2. Докажите, что следующая функция вычислима:

$$f(n) = \begin{cases} 1, \text{ если } n \text{ делится на 3;} \\ 0, \text{ если } n \text{ не делится на 3.} \end{cases}$$

В качестве доказательства напишите программы для машин Тьюринга и Поста, а также НАМ.

*3. Первой задачей, неразрешимость которой была доказана, была **проблема самоприменимости**: по заданному тексту программы P определить, останавливается ли программа P, если ей на вход подать текст этой же программы. Докажите, что проблема останова сводится к проблеме самоприменимости (именно так и была доказана неразрешимость проблемы останова).

§ 36 Сложность вычислений

Что такое сложность вычислений?

Центральная задача теории алгоритмов — выяснить, существует ли алгоритм решения той или иной задачи. Если существует, то возникает следующий вопрос: а можно ли им воспользоваться на практике, при современном уровне развития вычислительной техники? То есть способен ли компьютер за приемлемое время получить результат? Например, в игре в шахматы возможно лишь конечное количество позиций и, значит, только конечное количество различных партий. Следовательно, теоретически можно перебрать все возможные партии и выяснить, кто побеждает при правильной игре — белые или чёрные. Однако количество вариантов настолько велико, что современные компьютеры не могут выполнить такой перебор за приемлемое время.

Что мы хотим от алгоритма? Во-первых, чтобы он работал как можно быстрее. Во-вторых, чтобы объём необходимой памяти был как можно меньше. В-третьих, чтобы он был как можно более прост и понятен, что позволяет легче отлаживать программу. К сожалению, эти требования противоречивы, и в серьёзных задачах редко удаётся найти алгоритм, который был бы лучше остальных по всем показателям.

Часто говорят о *временно́й сложности* алгоритма (*быстродействии*) и *пространственной сложности*, которая определяется объёмом необходимой памяти. Поскольку память постоянно дешевеет, а быстродействие компьютеров растёт медленно, мы будем рассматривать главным образом временну́ю сложность — время выполнения программы, работающей по данному алгоритму.

В общем случае, говоря о сложности алгоритма, нужно уточнить, о каком исполнителе идёт речь, какие элементарные операции мы используем. Как правило, это один из универсальных исполнителей (во многих случаях универсальным исполнителем можно считать компьютер). Временем работы алгоритма называется количество выполненных им элементарных операций T. Такой подход позволяет оценивать именно качество алгоритма, а не свойства исполнителя (например, быстродействие компьютера, на котором выполняется алгоритм).

Как правило, величина T будет существенно зависеть от объёма исходных данных: поиск в списке из 10 элементов завершится гораздо быстрее, чем в списке из 10 000 элементов. Поэтому сложность алгоритма обычно связывают с размером входных данных n и определяют как функцию $T(n)$. Например, для алгоритмов обработки массивов в качестве размера n используют длину массива. Функция $T(n)$ называется **временно́й сложностью алгоритма**.

Примеры

Рассмотрим алгоритмы выполнения различных операций с массивом A длины n, который может быть объявлен в программе на школьном алгоритмическом языке как

```
целтаб A[1:n]
```

Пример 1. Требуется вычислить сумму первых трёх элементов массива (при $n \geq 3$).

Решение этой задачи содержит всего один оператор:

```
Sum:=A[1]+A[2]+A[3]
```

Этот алгоритм включает две операции сложения и одну операцию записи значения в память, поэтому его сложность $T(n) = 3$ не зависит от размера массива вообще.

Пример 2. Требуется вычислить сумму всех элементов массива. В этой задаче уже не обойтись без цикла:

```
Sum:=0
нц для i от 1 до n
  Sum:=Sum+A[i]
кц
```

Здесь выполняется n операций сложения и $n + 1$ операций записи в память[1], поэтому его сложность $T(n) = 2n + 1$ возрастает линейно с увеличением длины массива[2].

Пример 3. Требуется отсортировать все элементы массива по возрастанию методом выбора.

Напомним, что метод выбора предполагает поиск на каждом шаге минимального из оставшихся неупорядоченных значений (здесь i, j, $nMin$ и c — целочисленные переменные):

```
нц для i от 1 до n-1
  nMin:=i;
  нц для j от i+1 до n
    если A[j]<A[nMin] то nMin:=j все
  кц
  если nMin<>i то
    c:=A[i]; A[i]:=A[nMin]; A[nMin]:=c
  все
кц
```

Подсчитаем отдельно количество сравнений $T_c(n)$ и количество перестановок $T_p(n)$. Количество сравнений не зависит от данных и определяется числом шагов внутреннего цикла:

$$T_c(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n.$$

Число перестановок зависит от данных. Например, если массив уже отсортирован в нужном порядке, перестановок не будет вообще. В худшем случае на каждом шаге основного цикла происходит перестановка, всего их будет $T_p(n) = n - 1$.

---

1 Здесь и далее для упрощения выводов мы не учитываем команды, необходимые для организации цикла, потому что при больших n время их выполнения очень мало в сравнении со временем выполнения остальных операторов.

2 Предполагается, что сложение любых двух чисел выполняется одинаковое время.

Что такое асимптотическая сложность?

Допустим, что нужно выбрать между несколькими алгоритмами, которые имеют разную сложность. Какой из них лучше (работает быстрее)? Оказывается, для этого необходимо знать размер массива данных, которые нужно обрабатывать. Сравним, например, три алгоритма, сложность которых

$$T_1(n) = 10\,000 \cdot n, \quad T_2(n) = 100 \cdot n^2, \quad T_3(n) = n^3.$$

Построим эти зависимости на графике (рис. 5.8). При $n \le 100$ получаем $T_3(n) < T_2(n) < T_1(n)$, при $n = 100$ количество операций для всех трёх алгоритмов совпадает, а при больших n имеем $T_3(n) > T_2(n) > T_1(n)$.

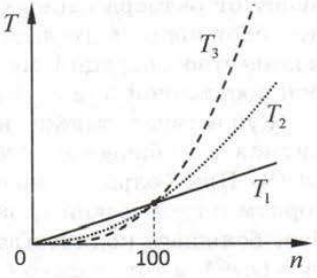

Рис. 5.8

Обычно в теоретической информатике при сравнении алгоритмов используется их **асимптотическая сложность**, т. е. скорость роста количества операций при больших значениях n. При этом запись $O(n)$ (читается «О большое от n») обозначает, что, начиная с некоторого значения $n = n_0$, количество операций ограничено функцией $c \cdot n$, где c — некоторая константа:

$$T(n) \le c \cdot n \text{ для } n \ge n_0.$$

Такие алгоритмы имеют **линейную сложность**, т. е. при увеличении размера данных в 10 раз объём вычислений увеличивается тоже примерно в 10 раз.

Пусть, например, $T(n) = 2n - 1$, как в алгоритме поиска суммы элементов массива. Очевидно, что при этом $T(n) \le 2n$ для всех $n \ge 1$, поэтому алгоритм имеет линейную сложность.

Многие известные алгоритмы имеют **квадратичную сложность** $O(n^2)$. Это значит, что сложность алгоритма ограничена функцией $c \cdot n^2$:

$$T(n) \le c \cdot n^2 \text{ для } n \ge n_0.$$

При этом если размер данных увеличивается в 10 раз, то количество операций (и время выполнения) увеличивается пример-

но в 100 раз. Пример такого алгоритма — сортировка методом прямого выбора, для которой число сравнений

$$T_c(n) = \frac{1}{2}n^2 - \frac{1}{2}n \le \frac{1}{2}n^2 \quad \text{для всех } n \ge 0.$$

Алгоритм имеет **асимптотическую сложность** $O(f(n))$, если найдётся такая постоянная c, что для всех $n \ge n_0$ выполняется условие $T(n) \le c \cdot f(n)$.

Это значит, что при $n \ge n_0$ график функции $c \cdot f(n)$ идёт выше, чем график функции $T(n)$ (рис. 5.9).

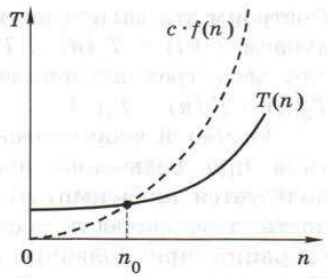

Рис. 5.9

Если количество операций не зависит от размера данных, то говорят, что сложность алгоритма $O(1)$, т. е. количество операций меньше некоторой постоянной при любых n.

Существует также немало алгоритмов с кубической сложностью — $O(n^3)$. При больших значениях n алгоритм с кубической сложностью требует большего количества вычислений, чем алгоритм со сложностью $O(n^2)$, а тот, в свою очередь, работает дольше, чем алгоритм с линейной сложностью. Заметьте, что при малых значениях n всё может быть наоборот; это зависит от постоянной c для каждого из алгоритмов.

Известны и алгоритмы, для которых количество операций растёт быстрее, чем любой полином, например как $O(2^n)$ или $O(n!)$. Они встречаются чаще всего в задачах оптимизации, которые решаются только методом полного перебора. Самая известная задача такого типа — это **задача коммивояжёра** (бродячего торговца), который должен посетить по одному разу каждый из указанных городов и вернуться в начальную точку. Для него нужно выбрать оптимальный маршрут, при котором стоимость поездки (или общая длина пути) будет минимальной.

Ещё один пример сложной задачи, которая решается только полным перебором всех вариантов, — **задача выполнимости**. Дано логическое выражение, которое содержит только имена логических переменных, скобки, а также операции «И», «ИЛИ» и «НЕ». Требуется определить, существует ли набор значений логических переменных, при котором заданное выражение истинно.

Алгоритмы поиска

Сравним вычислительную сложность двух наиболее известных алгоритмов поиска.

Пример 4 (линейный поиск). Дан массив, в котором элементы расположены в произвольном порядке. Требуется найти в нём заданное значение X или сообщить, что его нет.

Решение этой задачи сводится к последовательному просмотру всех элементов массива:

```
nX:=0
нц для i от 1 до n
  если A[i]=X то
    nX:=i
    выход
  все
кц
если nX>0 то
  вывод "A[", nX, "]=", X
иначе
  вывод "Элемент не найден"
все
```

В этом алгоритме число сравнений (в худшем случае) равно $T(n) = n$, поэтому он имеет линейную сложность.

Пример 5 (двоичный поиск). Дан массив, в котором элементы упорядочены по возрастанию. Требуется найти в нём заданное значение X или сообщить, что его нет.

По сравнению с предыдущей задачей, элементы массива отсортированы, и это ускоряет решение, потому что можно применить метод двоичного поиска (дихотомии):

```
L:=1; R:=n+1
нц пока L<R-1
  c:=div(L+R, 2)     | или c:=L+div(R-L,2)
  если X<A[c] то
    R:=c
  иначе
    L:=c
  все
кц
```

```
если A[L]=X то
  вывод "A[", L, "]=", X
иначе
  вывод "Элемент не найден"
все
```

Попробуем определить, сколько раз выполняется основной цикл при двоичном поиске. Сначала ширина интервала поиска — все n элементов массива. На каждом шаге этот интервал делится на 2, процесс завершается, когда левая и правая границы интервала совпадут. Предположим, что число элементов — это целая степень двойки, т. е. $n = 2^m$. Тогда за m шагов ширина интервала сужается до 1, а на следующем шаге его границы совпадут, и цикл закончится. Таким образом, количество шагов цикла равно $m + 1$. Из равенства $n = 2^m$ получаем $m = \log_2 n$, так что

$$T(n) = \log_2 n + 1.$$

Например, при $n = 2^{16}$ линейный поиск потребует в худшем случае $2^{16} = 65\,536$ сравнений, а двоичный — всего $16 + 1 = 17$ сравнений.

Таким образом, алгоритм двоичного поиска имеет асимптотическую сложность $O(\log n)$. Основание логарифма обычно не указывают, потому что выражения $\log_a n$ и $\log_b n$ различаются на постоянный множитель (который можно включить в постоянную c):

$$\log_a n = \frac{1}{\log_b a} \cdot \log_b n.$$

Можно ли сказать, что алгоритм двоичного поиска лучше алгоритма линейного поиска? Нет! Ведь алгоритм линейного поиска применим к любым массивам данных, а алгоритм двоичного поиска — только к упорядоченным (отсортированным). А если мы сначала отсортируем массив, а потом применим к нему двоичный поиск, то общее время работы будет больше, чем при линейном поиске.

Ситуация меняется, если нам нужно многократно выполнять операцию поиска для одних и тех же данных (так, как правило, бывает при работе с базами данных). Тогда имеет смысл заранее отсортировать массив (применить предварительную обработку данных), а затем, используя двоичный поиск, экономить время при каждом новом поисковом запросе.

Алгоритмы сортировки

Ранее мы проанализировали один из простых методов сортировки массивов — метод прямого выбора и выяснили, что его асимптотическая сложность $O(n^2)$. Повторим анализ для метода пузырька, который также изучался в 10 классе:

```
нц для i от 1 до n-1
  нц для j от n-1 до i шаг -1
    если A[j]>A[j+1] то
      c:= A[j]; A[j]:= A[j+1]; A[j+1]:= c;
    все
  кц
кц
```

На первом шаге основного цикла выполняется $n - 1$ шагов внутреннего цикла, т. е. $n - 1$ сравнений. Далее количество сравнений уменьшается до 1, так что общее количество сравнений равно:

$$T_c(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n,$$

так же как и у алгоритма прямого выбора. В то же время в худшем случае при каждом сравнении выполняется перестановка, что требует

$$T_p(n) = 3 \cdot \frac{n(n-1)}{2} = \frac{3}{2}n^2 - \frac{3}{2}n$$

операций присваивания. Таким образом, этот алгоритм имеет асимптотическую сложность $O(n^2)$ как по числу сравнений, так и по числу присваиваний.

Существуют ли более эффективные сортировки, имеющие, например, линейную сложность? Да, для некоторых особых случаев существуют. Например, если известно, что все значения исходного массива находятся в интервале от 1 до некоторого значения MAX, можно использовать **сортировку подсчётом**. Для этого выделяется дополнительный массив счётчиков:

```
целтаб C[1:MAX]
```

который предварительно обнуляется:

```
нц для i от 1 до MAX
  C[i]:= 0
кц
```

Затем в цикле проходим весь массив с данными и для каждого элемента $A[i]$ увеличиваем счётчик $C[A[i]]$. Например, если $A[i]=20$, счётчик $C[20]$ увеличивается на 1. После окончания цикла в каждом счётчике $C[i]$ находится количество значений исходного массива, равных i.

```
нц для i от 1 до n
  C[A[i]]:=C[A[i]]+1
кц
```

Теперь остаётся расставить числа в массиве A в нужном количестве. Например, если C[20] = 5, в массив A записываются последовательно 5 значений, равных 20:

```
k:= 1
нц для i от 1 до MAX
  нц для j от 1 до C[i]
    A[k]:=i
    k:=k+1
  кц
кц
```

Попробуем подсчитать количество операций для этого алгоритма. Заполнение массива C нулями требует MAX присваиваний. Цикл подсчёта элементов содержит n сложений и присваиваний, т. е. его сложность — линейная, $O(n)$. Наконец, последний вложенный цикл выполняет также n сложений и присваиваний (по числу элементов массива A), поэтому алгоритм в целом имеет линейную сложность по n.

Однако нужно учитывать, что принципиальное ускорение алгоритма в сравнении с предыдущими получено за счёт того, что:

- все значения — целые числа в ограниченном диапазоне;
- есть возможность использовать дополнительный массив размером MAX, который может значительно превышать размер исходного массива.

Здесь проявляется компромисс «скорость — память», который присутствует во многих задачах: ускорение алгоритма возможно за счёт использования дополнительной памяти и наоборот, экономия памяти приводит к замедлению работы алгоритма.

Доказано, что в общем случае вычислительная сложность сортировки, основанной только на использовании операций «сравнить» и «переставить», не может быть меньше, чем $O(n\log n)$. Именно такую сложность имеют, например, **сортировка слиянием** (англ. *merge sort*) и **пирамидальная сортировка** (англ. *heap sort*), которые применяются при работе с большими наборами данных. **Быстрая сортировка** (англ. *quick sort*), которая изучалась в 10 классе, в среднем тоже имеет сложность $O(n\log n)$, однако в худшем случае (когда на каждом шаге массив делится на две части, одна из которых состоит из одного элемента) требуется $O(n^2)$ обменов.

Вопросы и задания

1. Какие критерии используются для оценки качества алгоритмов?
2. Почему скорость работы алгоритма оценивается не временем выполнения, а количеством элементарных операций?
3. Как учитывается размер данных при оценке скорости алгоритма?
4. Что означают записи $O(1)$, $O(n)$, $O(n^2)$ и $O(2^n)$?
5. В каких случаях алгоритм, имеющий асимптотическую сложность $O(n^2)$, может работать быстрее, чем алгоритм с асимптотической сложностью $O(n)$?

Задачи

1. Оцените количество операций для алгоритмов:
   а) поиска всех делителей числа;
   б) нахождения минимального и максимального элементов массива;
   в) определения количества положительных элементов массива;
   г) проверки числа на простоту.
   В каждом случае опишите набор используемых элементарных операций. Определите асимптотическую сложность этих алгоритмов.

*2. Предложите алгоритм, позволяющий найти и вывести на экран те символы, которые встречаются в строке более одного раза. Оцените его асимптотическую сложность.

*3. Алфавит языка племени «тумба-юмба» содержит k символов. Предложите алгоритм построения всех возможных слов этого языка, имеющих длину n символов, и оцените его асимптотическую сложность.