Разбор задачи «Медианы объединений»

Лига A, задача H. Лига B, задача I

Задача предложена А. Давыдовым. Автор идеи неизвестен.
Автор формулировки условия – В. Матюхин. Подготовил задачу и разбор И. Григорьев

Условия задачи в разных лигах, по существу, отличаются только ограничениями, так что простые способы, работающие в лиге B, не проходят по времени в лиге A. Несколько замысловатый способ порождения последовательностей в лиге A интереса не представляет, он был выбран ради того, чтобы не приходилось тратить время на ввод 10 млн. чисел.

Сначала стоит проверить, хватит ли памяти для хранения всех последовательностей в двумерном массиве. В лиге B на это требуется всего лишь 100·300·2 байт ≈ 59 Кбайт, а в лиге A – 200·50 000·4 байт ≈ 38 Мбайт. В условии указано ограничение 64 Мбайт, так что памяти вполне достаточно. (Более того, в лиге B памяти хватает даже такой старой системе, как Borland/Turbo Pascal, в которой невозможно простым способом создать массив размером больше 64 Кбайт).

Поэтому первым делом запишем все последовательности в двумерный массив, а затем вызовем подпрограмму (функцию) нахождения медианы объединения для каждой пары последовательностей. (Как нетрудно посчитать, всего таких пар $N(N\text{-}1)/2$).

Самое интересное – в написании этой подпрограммы.

Простые способы (Лига B)

Решение «в лоб» – выполнить объединение двух массивов в один упорядоченный массив и найти ответ в нём на L-м месте. С учётом того, что исходные массивы уже упорядочены, здесь подходит алгоритм, хорошо известный по сортировке слиянием, который работает за время порядка L. Кроме того, небольшая его модификация позволяет, не тратя время и память на создание нового массива, перебирать в порядке возрастания все элементы «объединённой» последовательности – этого уже достаточно для нахождения медианы. Подробнее опишем этот алгоритм ниже, а пока оценим время работы этого и некоторых других очевидных решений.

Для грубой оценки числа операций при максимальных N и L просто вычислим N^2L. В условиях лиги B получим $3\cdot10^6$, а в лиге A – $2\cdot10^9$. Эта оценка позволяет достаточно уверенно сказать, что данное решение проходит в лиге B, но не проходит в лиге A[1].

Если же «забыть» о том, что исходные массивы уже упорядочены, то можно просто объединить их и отсортировать «с нуля». Но посмотрим, какое потребуется на это время. Если сортировать каким-нибудь простым квадратичным методом (например, пузырьковой сортировкой или сортировкой вставками), то на всё решение уйдёт время порядка N^2L^2, что при ограничениях лиги B даёт $9\cdot10^8$. Этот результат делает прохождение такого решения очень сомнительным. И, действительно, подобные решения не проходили по времени.

Если же воспользоваться каким-либо более эффективным алгоритмом сортировки, работающим за время $L \log L$, то получим оценку времени $N^2L \log_2 L \approx 2\cdot10^7$. Скорее всего, такое решение пройдёт по времени, если будет реализовано достаточно аккуратно.

Слияние двух массивов

Алгоритм объединения двух упорядоченных массивов в один, тоже упорядоченный, традиционно называется слиянием (merging). Однако в данной задаче не было необходимости реально создавать массив-объединение: достаточно просто перебрать в порядке возрастания (точнее – неубывания, поскольку могут встречаться равные элементы) его начальные L-1 элементов, после чего медианой будет минимальный из оставшихся.

---

[1] Приблизительное правило для современных процессоров с частотой немногоим более гигагерца таково: программа, содержащая только целочисленные операции, успеет завершиться за секунду, если подобная грубая оценка даёт не больше, чем 10^7–10^8.

Приведём фрагмент программы (здесь и далее считаем, что нумерация элементов массива начинается с 0).

```
{ На языке Паскаль }
i:=0; j:=0;
while i+j <> L-1 do begin
  if a[i] <= b[j] then inc(i)
  else inc(j);
end;
{ ответ – минимальный из a[i] и b[j] }
```

```
/* На языке Си */
i=j=0;
while (i+j != L-1) {
  if (a[i] <= b[j]) i++;
  else j++;
}
/* ответ = min(a[i], b[j]); */
```

Для доказательства правильности этого решения заметим, что до и после каждой итерации[2] цикла верно, что все пройденные элементы двух массивов (то есть элементы массива a с индексами меньше i и элементы массива b с индексами меньше j) не превышают $a[i]$ и $b[j]$. Следовательно, в момент завершения цикла, когда $i+j=L$-1, имеется L-1 такой элемент.

Дальнейшее следует из довольно очевидной леммы, которая будет использована также в одном из самых эффективных решений для лиги A:

Лемма 1. Пусть имеются две упорядоченные по неубыванию последовательности $a_0, a_1, a_2, \ldots a_{L-1}$ и $b_0, b_1, b_2, \ldots b_{L-1}$, в которых выбраны элементы a_i и b_j. Минимальный из a_i и b_j является медианой объединения последовательностей, если выполняются следующие три условия:

$i+j = L$-1;

a_{i-1} (если существует) не превосходит b_j

b_{j-1} (если существует) не превосходит a_i

Более эффективные способы (Лига A)

Поскольку теперь мы собираемся искать медиану не выписывая и не перебирая подряд элементы объединенного массива, то могут пригодиться несколько альтернативных *равносильных* определений (левой) медианы.

Медианой числовой последовательности длины 2L будем называть:

1) член последовательности, который окажется на L-м месте, если последовательность упорядочить по неубыванию.

2) такой член последовательности, что если его убрать из последовательности, то в ней останется хотя бы L-1 членов, которые не больше его и хотя бы L членов, которые не меньше его.

3) минимальное число, для которого найдется хотя бы L членов последовательности, не превосходящих его.

Ещё один способ определить медиану даёт лемма 1 (см. выше).

Заметим, что определения построены так, чтобы учесть случай равенства многих или даже всех элементов последовательности. Если б не это, всё было бы несколько проще.

Жюри известны, по меньшей мере, 3 способа решения, в которых используется бинарный поиск, а также один, основанный на похожей идее. Его оставим напоследок, а сначала расскажем способы, которые используют бинарный поиск непосредственно. Реализация самого бинарного поиска разобрана в Приложении.

Вложенный бинарный поиск (время – произведение логарифмов)

Будем ориентироваться на 3-е определение медианы. Нетрудно написать подпрограмму-функцию (назовём её f), которая по данному числу определяет, сколько элементов в двух наших упорядоченных массивах его не превосходят. Это делается за логарифмическое время с помощью бинарного поиска по каждому из этих массивов (то есть двумя вызовами – по одному на каждый массив – функции *upper_bound*, описанной в Приложении).

[2] Итерацией называется однократное выполнение тела цикла.

Теперь нужно найти минимальное значение аргумента, при котором значение функции f оказывается больше или равно L. Поскольку значение этой функции не убывает при возрастании аргумента, то это тоже делается с помощью бинарного поиска, а именно, вызовом функции *lower_bound* для функции f (см. Приложение, раздел «Бинарный поиск по значениям функции»). В качестве искомого значения y ей нужно передать L, а в качестве начальной области поиска задать полуинтервал $(-X_{max}-1; X_{max}]$. (Поскольку известно, что члены последовательностей а, следовательно, и медиана, не превосходят по модулю $X_{max} = 10^9$).

При этом потребуется порядка $\log 2X_{max}$ вызовов функции f (каждый из которых занимает время порядка $\log L$), поэтому общая оценка времени – $O(\log L \cdot \log X_{max})$. Это самое медленное из известных жюри «хороших» решений для лиги А.

Этот способ можно назвать бинарным поиском по величине медианы.

Несколько быстрее работает более сложная программа, в которой «внешний» бинарный поиск проводится не по всем возможным значениям, а по элементам одного из массивов: во-первых, их меньше, а, во-вторых, для вычисления функции f (которую придётся несколько переопределить) теперь достаточно однократного бинарного поиска – по другому массиву. Однако при этом возникают некоторые подводные камни, связанные с возможным равенством многих элементов. Также нужно не забыть поискать во втором массиве, если медиана не нашлась в первом. Время работы такой программы – $O((\log L)^2)$. Заметим, что при наших ограничениях $\log_2 X_{max} / \log_2 L_{max} \approx 2$, так что выигрыш не велик.

Простой бинарный поиск (время – log *L*)

Это самое короткое и быстрое решение (из известных нам).

Чтобы объяснить идею, временно предположим, что все элементы объединенного массива попарно различны. Допустим, медианой является $a[i]$. Выясним, как его значение должно соотноситься со значениями элементов второго массива. А именно, обозначим j индекс первого элемента массива b, который больше $a[i]$. (Такой непременно найдётся, поскольку медиана не может быть больше последнего элемента массива b). Сколько элементов объединенного массива при этом меньше, чем $a[i]$? В массиве a таковых i, а в массиве $b - j$, всего $i + j$. Чтоб $a[i]$ был медианой, необходимо и достаточно, чтобы $i + j = L$-1.

Алгоритм может быть основан на следующем соображении: возьмём некоторый элемент $a[i]$ и сравним его с $b[\text{L-1-}i]$. Если $a[i] > b[\text{L-1-}i]$, то $a[i]$ явно не может быть медианой, и медиана (если она есть в a) находится где-то левее. Если $a[i] < b[\text{L-1-}i]$, то $a[i]$ может быть медианой, но медиана может находиться и где-то правее. За счёт этого можно организовать бинарный поиск по массиву a. И работать он будет за время $O(\log L)$ (поскольку одна итерация будет выполняться за ограниченное константой время).

Правда, может случиться, что в массиве a медианы нет. В принципе, можно тем же способом поискать её в массиве b, но мы сможем без этого обойтись. Для этого, а также для того, чтобы разобраться, как поступать, если в массивах могут быть равные элементы, зайдём немного с другой стороны: вспомним лемму 1 и будем искать индексы, которые удовлетворяют её условиям.

Чтобы не возникало вопроса о существовании элементов $a_{i\text{-}1}$ и $b_{j\text{-}1}$ при нулевых i или j, мысленно добавим к обоим массивам $a[-1] = b[-1] = -\infty$.

Итак, требуется найти индекс i, удовлетворяющий следующим двум условиям:
$a[i\text{-}1] \le b[L\text{-}(i\text{-}1)\text{-}2]$ и $b[L\text{-}i\text{-}2] \le a[i]$ (мы избавились от j, подставив вместо него L-1-i).

Чтобы для нахождения такого i можно было применить бинарный поиск по значениям функции (см. Приложение), перепишем эти неравенства в терминах функции f, заданной формулой $f(x) = a[x] - b[L\text{-}x\text{-}2]$ $(x \in [\text{-1; L-1}])$.

$f(i\text{-}1) \le 0$; $f(i) \ge 0$.

Очевидно, что функция f не убывает. Следовательно, удовлетворяющее этим неравенствам значение i может быть найдено бинарным поиском. Причём здесь подходит и функция *lower_bound* для функции f (тогда получим $f(i\text{-}1) < 0$; $f(i) \ge 0$), и *upper_bound* (тогда $f(i\text{-}1) \le 0$; $f(i) > 0$). При вызове любой из этих функций ей в качестве искомого значения y

нужно передать 0, а в качестве начальной области поиска задать полуинтервал (-1; L-1][3].

Таким образом, решение состоит из двух этапов: находим i однократным вызовом функции *lower_bound* или *upper_bound*, а затем выбираем минимальный из $a[i]$ и $b[L\text{-}i\text{-}1]$.

Последний способ (время – O(log *L*))

Схема решения:

```
while длина каждого массива ≠ 1 do begin
   согласованно укоротить оба массива
   так, чтобы медиана объединения не изменилась
   (поддерживая длины массивов равными)
end;
ответ = медиана объединения двух 1-элементных массивов
```

Основной вопрос – как укорачивать массивы, не меняя медиану объединения. Здесь поможет следующая довольно очевидная лемма:

Лемма 2. Пусть имеется упорядоченная по неубыванию последовательность. Если вычеркнуть из неё k членов, стоящих левее медианы и k членов, стоящих правее неё, то медиана новой последовательности будет совпадать с медианой прежней последовательности.

Давайте сравним между собой средние элементы исходных последовательностей $a_0, a_1, a_2, \dots a_{L\text{-}1}$ и $b_0, b_1, b_2, \dots b_{L\text{-}1}$, то есть элементы с индексом $m = [(L\text{-}1)/2]$. (Квадратные скобки обозначают здесь целую часть, то есть округление вниз до ближайшего целого). Пусть, для определённости, $a_m \geq b_m$. Изобразим это схематично:

a_0_______a_m_______$a_{L\text{-}1}$

$\geq$

b_0_______b_m_______ $b_{L\text{-}1}$

Все элементы объединенной последовательности (кроме a_m и b_m) разбиваются на 4 примерно равные группы. В первом приближении (с возможной ошибкой ±1 элемент; уточнения будут ниже) можно выкинуть правую верхнюю и левую нижнюю группы.

Действительно, элементы правой верхней группы не меньше элементов левой верхней и левой нижней групп, а также a_m и b_m. Поэтому они будут правее медианы в объединенной упорядоченной последовательности.

Аналогично, элементы левой нижней группы не превосходит элементы правой верхней и правой нижней групп, а также a_m и b_m. Поэтому они будут левее медианы в объединенной упорядоченной последовательности.

Понять, какие в точности участки следует выкинуть, поможет следующая лемма (она следует из предыдущей леммы и определения 2).

Лемма 3. Пусть имеется последовательность длины 2L. Если вычеркнуть из неё элемент, для которого найдётся L+1 других элементов, неменьших его, и элемент, для которого найдётся L других элементов, небольших его, то медиана новой последовательности будет совпадать с медианой прежней последовательности.

Придётся отдельно рассмотреть случаи чётного и нечётного L.

Случай нечётного $L = 2m + 1$. Ясно, что заведомо правее медианы оказываются $a_m \dots a_{L\text{-}1}$ ($m + 1$ штука), а левее – $b_0 \dots b_{m\text{-}1}$ (m штук). Поскольку выкидывать нужно поровну, то выкидываем $a_{m+1} \dots a_{L\text{-}1}$ и $b_0 \dots b_{m\text{-}1}$.

Пример: L = 5; m = 2; $a_2 \geq b_2$

$a_0\ a_1\ \boldsymbol{a_2}\ \cancel{a_3\ a_4}$

$\geq$

$\cancel{b_0\ b_1}\ \boldsymbol{b_2}\ b_3\ b_4$

---

[3] Заметим, что формула для f даёт $f(-1) = a[-1]-b[L-1] = -\infty$, $f(L-1) = a[L-1]-b[-1] = +\infty$, поэтому важно, чтобы при вычислении *lower_bound* и *upper_bound* не происходило попыток вычисления $f(-1)$ и $f(L-1)$. При правильной реализации это так.

Случай чётного $L = 2m + 2$. Заведомо правее медианы оказываются $a_{m+1} \ldots a_{L-1}$ ($m + 1$ штука), левее – $b_0 \ldots b_m$ ($m + 1$ штука). Все их и выкидываем.

Пример: $L = 6$, $m = 2$; $a_2 \geq b_2$

$a_0\ a_1\ \boldsymbol{a_2}\ \cancel{a_3\ a_4\ a_5}$

\V

$\cancel{b_0\ b_1\ \boldsymbol{b_2}}\ b_3\ b_4\ b_5$

Таким образом, в a в любом случае (независимо от чётности) выбрасываем все конечные элементы начиная с a_{m+1}. А в массиве b индекс первого из оставшихся при чётном L будет $m + 1$, а при нечетном – m. Единообразно это можно записать так: $m + ((L+1) \bmod 2)$

Код программы на языке Паскаль может выглядеть примерно так:

```
left_a := 0;  right_a := L;
left_b := 0;  right_b := L;
{  До и после каждой итерации:
   искомая медиана совпадает с медианой объединения участков
     [left_a; right_a) массива a и [left_b; right_b) массива b;
   right_a – left_a = right_b – left_b   }
while right_a - left_a <> 1 do begin
  middle_a := (right_a + left_a - 1) div 2;
  middle_b := (right_b + left_b - 1) div 2;
  if a[middle_a] >= b[middle_b] then begin
      right_a := middle_a + 1;
      left_b := middle_b + (right_b-left_b+1) mod 2;
  end else begin
      right_b := middle_b + 1;
      left_a := middle_a + (right_a-left_a+1) mod 2;
  end;
end;
{  здесь right_a-left_a = right_b-left_b = 1, то есть полуинтервалы [left_a; right_a) и
   [left_b; right_b) содержат по 1 элементу: left_a и left_b соответственно  }
ответ - минимальный из a[left_a] и b[left_b]
```

Очевидный недостаток этого решения – оно работает только в случае массивов одинаковой длины. Остальные способы лишены этого недостатка.

Оценка времени работы

Выполним грубую оценку числа операций при максимальных N и L. Во-первых, в любом случае на генерирование всех массивов требуется порядка $NL = 10^7$ операций. Во-вторых, на поиск медианы уходит:

вложенный бинарный поиск по ответу:	$N^2 \log_2 L \cdot \log_2 X_{max} \approx 1{,}5 \cdot 10^7$
вложенный бинарный поиск по массиву:	$N^2 (\log_2 L)^2 \approx 6 \cdot 10^6$
простой бинарный поиск и последний способ:	$N^2 \log_2 L \approx 5 \cdot 10^5$

Любое из этих решений проходит по времени.

Приложение. Бинарный поиск

В решении рассматриваемой задачи используются разные виды бинарного поиска. Разберём их здесь, предполагая, что с идеей бинарного поиска читатель уже знаком.

Задачу бинарного (двоичного) поиска часто формулируют как быструю проверку наличия элемента с данным значением в неубывающем массиве. (Назовём массив a, а искомое значение – y). Нередко бывает нужно найти также и его местоположение (индекс). Но элементов с данным значением может быть больше одного, поэтому в общем случае может потребоваться найти их все, то есть все индексы i, для которых $a[i] = y$. Для этого достаточно найти границы участка, все элементы которого равны y.

Пусть левую границу будет искать функция *lower_bound*, а правую – *upper_bound*[4]. Забегая вперёд, заметим, что они будут выдавать осмысленные ответы и в случае отсутствия искомого значения y.

Будем считать, что элементы массива нумеруются от 0 до N-1 включительно. Удобно будет мысленно дополнить наш массив двумя фиктивными элементами: $a[-1] = -\infty$ и $a[N] = +\infty$. (Лишь мысленно! Эти элементы – лишь удобные «призраки», которые позволят нам упростить формулировки и объяснения, но в реальной программе не используются. Ниже мы проследим, чтобы программа не делала попыток обращения к этим элементам).

Наши рассуждения и программы будут корректны при любых целых неотрицательных N (то есть даже при $N = 0$, когда массив пуст).

Функция *lower_bound*

Хотелось бы определить *lower_bound* как индекс первого вхождения y в массив a. (То есть минимальное i, при котором $a[i] = y$). Но это определение не работает, если искомого y нет в массиве. Поэтому давайте определим *lower_bound* как минимальное i, при котором $a[i] \geq y$. Такое i непременно найдётся благодаря соглашению $a[N] = +\infty$.

Заметим, что значение *lower_bound* даёт заодно количество элементов массива, которые строго меньше y (без учёта «призрачного» $a[-1]$).

Давайте напишем код этой функции. Итак, ищем минимальное i, такое, что `a[i]` $\geq y$. С самого начала мы знаем, что ответ принадлежит отрезку $[0; N]$[5], за одну итерацию цикла будем уменьшать этот промежуток примерно вдвое. Остановиться нужно, когда он будет содержать одно целое число. (Можно доказать, что будет сделано меньше $\log_2 (N+1) + 1$ итераций, следовательно время работы бинарного поиска – $O(\log N)$).

Заметим, что промежуток $[0; N]$ можно обозначить и по-другому: $[0; N+1)$, $(-1; N+1)$; $(-1; N]$. Любой способ из этих 4-х годится, но некоторые преимущества имеет последний из них[6]. Им и воспользуемся. Дальнейшие объяснения – в комментариях к коду.

```
00  {  На языке Паскаль  }
01  function lower_bound(var a :array of longint; N, y:longint) :longint;
02  var left, right, m : longint;
03  begin
04    left := -1;  right := N;
05    { До и после каждой итерации ответ принадлежит (left; right] }
06    while right - left <> 1 do begin
07      m := (left + right) div 2;      {  см. ниже  }
08      if a[m] >= y then right := m    {  в этом случае ответ  ≤  m  }
09      else left := m;                 {  в этом случае ответ  >  m  }
10    end;
11    { здесь right-left = 1, то есть полуинтервал (left; right] содержит 1 элемент – right }
12    lower_bound := right;
13  end;
```

```
00  /*   На языке Си  */
01  int lower_bound(int a[], int N, int y)
02  {
03    int left  = -1;
04    int right = N;
05    /* До и после каждой итерации ответ принадлежит (left; right]  */
06    while (right-left != 1) {
07          int m = (left + right) / 2;  /* см. ниже */
```

[4] Названия заимствованы из STL C++, они переводятся с английского как «нижняя граница» и «верхняя граница». Подробнее о бинарном поиске в STL рассказано ниже.

[5] Подчеркнём ещё раз: элемента $a[N]$ в массиве нет, но N может быть ответом.

[6] При любом другом способе хотя бы в одной из строк 8-9 справа пришлось бы писать `m+1` вместо `m`.

```
08          if (a[m] >= y) right = m;      /* в этом случае ответ ≤ m */
09          else left = m;                 /* в этом случае ответ > m */
10      }
11      /* здесь right-left==1, то есть полуинтервал (left; right] содержит 1 элемент – right */
12      return right;
13  }
```

По какой формуле вычислять *m* в 7-й строке? Понятно, что с точностью до ±1 это будет [(*right*+*left*)/2]. Для уточнения формулы, заметим, что наш промежуток (*left*; *right*] делится на два: (*left*; *m*] и (*m*; *right*], и требуется, чтобы их длины всегда отличались не более, чем на 1. Нетрудно убедиться, что подходит как [(*right*+*left*+1)/2], так и просто [(*right*+*left*)/2][7]. (Достаточно проверить два случая, когда промежуток имеет чётную и нечётную длину).

Проверим, что программа действительно не пытается обращаться к «призракам» – элементам *a*[-1] и *a*[*N*]. Это так, потому что обращения к элементам массива происходят только в 8-й строке (к элементу *a*[*m*]), а для *m* в 7-й строке верно, что $left < m < right$.

Функция *upper_bound*

Для случая, когда искомый *y* присутствует в массиве, можно было бы определить *upper_bound* как индекс его последнего вхождения (то есть максимальное *i*, при котором $a[i] = y$). Но удобнее – как индекс элемента, следующего за последним вхождением *y*.

В общем случае (когда искомый *y* может входить или не входить в массив) определим *upper_bound* как минимальное *i*, при котором $a[i] > y$. Такое *i* непременно найдётся благодаря соглашению $a[N] = +\infty$.

Это определение дословно повторяет определение *lower_bound*, если заменить в нём знак "≥" на ">". Поэтому и код функции *upper_bound* может быть получен из кода *lower_bound* простой заменой ">=" на ">" в 8-й строке.

Заметим, что значение *upper_bound* даёт заодно количество элементов массива, которые не превосходят *y* (без учёта «призрачного» *a*[-1]).

Использование функций *lower_bound* и *upper_bound*

Для удобства приведём таблицу с основными сведениями об этих функциях:

	lower_bound	*upper_bound*
Индекс первого элемента, который	$\geq y$	$> y$
Количество элементов, которые	$< y$	$\leq y$
Если есть элементы равные *y*, то индекс	первого вхождения	последнего вхождения +1
	$a[i] = y$ при $i \in$ [*lower_bound*; *upper_bound*)	

В качестве примера приведём несколько значений этих функций для такого массива (*N*=14):

i	(-1)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	(14)
a[*i*]	-∞	-8	-3	0	2	5	5	5	5	7	10	22	30	40	50	+∞

Y	*lower_bound*	*upper_bound*		*y*	*lower_bound*	*upper_bound*
-10	0	0		6	8	8
-8	0	1		7	8	9
2	3	4		50	13	14
5	4	8		60	14	14

Проанализируем случай, когда искомого нет в массиве. Тогда *lower_bound* = *upper_bound* и *a*[*lower_bound*-1] < *y* < *a*[*lower_bound*]. С учётом «призрачных» элементов $a[-1] = -\infty$ и $a[N] = +\infty$ это верно, даже если все настоящие элементы массива меньше, чем *y* (при этом *lower_bound* = *upper_bound* = *N*), и если все они больше, чем *y* (при этом *lower_bound* = *upper_bound* = 0).

[7] Квадратные скобки обозначают здесь целую часть, то есть округление вниз до ближайшего целого.

Заметим, что полуинтервал [*lower_bound*; *upper_bound*) всегда задаёт множество индексов элементов, которые равны *y* – с учётом обычного соглашения, что при *lower_bound* = *upper_bound* этот полуинтервал пуст.

Если требуется просто проверить, имеется ли искомое значение *y* в массиве, то можно сравнить *lower_bound* и *upper_bound*. Но быстрее (почти вдвое) сделать это любым из двух следующих способов с помощью вызова одной *lower_bound*:

Длинный безопасный способ:

```
i:=lower_bound(a, N, y);
if i<>N then begin
  if a[i]=y then имеется
  else не имеется
end else не имеется
```

Этот короткий способ применим, если вы можете гарантировать, что в случае $i = N$ при проверке условия не будет происходить попытки обращения к несуществующему элементу $a[N]$:

```
i := lower_bound(a, N, y);
if (i<>N) and (a[i]=y) then имеется
else не имеется
```

Бинарный поиск по значениям функции

Нередко аналогичная задача формулируется не для массива, а для некоторой неубывающей функции целочисленного аргумента. (Такой поиск часто называют «бинарным поиском по ответу»).

Например, выше (см. раздел «Вложенный бинарный поиск») требовалось найти минимальное *i*, при котором $f(i) \geq y$ (там вместо *y* было *L*). Легко видеть, что эта задача отличается от задачи, решаемой описанной выше функцией *lower_bound* лишь тем, что вместо обращения к элементу массива $a[i]$ стоит вызов функции $f(i)$.

Более того, с точки зрения математики – это просто та же самая задача, поскольку математически массив – это всего лишь способ задания функции, аргументом которой является индекс массива, а значением – значение соответствующего элемента массива.

Поэтому всё, что сказано выше относительно бинарного поиска по массиву, легко переносится и на случай бинарного поиска по значениям неубывающей функции. В тексте программы надо только изменить начальные границы поиска (чтобы полуинтервал ($left_0$; $right_0$)[8] совпал с интересующим нас диапазоном аргумента функции); заменить `a[m]` на вызов функции `f(m)` и, само собой, убрать массив из списка формальных параметров.

Полученную таким образом функцию будем называть ***lower_bound* для функции *f***. Аналогично, функцию, находящую минимальное *i*, при котором $f(i) > y$, будем называть ***upper_bound* для функции *f***. Она строится тем же способом из *upper_bound* для массива.

Бинарный поиск в STL C++

Если вы используете стандартную библиотеку шаблонов C++, то полезно знать, что в ней уже имеется реализация бинарного поиска для вектора и массива, которая состоит из четырёх функций. Две главные из них именно так и называются – `upper_bound` и `lower_bound`. Они отличаются от описанных выше одноимённых функций лишь некоторыми особенностями, свойственными почти всем функциям STL: во-первых, они работают в терминах не индексов, а итераторов или указателей а, во-вторых, могут искать не только во всём векторе (массиве), но и в его произвольном фрагменте, заданном парой итераторов (указателей). Две другие – это `equal_range` (возвращает сразу пару итераторов (указателей), первый из которых равен *lower_bound*, а второй – *upper_bound*) и `binary_search` (возвращает *true*, если имеется искомое значение, иначе *false*; легко может быть реализована с помощью функции `lower_bound`, как показано выше).

Для использования этих функций необходимо подключить заголовочный файл `algorithm`. Их более подробное описание можно найти в документации по STL.

[8] Обе скобки круглые, поскольку программа не делает попыток вычисления $f(left_0)$ и $f(rigth_0)$. Для объяснения смысла возвращаемых значений иногда удобно считать что $f(left_0) = -\infty$, а $f(right_0) = +\infty$.