

# 4.

## Как писать правильные программы

В конце 60-х годов много говорилось о возможности автоматической верификации программ. К сожалению, за последующие десятилетия никаких систем автоматической проверки так и не появилось. Несмотря на отсутствие результата, исследования, которые были проведены в области верификации программ, дали нам гораздо больше, чем черный ящик, переваривающий программу и выдающий вердикт «хорошая» или «плохая», — люди достигли понимания глубинных основ программирования. Цель этой главы — показать, каким образом понимание этих основ может помочь в написании программ. Один из читателей охарактеризовал подход большинства программистов к этому вопросу следующим образом: «Напишите программу, перебросьте ее текст через стенку в соседний кабинет, и пусть отделы гарантии качества и тестирования разбираются с ошибками». В этой главе мы опишем альтернативный подход. Прежде чем обратиться к самой теме, мы рассмотрим ее со всех сторон. Умение правильно кодировать — лишь одно из множества требований для написания правильных программ. Самое сложное — то, о чем мы говорили в предыдущих трех главах, — правильно поставить задачу, разработать алгоритм и выбрать подходящую структуру данных. Если вы справитесь с этим, кодирование, скорее всего, не вызовет проблем.

### 4.1. Двоичный поиск бросает вызов

Даже если на этапе разработки все было проделано наилучшим образом, программиста зачастую ожидает написание весьма сложного кода. Эта глава посвящена одной задаче, для которой требуется особенно аккуратное кодирование, — двоичному поиску. Мы уже рассмотрели ранее эту проблему и сделали набросок алгоритма. Теперь мы воспользуемся основными принципами верификации, чтобы правильно написать программу.

Впервые мы столкнулись с двоичным поиском в разделе 2.2 (см. главу 2), где нужно было определить, содержит ли отсортированный массив  $x$   $[0..n-1]$  элемент  $t$ <sup>1</sup>. Мы точно знали, что  $n$  неотрицательно и что каждый последующий элемент массива не меньше предыдущего. Кроме того, мы знали, что если  $n=0$ , то массив пуст. Тип элементов  $t$  и  $x$  совпадает; псевдокод работает одинаково хорошо с целыми и вещественными числами и со строками. Ответ помещается в целое число  $p$  (от слова позиция): если элемент в массиве отсутствует, в  $p$  помещается число  $-1$ . В противном случае  $p$  неотрицательно и не превышает  $n-1$ , и  $t=x[p]$ .

Алгоритм двоичного поиска решает задачу, уменьшая размер поддиапазона, в котором может находиться  $t$  (если это число вообще есть в массиве). В начале диапазоном является весь массив. Затем его средний элемент сравнивается с  $t$ , после чего половина диапазона исключается из рассмотрения. Процесс продолжается до тех пор, пока число  $t$  не будет найдено или пока диапазон, в котором оно должно находиться, не окажется пуст. Для массива из  $n$  элементов двоичный поиск выполняет около  $\log_2 n$  сравнений.

Большинство программистов полагают, что по подобному описанию написать программу не составляет труда. Они ошибаются. Единственный способ заставить вас поверить в это: отложите книгу и попробуйте сами написать программу. Попробуйте!

Я предлагал проделать подобную работу слушателям курсов для профессиональных программистов. Им давалось несколько часов на то, чтобы сделать программу по данному им описанию на произвольном, выбранном ими языке, кроме того, разрешалось воспользоваться псевдокодом высокого уровня. К концу предложенного времени почти все отвечали, что они готовы. На проверку их кода с помощью тестов отводилось еще тридцать минут. Статистика по нескольким классам, в которых обучалось более ста человек, получилась такая: 90% программистов нашли ошибки в своих программах (и я не всегда был уверен в правильности текстов тех, у кого явных ошибок обнаружено не было).

Я был удивлен. Имея в запасе практически неограниченное время, только 10% профессиональных программистов смогли написать эту небольшую программу правильно. Эта задача оказалась не по зубам не только моим ученикам: раздел 6.2.1 третьего тома книги Кнута «Искусство программирования для ЭВМ: сортировка и поиск» посвящен истории этого алгоритма, и в нем Кнут отмечает, что, хотя первая программа двоичного поиска была опубликована в 1946 году, программа, не содержащая ошибок, появилась лишь в 1962 году.

## 4.2. Пишем программу

Основная идея двоичного поиска состоит в том, что мы всегда знаем, что если  $t$  вообще есть в массиве, то оно должно быть в выбранном нами поддиапазоне. Мы будем использовать выражение `mustbe (range)` для отражения того факта, что если  $t$  есть в массиве, то оно есть и в исследуемом на данном этапе диапазоне `range`. Мы

<sup>1</sup> Обратитесь к задаче 1 в главе 5 и ее решению, если вам нужна помощь в наведении критики на короткие имена переменных, формат определения функции двоичного поиска, обработку ошибок и другие проявления стиля программирования, жизненно важные для больших программных проектов.

воспользуемся этой записью, чтобы превратить описание двоичного поиска в набросок программы (листинг 4.1):

**Листинг 4.1.** Набросок программы двоичного поиска

```
initialize range to 0 n-1
/* инициализируем диапазон 0 n-1 */
loop
/* цикл */
{ invariant mustbe(range) }
/* инвариант число t должно быть в диапазоне range */
if range is empty. /* если диапазон пуст */
    break and report that t is not in array
/* цикл завершается. t в массиве нет */
compute m, the middle of the range
/* находим m - середину диапазона */
use m as a probe to shrink the range
/* используем значение m для сужения диапазона */
if t is found during the shrinking process
    break and report its position
/* если при этом находим t */
/* завершаем цикл и возвращаем позицию t */
```

Важной частью этой программы является *инвариант цикла* (loop invariant), заключенный в фигурные скобки. Это утверждение, относящееся к состоянию программы, называется *инвариантом*, поскольку оно остается истинным в начале и в конце каждой из итераций; это формализация того интуитивного утверждения, которое мы сделали выше.

Теперь займемся усовершенствованием нашей программы, заботясь о сохранности инварианта в ходе всех наших действий. Прежде всего, нам нужно как-то представить диапазон. Для этого мы будем использовать два индекса  $l$  и  $u$ , задающие нижнюю и верхнюю границы диапазона  $l..u$ . Функция двоичного поиска в разделе 9.3 главы 9 представляет диапазон другим способом: заданием его начала и длины. Логическая функция `mustbe(l, u)` утверждает, что если число  $t$  вообще есть в массиве, то оно обязательно принадлежит диапазону  $x[1..u]$ , включая границы диапазона.

Затем нужно заняться инициализацией. Какими должны быть значения  $t$  и  $u$ , чтобы утверждение `mustbe(l, u)` было истинным? Очевидный выбор:  $0$  и  $n-1$ : утверждение `mustbe(0, n-1)` сводится к тому, что если  $t$  есть в массиве  $x$ , то оно принадлежит  $x[0..n-1]$ , а именно это и известно нам с самого начала. Следовательно, инициализация будет состоять в присваивании значений  $l=0$  и  $u=n-1$ .

Затем нужно проверить диапазон на наличие в нем вообще каких-либо элементов, после чего найти его середину  $t$ . Диапазон  $l..u$  является пустым, если  $l > u$ , и в этом случае мы помещаем в переменную  $p$  специальное значение  $-1$ , завершая цикл.

```
if (l > u)
    p = -1; break
```

Оператор `break` завершает внутренний цикл, к которому он относится. Следующее выражение вычисляет значение  $m$  (середину диапазона):

```
m = (l+u)/2
```

Здесь используется целочисленное деление:  $6/2=3$ , и  $7/2=3$ . Итак, наша усовершенствованная программа выглядит теперь следующим образом (листинг 4.2):

**Листинг 4.2.** Вторая версия программы двоичного поиска

```

l=0;  u = n-1
loop
  {invariant: mustbe(l,u) }
  if l>u
    p = -1; break
  m = (l + u)/2
  use m as a probe to shrink the range l..u
  if t is found during the shrinking process
    break and note its position
/* русский перевод см в предыдущем листинге */

```

Теперь нужно в явном виде записать последние три строки. Для этого необходимо сравнить  $t$  и  $x[t]$ , после чего выполнить определенное действие для сохранения инварианта. Действие это будет выполняться по следующей схеме:

```

case
  x[m] < t  action a
  x[m] == t  action b
  x[m] > t  action c

```

Действие  $b$ , очевидно, будет состоять в присваивании  $p=t$  и выходе из цикла, поскольку выполнение условия означает, что мы нашли элемент, равный  $t$ . Оставшиеся два случая симметричны, мы займемся первым, а второй получим из него по соображениям симметрии (это одна из причин, по которым мы будем очень аккуратны при проверке кода в следующем разделе).

Если  $x[m]<t$ , мы понимаем, что  $x[0]<=x[1]<=..<=x[m]<t$ , поэтому  $t$  не может принадлежать диапазону  $x[0..m]$ . Поскольку мы знаем также, что  $t$  не может *не* принадлежать  $x[l..u]$ , мы понимаем, что если  $t$  где-нибудь есть, то оно принадлежит диапазону  $x[m+1..u]$ , что мы запишем в форме `mustbe(m+1, u)`. Поэтому мы восстановим инвариант `mustbe(l, u)` путем присваивания  $l = m+1$ . Уточнив, таким образом, предыдущий набросок, получим листинг 4.3.

**Листинг 4.3.** Двоичный поиск на псевдокоде: версия 3

```

l = 0  u = n - 1
loop
  { mustbe(l u) }
  if l > u
    p = -1 break
  m = (l + u) / 2
  case
    x[m] < t  l = m + 1
    x[m] == t  p = m break
    x[m] > t  u = m - 1

```

Программа получилась короткой: десяток строк кода и один инвариант. Простейший способ проверки программы — запись инварианта и сохранение его неизменным в процессе написания программы — оказался очень полезным при переписывании наброска алгоритма на псевдокоде. Этот процесс дал нам некоторую уверенность в программе, но мы ни в коем случае не можем считать ее правильной. Прежде чем читать дальше, потратьте несколько минут, чтобы проверить, правильно ли она работает.

## 4.3. Понимание программы

Когда мне приходится сталкиваться со сложной проблемой, требующей аккуратного программирования, я пытаюсь записать код с уровнем детализации, который

только что был продемонстрирован. Затем я использую методы верификации для повышения собственной уверенности в правильности программы. Верификация на этом уровне рассматривается в главах 9, 11 и 14.

В этом разделе мы изучим процесс верификации программы двоичного поиска в его наиболее подробном виде. На практике я анализирую программы гораздо менее формально. Текст программы, приведенный в листинге 4.4, наполнен утверждениями, формализующими интуитивные соображения, использовавшиеся при написании кода.

Хотя код разрабатывался сверху вниз (вначале — общая идея, затем — детализация), анализ корректности будет производиться снизу вверх: мы будем рассматривать строки по отдельности, а затем посмотрим, как они работают вместе при решении задачи.

---

#### ВНИМАНИЕ

Дальше будет скучно. Переходите сразу к разделу 4.4, если почувствуете сонливость.

---

Начнем с первых трех строк. Утверждение в строке 1: `mustbe(0, n-1)` является истинным по определению утверждения `mustbe` — если `t` вообще имеется в массиве, то оно должно быть в диапазоне `x[0..n-1]`. Операции присваивания в строке 2 (`l = 0` и `u = n-1`) делают истинным утверждение в строке 3: `mustbe(l, u)`.

Теперь мы переходим к сложной части: циклу в строках 4-27. Аргументировать его правильность будем тремя утверждениями, каждое из которых связано с инвариантом цикла.

- **Инициализация.** Инвариант является и остается истинным при первом проходе цикла.
- **Сохранность.** Если инвариант является истинным в начале цикла, то он останется истинным и после окончания выполнения тела цикла.
- **Завершение.** Выполнение цикла будет окончено, а желаемый результат будет сохранен (в данном случае результат состоит в том, что переменной `p` присваивается правильное значение).

Чтобы показать это, нам придется воспользоваться следствиями из истинности инварианта. Для доказательства правильности инициализации отметим, что утверждение в строке 3 совпадает с утверждением в строке 5. Для доказательства правильности сохранности и завершения мы будем обращаться к строкам 5-21. Когда мы начнем обсуждать строки 9 и 21 (операторы `break`), мы установим свойства этапа завершения, и если мы доберемся до строки 27, то гарантируем сохранность инварианта, поскольку утверждение в этой строке совпадает с утверждением в строке 5.

**Листинг 4.4.** Анализ корректности программы двоичного поиска

```

1. { mustbe(0, n-1) }
2. l = 0; u = n-1
3. { mustbe(l, u) }
4. loop
5.   { mustbe(l, u) }
6.   if l > u
7.     { l > u && mustbe(l, u) }
8.     { t is not in the array }
9.     p = -1; break
10.  { mustbe(l, u) && l <= u }
11.  m = (l + u) / 2

```

```

12.     { mustbe(l, u) && l <= m <= u }
13.     case
14.         x[m] < t.
15.         { mustbe(l, u) && cantbe(0, m) }
16.         { mustbe(m+1, u) }
17.         l = m + 1
18.         { mustbe(l, u) }
19.     x[m] == t:
20.         { x[m] == t }
21.         p = m; break
22.     x[m] > t:
23.         { mustbe(l, u) && cantbe(m, n) }
24.         { mustbe(l, m-1) }
25.         u = m-1
26.         { mustbe(l, u) }
27.     { mustbe(l, u) }

```

Успешная проверка в строке 6 делает возможным утверждение в строке 7: если  $t$  есть в массиве, оно должно быть между элементами  $l$  и  $u$  и при этом  $l > u$ . Из этого следует строка 8:  $t$  в массиве нет. Поэтому мы корректно завершаем цикл в строке 9, предварительно выполнив присваивание  $p = -1$ . Если утверждение в строке 6 ложно, мы переходим на строку 10. Инвариант все еще остается истинным (мы не делали ничего, что могло бы это изменить). Поскольку условие в строке 6 оказалось ложным, мы уверены, что  $l \leq u$ . Строка 11 устанавливает  $m$  равным среднему этих чисел, округленному к ближайшему меньшему целому. Поскольку среднее всегда лежит между двумя величинами и округление не может сделать его меньшим  $l$ , мы можем сделать утверждение в строке 12.

Теперь проанализируем все три ситуации оператора `case` в строках 13-27. Проще всего разобраться со второй альтернативой в строке 19. Вследствие утверждения из строки 20 мы имеем право присвоить переменной  $p$  значение  $m$  и выйти из цикла. Это второе из двух мест, где возможен выход из цикла, так что мы доказали корректность завершения процедуры.

Теперь рассмотрим одну из двух симметричных ветвей оператора `case`. Поскольку при написании кода мы работали с первой ветвью, теперь мы займемся второй (строки 22-26). Возьмем утверждение в строке 23. Первая часть его — инвариант, который нигде ранее в цикле не был изменен. Вторая часть истинна, поскольку  $t < x[m] \leq x[m+1] \leq \dots \leq x[n-1]$ , так что мы можем быть уверены, что  $t$  не может быть равным одному из элементов с индексом, большим  $m-1$ ; это записывается коротко, как `cantbe(m, n-1)`. Логика говорит нам, что если  $t$  должно быть между  $l$  и  $u$ , и оно не может быть «выше»  $m$  или находиться точно в этой позиции, следовательно, оно должно быть между  $l$  и  $m-1$  (если оно вообще где-то есть); отсюда следует и строка 24. Выполнение строки 25 при условии истинности строки 24 сохраняет истинность строки 26 — просто по определению операции присваивания. Таким образом, эта ветвь оператора `case` заново устанавливает инвариант в строке 27.

Доказательство корректности строк 14-18 выглядит точно так же, поэтому теперь мы можем считать, что мы доказали правильность всех трех ветвей оператора `case`. Одна из ветвей корректно завершает цикл, а другие две сохраняют истинность инварианта.

Этот анализ показывает, что если цикл завершается, то значение  $p$  оказывается истинным. Однако мы все еще можем попасть в ситуацию с бесконечным циклом.

На самом деле именно таким был результат ошибок, допущенных большинством профессиональных программистов.

Доказательство конечности использует другой аспект диапазона  $I$ .  $i$ . Этот диапазон изначально имеет некоторый конечный размер ( $n$ ), и строки с 6-й по 9-ю гарантируют, что цикл будет завершен, когда в диапазоне окажется менее одного элемента. Следовательно, нужно показать, что диапазон уменьшается при каждой итерации. Строка 12 говорит, что  $m$  всегда принадлежит текущему диапазону. Обе ветви оператора `case`, в которых выполнение цикла не прерывается (строки 14 и 22), исключают значение  $m$  из текущего диапазона и, следовательно, уменьшают его по меньшей мере на 1 элемент. Таким образом, программа обязательно должна завершиться.

Теперь я практически уверен, что мы можем пойти дальше и закодировать функцию на C. В следующей главе мы займемся именно этим — реализацией функции на языке C и проверкой ее корректности и эффективности.

## 4.4. Принципы

Это упражнение продемонстрировало множество преимуществ верификации программ: рассмотренная задача достаточно важна и требует аккуратного кодирования; при разработке программы мы пользуемся принципами верификации, а при анализе корректности используются достаточно общие методы. Главным недостатком этого примера является уровень детализации: на практике я работаю на гораздо менее формальном уровне. К счастью, все эти детали иллюстрируют набор общих принципов.

### Утверждения

Соотношения между входными данными, переменными программы и результатами описывают состояние программы. Утверждения дают возможность программисту явно записать эти соотношения. Их роль при написании программы обсуждается в следующем разделе данной книги.

### Последовательное выполнение

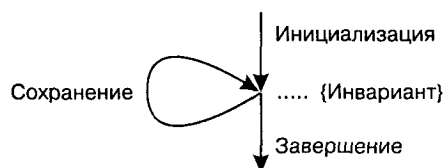
Простейшая структура программы: «сделай это, потом то». Эти структуры анализируются путем расстановки между ними утверждений и разбора каждого из операторов.

### Ветвление

Ветвление осуществляется операторами `if` и `case` в различных формах. При выполнении программы выбирается одна из возможных ветвей. Корректность доказывается путем индивидуального разбора каждой из ветвей. Факт выбора данной ветви дает возможность сделать некоторые утверждения в нашем доказательстве; если мы выполняем оператор, следующий за `if i > j`, то мы можем сделать утверждение, что  $i > j$ , и использовать это для вывода следующего подходящего утверждения.

## Циклы

Для доказательства корректности цикла нужно рассмотреть три его части: инициализацию, сохранение (инварианта) и завершение (рис. 4.1)



**Рис. 4.1.** Три составные части цикла

Сначала мы доказываем, что инвариант цикла устанавливается при инициализации, а затем показываем, что на всех итерациях его истинность сохраняется. Эти два этапа гарантируют (по методу математической индукции), что инвариант остается истинным перед и после каждой итерации цикла. Третий этап: доказательство того, что при любом варианте завершения цикла результат оказывается корректным. Все это вместе гарантирует, что если цикл завершится, то результат будет правильным. Сходимость доказывается отдельно (типичный пример — доказательство сходимости двоичного поиска, приведенное выше).

## Функции

Для верификации функции мы должны определить ее назначение с помощью двух утверждений. Предусловие определяет состояние, в котором должна находиться программа в момент вызова функции, а постусловие называется то, что будет гарантироваться функцией при ее завершении. Таким образом мы можем определить функцию двоичного поиска на языке C следующим образом:

```
int bsearch(int t, int x[], int n)
/* precondition  x[0] <= x[1] <= ... <= x[n-1]
   postcondition.
       result == -1 => t not present in x
       0 <= result < n => x[result] == t
*/
```

Эти условия представляют собой скорее соглашения, нежели факты. В них утверждается, что если выполнены предусловия, то выполнение функции приведет к истинности постусловий. После того как будет доказано то, что тело функции обладает этим свойством, соотношением между пред- и постусловиями можно будет пользоваться без углубления в подробности реализации. Этот подход к написанию программного обеспечения называется «программированием по контракту».

## 4.5. Смысл верификации программ

Когда один программист пытается убедить другого в правильности своей программы, он обычно прибегает к помощи тестов. Берется программа и какой-либо вариант выходных данных, а затем она выполняется вручную. Это мощное средство



для поиска ошибок, которым легко пользоваться. Одно из главных преимуществ верификации программ описанным выше способом заключается в том, что программист получает язык, на котором он может выразить свое понимание текста программы.

Далее в главах 9, 11 и 14 данной книги мы воспользуемся методами верификации при разработке сложных программ. Каждую новую строку кода мы будем объяснять на этом языке; он особенно удобен для выбора инвариантов цикла. Важные пояснения будут превращаться в утверждения в тексте программы. Выбор утверждений, которые нужно включать в настоящие программы, — это искусство, которым можно овладеть лишь на практике.

Язык верификации часто используется после написания кода при его мысленном прогоне. Нарушения утверждений во время тестирования помогают обнаружить ошибки, а анализ этих нарушений позволяет избавиться от ошибок, не наплодив новых. Во время отладки следует исправлять и код, и неправильные утверждения. Нужно понимать свою программу в каждый момент времени и не поддаваться искушению «менять что-нибудь, пока она не заработает». В следующей главе иллюстрируется роль утверждений при тестировании и отладке. Утверждения жизненно важны в процессе сопровождения программы: если вы пытаетесь разобраться в программе, которую никогда раньше не видели и которую никто другой тоже не видел уже много лет, утверждения о состоянии программы могут дать вам неоценимую информацию.

Эти методы — лишь малая часть того, что нужно знать, чтобы правильно писать программы. Ключом к правильности часто является простота кода. С другой стороны, несколько профессиональных программистов, знакомых с этими методиками, поделились со мной опытом, который был не чужд и мне самому: при написании программы «сложная» ее часть обычно работает с первого раза, тогда как ошибки обычно содержатся в простых частях. Когда вы сталкиваетесь с чем-то сложным, вы сосредотачиваетесь и используете мощные формальные методы. В легких местах вы возвращаетесь к старым методам программирования и получаете характерные для этих методов результаты. Я бы сам в это не поверил, если бы это не случилось со мной. Подобные вещи являются хорошей причиной для более частого использования правильных методов.

## 4.6. Задачи

1. Хотя наше доказательство правильности двоичного поиска было достаточно трудоемким, оно все еще не вполне закончено. Как вы докажете, что программа не содержит ошибок времени выполнения (деление на ноль, переполнение, выход за границы диапазона или массива)? Если вы знакомы с дискретной математикой, можете ли вы формализовать доказательство в некоторой логической системе?
2. Если этот вариант двоичного поиска был для вас слишком прост, попробуйте возвращать в переменной  $p$  позицию первого вхождения  $t$  в массив  $x$  (если вхождений несколько, наш алгоритм возвращал произвольное вхождение). Ваш код должен быть логарифмическим по числу сравнений; задачу можно решить за  $\log_2 n$  сравнений.

3. Напишите и проверьте рекурсивную программу двоичного поиска. Какие части кода и доказательства остаются неизменными по сравнению с итерационной версией, а какие меняются?
4. Добавьте вспомогательные переменные для определения количества сравнений и используйте методы проверки программ для доказательства того, что это количество действительно пропорционально логарифму размера массива.

5. Докажите, что программа завершает работу, если  $x$  — положительное целое число:

```
while x != 1 do
  if even(x)
    x = x/2
  else
    x = 3*x + 1
```

6. [C. Sholten] Дэвид Грис в своей книге «Наука программирования» (David Gries, Science of Programming) назвал эту задачу «Задачей о кофейной банке». Дается кофейная банка, в которой есть несколько черных бобов и несколько белых, и, кроме того, дается большая куча черных бобов. Затем следующий процесс выполняется до тех пор, пока в банке не останется один боб.

Случайным образом выбираются два боба. Если они одного цвета, они выкидываются, а вместо них из кучи берется черный и кладется в банку. Если они разных цветов, белый боб возвращается в банку, а черный выбрасывается.

Докажите, что процесс сходится. Что вы можете сказать о цвете последнего оставшегося боба в зависимости от начального количества белых и черных бобов?

7. Мой коллега столкнулся со следующей задачей при написании программы, рисовавшей линии на растровом экране. Массив из  $n$  пар вещественных чисел  $(a, b_i)$  определял  $n$  линий вида  $y = ax + b_i$ . Эти строки были упорядочены на интервале  $[0; 1]$  в том смысле, что  $y[i] < y[i+1]$  для всех  $i$  между 0 и  $n-2$  и всех значений  $x$  из отрезка  $[0; 1]$  (рис. 4.2).

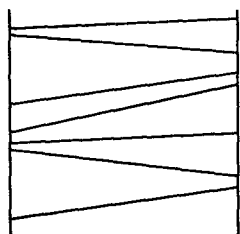


Рис. 4.2. Линии не имели общих точек на вертикальных прямых

Другими словами, линии не имели общих точек на вертикальных прямых ( $x = 0$  и  $x = 1$ ). Ему нужно было найти две линии, между которыми находилась произвольная точка с координатами  $(x, y)$ , где  $x$  принадлежала отрезку  $[0, 1]$ . Как можно быстро решить эту задачу?

#### 4.6. Задачи 63

8. Двоичный поиск работает принципиально быстрее последовательного. Для поиска в массиве из  $n$  элементов в двоичном поиске выполняется примерно  $\log_2 n$  сравнений, а для последовательного — в среднем  $n/2$ . Хотя часто этого вполне достаточно, иногда приходится добиваться еще более высокого быстродействия. Если нельзя улучшить логарифмическую зависимость числа сравнений, можно ли переписать код так, чтобы он стал работать быстрее? Для определенности предположите, что вам нужно найти число в отсортированном массиве из  $n=1000$  целых чисел.

9. В качестве упражнения в проверке программ попробуйте точно описать поведение на входе и выходе следующих программных блоков и покажите, что код соответствует требованиям. Первая программа реализует векторное сложение  $a = b + c$ .

```
i = 0
while i < n
    a[i] = b[i] + c[i]
    i = i+1
```

Этот и следующие два фрагмента используют цикл while вместо for  $i = [0, n)$ . В следующем фрагменте вычисляется максимальное значение в массиве  $x$ .

```
max = x[0]
i = 1
while i < n do
    if x[i] > max
        max = x[i]
    i = i+1
```

А эта программа последовательного поиска возвращает первое вхождение  $t$  в массив  $x[0..n-1]$ .

```
i = 0
while i < n && x[i] != t
    i = i+1
if i >= n
    p = -1
else
    p = i
```

Эта программа вычисляет  $n$ -ю степень числа  $x$  за время, пропорциональное логарифму  $n$ . Такую рекурсивную программу легко закодировать и проверить; итеративная версия достаточно сложна и предоставляется читателю в качестве дополнительного задания.

```
function exp(x n)
    pre n >= 0
    post result = x^n
    if n = 0
        return 1
    else if even(n)
        return square(exp(x n/2))
    else
        return x*exp(x n-1)
```

10. Добавьте ошибки в функцию двоичного поиска и посмотрите, сможете ли вы их обнаружить с помощью метода проверки и в чем они будут проявляться.

11. Напишите и докажите правильность рекурсивной функции двоичного поиска на языке C или C++, причем функция должна соответствовать следующей декларации:

```
int binarysearch(DataType x[], int n)
```

Используйте только одну эту функцию и не вызывайте никаких других.

## 4.7. Дополнительная литература

«Наука программирования» Дэвида Гриса (David Gries, *Science of Programming*, Springer-Verlag, 1987) является превосходным введением в область методик верификации программ. Начинается она с курса логики, затем рассматривает верификацию и разработку программ с формальной точки зрения и, наконец, переходит к обсуждению программирования на конкретных языках. В этой главе я попытался набросать потенциальные преимущества метода верификации программ; единственный способ для программиста научиться эффективно пользоваться этими методиками — изучить их по книге наподобие указанной.