритмов в качестве технологии реализации принята модель обощенной однопроцессорной машины с памятью с произвольным доступом: RAM (Random Access Machine). Машина в модели RAM обладает следующими свойствами:

- последовательное исполнение инструкций (нет параллельного исполнения)
- поддерживаются элементарные операции: арифметические, перемещение данных (чтение/запись, копирование), управляющие (ветвление, вызов подпрограммы)
- есть целочисленный тип данных и тип данных с плавающей точкой. Есть верхний предел слова данных
- фиксированное (константное) время всех операций с памятью (чтение/запись)

Временная сложность алгоритма - количество элементарных шагов (инструкций), требующихся для выполнения алгоритма. Временная сложность может быть представлена как функция, зависящая от входных данных или их параметра (например, размера входного массива для алгоритма сортировки). Чаще всего рассматривается наихудший случай, когда временная сложность алгоритма для заданного параметра входных данных максимальна.

Для нахождения временной сложности алгоритма с помощью модели RAM требуется просуммировать “стоимость” всех элементарных операций, исполняющихся алгоритмом, для заданного параметра входных данных.

Пример:

```
a, b, c: array of n elements
for i = 1..n:
      c[i] = a[i]*b[i]
```

Если предположить, что операция чтения из памяти требует C_R тактов процессора (секунд или других единиц времени), записи в память - C_W, умножение - C_M, итоговая временная сложность алгоритма $T(n) = 2*C_R*n + C_W*n + C_M*n = (2*C_R + C_W + C_M)*n = C*n$.

Для упрощение анализа алгоритмов используется понятие скорости или порядка роста. Временная сложность алгоритма $T(n) = O(g(n))$: $\exists\ c > 0, n_0: \forall\ n \geq n_0, 0 \leq T(n) \leq c*g(n)$, т.е. $g(n)$ - верхняя граница скорости роста $T(n)$ и $T(n)$ растет не быстрее чем $g(n)$. В этом случае говорят, что алгоритм имеет порядок $O(g(n))$ или порядок роста $g(n)$.

Для предудущего примера $T(n) = C*n = O(n)$, т.е. алгоритм имеет линейный порядок роста. В оценке порядка роста можно пренебречь коэффициентами и членами с меньшей скоростью роста. Пример: $T(n) = 10*n^2 + 5*n + 12 = O(n^2)$.

T(n) = O(g(n)) также называется асимптотической временной сложностью алгоритма. Нахождение асимптотической временной сложности алгоритма зачастую намного проще нахождения точной формулы для временной сложности. Следует искать наиболее точную оценку сложности: например, $O(n^2)$ и $O(n^3)$ обе являются асимптотическими оценками для $T(n) = 4*n^2 + 7$, но первая точнее второй. Алгоритм с меньшей асимптотической временной сложностью (порядком роста) считается более эффективным.

Типы и структуры данных

Тип данных (переменной) - множество значений, которые может принимать переменная.

Абстрактный тип данных - математическая модель плюс различные операторы, определенные в рамках этой модели.

Структура данных - представление абстрактного типа данных; способ хранения и организации данных, облегчающий доступ к этим данным и их модификацию. Любая структура данных имеет как преимущества, так и недостатки, и не является универсальной. Структура данных подразумевает под собой множество алгоритмов для работы с этой структурой данных; в свою очередь, многие алгоритмы включают в себя использование определенной структуры данных.

Примеры структур данных: массив, список, стек, очередь, дерево и т.д. Базовые структуры данных в C++ (библиотека STL, пространство имен std): vector, list, stack, queue, deque, set, map, string.

Домашнее задание

Номер варианта для выполнения = свой номер в списке группы % количество вариантов

1. Найти временную сложность алгоритма:

```
a, b, c: array of n elements
for i = 1..n:
      if a[i] > 0:
            c[i] = a[i]*b[i]
      else:
            c[i] = a[i] + b[i]
```

2. Найти временную сложность алгоритма:

```
fact(n: integer):
      if n == 0:
            return 1
      else
            return n*fact(n - 1)
```

3. Найти временную сложность алгоритма:

```
fib(n: integer):
      if (n == 0) || (n == 1):
            return 1
      else:
            return fib(n - 1) + fib(n - 2)
```

4. Найти временную сложность алгоритма:

```
a: array of n elements
for i = 1..n-1:
      for j = 1..n-i:
            if a[j] > a[j + 1]:
                  swap(a[j], a[j + 1])
```

5. Найти временную сложность алгоритма:

```
a: array of n elements
flag = false
do:
      flag = true
      for j = 1..N-1:
            if a[j] > a[j + 1]:
                  swap(a[j], a[j + 1])
                  flag = false
while (flag == false)
```

6. Найти временную сложность алгоритма:

```
mult_naive(a: integer, b: integer):
      x = a
      y = b
      z = 0
      while x > 0:
            z = z + y
            x = x - 1
      return z
```

7. Найти временную сложность алгоритма:

```
mult_egyptian(a: integer, b: integer):
      x = a
      y = b
      z = 0
      while x > 0:
            if x % 2 == 1:
                  z = z + y
            y = y * 2
            x = x / 2
      return z
```

8. Найти временную сложность алгоритма:

```
func(n: integer):
      x = n
      z = 0
      while x > 0:
            z = z + 1
            x = x / 2
      return z
```

9. Найти временную сложность алгоритма:

```
func(n: integer):
      if n == 1:
            return 1
      else:
            return 2*func(n/2)
```

10. Найти значение n, при котором алгоритм с временной сложностью $100*\ln(n)$ становится эффективнее алгоритма с временной сложностью $3*n^2$.
11. Найти значение n, при котором алгоритм с временной сложностью $150*n + 128$ становится эффективнее алгоритма с временной сложностью $9*n^2 + 2$.
12. Найти значение n, при котором алгоритм с временной сложностью $500*n*\ln(n)$ становится эффективнее алгоритма с временной сложностью $4*n^3$.
13. Найти значение n, при котором алгоритм с временной сложностью $1000*n^2*\ln(n)$ становится эффективнее алгоритма с временной сложностью $3*2^n$.
14. Компьютер 1 может выполнить 100 инструкций в секунду, компьютер 2 - 20000 инструкций в секунду. Найти значение n, начиная с которого выполнение требующего $2000*n$ инструкций алгоритма на компьютере 1 будет занимать меньше времени, чем выполнение требующего $2*n^3$ инструкций алгоритма на компьютере 2.
15. Компьютер 1 может выполнить 200 инструкций в секунду, компьютер 2 - 50000 инструкций в секунду. Найти значение n, начиная с которого выполнение требующего $100*n^2$ инструкций алгоритма на компьютере 1 будет занимать меньше времени, чем выполнение требующего n^3 инструкций алгоритма на компьютере 2.
16. Компьютер 1 может выполнить 1000 инструкций в секунду, компьютер 2 - 800000 инструкций в секунду. Найти значение n, начиная с которого выполнение требующего $500*n*\ln(n)$ инструкций алгоритма на компьютере 1 будет занимать меньше времени, чем выполнение требующего $3*2^n$ инструкций алгоритма на компьютере 2.
17. Компьютер 1 может выполнить 30 инструкций в секунду, компьютер 2 - 6000 инструкций в секунду. Найти значение n, начиная с которого выполнение требующего $2000*n^2*\ln(n)$ инструкций алгоритма на компьютере 1 будет занимать меньше времени, чем выполнение требующего $8*n*3^n$ инструкций алгоритма на компьютере 2.

18. Найти временную сложность (точную формулу) алгоритма подбора пароля длины n, с алфавитом из K символов (K ≥ n), если все символы в пароле могут повторяться произвольное число раз.
19. Условия из упражения 10, но символы в пароле не могут повторяться.
20. Условия из упражения 10, но символы в пароле не могут повторяться больше двух раз.
21. Условия из упражения 10, но первый и последний символы пароля совпадают.
22. Условия из упражения 10, но любой символ в пароле может повторяться не более трех раз.

Контрольные вопросы

1. Определение алгоритма. Свойства алгоритма.
2. Корректность алгоритма. Примеры некорректных алгоритмов.
3. Анализ алгоритмов. Критерии эффективности алгоритма.
4. Временная сложность алгоритма. Примеры алгоритмов с различной временной сложностью.
5. Random Access Machine: определение и свойства.
6. Порядок роста алгоритма. Примеры алгоритмов с различным порядком роста.
7. Структуры данных: определение и примеры.
8. Сравнение эффективности структур данных:
    a. std::vector и std::list
    b. обычное дерево и сбалансированное дерево поиска

Литература

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: Построение и анализ, 3-е издание. “Вильямс”, 2013.
2. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы. “Вильямс”, 2000.
3. Мальцев А.И. Алгоритмы и рекурсивные функции, 2-е издание. М.: Наука, 1986.
4. Кнут Д. Искусство программирования, том 1-3, 2-е издание. “Вильямс”, 2012.
5. Седжвик Р. Фундаментальные алгоритмы на C++, части 1-5. Изд-во “Диасофт”, 2001.
6. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.
7. Sedgewick R., Wayne K. Algorithms, 4th edition. Addison-Wesley, 2011.
8. Levitin A. Introduction to the design and analysis of algorithms, 3rd edition. Addison-Wesley, 2012.
9. Goodrich M., Tamassia R., Mount D. Data structures and algorithms in C++, 2nd edition, John Wiley & Sons, Inc., 2011.
10. Lafore R. Data structures and algorithms in Java, 2nd edition. Sams Publishing, 2003.