Решения избранных упражнений и задач

Оглавление

Решения для главы 2

Решение упражнения 2.2.2

```
SELECTION-SORT(A, n)
for i = 1 to n–1
    smallest = i
    for j = i + 1 to n
        if A[j] < A[smallest]
            smallest = j
    Поменять местами A[i] и A[smallest]
```

Алгоритм поддерживает инвариант цикла: в начале каждой итерации внешнего цикла **for** подмассив $A[1 : i - 1]$ состоит из $i - 1$ наименьших элементов массива $A[1 : n]$, и этот подмассив отсортирован. После первых $n - 1$ элементов подмассив $A[1 : n - 1]$ содержит наименьшие $n - 1$ элементов в отсортированном виде и, следовательно, элемент $A[n]$ должен быть наибольшим.

Время работы алгоритма составляет $\Theta(n^2)$ для всех случаев.

Решение упражнения 2.2.4

Модифицируйте алгоритм так, чтобы он сначала проверял, отсортирован ли входной массив, что заняло бы время $\Theta(n)$ для массива из n элементов. Если массив уже отсортирован, тогда алгоритм завершает работу. В противном случае массив сортируется обычным образом. Время работы в наилучшем случае, как правило, не является хорошим показателем эффективности алгоритма.

Решение упражнения 2.3.6

Процедура двоичного поиска принимает отсортированный массив A, значение x и диапазон $[low : high]$ массива, где ищется x. Процедура сравнивает x с элементом массива в середине диапазона и принимает решение относительно исключения половины диапазона из дальнейшего рассмотрения.

Ниже приведены итеративная и рекурсивная версии процедуры, каждая из которых возвращает либо индекс i, такой что $A[i] = x$, либо NIL, если ни один из элементов $A[low : high]$ не содержит x. При начальном вызове любой из версий должны указываться параметры A, x, 1, n.

```
ITERATIVE-BINARY-SEARCH(A, x, low, high)
 while low ≤ high
     mid = ⌊(low + high) /2⌋
     if x == A[mid]
             return mid
     elseif x > A[mid]
             low = mid + 1
     else high = mid – 1
 return NIL

RECURSIVE-BINARY-SEARCH(A, x, low, high)
 if low > high
     return NIL
 mid = ⌊(low + high) /2⌋
 if x == A[mid]
     return mid
 elseif x > A[mid]
     return RECURSIVE-BINARY-SEARCH(A, x, mid + 1, high)
 else return RECURSIVE-BINARY-SEARCH(A, x, low, mid – 1)
```

Обе процедуры завершают поиск неудачно, если диапазон пуст (т.е. $low > high$), и успешно, если значение x найдено. На основе сравнения x со средним элементом в искомом диапазоне поиск продолжается с уменьшенным вдвое диапазоном. Таким образом, рекуррентное соотношение для этих процедур выглядит как $T(n) = T(n/2) + \Theta(1)$ и имеет решение $T(n) = \Theta(\lg n)$.

Решение задачи 2.4

а) Инверсии: (1, 5), (2, 5), (3, 4), (3, 5), (4, 5). (Не забывайте, что инверсии указываются с помощью индексов, а не значений в массиве.)

б) Массив с элементами, взятыми из $\{1, 2, \ldots, n\}$, с наибольшим количеством инверсий: $\langle n, n-1, n-2, \ldots, 2, 1\rangle$. Для всех $1 \le i < j \le n$ существует инверсия (i, j). Количество таких инверсий составляет $\binom{n}{2} = n(n-1)/2$.

в) Предположим, что массив A начинается с инверсии (k, i). Тогда $k < i$ и $A[k] > A[i]$. В момент, когда внешний цикл **for** в строках 1–8 устанавливает $key = A[i]$, значение, которое было сначала в $A[k]$, по-прежнему располагается левее $A[i]$. То есть оно находится в $A[j]$, где $1 \le j < i$, поэтому инверсией становится (j, i). На какой-то итерации цикла **while** в строках 5–7 элемент $A[j]$ перемещается на одну позицию вправо. Строка 8 в конечном итоге поместит *key* слева от этого элемента, устранив тем самым инверсию. Поскольку строка 5 перемещает только элементы больше *key*, она перемещает только элементы, соответствующие инверсиям. Другими словами, каждая итерация цикла **while** в строках 5–7 соответствует устранению одной инверсии.

г) Мы следуем подсказке и модифицируем сортировку слиянием, чтобы подсчитать количество инверсий за время $\Theta(n \lg n)$.

Для начала определим ***инверсию слияния*** как ситуацию при выполнении сортировки слиянием, в которой процедура MERGE после копирования $A[p : q]$ в L и $A[q + 1 : r]$ в R имеет значения x в L и y в R, такие что $x > y$. Рассмотрим инверсию (i, j) и примем $x = A[i]$ и $y = A[j]$, так что $i < j$ и $x > y$. Мы утверждаем, что в случае запуска сортировки слиянием была бы ровно одна инверсия слияния, использующая x и y. Чтобы понять причину, для начала заметим, что процедура MERGE — единственный способ, которым элементы массива меняют свои позиции друг относительно друга. Более того, поскольку MERGE сохраняет элементы в L и R в том же относительном порядке, порядок следования элементов друг относительно друга изменяется, только когда в L появляется больший элемент, а в R — меньший. Таким образом, существует, по крайней мере, одна инверсия слияния с участием x и y. Чтобы удостовериться в наличии ровно одной такой инверсии слияния, важно понять, что после любого вызова MERGE, затрагивающего и x, и y, они находятся в одном и том же отсортированном подмассиве, а потому при любом последующем вызове оба появятся в L или оба появятся в R. Таким образом, выдвинутое выше утверждение доказано.

Мы показали, что каждая инверсия подразумевает одну инверсию слияния. Фактически соответствие между инверсиями и инверсиями слияния является однозначным. Предположим, что имеется инверсия слияния, включающая значения x и y, причем изначально значением x было $A[i]$, а значением y — $A[j]$. Так как у нас есть инверсия слияния, то $x > y$. И поскольку x принадлежит L, а y — R, значение x должно находиться внутри подмассива, который предшествует подмассиву, содержащему y. Следовательно, значение x изначально находился в позиции i, предшествующей исходной позиции j значения y, поэтому (i, j) является инверсией. Продемонстрировав взаимно однозначное соответствие между инверсиями и инверсиями слияния, осталось подсчитать количество инверсий слияния.

Рассмотрим инверсию слияния, которая задействует y в R. Пусть z — наименьшее значение в L, превышающее y. В какой-то момент процесса слияния z и y станут "открытыми" значениями в L и R, т.е. в строке 13 процедуры MERGE мы получим $z = L[i]$ и $y = R[j]$. В этот момент существуют инверсии слияния, включающие y и $L[i]$, $L[i + 1]$, $L[i + 2]$, $\ldots, L[n_L - 1]$, и эти $n_L - 1$ инверсий слияния будут единственными, которые задействуют y. Следовательно, необходимо выявить, когда z и y впервые становятся открытыми во время выполнения процедуры MERGE, и в этот момент добавить значение $n_L - i$ к общему количеству инверсий слияния. Именно так работает показанный ниже псевдокод, созданный по образцу сортировки слиянием. Он также сортирует массив A.

```
MERGE-INVERSIONS(A, p, q, r)
n_L = q − p + 1
n_R = r − q
Пусть L[0 : n_L − 1] и R[0 : n_R − 1] будут новыми массивами
for i = 0 to n_L − 1
    L[i] = A[p + i − 1]
for j = 0 to n_R − 1
    R[j] = A[q + j]
i = 0
j = 0
k = p
```

```
inversions = 0
while i < n_L и j < n_R
    if L[i] ≤ R[j]
            inversions = inversions + n_L − i
            A[k] = L[i]
            i = i + 1
    else A[k] = R[j]
            j = j + 1
    k = k + 1
while i < n_L
    A[k] = L[i]
    i = i + 1
    k = k + 1
while j < n_R
    A[k] = R[j]
    j = j + 1
    k = k + 1
return inversions
```

```
COUNT-INVERSIONS(A, p, r)
inversions = 0
if p < r
    q = ⌊(p + r)/2⌋
    inversions = inversions + COUNT-INVERSIONS(A, p, q)
    inversions = inversions + COUNT-INVERSIONS(A, q + 1, r)
    inversions = inversions + MERGE-INVERSIONS(A, p, q, r)
return inversions
```

Начальный вызов выглядит как COUNT-INVERSIONS(A, 1, n).

Всякий раз, когда в процедуре MERGE-INVERSIONS раскрывается $R[j]$ и в массиве L обнаруживается значение, превышающее $R[j]$, мы увеличиваем *inversions* на количество оставшихся элементов в L. Затем, т.к. раскрывается $R[j + 1]$, $R[j]$ больше никогда не сможет быть раскрыто.

Поскольку мы добавили лишь постоянный объем дополнительных действий к каждому вызову процедуры и к каждой итерации последнего цикла **for** процедуры слияния, общее время работы приведенного выше псевдокода остается таким же, как и время работы сортировки слиянием: $\Theta(n \lg n)$.

Решения для главы 3

Решение упражнения 3.2.2

Поскольку O-запись предоставляет только верхнюю, а не точную оценку, утверждение означает, что время работы алгоритма A является, по меньшей мере, функцией, скорость роста которой не превышает n^2.

Решение упражнения 3.2.3

$2^{n+1} = O(2^n)$, но $2^{2n} \neq O(2^n)$.

Чтобы показать, что $2^{n+1} = O(2^n)$, потребуется отыскать константы c, $n_0 > 0$, такие что

$$0 \leq 2^{n+1} \leq c \cdot 2^n \quad \text{для всех} \quad n \geq n_0.$$

Поскольку $2^{n+1} = 2 \cdot 2^n$ для всех n, мы можем удовлетворить определение при $c = 2$ и $n_0 = 1$.

Чтобы показать, что $2^{2n} \neq O(2^n)$, предположим, что существуют константы c, $n_0 > 0$, такие что

$$0 \leq 2^{2n} \leq c \cdot 2^n \quad \text{для всех} \quad n \geq n_0.$$

Тогда $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n \Rightarrow 2^n \leq c$. Но ни одна константа не больше всех 2^n, поэтому предположение приводит к противоречию.

Решение упражнения 3.3.5

Функция $\lceil \lg n \rceil !$ не является полиномиально ограниченной, но функция $\lceil \lg \lg n \rceil !$ полиномиально ограничена.

Доказательство полиномиальной ограниченности функции $f(n)$ эквивалентно доказательству того, что $\lg f(n) = O(\lg n)$ по следующим причинам.

- Если функция $f(n)$ является полиномиально ограниченной, то существуют положительные константы c, k и n_0, такие что $0 \leq f(n) \leq cn^k$ для всех $n \geq n_0$. Без утраты общности предположим, что $c \geq 1$, т.к. если $c < 1$, то из неравенства $f(n) \leq cn^k$ вытекает $f(n) \leq n^k$. Предположим также, что $n_0 \geq 2$, поэтому из $n \geq n_0$ вытекает $\lg c \leq (\lg c)(\lg n)$. Тогда имеем

  $$\lg f(n) \leq \lg c + k \lg n \leq$$
  $$\leq (\lg c + k) \lg n,$$

  откуда следует, что $\lg f(n) = O(\lg n)$, поскольку c и k — константы.

- Предположим теперь, что $\lg f(n) = O(\lg n)$. Тогда существуют положительные константы c и n_0, такие что $0 \le \lg f(n) \le c \lg n$ для всех $n \ge n_0$. В данном случае мы имеем

  $$0 \le f(n) = 2^{\lg f(n)} \le 2^{c \lg n} = (2^{\lg n})^c = n^c$$

  для всех $n \ge n_0$, так что функция $f(n)$ полиномиально ограничена.

В последующих доказательствах мы будем использовать описанные ниже два факта.

1. $\lg(n!) = \Theta(n \lg n)$ (согласно уравнению (3.28)).
2. $\lceil \lg n \rceil = \Theta(\lg n)$, потому что
   - $\lceil \lg n \rceil \ge \lg n$ и
   - $\lceil \lg n \rceil < \lg n + 1 \le 2 \lg n$ для всех $n \ge 2$.

Мы имеем

$$\begin{aligned}\lg(\lceil \lg n \rceil !) &= \Theta(\lceil \lg n \rceil \lg \lceil \lg n \rceil) = \\ &= \Theta((\lg n)(\lg \lg n)) = \\ &= \omega(\lg n)\,.\end{aligned}$$

Следовательно, $\lg(\lceil \lg n \rceil !)$ не равно $O(\lg n)$, а потому функция $\lceil \lg n \rceil !$ не является полиномиально ограниченной.

Мы также имеем

$$\begin{aligned}\lg(\lceil \lg \lg n \rceil !) &= \Theta(\lceil \lg \lg n \rceil \lg \lceil \lg \lg n \rceil) = \\ &= \Theta((\lg \lg n)(\lg \lg \lg n)) = \\ &= o((\lg \lg n)^2) = \\ &= o(\lg^2(\lg n)) = \\ &= o(\lg n)\,.\end{aligned}$$

Последний шаг вытекает из того, что любая полилогарифмическая функция растет медленнее любой положительной полиномиальной функции, т.е. для констант a, $b > 0$ имеем $\lg^b n = o(n^a)$. Подставив $\lg n$ вместо n, 2 вместо b и 1 вместо a, мы получаем $\lg^2(\lg n) = o(\lg n)$.

Таким образом, $\lg(\lceil \lg \lg n \rceil !) = O(\lg n)$ и, следовательно, функция $\lceil \lg \lg n \rceil !$ полиномиально ограничена.

Решения для главы 4

Решение упражнения 4.2.3

Если есть возможность перемножения матриц 3×3 с применением k умножений, то можно перемножать и матрицы размером $n \times n$, рекурсивно перемножая матрицы $n/3 \times n/3$ за время $T(n) = kT(n/3) + \Theta(n^2)$.

Используя основной метод для решения этого рекуррентного уравнения, рассмотрим отношение между $n^{\log_3 k}$ и n^2.

- Если $\log_3 k = 2$, тогда применим случай 2 и $T(n) = \Theta(n^2 \lg n)$. Мы имеем $k = 9$ и $T(n) = o(n^{\lg 7})$.
- Если $\log_3 k < 2$, тогда применим случай 3 и $T(n) = \Theta(n^2)$. Мы имеем $k < 9$ и $T(n) = o(n^{\lg 7})$.
- Если $\log_3 k > 2$, тогда применим случай 1 и $T(n) = \Theta(n^{\log_3 k})$. Мы имеем $k > 9$, а $T(n) = o(n^{\lg 7})$ при $\log_3 k < \lg 7$, т.е. когда $k < 3^{\lg 7} \approx 21.85$. Наибольшее такое целое число k равно 21.

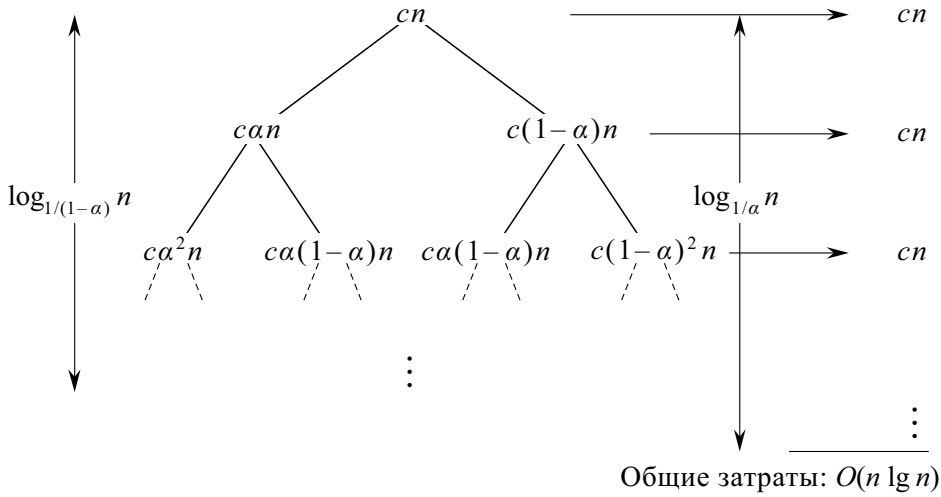
Таким образом, $k = 21$, а время работы составляет $\Theta(n^{\log_3 k}) = \Theta(n^{\log_3 21}) = O(n^{2.80})$ (т.к. $\log_3 21 \approx 2.77$).

Решение упражнения 4.4.4

$T(n) = T(\alpha n) + T((1-\alpha)n) + cn$.

Это рекуррентное соотношение решается аналогично рекуррентному соотношению $T(n) = T(n/3) + T(2n/3) + cn$, решение которого приводилось в книге.

Не утрачивая общности, предположим, что $\alpha \geq 1 - \alpha$, откуда $0 < 1 - \alpha \leq 1/2$ и $1/2 \leq \alpha < 1$.



Дерево рекурсии является полным для $\log_{1/(1-\alpha)} n$ уровней, каждый из которых привносит затраты cn, поэтому мы допускаем, что $\Omega(n \log_{1/(1-\alpha)} n) = \Omega(n \lg n)$. Оно содержит $\log_{1/\alpha} n$ уровней, каждый из которых привносит затраты, не превышающие cn, поэтому мы допускаем, что $O(n \log_{1/\alpha} n) = O(n \lg n)$.

Теперь покажем методом подстановки, что $T(n) = \Theta(n \lg n)$. Для доказательства верхней оценки нужно показать, что $T(n) \le dn \lg n$ для подходящей константы $d > 0$:

$$\begin{aligned} T(n) &= T(\alpha n) + T((1-\alpha)n) + cn \le \\ &\le d\alpha n \lg(\alpha n) + d(1-\alpha)n \lg((1-\alpha)n) + cn = \\ &= d\alpha n \lg \alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + d(1-\alpha)n \lg n + cn = \\ &= dn \lg n + dn(\alpha \lg \alpha + (1-\alpha) \lg(1-\alpha)) + cn \le \\ &\le dn \lg n, \end{aligned}$$

если $dn(\alpha \lg \alpha + (1-\alpha) \lg(1-\alpha)) + cn \le 0$. Это условие эквивалентно условию

$$d(\alpha \lg \alpha + (1-\alpha) \lg(1-\alpha)) \le -c.$$

Так как $1/2 \le \alpha < 1$ и $0 < 1-\alpha \le 1/2$, то $\lg \alpha < 0$ и $\lg(1-\alpha) < 0$. Таким образом, $\alpha \lg \alpha + (1-\alpha) \lg(1-\alpha) < 0$, так что при умножении обеих частей неравенства на этот коэффициент понадобится обратить неравенство:

$$d \ge \frac{-c}{\alpha \lg \alpha + (1-\alpha) \lg(1-\alpha)}$$

или

$$d \ge \frac{c}{-\alpha \lg \alpha + -(1-\alpha) \lg(1-\alpha)}.$$

Дробь в правой части — положительная константа, так что достаточно выбрать любое значение d, которое больше или равно этой дроби.

Чтобы доказать нижнюю оценку, необходимо показать, что $T(n) \ge dn \lg n$ для подходящей константы $d > 0$. Мы можем использовать доказательство для верхней оценки, поменяв $\le$ на $\ge$, и получим требование вида

$$0 < d \le \frac{c}{-\alpha \lg \alpha - (1-\alpha) \lg(1-\alpha)}.$$

Следовательно, $T(n) = \Theta(n \lg n)$.

Решения для главы 5

Решение упражнения 5.2.1

Поскольку процедура HIRE-ASSISTANT всегда нанимает кандидата 1, наем производится ровно один раз, если и только если не наняты другие кандидаты кроме кандидата 1. Такое событие происходит, когда кандидат 1 является наилучшим из n кандидатов, и вероятность этого события составляет $1/n$.

Процедура HIRE-ASSISTANT выполняет наем n раз, если каждый кандидат лучше всех кандидатов, с которыми ранее были проведены собеседования (и они были наняты). Это событие происходит в точности тогда, когда список рангов, переданных процедуре, выглядит как $\langle 1, 2, \ldots, n\rangle$, и вероятность этого события составляет $1/n!$.

Решение упражнения 5.2.5

Еще один способ представить себе задачу о гардеробщице заключается в том, что мы хотим определить ожидаемое количество фиксированных точек в случайной перестановке. (***Фиксированная точка*** перестановки π — это значение i, для которого $\pi(i) = i$.) Мы могли бы пройти по всем $n!$ перестановкам, подсчитать общее количество фиксированных точек и разделить на $n!$, чтобы определить среднее количество фиксированных точек на перестановку. Процесс довольно кропотливый, и ответом могло оказаться значение 1. Однако мы можем воспользоваться индикаторными случайными величинами и получить тот же ответ гораздо проще.

Определим случайную величину X, равную количеству посетителей, которые получили обратно свои шляпы, так что необходимо вычислить $E[X]$.

Определим для $i = 1, 2, \ldots, n$ индикаторную случайную величину

$$X_i = I\{\text{посетитель } i \text{ получает обратно свою шляпу}\}.$$

Тогда $X = X_1 + X_2 + \ldots + X_n$.

Поскольку порядок возвращения шляп случайный, вероятность того, что каждый посетитель получит свою шляпу, равна $1/n$. Другими словами, $\Pr\{X_i = 1\} = 1/n$, откуда согласно лемме 5.1 вытекает, что $E[X_i] = 1/n$.

Таким образом,

$$\begin{aligned}\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \\ &= \sum_{i=1}^{n} \mathrm{E}[X_i] = \\ &= \sum_{i=1}^{n} 1/n = \\ &= 1,\end{aligned}$$

(согласно свойству линейности математического ожидания)

и поэтому мы ожидаем, что ровно 1 посетитель получит обратно свою шляпу.

Обратите внимание, что в данной ситуации индикаторные случайные величины не являются независимыми. Например, если $n = 2$ и $X_1 = 1$, то X_2 также должно быть равно 1. И наоборот, если $n = 2$ и $X_1 = 0$, то X_2 также должно быть равно 0. Несмотря на зависимость, $\Pr\{X_i = 1\} = 1/n$ для всех i, и свойство линейности математического ожидания предохраняется. Таким образом, мы можем применять метод индикаторных случайных величин даже при наличии зависимости.

Решение упражнения 5.2.6

Пусть X_{ij} — индикаторная случайная величина для события, когда пара $A[i]$, $A[j]$ для $i < j$ инвертируется, т.е. $A[i] > A[j]$. Точнее говоря, мы определяем $X_{ij} = \mathrm{I}\{A[i] > A[j]\}$ для $1 \le i < j \le n$. Мы имеем $\Pr\{X_{ij} = 1\} = 1/2$, т.к. для двух разных случайных чисел вероятность того, что первое больше второго, равна $1/2$. Согласно лемме 5.1 мы получаем $\mathrm{E}[X_{ij}] = 1/2$.

Пусть X — случайная величина, обозначающая общее количество инвертированных пар в массиве, так что

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Нас интересует ожидаемое количество инвертированных пар, поэтому мы берем математическое ожидание обеих сторон приведенного выше уравнения и получаем

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right].$$

Воспользовавшись свойством линейности математического ожидания, мы получаем

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] = \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1/2 = \\
&= \binom{n}{2}\frac{1}{2} = \\
&= \frac{n(n-1)}{2}\cdot\frac{1}{2} = \\
&= \frac{n(n-1)}{4}.
\end{aligned}$$

Таким образом, ожидаемое количество инверсий составляет $n(n-1)/4$.

Решение упражнения 5.3.2

Наряду с тождественной перестановкой существуют и другие перестановки, которые процедуре PERMUTE-WITHOUT-IDENTITY не удается создать. Например, рассмотрим ее работу при $n = 3$, когда она должна быть способна создать $n! - 1 = 5$ нетождественных перестановок. Цикл **for** выполняется для $i = 1$ и $i = 2$. При $i = 1$ вызов RANDOM возвращает одно из двух возможных значений (2 или 3), а при $i = 2$ вызов RANDOM возвращает только одно значение (3). Таким образом, процедура PERMUTE-WITHOUT-IDENTITY может создать только $2 \cdot 1 = 2$ возможных перестановки вместо требуемых пяти перестановок.

Решение упражнения 5.3.4

Процедура PERMUTE-BY-CYCLE выбирает для *offset* случайное целое число в диапазоне $1 \le offset \le n$, а затем выполняет циклический поворот массива. То есть $B[((i + offset - 1) \bmod n) + 1] = A[i]$ для $i = 1, 2, \ldots, n$. (Вычитание и прибавление 1 при вычислении индекса обусловлено индексацией с началом от 1. Если бы использовалась индексация, начинающаяся с 0, тогда вычисление индекса упростилось бы до $B[(i + offset) \bmod n] = A[i]$ для $i = 0, 1, \ldots, n - 1$.)

Таким образом, как только смещение определено, определяется и вся перестановка. Поскольку каждое значение смещения встречается с вероятностью $1/n$, каждый элемент $A[i]$ имеет вероятность оказаться в позиции $B[j]$ с вероятностью $1/n$.

Тем не менее, эта процедура не создает равномерную случайную перестановку, потому что способна производить только n различных перестановок. Таким образом, n перестановок происходят с вероятностью $1/n$, а оставшиеся $n! - n$ перестановок встречаются с вероятностью 0.

Решения для главы 6

Решение упражнения 6.1.1

Поскольку пирамида представляет собой почти полное двоичное дерево (полное на всех уровнях за исключением, возможно, самого нижнего), она содержит не более $2^{h+1}-1$ элементов (когда она полная) и не менее $2^h - 1 + 1 = 2^h$ элементов (когда на самом нижнем уровне находится только один элемент, а остальные уровни полные).

Решение упражнения 6.1.2

Для n-элементной пирамиды высотой h из упражнения 6.1.1 известно, что

$$2^h \le n \le 2^{h+1}-1 < 2^{h+1}.$$

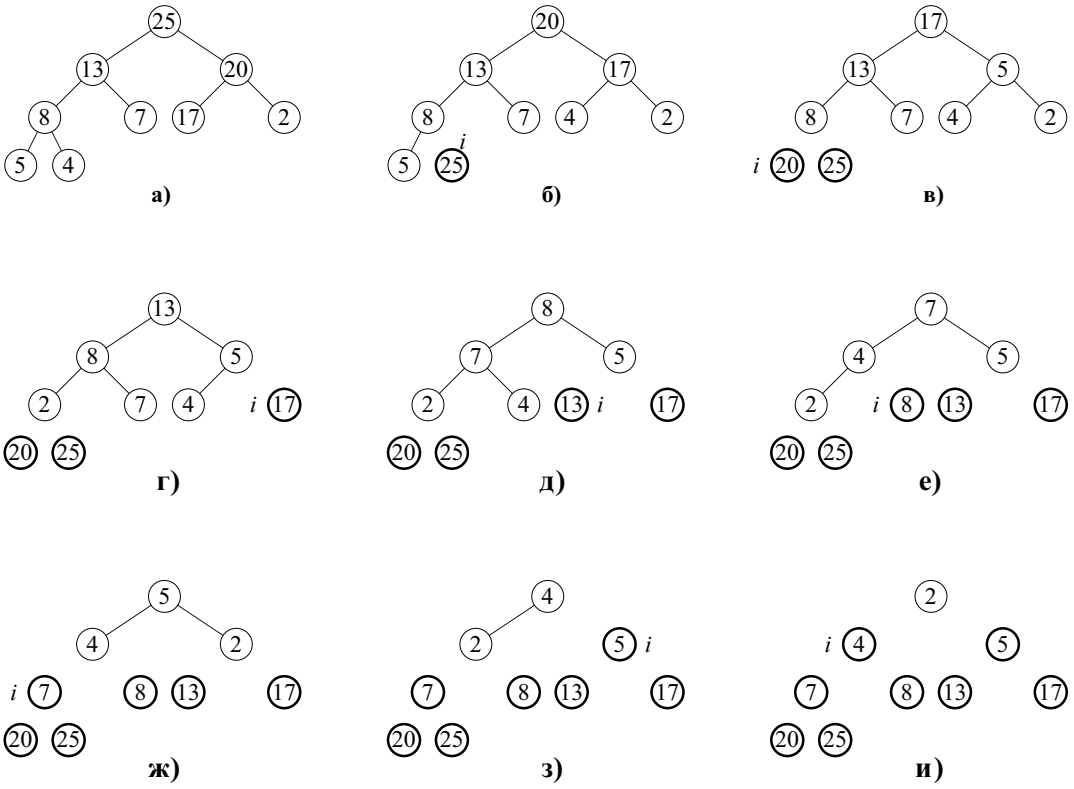
Таким образом, $h \le \lg n < h + 1$. Поскольку h — целое число, $h = \lfloor \lg n \rfloor$ (по определению $\lfloor\ \rfloor$).

Решение упражнения 6.2.7

Если поместить в корень значение, которое меньше всех значений в левом и правом поддеревьях, то процедура Max-Heapify будет вызываться рекурсивно до тех пор, пока не доберется до листа. Чтобы рекурсивные вызовы проходили по самому длинному пути к листу, понадобится выбрать значения, которые заставят Max-Heapify всегда рекурсивно выполняться на левом дочернем узле. Процедура следует по левой ветви, когда левый дочерний узел больше или равен правому дочернему узлу, так что добиться этого позволит помещение, например, 0 в корень и 1 во все остальные узлы. С такими значениями процедура Max-Heapify будет вызываться h раз (где h — высота пирамиды, которая представляет собой количество ребер в самом длинном

пути от корня до листа). Таким образом, время работы MAX-HEAPIFY составит $\Theta(h)$ (т.к. каждый вызов выполняет $\Theta(1)$-ю часть работы), что равно $\Theta(\lg n)$. Поскольку имеется случай, когда время работы процедуры MAX-HEAPIFY составляет $\Theta(\lg n)$, то время ее работы в наихудшем случае будет равно $\Omega(\lg n)$.

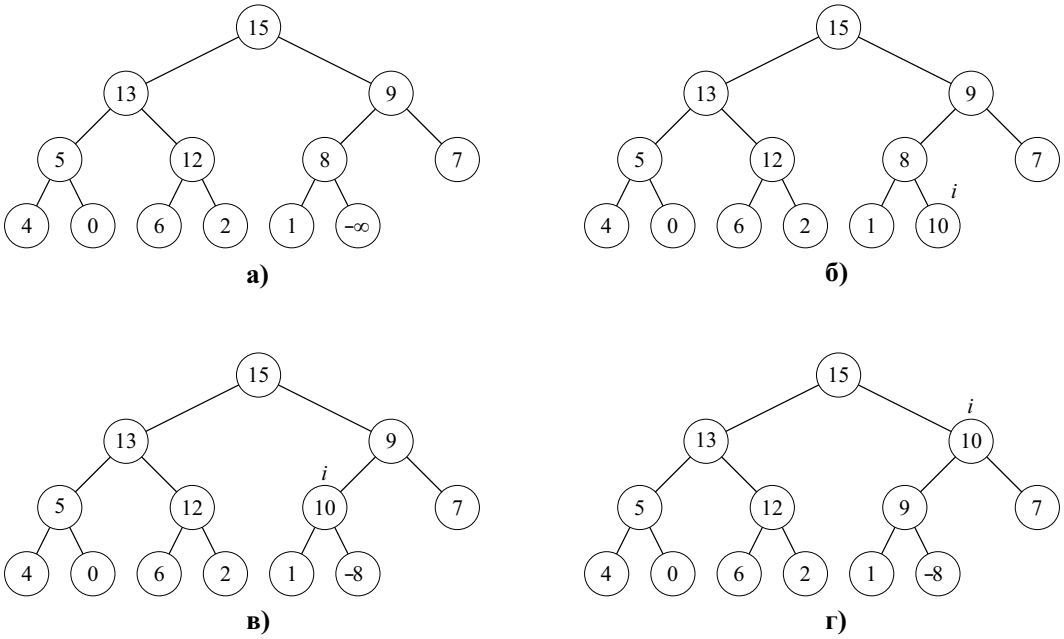
Решение упражнения 6.4.1



A	2	4	5	7	8	13	17	20	25

Решение упражнения 6.5.2

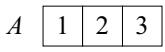
Время работы составляет $O(\lg n)$ плюс накладные расходы, связанные с отображением объектов очереди с приоритетами на индексы массива.



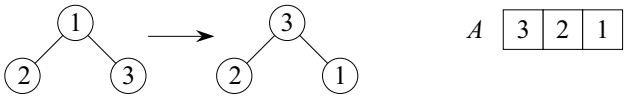
Решение задачи 6.1

а) Процедуры BUILD-MAX-HEAP и BUILD-MAX-HEAP′ не всегда создают одну и ту же пирамиду при запуске на том же самом входном массиве. Рассмотрим следующий контрпример.

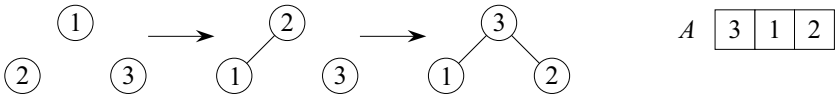
Входной массив A:



BUILD-MAX-HEAP(A):



BUILD-MAX-HEAP′(A):



б) Верхняя оценка времени $O(n \lg n)$ непосредственно следует из $n - 1$ вызовов процедуры MAX-HEAP-INSERT, каждый из которых занимает время $O(\lg n)$. Для получения нижней оценки $\Omega(n \lg n)$ рассмотрим случай, когда элементы входного массива расположены в строго возрастающем порядке. Каждый вызов MAX-HEAP-INSERT заставляет процедуру MAX-HEAP-INCREASE-KEY проходить весь путь вплоть до корня. Так как глубина узла i равна $\lfloor \lg i \rfloor$, общее время составляет

$$\begin{aligned}
\sum_{i=1}^{n} \Theta\left(\lfloor \lg i \rfloor\right) &\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta\left(\lfloor \lg \lceil n/2 \rceil \rfloor\right) \geq \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta\left(\lfloor \lg(n/2) \rfloor\right) = \\
&= \sum_{i=\lceil n/2 \rceil}^{n} \Theta\left(\lfloor \lg n - 1 \rfloor\right) \geq \\
&\geq (n/2) \cdot \Theta(\lg n) = \\
&= \Omega(n \lg n).
\end{aligned}$$

Следовательно, построение пирамиды из n элементов процедурой BUILD-MAX-HEAP′ занимает в наихудшем случае время $\Theta(n \lg n)$.

Решения для главы 7

Решение упражнения 7.2.3

Предположим, что PARTITION вызывается для подмассива $A[p : r]$, элементы которого отличаются друг от друга и расположены в порядке убывания. Процедура PARTITION выбирает наименьший элемент $A[r]$ в качестве опорного элемента. Любая проверка в строке 4 не проходит, поэтому во время выполнения цикла **for** обмен элементами не производится. Перед тем, как PARTITION возвращает управление, в строке 6 обнаруживается, что $i = p - 1$, поэтому элементы $A[p]$ и $A[r]$ меняются местами, а p возвращается в качестве позиции опорного элемента. Подмассив, который содержит элементы, меньшие или равные опорному элементу, пуст. Подмассив $A[p + 1 : r]$ для элементов, превышающих опорный, содержит все элементы кроме опорного, расположенные в порядке убывания, за исключением того, что максимальный элемент данного подмассива находится в $A[r]$.

Когда процедура QUICKSORT вызывает PARTITION для $A[p : q - 1]$, ничего не меняется, т.к. этот подмассив пуст. Когда QUICKSORT вызывает PARTITION для $A[q + 1 : r]$, опорный элемент является наибольшим элементом в подмассиве. Хотя все проверки в строке 4 проходят успешно, в строке 6 индексы i и j всегда равны, поэтому, как и в случае, когда опорный элемент является наименьшим элементом, во время выполнения цикла **for** обмен элементами не происходит. Перед выходом из PARTITION в строке 6 обнаруживается, что $i = r - 1$, так что обмен в строке 6 оставляет опорный элемент в $A[r]$. Процедура PARTITION возвращает r в качестве позиции опорного элемента. Теперь подмассив с элементами, которые меньше или равны опорному, содержит все элементы кроме опорного, расположенные в порядке убывания, а подмассив с элементами больше опорного пуст. Таким образом, следующий вызов PARTITION выполняется для подмассива, элементы которого расположены в порядке убывания, и снова актуален описанный выше первый случай.

Следовательно, каждый рекурсивный вызов выполняется с подмассивом, который меньше только на один элемент, что дает рекуррентное соотношение для времени работы $T(n) = T(n-1) + \Theta(n)$ с решением $\Theta(n^2)$.

Решение упражнения 7.2.5

Минимальная глубина присуща пути, который всегда занимает меньшую часть разбиения, т.е. получается умножением количества элементов на α. Один уровень рекурсии уменьшает количество элементов с n до αn, а i уровней рекурсии сокращают количество элементов до $\alpha^i n$. В листовом узле остается только один элемент, поэтому в листовом узле с минимальной глубиной m мы имеем $\alpha^m n = 1$. Таким образом, $\alpha^m = 1/n$. Взяв логарифмы, мы получаем $m \lg \alpha = \lg n$, или $m = -\lg n / \lg \alpha$. (Эта величина является положительной, потому что из $0 < \alpha < 1$ следует $\lg \alpha < 0$.)

Аналогичным образом путь с максимальной глубиной соответствует тому, что всегда занимает бо́льшую часть разбиения, т.е. каждый раз сохраняется доля β элементов. Максимальная глубина M достигается, когда остается один элемент, т.е. когда $\beta^M n = 1$. Таким образом, $M = -\lg n / \lg \beta$. (И снова эта величина является положительной, поскольку $0 < \beta < 1$ подразумевает, что $\lg \beta < 0$.)

Все приведенные уравнения приблизительны, т.к. мы игнорируем округление до целых чисел.

Решения для главы 8

Решение упражнения 8.1.3

Если сортировка выполняется за линейное время для m входных перестановок, то высота h части дерева решений, состоящей из m соответствующих листьев и их предков, линейна.

Воспользуемся такими же рассуждениями, как и при доказательстве теоремы 8.1, и покажем, что алгоритм сортировки сравнением, время выполнения которого линейно, не существует для значений m, равных $n!/2$, $n!/n$ или $n!/2^n$.

Мы имеем $2^h \geq m$, что дает $h \geq \lg m$. Для всех приведенных здесь возможных значений m справедливо $\lg m = \Omega(n \lg n)$, следовательно, $h = \Omega(n \lg n)$. В частности, согласно уравнению (3.25) мы получаем

$$\lg \frac{n!}{2} = \ln n! - 1 \geq n \lg n - n \lg e - 1,$$

$$\lg \frac{n!}{n} = \ln n! - \lg n \geq n \lg n - n \lg e - \lg n,$$

$$\lg \frac{n!}{2^n} = \ln n! - n \geq n \lg n - n \lg e - n.$$

Решение упражнения 8.2.3

Предложенное решение также можно рассматривать как решение упражнения 8.2.2.

Обратите внимание, что довод в пользу корректности, приведенный в книге, не зависит от порядка обработки A. Алгоритм корректен независимо от того, обрабатывается массив A от начала к концу или от конца к началу.

Но модифицированный алгоритм нестабилен. Как и ранее, в последнем цикле **for** элемент, равный ранее взятому из A, помещается перед предыдущим (т.е. в позицию с меньшим индексом) в выходном массиве B. Исходный алгоритм был стабилен, потому что элемент, взятый из A позже, изначально имел меньший индекс, чем взятый раньше. Но в модифицированном алгоритме элемент, взятый из A позже, изначально имеет больший индекс, чем элемент, взятый из A раньше.

В частности, алгоритм по-прежнему размещает элементы со значением k в позициях от $C[k-1]+1$ до $C[k]$, но в порядке, обратном тому, в котором они встречаются в A.

Перепишем процедуру COUNTING-SORT, которая помещает элементы с одинаковыми значениями в выходной массив в порядке возрастания индекса и является устойчивой:

```
COUNTING-SORT(A, n, k)
 Пусть B[1 : n], C[0 : k] и L[0 : k] будут новыми массивами
 for i = 0 to k
     C[i] = 0
 for j = 1 to n
     C[A[j]] = C[A[j]] + 1
 // C[i] теперь содержит количество элементов, равных i
 L[0] = 1
 for i = 1 to k
     L[i] = L[i – 1] + C[i – 1]
 // L[i] теперь содержит индекс первого элемента A,
 // значение которого равно i
 for j = 1 to n
     B[L[A[j]]] = A[j]
     L[A[j]] = L[A[j]] + 1
 return B
```

Решение упражнения 8.3.3

Базовый случай. Если $d = 1$, то имеется только одна цифра, так что сортировка по этой цифре приводит к сортировке самого массива.

Индуктивный шаг. Предполагая, что поразрядная сортировка работает для $d – 1$ цифр, мы покажем, что она работает и для d цифр.

Поразрядная сортировка сортирует отдельно по каждой цифре, начиная с первой цифры. Таким образом, поразрядная сортировка по d цифрам, которая сортирует по цифрам $1, \ldots, d$, эквивалентна поразрядной сортировке по младшим $d – 1$ цифрам, за которой следует сортировка по цифре d. Согласно нашему индуктивному предположению сортировка по младшим $d – 1$ цифрам работает, поэтому непосредственно перед сортировкой по цифре d элементы располагаются в порядке, соответствующем их младшим $d – 1$ цифрам.

Сортировка по цифре d упорядочит элементы по их d-й цифре. Рассмотрим два элемента, a и b, с d-ми цифрами a_d и b_d соответственно.

- Если $a_d < b_d$, тогда процедура сортировки поместит a перед b, что корректно, т.к. $a < b$ независимо от младших цифр.

- Если $a_d > b_d$, тогда процедура сортировки поместит a после b, что корректно, потому что $a > b$ независимо от младших цифр.
- Если $a_d = b_d$, тогда процедура сортировки оставит a и b в том же порядке, в котором они были, поскольку он стабилен. Но этот порядок уже корректен, т.к. корректный порядок a и b определяют младшие $d - 1$ цифр, когда их d-е цифры равны, и элементы уже отсортированы по младшим $d - 1$ цифрам.

Если бы промежуточная сортировка была нестабильной, то она могла бы переставить элементы, у которых d-е цифры были равны — элементы, которые находились бы в правильном порядке после сортировки по их младшим цифрам.

Решение упражнения 8.3.5

Числа необходимо трактовать как трехзначные в системе счисления с основанием n. Каждая цифра находится в диапазоне от 0 до $n-1$. Эти трехзначные числа должны сортироваться с помощью поразрядной сортировки.

Выполняются три вызова процедуры сортировки подсчетом, каждый из которых занимает время $\Theta(n + n) = \Theta(n)$, так что общее время составляет $\Theta(n)$.

Решение задачи 8.1

а) Для сортировки алгоритмом на основе сравнения A никакие две входные перестановки не могут достичь одного и того же листа дерева решений, поэтому в T_A должно быть достигнуто не менее $n!$ листьев, по одному для каждой возможной входной перестановки. Поскольку алгоритм A является детерминированным, он всегда должен достигать одного и того же листа при заданной в качестве входных данных перестановке, так что достигается не более $n!$ листьев (по одному для каждой перестановки). Следовательно, достигается в точности $n!$ листьев, по одному для каждой входной перестановки.

Каждый из этих $n!$ листьев будет достигаться с вероятностью $1/n!$, т.к. каждая из $n!$ возможных перестановок является входными данными с вероятностью $1/n!$. Все остальные листья будут достигаться с вероятностью 0, потому что они не достижимы ни для каких входных данных.

Не теряя общности, можно предположить, что для оставшейся части этой задачи пути, ведущие только к листьям с нулевой вероятностью, в дереве отсутствуют, поскольку они не могут повлиять на время работы сортировки. То есть мы можем предположить, что T_A состоит только из $n!$ листьев, достигаемых с вероятностью $1/n!$, и их предков.

б) Если $k > 1$, то корень T не является листом. Все листья T должны быть листьями в LT и RT. Поскольку каждый лист на глубине h в LT или RT имеет глубину $h + 1$ в T, то длина $D(T)$ должна быть суммой $D(LT)$, $D(RT)$ и общего количества листьев k. Чтобы доказать последнее утверждение, предположим, что $d_T(x)$ — глубина узла x в дереве T. Тогда

$$\begin{aligned}
D(T) &= \sum_{x \in \text{листья}(T)} d_T(x) = \\
&= \sum_{x \in \text{листья}(LT)} d_T(x) + \sum_{x \in \text{листья}(RT)} d_T(x) = \\
&= \sum_{x \in \text{листья}(LT)} (d_{LT}(x) + 1) + \sum_{x \in \text{листья}(RT)} (d_{RT}(x) + 1) = \\
&= \sum_{x \in \text{листья}(LT)} d_{LT}(x) + \sum_{x \in \text{листья}(RT)} d_{RT}(x) + \sum_{x \in \text{листья}(T)} 1 = \\
&= D(LT) + D(RT) + k.
\end{aligned}$$

в) Чтобы показать, что $d(k) = \min\{d(i) + d(k - i) + k : 1 \le i \le k - 1\}$, мы покажем по отдельности, что $d(k) \le \min\{d(i) + d(k - i) + k : 1 \le i \le k - 1\}$ и $d(k) \ge \min\{d(i) + d(k - i) + k : 1 \le i \le k - 1\}$.

- Мы продемонстрируем, что $d(k) \le \min\{d(i) + d(k - i) + k : 1 \le i \le k - 1\}$, показав, что $d(k) \le d(i) + d(k - i) + k$ для $i = 1, 2, \ldots, k - 1$. Согласно упражнению Б.5.4 существуют полные двоичные деревья с i листьями для любого i от 1 до $k - 1$. Следовательно, мы можем создать деревья решений LT с i листьями и RT с $k - i$ листьями, такие что $D(LT) = d(i)$ и $D(RT) = d(k - i)$. Дерево T должно быть построено так, чтобы LT и RT были левым и правым поддеревьями корня T соответственно. Тогда

$$\begin{aligned}
&d(k) \le \\
&\quad \le D(T) = \quad \text{(по определению } d \text{ как минимального значения } D(T)) \\
&\quad = D(LT) + D(RT) + k = \quad \text{(согласно части (б))} \\
&\quad = d(i) + d(k - i) + k \qquad \text{(согласно выбору } LT \text{ и } RT).
\end{aligned}$$

- Мы продемонстрируем, что $d(k) \geq \min\{d(i) + d(k - i) + k : 1 \leq i \leq k - 1\}$, показав, что $d(k) \geq d(i) + d(k - i) + k$ для некоторого i из $\{1, 2, \ldots, k - 1\}$. Имея дерево T с k листьями, такое что $D(T) = d(k)$, пусть LT и RT будут левым и правым поддеревьями T соответственно, а i — количеством листьев в LT. Тогда $k - i$ представляет собой количество листьев в RT и

$$
\begin{aligned}
&d(k) = \\
&\quad = D(T) = && \text{(согласно выбору } T\text{)} \\
&\quad = D(LT) + D(RT) + k = && \text{(согласно части (б))} \\
&\quad = d(i) + d(k - i) + k && \text{(по определению } d \text{ как} \\
&&& \text{минимального значения } D(T)\text{)}.
\end{aligned}
$$

Ни значение i, ни значение $k - i$ не может быть равно 0 (и, следовательно, $1 \leq i \leq k - 1$), т.к. если бы одно из них было равно 0, то либо LT, либо RT содержали бы все k листьев T. Корень T имел бы только одного потомка, так что T не было бы полным двоичным деревом, а потому не было бы деревом решений.

г) Пусть $f_k(i) = i \lg i + (k - i) \lg(k - i)$. Чтобы отыскать значение i, которое минимизирует f_k, необходимо найти i, для которого производная f_k по i равна 0 при $i = k/2$:

$$
\begin{aligned}
f_k'(i) &= \frac{d}{di}\left(\frac{i\ln i + (k-i)\ln(k-i)}{\ln 2}\right) = \\
&= \frac{\ln i + 1 - \ln(k-i) - 1}{\ln 2} = \\
&= \frac{\ln i - \ln(k-i)}{\ln 2}.
\end{aligned}
$$

Чтобы убедиться, что это действительно минимум (а не максимум), понадобится проверить, положительна ли вторая производная f_k при $i = k/2$:

$$
\begin{aligned}
f_k''(i) &= \frac{d}{di}\left(\frac{\ln i - \ln(k-i)}{\ln 2}\right) = \\
&= \frac{1}{\ln 2}\left(\frac{1}{i} + \frac{1}{k-i}\right).
\end{aligned}
$$

$$
\begin{aligned}
f_k''(k/2) &= \frac{1}{\ln 2}\left(\frac{2}{k} + \frac{2}{k}\right) = \\
&= \frac{1}{\ln 2} \cdot \frac{4}{k} > \\
&> 0 \qquad \text{(поскольку } k > 1\text{)}.
\end{aligned}
$$

Теперь воспользуемся методом подстановки, чтобы доказать, что $d(k) = \Omega(k \lg k)$. Базовый случай индукции удовлетворяется, т.к. $d(1) \geq 0 = c \cdot 1 \cdot \lg 1$ для любой константы c. В качестве индукционного шага предположим, что $d(i) \geq ci \lg i$ для $1 \leq i \leq k - 1$, где c — константа, которую необходимо определить:

$$\begin{aligned}
d(k) &= \min\{d(i) + d(k-i) + k : 1 \leq i \leq k-1\} \geq \\
&\geq \min\{c(i \lg i + (k-i)\lg(k-i)) + \boldsymbol{k} : 1 \leq i \leq k-1\} = \\
&= c\left(\frac{k}{2}\lg\frac{k}{2} + \left(k - \frac{k}{2}\right)\lg\left(k - \frac{k}{2}\right)\right) + k = \\
&= ck \lg\left(\frac{k}{2}\right) + k = \\
&= c(k \lg k - k) + k = \\
&= ck \lg k + (k - ck) \geq \\
&\geq ck \lg k, \text{ если } \; c \leq 1,
\end{aligned}$$

поэтому $d(k) = \Omega(k \lg k)$.

д) Используя результат части (г) и тот факт, что дерево T_A (модифицированное в нашем решении части (а)) имеет $n!$ листьев, мы можем прийти к следующему заключению:

$$D(T_A) \geq d(n!) = \Omega(n! \lg(n!)).$$

$D(T_A)$ — это сумма длин путей в дереве решений для сортировки всех входных перестановок, а длины путей пропорциональны времени работы. Поскольку $n!$ перестановок имеют одинаковую вероятность $1/n!$, ожидаемое время сортировки n случайных элементов (одной входной перестановки) равно совокупному времени всех перестановок, деленному на $n!$:

$$\frac{\Omega(n!\lg(n!))}{n!} = \Omega(\lg(n!)) = \Omega(n \lg n).$$

е) Мы покажем, как модифицировать дерево решений для рандомизированного алгоритма сортировки, чтобы определить дерево решений для детерминированного алгоритма сортировки, который, по крайней мере, не хуже рандомизированного алгоритма с точки зрения среднего количества сравнений.

В каждом узле дерева для рандомизированной сортировки выберите дочерний узел с наименьшим поддеревом (поддерево с наименьшим средним количеством сравнений на пути к листу). Удалите все остальные дочерние узлы узла дерева для рандомизированной сортировки и вырежьте сам узел.

Детерминированный алгоритм, соответствующий такому модифицированному дереву, по-прежнему работает, т.к. рандомизированный алгоритм работал независимо от того, какой путь выбирался из каждого узла дерева для рандомизированной сортировки.

Среднее количество сравнений для модифицированного алгоритма не превышает среднего количества сравнений для исходного рандомизированного алгоритма, поскольку мы отбрасываем поддеревья с более высоким средним значением в каждом случае. В частности, каждый раз, когда мы удаляем узел дерева для рандомизированной сортировки, мы оставляем общее среднее значение меньше или равным тому, каким оно было раньше, потому что

- тот же самый набор входных перестановок достигает того же модифицированного поддерева, что и раньше, но эти входные данные обрабатываются за меньшее или равное среднему время, нежели раньше;
- остальная часть дерева не изменяется.

Таким образом, рандомизированный алгоритм занимает в среднем не меньше времени, чем соответствующий детерминированный алгоритм. (Мы показали, что среднее время выполнения детерминированной сортировки сравнением составляет $\Omega(n \lg n)$, следовательно, ожидаемое время рандомизированной сортировки сравнением тоже составляет $\Omega(n \lg n)$.)

Решения для главы 9

Решение упражнения 9.3.1

Для групп из 7 элементов алгоритм по-прежнему работает за линейное время. Количество групп g не превышает $n/7$. Существует не менее $4(\lfloor g/2 \rfloor + 1) \geq 2g$ элементов, которые больше или равны опорному элементу, и не менее $4\lceil g/2 \rceil \geq 2g$ элементов, которые меньше или равны опорному элементу. В результате в рекурсивном вызове остается не более $7g - 2g = 5g \leq 5n/7$ элементов. Рекуррентное со-

отношение принимает вид $T(n) \le T(n/7) + T(5n/7) + O(n)$, что можно показать методом подстановки и получить в результате $T(n) = O(n)$.

На самом деле любая нечетная группа размером, большим или равным 5, обеспечивает работу алгоритма за линейное время.

Решение упражнения 9.3.3

Модификация быстрой сортировки, позволяющая ей работать за время $O(n \lg n)$ в наихудшем случае, использует детерминированную процедуру PARTITION-AROUND, которая принимает элемент для разбиения в качестве входного параметра.

Процедура SELECT принимает массив A, границы p и r подмассива в A и ранг i порядковой статистики, и за время, линейно зависящее от размера подмассива $A[p : r]$, возвращает i-й наименьший элемент в $A[p : r]$.

```
BEST-CASE-QUICKSORT(A, p, r)
 if p < r
     i = ⌊(r − p + 1)/2⌋
     x = SELECT(A, p, r, i)
     q = PARTITION-AROUND(A, p, r, x)
     BEST-CASE-QUICKSORT(A, p, q − 1)
     BEST-CASE-QUICKSORT(A, q + 1, r)
```

Для n-элементного массива наибольший подмассив, на котором рекурсивно вызывается BEST-CASE-QUICKSORT, содержит $n/2$ элемента. Такая ситуация возникает, когда значение $n = r - p + 1$ является четным; тогда подмассив $A[q + 1 : r]$ содержит $n/2$ элементов, а в подмассиве $A[p : q - 1]$ находится $n/2 - 1$ элементов.

Поскольку процедура BEST-CASE-QUICKSORT всегда рекурсивно вызывается на подмассивах, размер которых не превышает половины исходного массива, рекуррентное соотношение для наихудшего случая выглядит как $T(n) \le 2T(n/2) + \Theta(n) = O(n \lg n)$.

Решение упражнения 9.3.6

Пусть процедура MEDIAN принимает в качестве параметров массив A и индексы подмассива p и r, а возвращает значение медианного элемента подмассива $A[p : r]$ за время $O(n)$ в наихудшем случае.

Процедура MEDIAN вызывает процедуру SELECT′ с линейным временем работы, которая ищет i-й наименьший элемент в $A[p : r]$. Она использует детерминированную процедуру PARTITION-AROUND, принимающую во входном параметре элемент для разбиения.

```
SELECT′(A, p, r, i)
 if p == r
     return A[p]
 x = MEDIAN(A, p, r)
 q = PARTITION-AROUND(A, p, r, x)
 k = q − p + 1
 if i == k
     return A[q]
 elseif i < k
     return SELECT′(A, p, q − 1, i)
 else return SELECT′(A, q + 1, r, i − k)
```

Поскольку x — медианный элемент массива $A[p : r]$, каждый подмассив $A[p : q - 1]$ и $A[q + 1 : r]$ содержит не более половины элементов $A[p : r]$. Рекуррентное соотношение для времени работы SELECT′ в наихудшем случае имеет вид $T(n) \le T(n/2) + O(n) = O(n)$.

Решение задачи 9.1

Предположим, что числа изначально находятся в массиве.

а) Отсортируйте числа с помощью сортировки слиянием или пирамидальной сортировки, что займет время $\Theta(n \lg n)$ в наихудшем случае. (Не используйте быструю сортировку или сортировку вставками, которые могут потребовать времени $\Theta(n^2)$.) Поместите i наибольших элементов (доступных напрямую в отсортированном массиве) в выходной массив, что займет время $\Theta(i)$.

Общее время работы в наихудшем случае составляет $\Theta(n \lg n + i) = \Theta(n \lg n)$ (потому что $i \le n$).

б) Реализуйте очередь с приоритетами в виде пирамиды. Постройте пирамиду с помощью BUILD-HEAP, что займет время $\Theta(n)$, затем i раз вызовите HEAP-EXTRACT-MAX, чтобы получить i наибольших элементов за время $\Theta(i \lg n)$ в наихудшем случае, и сохраните их в выходном массиве в порядке, обратном извлечению. Время извлечения в наихудшем случае составляет $\Theta(i \lg n)$, т.к.

- i извлечений из пирамиды, содержащей $O(n)$ элементов, занимает время $i \cdot O(\lg n) = O(i \lg n)$;
- половина из i извлечений выполняются из пирамиды, в которой содержится больше или равно $n/2$ элементов, так что эти $i/2$ извлечения занимают время $(i/2)\Omega(\lg(n/2)) = \Omega(i \lg n)$ в наихудшем случае.

Общее время выполнения в наихудшем случае равно $\Theta(n + i \lg n)$.

в) Примените процедуру SELECT из раздела 9.3 для нахождения i-го наибольшего числа за время $\Theta(n)$. Выполните разбиение относительно этого числа за время $\Theta(n)$. Отсортируйте i наибольших чисел за время $\Theta(i \lg i)$ в наихудшем случае (с помощью сортировки слиянием или пирамидальной сортировки).

Общее время выполнения в наихудшем случае составляет

$$\Theta(n + i \lg i).$$

Обратите внимание, что метод (в) всегда асимптотически не хуже двух других методов, а метод (б) асимптотически не хуже метода (а).

Решения для главы 11

Решение упражнения 11.2.1

Для каждой пары ключей k, l, где $k \neq l$, определим индикаторную случайную величину $X_{kl} = \mathrm{I}\{h(k) = h(l)\}$. Поскольку мы предполагаем, что используется независимое равномерное хеширование, то $\Pr\{X_{kl} = 1\} = \Pr\{h(k) = h(l)\} = 1/m$ и, следовательно, $\mathrm{E}[X_{kl}] = 1/m$.

Теперь определим случайную величину Y как общее количество коллизий, так что $Y = \sum_{k \neq l} X_{kl}$. Ожидаемое количество коллизий равно

$$\begin{aligned}
\mathrm{E}[Y] &= \mathrm{E}\left[\sum_{k \neq l} X_{kl}\right] = \\
&= \sum_{k \neq l} \mathrm{E}[X_{kl}] = \\
&= \binom{n}{2} \frac{1}{m} = \\
&= \frac{n(n-1)}{2} \cdot \frac{1}{m} = \\
&= \frac{n(n-1)}{2m}.
\end{aligned}$$

(согласно свойству линейности математического ожидания)

Решение упражнения 11.2.4

Флаг в каждом слоте указывает, свободен ли он.

- Свободный слот находится в списке свободных элементов — двусвязном списке всех свободных элементов в таблице. Таким образом, слот содержит два указателя.
- Используемый слот содержит элемент и указатель (возможно, NIL) на следующий элемент, хеш которого соответствует этому слоту. (Разумеется, этот указатель указывает на другой элемент в таблице.)

Операции

Вставка.

- Если элемент хешируется в свободный слот, тогда просто удалите слот из списка свободных мест и сохраните элемент там (с указателем NIL). Список свободных мест должен быть двусвязным, чтобы удаление выполнялось за время $O(1)$.
- Если элемент хешируется в занятый слот j, тогда проверьте, "принадлежит" ли ему уже находящийся там элемент x (его ключ тоже хешируется в слот j).
  - Если да, тогда добавьте новый элемент к цепочке элементов в данном слоте. Для этого выделите память под новый слот (например, из начала списка свободных мест) для нового элемента и поместите этот новый слот во главе списка, на который указывает хешируемый слот (j).
  - Если нет, то x является частью цепочки другого слота. Переместите x в новый слот, выбрав его из списка свободных слотов, скопируйте содержимое старого слота (j) (элемент x и указатель) в новый слот и обновите указатель в слоте, указывающий на j, чтобы он указывал на новый слот. Затем вставьте новый элемент в теперь пустой слот обычным образом.

    Чтобы обновить указатель на j, необходимо найти его в цепочке элементов, начинающейся с хеша слота x.

Удаление. Пусть j является слотом, к которому привязан хешируемый элемент x, подлежащий удалению.

- Если x — единственный элемент в j (j не указывает ни на какие другие элементы), тогда просто освободите слот, вернув его в начало списка свободных слотов.

- Если x находится в j, но есть указатель на цепочку других элементов, тогда переместите первый элемент, на который он указывает, в слот j и освободите слот, в котором он находился.
  - Если x найден по указателю из j, тогда просто освободите слот x и вырежьте его из цепочки (т.е. обновите слот, указывающий на x, чтобы он указывал на элемент, следующий за x).
- **Поиск.** Проверьте слот, к которому привязан ключ, и если это не нужный элемент, тогда проследите за цепочкой указателей из слота. Все операции вполне ожидаемо выполняются за время $O(1)$ по той же причине, по которой они выполняются в версии из книги: ожидаемое время поиска по цепочкам составляет $O(1 + \alpha)$ независимо от того, где хранятся цепочки, а тот факт, что все элементы хранятся в таблице, означает, что $\alpha \le 1$. Если бы свободный список был односвязным, то операции, включающие удаление произвольного слота из свободного списка, не выполнялись бы за время $O(1)$.

Решение задачи 11.3

а) Конкретный ключ хешируется в конкретный слот с вероятностью $1/n$. Предположим, что выбрано специфическое множество из k ключей. Вероятность того, что эти k ключей будут вставлены в рассматриваемый слот, а все остальные ключи — в другие места, равна

$$\left(\frac{1}{n}\right)^k \left(1-\frac{1}{n}\right)^{n-k}.$$

Поскольку существует $\binom{n}{k}$ способов выбора k ключей, мы получаем

$$Q_k = \left(\frac{1}{n}\right)^k \left(1-\frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

б) Пусть X_i для $i = 1, 2, \ldots, n$ будет случайной величиной, обозначающей количество ключей, которые хешируются в слот i, и A_i — событие, заключающееся в том, что $X_i = k$, т.е. ровно k ключей хешируются в слот i. Из части (а) имеем $\Pr\{A\} = Q_k$. Тогда

$$\begin{aligned}
P_k &= \Pr\{M = k\} = \\
&= \Pr\{\max\{X_i : 1 \le i \le n\} = k\} = \\
&= \Pr\{\text{существует } i, \text{ такое что } X_i = k \text{ и } X_i \le k \text{ для } i = 1, 2, \ldots, n\} \le \\
&\le \Pr\{\text{существует } i, \text{ такое что } X_i = k\} = \\
&= \Pr\{A_1 \cup A_2 \cup \ldots \cup A_n\} \le \\
&\le \Pr\{A_1\} + \Pr\{A_2\} + \ldots + \Pr\{A_n\} = \text{(согласно неравенству (B.21))} \\
&= nQ_k.
\end{aligned}$$

в) Начнем с демонстрации двух фактов. Во-первых, $1 - 1/n < 1$ и $n - k \geq 0$, откуда следует, что $(1 - 1/n)^{n-k} \leq 1$. Во-вторых, $n!/(n-k)! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-k+1) < n^k$. Используя эти факты, а также упрощение $k! > (k/e)^k$ уравнения (3.25), мы получаем

$$\begin{aligned} Q_k = \left(\frac{1}{n}\right)^k \left(1-\frac{1}{n}\right)^{n-k} \leq & \qquad \frac{n!}{k!(n-k)!} \\ \leq \frac{n!}{n^k k!(n-k)!} < & \qquad ((1-1/n)^{n-k} < 1) \\ < \frac{1}{k!} < & \qquad (n!/(n-k)! < n^k) \\ < \frac{e^k}{k^k} & \qquad (k! > (k/e)^k). \end{aligned}$$

г) Обратите внимание, что при $n = 2$ справедливо равенство $\lg\lg n = 0$, поэтому ради точности нужно предположить, что $n \geq 3$.

В части (в) мы показали, что $Q_k < e^k/k^k$ для любого k; в частности, это неравенство выполняется для k_0. Таким образом, достаточно показать, что $e^{k_0}/{k_0}^{k_0} < 1/n^3$ или, что то же самое, $n^3 < {k_0}^{k_0}/e^{k_0}$.

Логарифмирование обеих частей уравнения дает эквивалентное условие:

$$\begin{aligned} 3\lg n < k_0(\lg k_0 - \lg e) = \\ = \frac{c\lg n}{\lg\lg n}(\lg c + \lg\lg n - \lg\lg\lg n - \lg e). \end{aligned}$$

Деление обеих частей на $\lg n$ дает условие:

$$\begin{aligned} 3 < \frac{c}{\lg\lg n}(\lg c + \lg\lg n - \lg\lg\lg n - \lg e) = \\ = c\left(1 + \frac{\lg c - \lg e}{\lg\lg n} - \frac{\lg\lg\lg n}{\lg\lg n}\right). \end{aligned}$$

Пусть x — последнее выражение в скобках:

$$x = \left(1 + \frac{\lg c - \lg e}{\lg\lg n} - \frac{\lg\lg\lg n}{\lg\lg n}\right).$$

Нам нужно показать, что существует константа $c > 1$, такая что $3 < cx$.

Заметив, что $\lim_{n\to\infty} x = 1$, мы видим, что существует n_0, такое что $x \geq 1/2$ для всех $n \geq n_0$. Таким образом, для удовлетворения условия $n \geq n_0$ подходит любая константа $c > 6$.

Меньшие значения n, в частности, $3 \leq n < n_0$, поддерживаются следующим образом. Поскольку значение n ограничено целым числом, в диапазоне $3 \leq n < n_0$ имеется конечное количество n. Для каждого такого значения n можно вычислить выражение x и определить значение c, удовлетворяющее условию $3 < cx$ для всех значений n. Финальным значением c, которое используется далее, будет большее из перечисленных ниже значений:

- 6, которое работает для всех $n \geq n_0$;
- $\max\{c : 3 < cx \text{ и } 3 \leq n < n_0\}$, т.е. наибольшее значение c, которое было выбрано для диапазона $3 \leq n < n_0$.

Таким образом, мы показали, что $Q_{k_0} < 1/n^3$, как и требовалось.

Чтобы продемонстрировать соблюдение условия $P_k < 1/n^2$ для $k \geq k_0$, заметим, что согласно части (б) мы имеем $P_k \leq nQ_k$ для всех k. Выбор $k = k_0$ дает $P_{k_0} \leq nQ_{k_0} < n \cdot (1/n^3) = 1/n^2$. При $k > k_0$ мы покажем, что можно выбрать константу c так, чтобы удовлетворялось условие $Q_k < 1/n^3$ для всех $k \geq k_0$, и прийти к заключению о том, что $P_k < 1/n^2$ для всех $k \geq k_0$.

Для надлежащего выбора c мы примем, что значение c достаточно большое, чтобы соблюдалось условие $k_0 > 3 > e$. Тогда $e/k < 1$ для всех $k \geq k_0$ и поэтому e^k/k^k уменьшается с ростом k. В итоге получаем

$$\begin{aligned} Q_k &< e^k / k^k \leq \\ &\leq e^{k_0} / k^{k_0} = \\ &= Q_{k_0} < \\ &< 1/n^3 \end{aligned}$$

д) Математическое ожидание M равно

$$\begin{aligned} \mathrm{E}[M] &= \sum_{k=0}^{n} k \cdot \Pr\{M = k\} = \\ &= \sum_{k=0}^{k_0} k \cdot \Pr\{M = k\} + \sum_{k=k_0+1}^{n} k \cdot \Pr\{M = k\} \leq \\ &\leq \sum_{k=0}^{k_0} k_0 \cdot \Pr\{M = k\} + \sum_{k=k_0+1}^{n} n \cdot \Pr\{M = k\} = \\ &= k_0 \cdot \Pr\{M \leq k_0\} + n \cdot \Pr\{M > k_0\}, \end{aligned}$$

что и требовалось показать, т.к. $k_0 = c \lg n / \lg\lg n$.

Чтобы показать, что $\mathrm{E}[M] = O(\lg n / \lg\lg n)$, отметим соблюдение условий $\Pr\{M \le k_0\} \le 1$ и

$$\begin{aligned}
\Pr\{M > k_0\} &= \sum_{k=k_0+1}^{n} \Pr\{M = k\} = \\
&= \sum_{k=k_0+1}^{n} P_k < \\
&< \sum_{k=k_0+1}^{n} 1/n^2 < \qquad \text{(согласно части (г))} \\
&< n \cdot (1/n^2) = \\
&= 1/n.
\end{aligned}$$

Отсюда мы заключаем, что

$$\begin{aligned}
\mathrm{E}[M] &\le k_0 \cdot 1 + n \cdot (1/n) = \\
&= k_0 + 1 = \\
&= O(\lg n / \lg\lg n).
\end{aligned}$$

Решения для главы 12

Решение упражнения 12.1.2

В пирамиде ключ узла больше или равен ключам обоих его дочерних узлов. В двоичном дереве поиска ключ узла больше или равен ключу его левого дочернего узла, но меньше или равен ключу его правого дочернего узла.

В отличие от свойства дерева двоичного поиска свойство пирамиды не помогает выдавать узлы в отсортированном порядке, поскольку оно не указывает, какое поддерево узла содержит узел, подлежащий выдаче перед этим узлом. В пирамиде наибольший узел, меньший заданного узла, может находиться в любом из поддеревьев.

Обратите внимание, что если бы свойство пирамиды можно было применять для выдачи ключей в отсортированном порядке за время $O(n)$, тогда был бы получен алгоритм сортировки со временем работы $O(n)$, поскольку построение пирамиды занимает только время $O(n)$. Но из теоремы 8.1 известно, что сортировка сравнением должна занимать время $\Omega(n \lg n)$.

Решение упражнения 12.2.7

Обратите внимание, что вызов TREE-MINIMUM с последующими $n-1$ вызовами TREE-SUCCESSOR выполняет точно такой же симметричный обход дерева, как и процедура INORDER-TREE-WALK. Процедура INORDER-TREE-WALK сначала выводит TREE-MINIMUM, а по определению узел, возвращаемый TREE-SUCCESSOR, является следующим узлом в отсортированном порядке, который определяется симметричным обходом дерева.

Время работы этого алгоритма составляет $\Theta(n)$, потому что:

- он требует времени $\Omega(n)$ для выполнения n вызовов процедур;
- он обходит каждое из $n-1$ ребер дерева не более двух раз, что занимает время $O(n)$.

Чтобы убедиться в том, что каждое ребро проходится максимум дважды (один раз вниз по дереву и один раз вверх), рассмотрим ребро между произвольным узлом u и любым из его дочерних узлов, скажем, v. Начиная с корня, обход должен пройти по ребру (u, v) вниз от u к v перед тем, как проходить по нему вверх от v к u. Обход дерева вниз выполняется только в коде TREE-MINIMUM, а вверх — только в коде TREE-SUCCESSOR при поиске следующего узла для узла, не имеющего правого поддерева.

Предположим, что v является левым дочерним узлом узла u.

- Перед выводом u обход должен вывести все узлы в своем левом поддереве, корнем которого является v, что гарантирует нисходящий проход по ребру (u, v).
- После того, как все узлы в левом поддереве u выведены, следующим должен выводиться узел u. Процедура TREE-SUCCESSOR делает восходящий проход к u от максимального элемента (у которого нет правого поддерева) в поддереве с корнем в v. Очевидно, что этот путь включает ребро (u, v), а поскольку выведены все узлы в левом поддереве u, то ребро (u, v) больше проходиться не будет.

Теперь предположим, что v — правый дочерний узел узла u.

- После того, как узел u выведен, вызывается TREE-SUCCESSOR(u). Чтобы добраться до минимального элемента в правом поддереве u (корнем которого является v), необходимо пройти вниз по ребру (u, v).

- После того, как все значения в правом поддереве u выведены, TREE-SUCCESSOR вызывается для максимального элемента (опять-таки не имеющего правого поддерева) в поддереве с корнем v. Процедура TREE-SUCCESSOR проходит путь вверх по дереву до элемента после u, т.к. u уже был выведен. Ребро (u, v) должно быть пройдено вверх по этому пути, и поскольку все узлы в правом поддереве u были выведены, ребро (u, v) больше проходиться не будет.

Следовательно, ни одно ребро не проходится дважды в одном направлении.

Таким образом, этот алгоритм выполняется за время $\Theta(n)$.

Решение упражнения 12.3.3

Вот алгоритм:

```
TREE-SORT(A)
 Пусть T будет пустым двоичным деревом поиска
 for i = 1 to n
     TREE-INSERT(T, A[i])
 INORDER-TREE-WALK(T.root)
```

Наихудший случай: время работы $\Theta(n^2)$; ситуация возникает, когда в результате повторяющихся операций TREE-INSERT получается линейная цепочка узлов.

Наилучший случай: время работы $\Theta(n \lg n)$; ситуация возникает, когда в результате повторяющихся операций TREE-INSERT получается двоичное дерево высотой $\Theta(\lg n)$.

По сравнению с процедурой TREE-INSERT, приведенной в книге, в этой версии опущено присваивание $z.p$, но она должна корректно поддерживать атрибуты *succ*. Новый узел z становится дочерним узлом узла y. Если z становится левым дочерним узлом y, то y должен быть последователем узла z. Код также должен найти предшественника w узла y и установить z в качестве последователя узла w. Если z становится правым дочерним узлом узла y, то все немного упрощается. Необходимо просто сделать последователем узла z последователя узла y, а затем установить z в качестве последователя узла y.

Процедура TRANSPLANT заменяет значения атрибута p узлом, который возвращается вызовом TREE-PARENT.

```
TRANSPLANT(T, u, v)
 z = TREE-PARENT(T, u)
 if z == NIL
     T.root = v
 elseif u == z.left
     z.left = v
 else z.right = v
```

Наконец, в процедуре TREE-DELETE опущены ссылки на атрибут p, а также сделано так, что предшественник удаляемого узла z становится его последователем.

```
TREE-DELETE(T, z)
 x = TREE-PREDECESSOR(T, z)
 if x ≠ NIL
     x.succ = z.succ
 if z.left == NIL
     TRANSPLANT(T, z, z.right)
 elseif z.right == NIL
     TRANSPLANT(T, z, z.left)
 else y = TREE-MINIMUM(z.right)
     if y ≠ z.right
             TRANSPLANT(T, y, y.right)
             y.right = z.right
     TRANSPLANT(T, z, y)
     y.left = z.left
```

Поскольку каждый вызов TREE-PREDECESSOR и TREE-PARENT занимает время $O(h)$, процедуры TREE-INSERT и TREE-DELETE тоже занимают время $O(h)$.

Решение задачи 12.2

Чтобы отсортировать строки из множества S, понадобится сначала вставить их в дерево счисления, а затем использовать обход дерева в прямом порядке для их извлечения в лексикографически отсортированном порядке. Обход дерева выдает строки только для узлов, которые указывают на существование строки (т.е. тех, которые на рис. 12.5 показаны светло-серым цветом).

Корректность

Упорядочение в прямом порядке корректно по приведенным далее причинам.

- Строка любого узла является префиксом строк всех его потомков и потому находится перед ними в отсортированном порядке (правило 2).
- Левые потомки узла располагаются перед его правыми потомками, поскольку соответствующие строки идентичны вплоть до этого родительского узла, и в следующей позиции строки левого поддерева имеют значение 0, а строки правого поддерева — значение 1 (правило 1).

Время

$\Theta(n)$

- Процедура вставки требует времени $\Theta(n)$, т.к. вставка каждой строки занимает время, пропорциональное ее длине (проход по пути в дереве, длина которого равна длине строки), а сумма длин всех строк составляет n.
- Процедура обхода дерева в прямом порядке требует времени $O(n)$. Она выводит текущий узел и рекурсивно вызывает себя на левом и правом поддеревьях, так что время пропорционально количеству узлов в дереве. Количество узлов не превышает 1 плюс сумма (n) длин двоичных строк в дереве, поскольку строка длиной i соответствует пути через корень и i других узлов, но один узел может быть общим для нескольких путей для строк.

Ниже представлен псевдокод для процедуры обхода дерева в прямом порядке. Предполагается, что у каждого узла есть атрибуты *left* и *right*, указывающие на его дочерние узлы (или NIL, если дочерние узлы отсутствуют), и логический атрибут *string*, который отражает, указывает ли узел на фактическую строку (т.е. узел светло-серого цвета на рис. 12.5). Начальный вызов выглядит как PREORDER-RADIX-TREE-WALK($T.root$, ε), где ε обозначает пустую строку. С помощью || обозначается конкатенация строк.

```
PREORDER-RADIX-TREE-WALK(x, string-so-far)
 if x.string == TRUE
    Вывести string-so-far
 if x.left ≠ NIL
    PREORDER-RADIX-TREE-WALK(x.left, string-so-far || 0)
 if x.right ≠ NIL
    PREORDER-RADIX-TREE-WALK(x.left, string-so-far || 1)
```

Решения для главы 13

Решение упражнения 13.1.4

После поглощения каждого красного узла его черным родительским узлом степень каждого черного узла равна

- 2, если оба дочерних узла уже были черными;
- 3, если один дочерний узел был черным, а другой — красным;
- 4, если оба дочерних узла были красными.

Все листья результирующего дерева имеют одну и ту же глубину.

Решение упражнения 13.1.5

В самом длинном пути как минимум каждый второй узел является черным. В самом коротком пути как максимум каждый узел будет черным. Так как два пути содержат то же самое количество черных узлов, длина самого длинного пути не более чем вдвое превышает длину самого короткого пути.

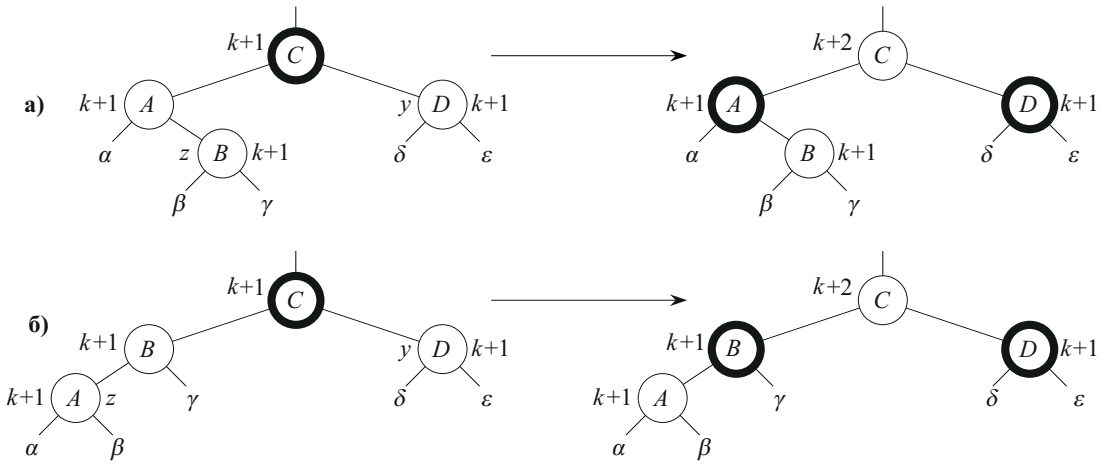
Точнее это можно выразить следующим образом.

> Поскольку каждый путь содержит $\mathrm{bh}(x)$ черных узлов, даже самый короткий путь от x до листа-потомка имеет длину не менее $\mathrm{bh}(x)$. По определению наиболее длинный путь от x до листа-потомка имеет длину $\mathrm{height}(x)$. Поскольку наиболее длинный путь содержит $\mathrm{bh}(x)$ черных узлов, и как минимум половина узлов в самом длинном пути являются черными (согласно свойству 4), то $\mathrm{bh}(x) \geq \mathrm{height}(x)/2$, так что
>
> *длина самого длинного пути* $= \mathrm{height}(x) \leq$
>
> $\leq 2 \cdot \mathrm{bh}(x) \leq$
>
> $\leq$ *удвоенная длина самого короткого пути*.

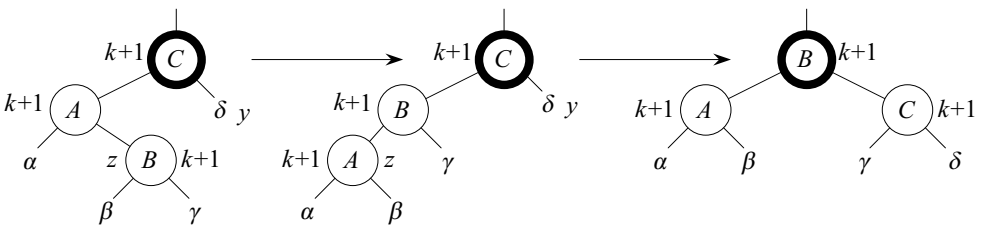
Решение упражнения 13.3.3

Примечание: на рисунках ниже узлы с толстым контуром являются черными, а узлы с обычным контуром — красными.

На рис. 13.5 узлы A, B и D имеют черную высоту $k + 1$ во всех случаях, потому что каждое из их поддеревьев имеет черную высоту k и черный корень. Узел C имеет черную высоту $k + 1$ слева (т.к. его красные потомки имеют черную высоту $k + 1$) и черную высоту $k + 2$ справа (поскольку его черные потомки имеют черную высоту $k + 1$).



На рис. 13.6 узлы A, B и C во всех случаях имеют черную высоту $k + 1$. Слева и посередине каждое из поддеревьев A и B имеет черную высоту k и черный корень, в то время как у C есть одно такое поддерево и красный дочерний узел с черной высотой $k + 1$. Справа каждое из поддеревьев A и C имеет черную высоту k и черный корень, в то время как красные дочерние узлы B имеют черную высоту $k + 1$.



Случай 2 Случай 3

При трансформациях свойство 5 предохраняется. Выше было показано, что черная высота корректно определена внутри изображенных поддеревьев, поэтому свойство 5 предохраняется и внутри них. Свойство 5 предохраняется и для дерева, содержащего изображенные поддеревья, т.к. каждый путь через эти поддеревья к листу добавляет $k + 2$ черных узлов.

Решение задачи 13.1

а) Когда вставляется новый узел, все узлы на пути от корня до добавляемого узла (нового листа) должны быть изменены, т.к. необходимость в указателе на новый дочерний узел распространяется от нового узла ко всем его предкам.

Когда удаляется узел z, возможны три варианта.

- Если узел z имеет максимум один дочерний узел, то z будет удален, так что все предки z должны быть изменены. (Как и при вставке, необходимость в указателе на новый дочерний узел распространяется от удаленного узла.)
- Если узел z имеет два дочерних узла и его последователь y является правым дочерним узлом z, тогда узел z заменяется узлом y, так что все предки z должны быть изменены (т.е. та же ситуация, как если бы у z было не более одного дочернего узла).
- Если узел z имеет два дочерних узла и его последователь y не является правым дочерним узлом z, тогда узел z заменяется узлом y, а y — правым дочерним узлом x узла y. Поскольку y и z — предки узла x, все предки y должны быть изменены.

Так как атрибут с указателем на родительский узел отсутствует, изменять другие узлы не требуется.

б) Ниже приведены две реализации процедуры PERSISTENT-TREE-INSERT. Первая реализация представляет собой версию TREE-INSERT, модифицированную с целью создания новых узлов вдоль пути, куда попадет новый узел, без использования атрибутов с указателями на родительские узлы.

```
PERSISTENT-TREE-INSERT(T, z)
 Создать новое постоянное двоичное дерево поиска T'
 T'.root = COPY-NODE(T.root)
 y  = NIL
 x = T'.root
 while x ≠ NIL
    y = x
  if z.key < x.key
          x = COPY-NODE(x.left)
          y.left = x
  else x = COPY-NODE(x.right)
          y.right = x
 if y == NIL
    new-root = z
 elseif z.key < y.key
    y.left = z
 else y.right = z
 return T'
```

Во второй реализации применяется рекурсивная подпрограмма PERSISTENT-SUBTREE-INSERT(r, z), которая вставляет узел z внутрь поддерева с корнем в узле r в T, копируя узлы по мере необходимости и возвращая либо узел z, либо копию в T' узла r.

```
PERSISTENT-TREE-INSERT(T, z)
 Создать новое постоянное двоичное дерево поиска T'
 T'.root = PERSISTENT-SUBTREE-INSERT(T.root, z)
 return T'
PERSISTENT-SUBTREE-INSERT(r, z)
 if r == NIL
     x = z
 else x = COPY-NODE(r)
     if z.key < r.key
             x.left = PERSISTENT-SUBTREE-INSERT(r.left, z)
     else x.right = PERSISTENT-SUBTREE-INSERT(r.right, z)
 return x
```

в) Как и TREE-INSERT, процедура PERSISTENT-SUBTREE-INSERT выполняет константный объем работы в каждом узле на пути от корня к новому узлу. Поскольку длина пути не превышает h, она занимает время $O(h)$.

Из-за того, что для каждого предка вставленного узла выделяется новый узел (константный объем пространства), процедура также нуждается в пространстве объемом $O(h)$.

г) Если бы существовали атрибуты с указателями на родительские узлы, то по причине появления нового корня каждый узел дерева пришлось бы копировать при вставке нового узла. Чтобы понять, почему, обратите внимание, что дочерние узлы корня изменились бы так, чтобы указывать на новый корень, затем их дочерние узлы изменились бы так, чтобы указывать на них, и т.д. Поскольку имеется n узлов, такое изменение привело бы к созданию $\Omega(n)$ новых узлов при вставке и потребовало бы времени $\Omega(n)$.

д) Из частей (а) и (в) известно, что вставка в постоянное двоичное дерево поиска высотой h, как и вставка в обычное двоичное дерево поиска, занимает в наихудшем случае время $O(h)$. Красно-черное дерево имеет высоту $h = O(\lg n)$, поэтому вставка в обычное красно-черное дерево занимает время $O(\lg n)$. Нам нужно показать, что если красно-черное дерево постоянно, то вставка

все еще может быть выполнена за время $O(\lg n)$. (Мы рассмотрим удаление чуть позже.) Для этого понадобится продемонстрировать два аспекта.

- Как находить необходимые указатели на родительские узлы за время $O(\lg n)$ без использования атрибута с указателем на родительский узел. Упомянутый атрибут нельзя использовать, потому что вставка в постоянное дерево с такими атрибутами занимает время $\Omega(n)$ (согласно части (г)).
- Что дополнительные изменения узлов, внесенные во время выполнения операций над красно-черным деревом (из-за поворота и перекрашивания), не приводят к изменению более чем $O(\lg n)$ дополнительных узлов.

Ниже объясняется, как найти каждый необходимый во время вставки указатель на родительский узел за время $O(1)$, не имея соответствующего атрибута. Для вставки в красно-черное дерево мы вызываем процедуру RB-Insert, которая в свою очередь вызывает RB-Insert-Fixup. Внесите те же изменения в RB-Insert, которые делались в процедуре Tree-Insert для обеспечения постоянства. Кроме того, когда процедура RB-Insert проходит по дереву, чтобы найти место для вставки нового узла, пусть она построит стек проходимых узлов и передаст этот стек процедуре RB-Insert-Fixup. Процедура RB-Insert-Fixup нуждается в указателях на родительские узлы, чтобы пройти обратно по тому же пути, и в любой момент времени ей нужны такие указатели только для выяснения родителя и прародителя узла, с которым она работает. Поскольку процедура RB-Insert-Fixup перемещается вверх по стеку родителей, ей требуются только указатели на родительские узлы, которые находятся в известных местах на константном расстоянии в стеке. Таким образом, информацию о родителях можно найти за время $O(1)$, как если бы она хранилась в атрибуте с указателем на родительский узел.

Поворот и перекрашивание изменяют узлы следующим образом.

- Процедура RB-Insert-Fixup выполняет не более двух поворотов, и каждый поворот обновляет указатели на дочерние узлы в трех узлах (в узле, относительно которого выполняется поворот, в родительском узле этого узла и в одном из дочерних узлов, относительно которого выполняется поворот). Таким образом, в ходе выполнения RB-Insert-Fixup поворота напрямую модифицируются не более шести узлов. В постоянном дереве все предки изме-

ненного узла копируются, поэтому поворот в RB-Insert-Fixup тратит время $O(\lg n)$ на изменение узлов из-за поворота. (На самом деле в данном случае измененные узлы имеют один общий путь предков длиной $O(\lg n)$.)

- Процедура RB-Insert-Fixup перекрашивает определенных предков вставленного узла, которые в любом случае изменяются при постоянной вставке, и определенных дочерних узлов предков (в описании алгоритма они упоминаются как "дяди"). Имеется $O(\lg n)$ предков, следовательно, $O(\lg n)$ изменений цвета "дядей". Перекрашивание "дядей" не приводит к дополнительным изменениям узлов из-за постоянства, т.к. предками "дядей" являются те же узлы (предки вставленного узла), которые и так изменяются по причине постоянства. Таким образом, перекрашивание не влияет на время работы $O(\lg n)$ даже при поддержке постоянства.

Аналогичным образом можно показать, что удаление в постоянном дереве также занимает время $O(h)$ в наихудшем случае.

- Мы уже видели в части (а), что $O(h)$ узлов изменяются.
- Мы могли бы написать процедуру RB-Delete с постоянством, которая выполняется за время $O(h)$, аналогично изменениям, которые были внесены для поддержки постоянства при вставке. Но чтобы сделать это без использования указателей на дочерние узлы, процедура должна пройти по дереву до самого глубокого изменяемого узла, построив стек родительских узлов, как обсуждалось выше для вставки. Такой проход опирается на различие ключей.

Тогда задача демонстрации того, что удаление занимает всего лишь время $O(\lg n)$ в постоянном красно-черном дереве, оказывается той же самой, как в случае вставки.

- Что касается вставки, то мы можем показать, что родительские узлы, необходимые RB-Delete-Fixup, могут быть найдены за время $O(\lg n)$ (с использованием такого же приема, как для вставки).
- Кроме того, процедура RB-Delete-Fixup выполняет не более трех поворотов, что, как обсуждалось выше для вставки, требует времени $O(\lg n)$ для изменения узлов из-за постоянства. Кроме того, она производит $O(\lg n)$ изменений цветов, что (аналогично вставке) занимает время $O(\lg n)$ для изменения предков из-за постоянства, поскольку количество скопированных узлов составляет $O(\lg n)$.

Решения для главы 14

Решение упражнения 14.2.5

Каждый раз, когда выполняется цикл по l, цикл по i выполняется $n - l + 1$ раз. Каждый раз, когда выполняется цикл по i, цикл по k выполняется $j - i = l - 1$ раз, дважды ссылаясь на элемент m при каждом прогоне. Следовательно, общее количество обращений к элементу m при вычислении других элементов равно $\sum_{l=2}^{n} 2(n-l+1)(l-1)$. Таким образом,

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{n} R(i,j) &= \sum_{l=2}^{n} 2(n-l+1)(l-1) = \\
&= 2\sum_{l=1}^{n-1}(n-l)l = \\
&= 2\sum_{l=1}^{n-1} nl - 2\sum_{l=1}^{n-1} l^2 = \\
&= 2\frac{n(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6} = \\
&= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3} = \\
&= \frac{n^3 - n}{3}.
\end{aligned}$$

Решение упражнения 14.3.1

Запуск процедуры RECURSIVE-MATRIX-CHAIN асимптотически эффективнее, чем перечисление всех способов расстановки скобок в произведении и расчет количества умножений для каждого способа.

Рассмотрим трактовку подзадач при этих двух подходах.

- Для каждого возможного места разделения цепочки матриц при подходе с перечислением ищутся все способы расстановки скобок в левой и правой половинах, а затем просматриваются все возможные комбинации левой и правой половин. Таким образом, объем работы, связанной с просмотром каждой комбинации результатов подзадачи для левой и правой половин, равен произведению числа способов расстановки скобок в левой половине и числа способов расстановки скобок в правой половине.

- Для каждого возможного места разделения цепочки матриц RECURSIVE-MATRIX-CHAIN находит наилучший способ расстановки скобок в левой и правой половинах, а затем объединяет только эти два результата. Таким образом, объем работы по объединению результатов подзадачи для левой и правой половин составляет $O(1)$.

В разделе 14.2 утверждалось, что время работы перечисления равно $\Omega(4^n / n^{3/2})$. Мы покажем, что время работы процедуры RECURSIVE-MATRIX-CHAIN составляет $O(n3^{n-1})$.

Чтобы получить верхнюю оценку времени работы RECURSIVE-MATRIX-CHAIN, мы воспользуемся приемом из раздела 14.2, который применялся для получения нижней оценки: выведем рекуррентное соотношение вида $T(n) \le \ldots$и решим его методом подстановки. Для рекуррентного соотношения с нижней оценкой в книге предполагалось, что выполнение строк 1, 2 и 6, 7 занимает не менее единицы времени. Для рекуррентного соотношения с верхней оценкой мы допускаем, что выполнение этих пар строк занимает не больше константного времени c. Таким образом, мы имеем рекуррентное соотношение

$$T(n) \le \begin{cases} c, & \text{если } n = 1, \\ c + \sum_{k=1}^{n-1} (T(k) + T(n-k) + c), & \text{если } n \ge 2. \end{cases}$$

Оно похоже на рассмотренное в книге рекуррентное соотношение с операцией $\ge$, но только вместо 1 указана константа c, так что мы можем переписать его как

$$T(n) \le 2\sum_{i=1}^{n-1} T(i) + cn.$$

Мы докажем, что $T(n) = O(n3^{n-1})$, используя метод подстановки. (*Примечание*: будет достаточно любой верхней оценки $T(n)$, равной $o(4^n / n^{3/2})$. Вы можете предпочесть доказать ту, которую легче обдумать, скажем, $T(n) = O(3.5^n)$.) В частности, мы покажем, что $T(n) \le cn3^{n-1}$ для всех $n \ge 1$.

Основание простое, т.к. $T(1) \le c = c \cdot 1 \cdot 3^{1-1}$. Согласно индуктивному предположению для $n \le 2$ мы имеем

$$
\begin{aligned}
T(n) &\le 2\sum_{i=1}^{n-1} T(i) + cn \le \\
&\le 2\sum_{i=1}^{n-1} ci3^{i-1} + cn = \\
&= c\cdot\left(2\sum_{i=1}^{n-1} i3^{i-1} + n\right) = \\
&= \left(2\cdot\left(\frac{n3^{n-1}}{3-1} + \frac{1-3^n}{(3-1)^2}\right) + n\right) = \quad \text{(см. ниже)} \\
&= cn3^{n-1} + c\cdot\left(\frac{1-3^n}{2} + n\right) = \\
&= cn3^{n-1} + \frac{c}{2}(2n+1-3^n) \le \\
&\le cn3^{n-1} \quad \text{для всех } c>0, n\ge 1.
\end{aligned}
$$

Выполнение процедуры RECURSIVE-MATRIX-CHAIN занимает время $O(n3^{n-1})$, а перечисление всех способов расстановки скобок — время $\Omega(4^n / n^{3/2})$, поэтому процедура RECURSIVE-MATRIX-CHAIN эффективнее перечисления. *Примечание*: в приведенной выше подстановке задействован следующий факт:

$$\sum_{i=1}^{n-1} ix^{i-1} = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2}.$$

Это уравнение можно вывести из уравнения (A.6) путем взятия производной. Пусть

$$f(x) = \sum_{i=1}^{n-1} x^i = \frac{x^n-1}{x-1} - 1.$$

Тогда

$$\sum_{i=1}^{n-1} ix^{i-1} = f'(x) = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2}.$$

Решение упражнения 14.4.4

При вычислении конкретной строки таблицы c строки перед предыдущей строкой не нужны. Таким образом, в памяти одновременно должно храниться только две строки, т.е. $2n$ элементов. (*Примечание*: каждая строка таблицы c фактически содержит $n+1$ элементов, но хранить столбец нулей не требуется — взамен мы можем снабдить

программу знанием о том, что данные элементы равны 0.) За счет применения такой идеи нам понадобится всего лишь $2 \cdot \min\{m, n\}$ элементов, если мы всегда вызываем процедуру LCS-Length с более короткой последовательностью в качестве аргумента *Y*.

Таким образом, мы можем избавиться от таблицы *c* так, как описано ниже.

- Использовать для хранения соответствующих строк таблицы *c* два массива длиной $\min\{m, n\}$ — *previous-row* и *current-row*.
- Инициализировать *previous-row* всеми нулями и вычислять *current-row* слева направо.
- Если после заполнения *current-row* все еще есть строки, подлежащие вычислению, тогда скопировать *current-row* в *previous-row* и вычислить новый массив *current-row*.

На самом деле для вычисления требуется лишь немногим больше, чем одна строка элементов таблицы *c*, а именно — $\min\{m, n\} + 1$ элементов. Когда наступает время вычисления $c[i, j]$, в таблице необходимы только элементы $c[i, k]$ для $k \le j - 1$ (т.е. предшествующие элементы в текущей строке, которые потребуются для вычисления следующей строки), и $c[i - 1, k]$ для $k \ge j - 1$ (т.е. элементы в предыдущей строке, которые все еще нужны для вычисления оставшейся части текущей строки). Речь идет об одном элементе для каждого k от 1 до $\min\{m, n\}$ за исключением того, что имеются два элемента с $k = j - 1$, поэтому необходим дополнительный элемент помимо элементов, составляющих одну строку.

Далее объясняется, как можно избавиться от таблицы *c*.

- Использовать массив *a* длиной $\min\{m, n\} + 1$ для хранения соответствующих элементов *c*. На момент вычисления $c[i, j]$ массив *a* содержит следующие элементы:
  - $a[k] = c[i, k]$ для $1 \le k < j - 1$ (т.е. предшествующие элементы в текущей “строке”);
  - $a[k] = c[i - 1, k]$ для $k \ge j - 1$ (т.е. элементы в предыдущей “строке”);
  - $a[0] = c[i, j - 1]$ (т.е. предыдущий вычисленный элемент, который не удалось поместить в “правильное” место внутри массива *a*, не удалив по-прежнему необходимый элемент $c[i - 1, j - 1]$).

- Инициализировать массив a всеми нулями и вычислять элементы слева направо.
  - Обратите внимание, что три значения, необходимые при вычислении $c[i, j]$ для $j > 1$, находятся в $a[0] = c[i, j - 1]$, $a[j - 1] = c[i - 1, j - 1]$ и $a[j] = c[i - 1, j]$.
  - После вычисления $c[i, j]$ элемент $a[0]$ ($c[i, j - 1]$) необходимо переместить в его "правильное" место, т.е. $a[j - 1]$, а в $a[0]$ поместить $c[i, j]$.

Решение задачи 14.4

Начнем с определения некоторых величин, чтобы можно было единообразно сформулировать задачу. В этих определениях будут учтены особые случаи, связанные с последней строкой, а также момент, касающийся того, помещается ли последовательность слов в строку, чтобы можно было забыть о них при выработке общей стратегии.

- Определим $extras[i, j] = M - j + i - \sum_{k=i}^{j} l_k$ как количество дополнительных пробелов в конце строки, содержащей слова с i по j. Обратите внимание, что величина *extras* может быть отрицательной.
- Далее определим затраты на включение строки, содержащей слова с i по j, в сумме, которую мы хотим минимизировать:

$$lc[i, j] = \begin{cases} \infty, & \text{если } extras[i\ j] < 0 \text{ (т.е. слова } i,\ldots, j \text{ не умещаются)}, \\ 0, & \text{если } j = n \text{ и } extras[i, j] \geq 0 \text{ (затраты на последнюю строку равны 0)}, \\ (extras[i, j])^3, & \text{в противном случае.} \end{cases}$$

  Делая затраты на строку бесконечными, когда слова в нее не умещаются, мы не позволяем такому расположению стать частью минимальной суммы, а делая затраты на последнюю строку нулевыми (если слова умещаются), мы предотвращаем влияние расположения последней строки на минимизируемую сумму.

Мы хотим минимизировать сумму *lc* по всем строкам абзаца.

Наши подзадачи будут касаться того, как оптимально расположить слова $1, \ldots, j$, где j изменяется от 1 до n.

Рассмотрим оптимальное расположение слов $1, \ldots, j$. Предположим, нам известно, что последняя строка, которая заканчивается словом j, начинается со слова i. Следовательно, предшествующие строки содержат слова $1, \ldots, i - 1$. Фактически они должны содержать оптимальное расположение слов $1, \ldots, i - 1$. (Применяется обычный метод копирования и замены.)

Пусть $c[j]$ – затраты на оптимальное расположение слов $1, \ldots, j$. Если известно, что последняя строка содержит слова $i, \ldots, j$, то $c[j] = c[i - 1] + lc[i, j]$. В базовом случае, когда вычисляется $c[1]$, требуется $c[0]$. Если установить $c[0] = 0$, то $c[1] = lc[1, 1]$, что и было желательно.

Но, разумеется, необходимо выяснить, с какого слова начинается последняя строка для подзадачи со словами $1, \ldots, j$, так что опробуются все варианты для слова i и выбирается тот, который дает наименьшие затраты. Здесь i варьируется от 1 до j. Таким образом, мы можем определить $c[j]$ рекурсивно:

$$c[j] = \begin{cases} 0, & \text{если } j = 0, \\ \min\{c[i-1] + lc[i, j] : 1 \le i \le j\}, & \text{если } j > 0. \end{cases}$$

Обратите внимание, что способ определения lc гарантирует следующие моменты:

- все выбранные слова уместятся в строку (поскольку расположение с $lc = \infty$ не может быть выбрано в качестве минимума);
- затраты на размещение слов $i, \ldots, j$ в последней строке не могут быть равны 0, если только это действительно не последняя строка абзаца ($j = n$) или слова $i, \ldots, j$ не заполняют всю строку.

Мы можем вычислить таблицу значений c слева направо, потому что каждое значение зависит только от предшествующих значений.

Чтобы отслеживать, какие слова в каких строках находятся, мы можем вести параллельную таблицу p, которая указывает, откуда взялось каждое значение c. Если при вычислении значения $c[j]$ обнаруживается, что оно основано на значении $c[k - 1]$, тогда мы устанавливаем $p[j] = k$. Затем после вычисления $c[n]$ мы можем отслеживать указатели, чтобы увидеть, где следует разрывать строки. Последняя строка начинается со слова $p[n]$ и проходит через слово n. Предыдущая строка начинается со слова $p[p[n]]$ и проходит через слово $p[n] - 1$ и т.д.

В псевдокоде таблицы конструируются следующим образом:

```
PRINT-NEATLY(l, n, M)
 Пусть extras[1 : n, 1 : n], lc[1 : n, 1 : n], c[0 : n] и p[1 : n]
 будут новыми таблицами
 // Вычислить extras[i, j] для 1 ≤ i ≤ j ≤ n
 for i = 1 to n
     extras[i, i] = M – l_i
```

```
    for j = i + 1 to n
        extras[i, j] = extras[i, j − 1] − l_j − 1
// Вычислить lc[i, j] для 1 ≤ i ≤ j ≤ n
for i = 1 to n
    for j = i to n
        if extras[i, j] < 0
            lc[i, j] = ∞
        elseif j == n и extras[i, j] ≥ 0
            lc[i, j] = 0
        else lc[i, j] = (extras[i, j])^3
// Вычислить c[j] для 0 ≤ j ≤ n и p[j] для 1 ≤ j ≤ n
c[0] = 0
for j = i to n
    c[j] = ∞
    for i = 1 to j
        if c[i − 1] + lc[i, j] < c[j]
            c[j] = c[i − 1] + lc[i, j]
            p[j] = i
return c и p
```

Совершенно очевидно, что и время, и объем памяти составляют $\Theta(n^2)$.

На самом деле мы можем добиться немного большего: сократить и время, и объем памяти до $\Theta(nM)$. Ключевое наблюдение заключается в том, что в строку может уместиться не более $\lceil M/2 \rceil$ слов. (Каждое слово имеет длину не менее одного символа, и между словами присутствует пробел.) Поскольку строка со словами $i, \ldots, j$ содержит $j - i + 1$ слов, если $j - i + 1 > \lceil M/2 \rceil$, то известно, что $lc[i, j] = \infty$. Нам нужно вычислить и сохранить только $extras[i, j]$ и $lc[i, j]$ для $j - i + 1 \le \lceil M/2 \rceil$. Заголовок внутреннего цикла **for** при вычислении $c[j]$ и $p[j]$ может выполняться от $\max\{1, j - \lceil M/2 \rceil + 1\}$ до j.

Мы можем сократить объем потребляемой памяти еще больше — до $\Theta(n)$. Для этого таблицы lc и $extras$ не ведутся, а взамен значение $lc[i, j]$ вычисляется по мере необходимости в последнем цикле. Идея в том, что $lc[i, j]$ можно было бы вычислить за время $O(1)$, если знать значение $extras[i, j]$. И если мы будем искать минимальное значение в порядке убывания i, то можем вычислить его как $extras[i, j] = extras[i + 1, j] - l_i - 1$. (Изначально $extras[i, j] = M - l_j$.) Такое усовершенствование сокращает объем потребляемой памяти до $\Theta(n)$, потому что теперь хранятся только таблицы c и p.

Ниже демонстрируется вывод результатов. Вызов процедуры PRINT-LINES(p, j) выводит все слова, начиная с 1-го и заканчивая j-м.

```
PRINT-LINES(p, j)
 if j > 0
     i = p[j]
     PRINT-LINES(p, i – 1)
     Вывести строку, содержащую слова с i по j, с одним
           пробелом между каждой парой слов
```

Начальный вызов выглядит как PRINT-LINES(p, n). Учитывая, что с каждым рекурсивным вызовом значение j уменьшается, процедура PRINT-LINES затрачивает на вывод всех n слов в общей сложности время $O(n + k)$, где k — суммарная длина всех слов. (Обратите внимание, что поскольку каждое слово содержит хотя бы один символ, даже если считать пробелы и переводы строки печатаемыми символами, то общее количество печатаемых символов не превышает $2k$.)

Решения для главы 15

Решение упражнения 15.1.4

Пусть S — множество из n действий.

"Очевидное" решение предусматривает применение процедуры GREEDY-ACTIVITY-SELECTOR для поиска множества максимального размера S_1 совместимых действий из S, которые будут проводиться в первом лекционном зале, затем ее использование для поиска множества максимального размера S_2 совместимых действий из $S - S_1$, которые будут проводиться во втором лекционном зале (и т.д. пока не будут распределены все действия). Такое решение требует времени $\Theta(n^2)$ в наихудшем случае. Более того, оно может привести к результату, при котором используется больше лекционных залов, чем необходимо. Рассмотрим действия с интервалами $\{[1, 4), [2, 5), [6, 7), [4, 8)\}$. Процедура GREEDY-ACTIVITY-SELECTOR выбрала бы действия с интервалами [1, 4) и [6, 7) для первого лекционного зала, а затем каждое действие с интервалами [2, 5) и [4, 8) пришлось бы разместить в отдельном лекционном зале, что в итоге дало бы три используемых зала. Оптимальным решением было бы размещение действий с интервалами [1, 4) и [4, 8) в одном зале, а действий с интервалами [2, 5) и [6, 7) — в другом зале, что привело бы к использованию всего двух залов.

Однако существует корректный алгоритм, асимптотическое время которого равно времени, необходимого для сортировки действий по значениям времени — $O(n \lg n)$ для произвольных значений времени или, возможно, $O(n)$, если значения времени представляют собой небольшие целые числа.

Общая идея заключается в том, чтобы перебирать действия в порядке времени их начала, распределяя каждое в любой доступный в это время зал. Для этого нужно пройти по множеству событий, состоящему из моментов начала и окончания действий, в порядке времен событий. Понадобится вести два списка лекционных залов: залы, занятые в текущий момент времени t (потому что им назначено занятие i, которое началось в $s_i \le t$, но не завершится, пока $f_i > t$), и залы, свободные в момент времени t. (Как и в задаче о выборе действий в разделе 15.1, мы предполагаем, что интервалы времени действий полуоткрыты, т.е. если $s_i \ge f_j$, то действия i и j совместимы.) Если t — время начала некоторого действия, тогда необходимо распределить это действие в свободный зал и переместить зал из списка свободных в список занятых. Если t — время окончания некоторого действия, тогда нужно переместить зал действия из списка занятых в список свободных. (Действие определенно находится в каком-то зале, потому что значения времени событий обрабатываются по порядку, и действие должно было начаться до своего времени окончания t, следовательно, должно было быть распределено в зал.)

Чтобы избежать использования большего количества лекционных залов, чем необходимо, по возможности всегда следует выбирать зал, которому уже назначено действие, прежде чем выбирать неиспользуемый зал. (Это можно сделать, всегда работая с началом списка свободных лекционных залов — помещая освобожденные залы в начало списка и извлекая залы из начала списка — так, чтобы новый зал не оказался впереди и не был выбран, если есть ранее занятые залы.) Такой подход гарантирует использование алгоритмом как можно меньшего количества лекционных залов: алгоритм завершит работу с расписанием, требующим $m \le n$ залов. Пусть действие i будет первым действием, запланированным в зале m. Причина, по которой действие i было размещено в m-м лекционном зале, связана с тем, что первые $m-1$ лекционных залов в момент времени s_i были заняты. Таким образом, в этот период одновременно происходят m действий. Следовательно, любое расписание должно использовать не менее m S_2, так что расписание, возвращаемое алгоритмом, является оптимальным.

Время работы.

- Сортировка $2n$ событий начала и конца действий. (В отсортированном порядке событие "конец действия" должно предшествовать событию "начало действия", которое происходит в то же время.) Время работы составляет $O(n \lg n)$ для произвольных значений времени и возможно $O(n)$, если время ограничено (например, небольшими целыми числами).
- Обработка событий за время $O(n)$: сканирование $2n$ событий, выполнение $O(1)$ работы для каждого (перемещение зала из одного списка в другой и, возможно, связывание с ним действия).

Общее время работы: $O(n$ + время на сортировку$)$.

Решение упражнения 15.2.2

Решение основано на наблюдении за оптимальной подструктурой. Пусть i будет предметом с наибольшим номером в оптимальном решении S для W фунтов и предметов $1, \ldots, n$. Тогда $S' = S - \{i\}$ должно быть оптимальным решением для $W - w_i$ фунтов и предметов $1, \ldots, i-1$, а значение решения S равно v_i плюс значение решения подзадачи S'.

Мы можем выразить это отношение следующей формулой: определим $c[i, w]$ как значение решения для предметов $1, \ldots, i$ и максимального веса w. Мы имеем

$$c[i,w] = \begin{cases} 0, & \text{если } i = 0 \text{ или } w = 0, \\ c[i-1,w], & \text{если } w_i > w, \\ \max\{v_i + c[i-1, w-w_i], c[i-1,w]\}, & \text{если } i > 0 \text{ и } w \geq w_i. \end{cases}$$

В последнем случае видно, что значение решения для i предметов либо включает предмет i и тогда оно равно v_i плюс значение решения подзадачи для $i-1$ предметов и веса без учета w_i, либо не включает предмет i и тогда оно равно значению решения подзадачи для $i-1$ предметов и того же самого веса. То есть если вор выбирает предмет i, то добавляется значение v_i, и вор может выбрать из предметов $1, \ldots, i-1$ до достижения предельного веса $w - w_i$, получая дополнительную ценность $c[i-1, w-w_i]$. С другой стороны, если вор решает не брать предмет i, тогда остается выбор из предметов $1, \ldots, i-1$ до достижения предельного веса w, давая ценность $c[i-1, w]$. Потребуется выбрать лучший из этих двух вариантов.

Алгоритм принимает в качестве входных данных максимальный вес W, количество предметов n и две последовательности $v = \langle v_1, v_2, \ldots, v_n \rangle$ и $w = \langle w_1, w_2, \ldots, w_n \rangle$. Он сохраняет значения $c[i, j]$ в таблице $c[0 : n, 0 : W]$, элементы которой вычисляются в порядке по строкам. (То есть сначала слева направо заполняется первая строка таблицы c, затем вторая строка и т.д.) В конце вычисления $c[n, W]$ содержит максимальную ценность предметов, которые может унести вор.

```
DYNAMIC-0-1-KNAPSACK(v, w, n, W)
 Пусть c[0 : n, 0 : W] будет новым массивом
 for w = 0 to W
     c[0, w] = 0
 for i = 1 to n
     c[i, 0] = 0
     for w = 1 to W
             if w_i ≤ w и v_i + c[i − 1, w − w_i] > c[i − 1, w]
                     c[i, w] = v_i + c[i − 1, w − w_i]
             else c[i, w] = c[i − 1, w]
```

Таблицу c можно использовать для выведения множества элементов, начиная с $c[n, W]$ и отслеживая, откуда поступили оптимальные значения. Если $c[i, w] = c[i - 1, w]$, то элемент i не является частью решения, и мы продолжаем отслеживание с $c[i - 1, w]$. В противном случае элемент i является частью решения, и мы продолжаем отслеживание с $c[i - 1, w - w_i]$.

Общее время работы приведенной выше процедуры составляет $\Theta(nW)$:

- время $\Theta(nW)$ на заполнение таблицы c, содержащей $(n + 1) \cdot (W + 1)$ элементов, вычисление каждого из которых занимает время $\Theta(1)$;
- время $O(n)$ на отслеживание решения (т.к. оно начинается со строки n таблицы и перемещается вверх на одну строку на каждом шаге).

Решение упражнения 15.2.7

Отсортируйте множества A и B в порядке монотонного убывания.

Вот доказательство того, что этот метод дает оптимальное решение. Рассмотрим любые индексы i и j, такие что $i < j$, и рассмотрим члены $a_i^{b_i}$ и $a_j^{b_j}$. Мы хотим показать, что включение этих членов в максимизируемую величину не хуже, чем включение $a_i^{b_j}$ и $a_j^{b_i}$, т.е. что $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$.

Поскольку множества A и B отсортированы в порядке монотонного убывания и $i < j$, мы имеем $a_i \geq a_j$ и $b_i \geq b_j$. Так как элементы a_i и a_j положительны, а величина $b_i - b_j$ неотрицательна, справедливо неравенство $a_i^{b_i - b_j} \geq a_j^{b_i - b_j}$. Умножение обеих частей на $a_i^{b_j} a_j^{b_j}$ дает $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$.

Из-за того, что порядок умножения не имеет значения, сортировка A и B в монотонно возрастающем порядке тоже работает.

Решения для главы 16

Решение упражнения 16.1.3

Пусть c_i — затраты, приходящиеся на i-ю операцию:

$$c_i = \begin{cases} i, \text{ если } i \text{ является точной степенью 2,} \\ 1, \text{ в противном случае.} \end{cases}$$

Операция	Затраты
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
10	1
. . .	. . .

Затраты на n операций составляют

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lg n} 2^j = n + (2n - 1) < 3n.$$

(*Примечание*: округление до ближайшего меньшего целого в верхней границе суммы $\sum 2^j$ игнорируется.)

Средние затраты на операцию составляют

(*Общие затраты*/*Количество операций*) < 3.

По данным группового анализа амортизированные затраты, приходящиеся на одну операцию, составляют $O(1)$.

Решение упражнения 16.2.2

Пусть c_i — затраты, приходящиеся на i-ю операцию:

$$c_i = \begin{cases} i, \text{ если } i \text{ является точной степенью 2,} \\ 1, \text{ в противном случае.} \end{cases}$$

С каждой операции взимается 3 долл. (амортизированные затраты $\hat{c}_i$).

- Если i не является степенью 2, то 1 долл. оплачивается и 2 долл. сохраняется в виде кредита.
- Если i является степенью 2, то i долл. оплачивается с использованием сохраненного кредита.

Операция	Амортизированные затраты	Фактические затраты	Оставшийся кредит
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9
. . .	. . .	. . .	. . .

Так как амортизированные затраты составляют 3 долл. за операцию, то $\sum_{i=1}^{n} \hat{c}_i = 3n$.

Из упражнения 16.1.3 известно, что $\sum_{i=1}^{n} c_i < 3n$.

Тогда мы имеем $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \Rightarrow$ кредит = амортизированные затраты — фактические затраты ≥ 0.

Поскольку амортизированные затраты, приходящиеся на каждую операцию, равны $O(1)$, а сумма кредита никогда не становится отрицательной, то общие затраты на n операций составляют $O(n)$.

Решение упражнения 16.2.3

Мы вводим новое поле $A.max$ для хранения индекса старшей единицы в A. Изначально поле $A.max$ установлено в –1, т.к. младший бит A находится по индексу 0, а единицы в A первоначально отсутствуют. Значение $A.max$ обновляется по мере необходимости при увеличении или сбросе счетчика, и это значение ограничивает объем битового массива A, который придется просмотреть для его сброса. Управляя затратами RESET подобным образом, мы можем ограничить их суммой, которая может быть покрыта кредитом от предшествующих операций INCREMENT.

```
INCREMENT(A, k)
 i = 0
 while i < k и A[i] == 1
     A[i] = 0
     i = i + 1
 if i < k
     A[i] = 1
     //Здесь начинаются добавления к процедуре INCREMENT из книги
     A.max = max{A.max, i}
 else A.max = –1

RESET(A)
 for i = 0 to A.max
     A[i] = 0
 A.max = –1
```

Что касается счетчика, приведенного в книге, то мы предполагаем, что затраты на переключение бита составляют 1 долл. Вдобавок мы предполагаем, что затраты на обновление $A.max$ тоже составляют 1 долл.

Установка и сброс битов с помощью INCREMENT будет работать точно так же, как для исходного счетчика, рассмотренного в книге: 1 долл. оплачивается за установку одного бита в 1, 1 долл. размещается в виде кредита для бита, установленного в 1, а кредит для каждого единичного бита идет на оплату сброса бита при инкрементировании.

Кроме того, 1 долл. оплачивается за обновление поля *max*, и если *max* увеличивается, тогда для новой старшей единицы добавляется дополнительная сумма 1 долл. (Если *max* не увеличивается, тогда можно просто потратить эту сумму 1 долл., т.к. она не понадобится.) Поскольку процедура RESET манипулирует битами только в позициях до $A.max$ и каждый бит до $A.max$ должен был стать старшей единицей в какой-то момент перед тем, как старшая единица достигла $A.max$, каждый бит, видимый процедуре RESET, имеет кредит 1 долл. Следовательно, обнуление битов в A с помощью RESET может быть полностью оплачено кредитом, хранящимся для битов. Нужна всего лишь сумма 1 долл. для оплаты сброса *max*.

Таким образом, взимания 4 долл. за каждую операцию INCREMENT и 1 долл. за каждую операцию RESET вполне достаточно, поэтому выполнение последовательности из n операций INCREMENT и RESET требует времени $O(n)$.

Решения для главы 17

Решение упражнения 17.1.7

Пусть $A[1 : n]$ — массив из n различных чисел.

Один из способов подсчета количества инверсий предусматривает нахождение для каждого элемента суммы количеств более крупных элементов, которые предшествуют ему в массиве:

$$\textit{Количество инверсий} = \sum_{j=1}^{n} |Inv(j)|,$$

где $Inv(j) = \{i : i < j \text{ и } A[i] > A[j]\}$.

Обратите внимание, что $|Inv(j)|$ относится к рангу $A[j]$ в подмассиве $A[1 : j]$, т.к. элементы в $Inv(j)$ являются причиной того, что $A[j]$ не позиционируется в соответствии со своим рангом. Пусть $r(j)$ — ранг $A[j]$ в $A[1 : j]$. Тогда $j = r(j) + |Inv(j)|$, так что вычислить

$$|Inv(j)| = j - r(j)$$

можно, вставив $A[1], \ldots, A[n]$ в дерево порядковой статистики и применив процедуру OS-RANK для нахождения ранга каждого элемента $A[j]$ в дереве сразу после его вставки. (Значение, возвращаемое OS-RANK, равно $r(j)$.)

Операции вставки и OS-RANK занимают время $O(\lg n)$ каждая, поэтому общее время для n элементов составляет $O(n \lg n)$.

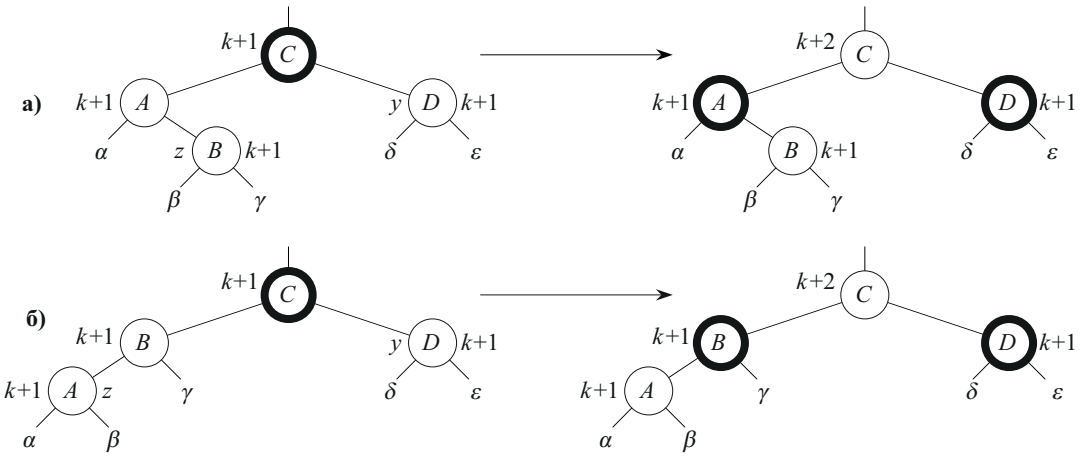
Решение упражнения 17.2.2

Да, черные высоты можно поддерживать в виде атрибутов в узлах красно-черного дерева, не влияя на асимптотическую производительность операций с красно-черным деревом. Мы отсылаем к теореме 17.1, т.к. черную высоту узла можно вычислить, используя информацию об узле и двух его дочерних узлах. Фактически черную высоту можно вычислить с применением информации только об одном дочернем узле: черная высота узла равна черной высоте красного дочернего узла или черной высоте черного дочернего узла плюс один. Второй дочерний узел не требует проверки благодаря свойству 5 красно-черных деревьев.

Процедуры RB-Insert-Fixup и RB-Delete-Fixup изменяют цвета узлов, и каждое изменение цвета потенциально может привести к $O(\lg n)$ изменениям черных высот. Мы покажем, что изменения цветов в процедурах исправления вызывают только локальные изменения черных высот и, следовательно, являются операциями с постоянным временем работы. Предположим, что черная высота каждого узла x хранится в атрибуте $x.bh$.

Для процедуры RB-Insert-Fixup необходимо рассмотреть три случая.

Случай 1: дядя узла z является красным узлом.

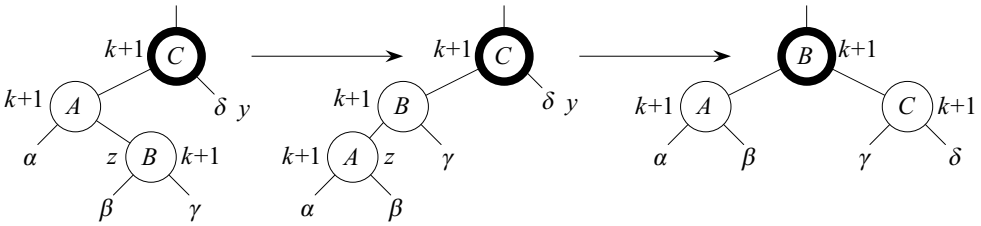


- Перед изменением цвета предположим, что все поддеревья α, β, γ, δ, ε имеют одну и ту же черную высоту k с черным корнем, так что узлы A, B, C и D имеют черные высоты $k + 1$.
- После изменения цвета единственным узлом, чья черная высота изменилась, будет узел C. Чтобы исправить это, понадобится добавить $z.p.p.bh = z.p.p.bh + 1$ после строк 7 и 21 в процедуре RB-Insert-Fixup.

- Поскольку количество черных узлов между $z.p.p$ и z остается неизменным, узлы выше $z.p.p$ изменением цвета не затрагиваются.

Случай 2: дядя y узла z является черным узлом, а z — правым дочерним узлом.

Случай 3: дядя y узла z' является черным узлом, а z — левым дочерним узлом.

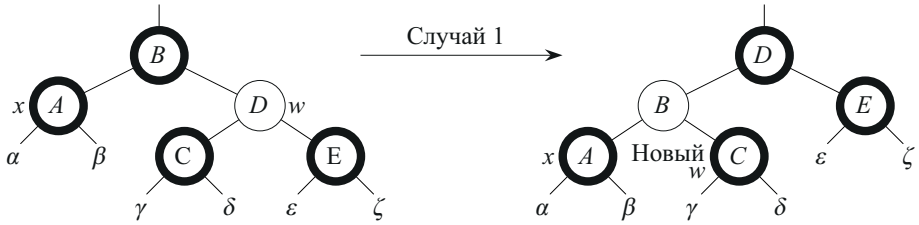


- Для поддеревьев α, β, γ, δ, ε с черной высотой k даже при изменениях цветов и поворотах черные высоты узлов A, B и C остаются теми же самыми, т.е. $(k+1)$.

Таким образом, процедура RB-Insert-Fixup сохраняет исходное время работы $O(\lg n)$.

Для процедуры RB-Delete-Fixup потребуется рассмотреть четыре случая.

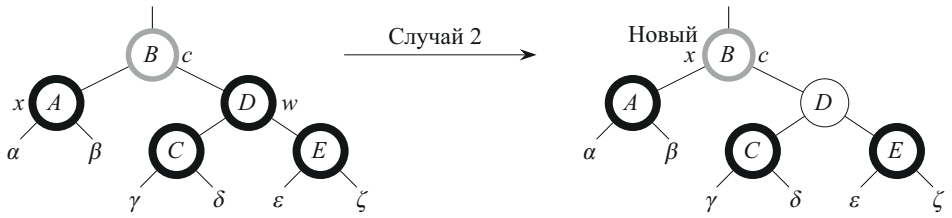
Случай 1: родственный узел w узла x является красным.



- Несмотря на то что в случае 1 изменяются цвета узлов и выполняется поворот, черные высоты не изменяются.
- В случае 1 изменяется структура дерева, но ожидаются случаи 2, 3 и 4, чтобы справиться с "добавочным" черным цветом узла x.

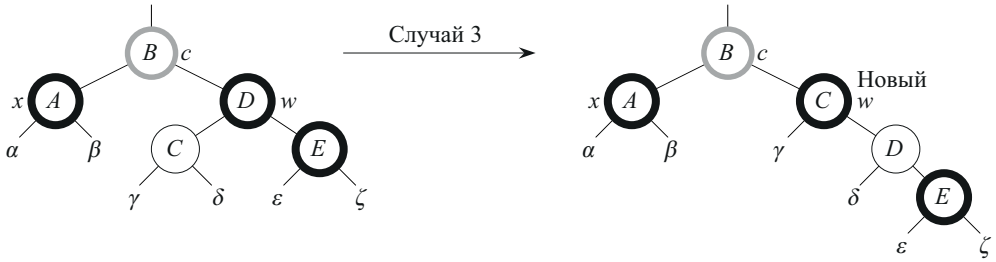
Случай 2: родственный узел w узла x является черным, а оба дочерних узла w — черными.

- Узел w окрашен в красный цвет, а "добавочный" черный цвет узла x перемещается в $x.p$.



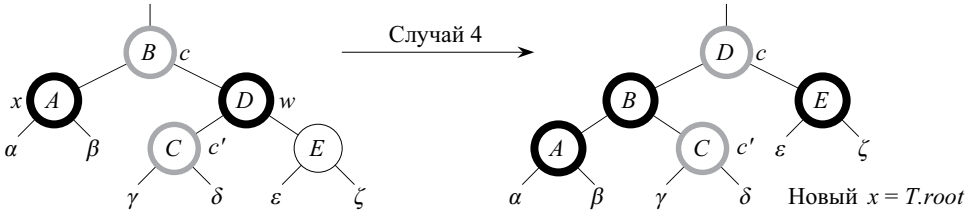
- Нужно добавить $x.p.bh = x.bh$ после строк 10 и 31 в процедуре RB-DELETE-FIXUP.
- Это обновление выполняется за постоянное время. Затем цикл продолжается, чтобы справиться с "добавочным" черным цветом узла $x.p$.

Случай 3: родственный узел w узла x является черным, левый дочерний узел узла w — красным, а правый дочерний узел узла w — черным.



- Независимо от изменений цветов и поворота в данном случае, черные высоты не меняются.
- В случае 3 просто настраивается структура дерева, чтобы можно было корректно перейти к случаю 4.

Случай 4: родственный узел w узла x является черным, а правый дочерний узел узла w — красным.



- Узлы A, C и E сохраняют те же самые поддеревья, поэтому их черные высоты не меняются.
- Необходимо добавить следующие две строки с постоянным временем работы после строк 21 и 42 в процедуре RB-DELETE-FIXUP:

$$x.p.bh = x.bh + 1$$
$$x.p.p.bh = x.p.bh + 1$$

- "Добавочный" черный цвет учтен и цикл завершается.

Таким образом, процедура RB-DELETE-FIXUP сохраняет свое исходное время работы $O(\lg n)$.

Следовательно, мы приходим к выводу, что черные высоты узлов можно поддерживать в виде атрибутов в красно-черных деревьях, не влияя на асимптотическую производительность операций с красно-черными деревьями.

Что касается второй части вопроса, то нет, мы не можем поддерживать глубину узлов, не влияя на асимптотическую производительность операций с красно-черными деревьями. Глубина узла зависит от глубины его родительского узла. При изменении глубины узла глубины всех узлов, расположенных ниже него в дереве, должны быть обновлены. Обновление корневого узла приводит к обновлению $n-1$ других узлов, откуда следует, что операции в дереве, изменяющие глубину узлов, могут не выполняться за время $O(n \lg n)$.

Решение упражнения 17.3.6

Общая идея.

Перемещаем строку развертки слева направо, сохраняя при этом набор прямоугольников, пересекаемых ею в данный момент, в дереве отрезков. Дерево отрезков организует все прямоугольники, отрезок x которых включает текущее положение строки развертки, и будет основано на отрезках y прямоугольников, так что любые перекрывающиеся отрезки y в дереве отрезков соответствуют перекрывающимся прямоугольникам.

Детали.

1. Сортируем прямоугольники по их координатам x. (На самом деле каждый прямоугольник должен встречаться в отсортированном списке дважды — один раз для своей левой координаты x и один раз для своей правой координаты x.)
2. Просматриваем отсортированный список (от наименьшей к наибольшей координате x).
   - При нахождении координаты x левого края проверяем, перекрывает ли отрезок координаты y прямоугольника отрезок в дереве, и вставляем прямоугольник (с указанием его отрезка координаты y) в дерево.
   - При нахождении координаты x правого края удаляем прямоугольник из дерева отрезков.

Дерево отрезков всегда содержит множество "открытых" прямоугольников, пересекаемых строкой развертки. Если в дереве отрезков обнаруживается перекрытие, то это означает наличие перекрывающихся прямоугольников.

Время работы: $O(n \lg n)$

- $O(n \lg n)$ для сортировки прямоугольников (с использованием сортировки слиянием или пирамидальной сортировки).
- $O(n \lg n)$ для операций с деревом отрезков (вставка, удаление и проверка на перекрытие).

Решения для главы 19

Решение упражнения 19.2.3

Мы хотим показать, как назначить операциям MAKE-SET и FIND-SET затраты $O(1)$ и операции UNION затраты $O(\lg n)$, чтобы затраты на выполнение последовательности этих операций были достаточными для покрытия затрат $O(m + n \lg n)$, обоснованных в теореме. Говоря о затратах на каждый вид операций, полезно также иметь возможность рассуждать о количестве операций каждого вида.

Рассмотрим обычную последовательность из m операций MAKE-SET, UNION и FIND-SET, n из которых являются операциями MAKE-SET, и пусть $u < n$ будет количеством операций UNION. (Вспомните обсуждение в разделе 19.1 о том, что существует не более $n-1$ операций UNION.) Тогда имеется n операций MAKE-SET, u операций UNION и $m - n - u$ операций FIND-SET.

В теореме не указано отдельно количество u операций UNION; взамен это количество ограничивается числом n. Если провести доказательство теоремы с u операциями UNION, то получим оценку времени работы для последовательности операций $O(m - u + u \lg u) = O(m + u \lg u)$. То есть фактическое время, в течение которого выполняется последовательность операций, не превышает $c(m + u \lg u)$ для некоторой константы c.

Таким образом, мы хотим назначить затраты на операции так, чтобы

$$\begin{aligned} &(\text{затраты на MAKE-SET}) \cdot n \\ +\ &(\text{затраты на FIND-SET}) \cdot (m - n - u) \\ +\ &(\text{затраты на UNION}) \cdot u \\ \hline \geq\ & c(m + u \lg u), \end{aligned}$$

так что амортизированные затраты дают верхнюю оценку фактических затрат.

Подходят следующие назначения затрат, где $c' \geq c$ — некоторая константа:

- MAKE-SET: c'
- FIND-SET: c'
- UNION: $c'(\lg n + 1)$

Подстановка в представленную выше сумму дает

$$\begin{aligned} c'n + c'(m - n - u) + c'(\lg n + 1)u &= c'm + c'u \lg n = \\ &= c'(m + u \lg n) > \\ &> c(m + u \lg u). \end{aligned}$$

Решение упражнения 19.2.6

Назовем два списка A и B и предположим, что представитель нового списка будет представителем A. Вместо того чтобы добавлять B в конец A, поместим B в A сразу после первого элемента A. В любом случае нам придется пройти по B для обновления указателей на объект множества, поэтому мы можем просто сделать так, чтобы последний элемент B указывал на второй элемент A.

Решения для главы 20

Решение упражнения 20.1.7

$$BB^{\mathrm{T}}(i, j) = \sum_{e \in E} b_{ie} b_{ej}^{\mathrm{T}} = \sum_{e \in E} b_{ie} b_{je}.$$

- Если $i = j$, тогда $b_{ie}b_{je} = 1$ (результат $1 \cdot 1$ или $(-1) \cdot (-1)$) всякий раз, когда e входит или покидает вершину i, и 0 в противном случае.
- Если $i \neq j$, тогда $b_{ie}b_{je} = -1$, когда $e = (i, j)$ или $e = (j, i)$, и 0 в противном случае.

Таким образом,

$$BB^{\mathrm{T}}(i, j) = \begin{cases} \text{полустепень захода } i + \text{полустепень исхода } i, & \text{если } i = j, \\ -(\text{количество ребер, соединяющих } i \text{ и } j), & \text{если } i \neq j. \end{cases}$$

Решение упражнения 20.2.5

Доказательство корректности процедуры BFS показывает, что $u.d = \delta(s, u)$, и алгоритм не предполагает, что списки смежности находятся в каком-то определенном порядке.

На рис. 20.3, если t предшествует x в $Adj[w]$, то мы можем получить дерево поиска в ширину, показанное на рисунке. Но если x предшествует t в $Adj[w]$, а u предшествует y в $Adj[x]$, тогда мы можем получить ребро (x, u) в дереве поиска в ширину.

Решение упражнения 20.3.12

Ниже приведен псевдокод модифицированных процедур DFS и DFS-Visit, в котором атрибутам *cc* вершин присваиваются значения.

```
DFS(G)
 for каждая вершина u ∈ G.V
     u.color = WHITE
     u.n = NIL
 time = 0
 counter = 0
 for каждая вершина u ∈ G.V
     if u.color == WHITE
             counter = counter + 1
             DFS-Visit(G, u, counter)

DFS-Visit(G, u, counter)
 u.cc = counter              // пометить вершину
 time = time + 1
 u.d = time
 u.color = GRAY
 for каждая вершина v в G.Adj[u]
     if v.color == WHITE
             v.n = u
             DFS-Visit(G, v, counter)
 time = time + 1
 u.f  = time
 u.color = BLACK
```

Эта процедура DFS увеличивает счетчик каждый раз, когда вызывается DFS-Visit для построения нового дерева в лесу поиска в глубину. Каждая вершина, посещенная процедурой DFS-Visit (и добавленная

в дерево), помечается тем же самым значением счетчика. Таким образом, $u.cc = v.cc$, если и только если u и v посещаются в одном и том же вызове DFS-Visit из DFS, а конечное значение счетчика равно количеству вызовов DFS-Visit, сделанных в DFS. Кроме того, поскольку в конечном итоге посещаются все вершины, каждая вершина помечается.

Таким образом, нам нужно показать, что вершины, посещаемые каждым вызовом DFS-Visit из DFS, являются в точности вершинами в одном компоненте связности графа G.

- Все вершины в компоненте связности посещаются одним вызовом DFS-Visit из DFS.

  Пусть u — первая вершина в компоненте C, посещаемая DFS-Visit. Поскольку вершина становится небелой только при посещении, перед вызовом DFS-Visit для u все вершины в C являются белыми. Таким образом, согласно теореме о белом пути все вершины в C становятся потомками u в лесу, что означает, что все вершины в C посещаются (рекурсивными вызовами DFS-Visit) до того, как происходит возврат из DFS-Visit в DFS.

- Все вершины, посещаемые одним вызовом DFS-Visit из DFS, находятся в том же самом компоненте связности.

  Если две вершины посещаются одним вызовом DFS-Visit из DFS, тогда они находятся в том же самом компоненте связности, т.к. вершины посещаются только при следовании по путям в G (за счет прохождения по найденным в списках смежности ребрам, начиная с некоторой вершины).

Решение упражнения 20.4.3

Неориентированный граф является ациклическим (т.е. лесом), если и только если поиск в глубину не выдает обратных ребер.

- Если есть обратное ребро, то имеется и цикл.
- Если нет обратного ребра, то согласно теореме 20.10 есть только ребра дерева. Следовательно, граф ациклический.

Таким образом, чтобы определить, содержит ли неориентированный граф цикл, необходимо запустить поиск в глубину и классифицировать ребра: если какое-то ребро является обратным, то имеется цикл.

- Время: $O(V)$.

  Не $O(V + E)$: после того, как найдено $|V|$ различных ребер, хотя бы одно из них должно быть обратным ребром, поскольку (согласно теореме Б.2) в ациклическом (неориентированном) лесу справедливо неравенство $|E| \le |V| - 1$.

Решение задачи 20.1

а) 1. Предположим, что (u, v) — обратное или прямое ребро при поиске в ширину в неориентированном графе. Без потери общности представим, что u — собственный предок v в дереве поиска в ширину. Поскольку все ребра u исследуются до исследования любых ребер любого из потомков u, ребро (u, v) должно быть исследовано при исследовании из u. Но тогда (u, v) должно быть ребром дерева.

2. При поиске в ширину ребро (u, v) является ребром дерева, когда процедура устанавливает $v.\pi = u$. Но это происходит только тогда, когда процедура также устанавливает $v.d = u.d + 1$. Поскольку ни $u.d$, ни $v.d$ никогда впоследствии не изменяются, после завершения поиска в ширину мы имеем $v.d = u.d + 1$.

3. Рассмотрим перекрестное ребро (u, v), в котором без утраты общности вершина u посещается раньше v. При исследовании ребер, инцидентных u, вершина v уже должна находиться в очереди, иначе (u, v) было бы ребром дерева. Поскольку v находится в очереди, согласно лемме 20.3 мы имеем $v.d \le u.d + 1$. По следствию 20.4 мы получаем $v.d \ge u.d$. Таким образом, либо $v.d = u.d$, либо $v.d = u.d + 1$.

б) 1. Предположим, что (u, v) — прямое ребро. Тогда оно было бы исследовано при исследовании из u и оказалось бы ребром дерева.

2. То же, что и для неориентированных графов.

3. Для любого ребра (u, v) вне зависимости от того, является ли оно перекрестным, не может быть $v.d > u.d + 1$, т.к. поиск в ширину посещает v не позднее, чем при исследовании ребра (u, v). Таким образом, $v.d \le u.d + 1$.

4. Очевидно, $v.d \ge 0$ для всех вершин v. Для обратного ребра (u, v) вершина v является предком u в дереве поиска в ширину, а это означает, что $v.d \le u.d$. (Заметим, что поскольку петли считаются обратными ребрами, мы могли бы иметь $u = v$.)

Решения для главы 21

Решение упражнения 21.1.1

Это показано в теореме 21.1.

Пусть A — пустое множество, а S — любое множество, содержащее u, но не содержащее v.

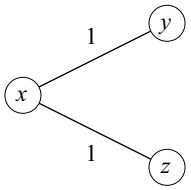
Решение упражнения 21.1.4

Треугольник, все ребра которого имеют одинаковые веса, представляет собой граф, в котором каждое ребро является легким ребром, пересекающим некоторый разрез. Однако такой треугольник представляет собой цикл, поэтому он не будет минимальным остовным деревом.

Решение упражнения 21.1.6

Предположим, что для каждого разреза графа G существует единственное легкое ребро, пересекающее разрез. Рассмотрим два различных минимальных остовных дерева T и T' графа G. Поскольку T и T' различны, T содержит ребро (u, v), которое не принадлежит T'. Если удалить (u, v) из T, то дерево T становится несвязным, что приводит к разрезу $(S, V - S)$. Ребро (u, v) является легким ребром, пересекающим разрез $(S, V - S)$ (согласно упражнению 21.1.3), и по нашему предположению это единственное легкое ребро, пересекающее данный разрез. Поскольку (u, v) является единственным легким ребром, пересекающим $(S, V - S)$, и (u, v) не принадлежит T', каждое ребро в T', которое пересекает $(S, V - S)$, должно иметь вес строго больший, чем $w(u, v)$. Как и в доказательстве теоремы 21.1, мы можем идентифицировать единственное ребро (x, y) в T', которое пересекает $(S, V - S)$ и лежит на цикле, получаемом в случае добавления (u, v) к T'. По нашему предположению известно, что $w(u, v) < w(x, y)$. Тогда мы можем удалить ребро (x, y) из T' и заменить его ребром (u, v), получив остовное дерево с весом строго меньшим, чем $w(T')$. Таким образом, T' не является минимальным остовным деревом, что противоречит предположению о том, что граф имеет два уникальных минимальных остовных дерева.

Вот контрпример:



Здесь граф является собственным минимальным остовным деревом, поэтому минимальное остовное дерево единственно. Рассмотрим разрез ($\{x\}$, $\{y, z\}$). Оба ребра (x, y) и (x, z) являются легкими ребрами, пересекающими разрез.

Решения для главы 22

Решение упражнения 22.1.3

Если наибольшее количество ребер на любом кратчайшем пути из исходной вершины равно m, то свойство ослабления пути говорит о том, что после m итераций процедуры BELLMAN-FORD каждая вершина v достигнет своего веса кратчайшего пути в $v.d$. Согласно свойству верхней оценки после m итераций ни одно значение d никогда не изменится. Следовательно, ни одно значение d не изменится в $(m + 1)$-й итерации. Поскольку мы заранее не знаем m, то не можем заставить алгоритм повториться ровно m раз и завершить работу. Но если алгоритм просто остановится, когда ничего больше не изменится, то он остановится после $m + 1$ итераций.

```
BELLMAN-FORD-EARLY-TERMINATION(G, w, s)
 INITIALIZE-SINGLE-SOURCE(G, s)
 repeat
     changes = FALSE
     for каждое ребро (u, v) ∈ G.E
           if RELAX′(u, v, w)
                changes = TRUE
 until changes == FALSE

RELAX′(u, v, w)
 if v.d > u.d + w(u, v)
     v.d = u.d + w(u, v)
     v.π = u
     return TRUE
 else return FALSE
```

Поскольку в упражнении указано, что в G нет циклов с отрицательным весом, проверка на наличие цикла с отрицательным весом (основанная на наличии значения d, которое изменится при выполнении еще одного шага релаксации) была удалена. Если бы цикл с отрица-

тельным весом существовал, тогда эта версия алгоритма никогда бы не вышла из цикла **repeat**, потому что какое-то значение d изменялось бы на каждой итерации.

Решение упражнения 22.3.3

Да, алгоритм по-прежнему работает. Пусть u — оставшаяся вершина, которая не извлекается из очереди с приоритетами Q. Если u недостижима из s, то $u.d = \delta(s, u) = \infty$. Если вершина u достижима из s, то существует кратчайший путь $p = s \rightsquigarrow x \rightarrow u$. Когда вершина x была извлечена, $x.d = \delta(s, x)$ и затем ребро (x, u) было ослаблено; таким образом, $u.d = \delta(s, u)$.

Решение упражнения 22.3.7

Чтобы найти наиболее надежный путь между вершинами s и t, необходимо запустить алгоритм Дейкстры с весами ребер $w(u, v) = -\lg r(u, v)$ для поиска кратчайших путей из s за время $O(E + V \lg V)$. Самым надежным путем является кратчайший путь из s в t, и надежность этого пути равна произведению надежностей его ребер.

Вот почему такой метод работает. Поскольку вероятности независимы, вероятность того, что путь не будет разрушен, равна произведению вероятностей того, что его ребра не будут разрушены. Мы хотим найти такой путь $s \overset{p}{\rightsquigarrow} t$, чтобы величина $\prod_{(u,v)\in p} r(u,v)$ была максимальной. Это эквивалентно максимизации $\lg\left(\prod_{(u,v)\in p} r(u,v)\right) = \sum_{(u,v)\in p} \lg r(u,v)$, что в свою очередь эквивалентно минимизации $\sum_{(u,v)\in p} -\lg r(u,v)$. (*Примечание*: $r(u, v)$ может быть равно 0, а $\lg 0$ не определено, поэтому в данном алгоритме примем, что $\lg 0 = -\infty$.) Таким образом, если мы назначим веса $w(u, v) = -\lg r(u, v)$, то у нас будет задача поиска кратчайшего пути. Поскольку $\lg 1 = 0$, $\lg x < 0$ для $0 < x < 1$, и мы определили $\lg 0 = -\infty$, все веса w неотрицательны и можно применять алгоритм Дейкстры для поиска кратчайших путей из вершины s за время $O(E + V \lg V)$.

Альтернативное решение

Вы также можете работать с исходными вероятностями, запустив модифицированную версию алгоритма Дейкстры, которая максимизирует произведение надежностей вдоль пути вместо минимизации суммы весов вдоль пути.

В алгоритме Дейкстры нужно будет использовать надежности в качестве весов ребер и внести в него следующие изменения.

- В процедуре INITIALIZE-SINGLE-SOURCE строка 2 принимает вид
  $v.d = -\infty$
- Процедура RELAX принимает вид

  RELAX(u, v, r)
  if $v.d < u.d \cdot r(u, v)$
  $v.d = u.d \cdot r(u, v)$
  $v.\pi = u$
- В процедуре DIJKSTRA переменная Q становится очередью с невозрастающими приоритетами, строка 7 принимает вид

  u = EXTRACT-MAX(Q)

  а строки 11, 12 — вид

  if вызов RELAX увеличивает $v.d$
  INCREASE-KEY(Q, v, $v.d$)

Этот алгоритм изоморфен предыдущему: он выполняет те же операции за исключением того, что работает с исходными вероятностями, а не с трансформированными.

Решение упражнения 22.4.7

Обратите внимание, что после первого прохода все значения d не превышают 0 и ослабление ребер (v_0, v_i) больше не изменит значение d. Следовательно, мы можем исключить вершину v_0, запустив алгоритм Беллмана-Форда на графе ограничений без v_0, но инициализируя все оценки кратчайших путей значением 0 вместо 1.

Решение упражнения 22.5.4

Всякий раз, когда процедура RELAX устанавливает атрибут n для какой-либо вершины, она также уменьшает значение атрибута d этой вершины. Таким образом, если $s.n$ принимает значение, отличающееся от NIL, то значение $s.d$ уменьшается с начального 0 до отрицательного числа. Но $s.d$ — это вес некоторого пути из s в s, который является циклом, включающим s. Таким образом, существует цикл с отрицательным весом.

Решение задачи 22.3

а) Мы можем использовать алгоритм Беллмана-Форда на подходящем взвешенном ориентированном графе $G = (V, E)$, который формируется следующим образом. Для каждой валюты в V предусмотрена одна вершина, а для каждой пары валют c_i и c_j — ориентированные ребра (v_i, v_j) и (v_j, v_i). (Таким образом, $|V| = n$ и $|E| = n(n-1)$.)

Мы ищем цикл $\langle i_1, i_2, i_3, \ldots, i_k, i_1 \rangle$, такой что

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdot \ldots \cdot R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$

Логарифмируя обе части этого неравенства, получаем

$$\lg R[i_1, i_2] + \lg R[i_2, i_3] + \ldots + \lg R[i_{k-1}, i_k] + \lg R[i_k, i_1] > 0.$$

Если взять отрицание обеих частей, тогда получится

$$(-\lg R[i_1, i_2]) + (-\lg R[i_2, i_3]) + \ldots + (-\lg R[i_{k-1}, i_k]) + (-\lg R[i_k, i_1]) < 0,$$

поэтому при таких весах ребер необходимо выяснить, содержит ли граф G цикл с отрицательным весом.

Мы можем определить, существует ли цикл с отрицательным весом в G, для чего добавить дополнительную вершину v_0 с ребрами нулевого веса (v_0, v_i) для всех $v_i \in V$, запустить процедуру BELLMAN-FORD из v_0 и применить булевский результат BELLMAN-FORD (равный TRUE, если нет циклов с отрицательным весом, и FALSE, если есть цикл с отрицательным весом) для получения ответа. То есть мы инвертируем булевский результат, возвращенный алгоритмом Беллмана-Форда.

Описанный метод работает, т.к. добавление новой вершины v_0 с ребрами нулевого веса, соединяющими v_0 и все остальные вершины, не создает новых циклов, но гарантирует, что все циклы с отрицательным весом достижимы из v_0.

Создание графа G, имеющего ребра с нулевым весом, занимает время $\Theta(n^2)$. Последующее выполнение алгоритма Беллмана-Форда занимает время $O(n^3)$. Таким образом, совокупное время составляет $O(n^3)$.

Еще один способ выяснения, существует ли цикл с отрицательным весом, предусматривает создание графа G и запуск любого алгоритма поиска кратчайших путей для всех пар вершин без предварительного добавления вершины v_0 и инцидентных ей ребер. Если результирующая матрица расстояний кратчайших путей содержит отрицательные значения на диагонали, тогда существует цикл с отрицательным весом.

б) *Примечание*: решение этой части также служит решением упражнения 22.1.7.

Предполагая, что запускалась процедура BELLMAN-FORD для решения части (а), необходимо отыскать только вершины цикла с отрицательным весом. Это можно сделать следующим образом. Пройдем по ребрам еще раз. Обнаружение ребра (u, v), для которого $u.d + w(u, v) < v.d$, означает, что вершина v либо находится в цикле с отрицательным весом, либо достижима из него. Мы можем найти вершину в цикле с отрицательным весом, отслеживая значения π из v и запоминая посещенные вершины, пока не встретится вершина x, которая была посещена ранее. Затем мы можем отследить значения π из вершины x, пока не вернемся в x, и все вершины между ними вместе с x будут формировать цикл с отрицательным весом. Мы можем воспользоваться рекурсивным методом, реализованным в процедуре PRINT-PATH из раздела 20.2, но остановить его при возвращении к вершине x.

Время работы процедуры BELLMAN-FORD составляет $O(n^3)$ плюс $O(m)$ на проверку всех ребер и $O(n)$ на вывод вершин цикла, что в сумме дает $O(n^3)$.

Решения для главы 23

Решение упражнения 23.1.3

Матрица $L^{(0)}$ соответствует единичной матрице

$$I = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

из обычного перемножения матриц. Необходимо подставить 0 (тождество для +) вместо ∞ (тождество для min) и 1 (тождество для $\cdot$) вместо 0 (тождество для +).

Решение упражнения 23.1.5

Алгоритм поиска кратчайших путей для всех пар вершин, описанный в разделе 23.1, вычисляет

$$L^{(n-1)} = W^{n-1} = L^{(0)} \cdot W^{n-1},$$

где $l_{ij}^{(n-1)} = \delta(i, j)$ и $L^{(0)}$ — единичная матрица. То есть элемент в i-й строке и j-м столбце матричного "произведения" является расстоянием кратчайшего пути от вершины i до вершины j, а строка i произведения — решение задачи поиска кратчайшего пути с одним источником для вершины i.

Обратите внимание, что в матричном "произведении" $C = A \cdot B$ строка C под номером i представляет собой i-ю строку A, "умноженную" на B. Поскольку нас интересует только i-я строка C, нам никогда не понадобится что-то большее, чем i-я строка A. Таким образом, решением задачи поиска кратчайших путей с одним источником из вершины i будет $L_i^{(0)} \cdot W^{n-1}$, где $L_i^{(0)}$ является i-й строкой $L^{(0)}$, т.е. вектором, i-й элемент которого равен 0, а остальные элементы равны 1.

Выполнение описанных выше "умножений", начиная слева, по существу аналогично выполнению процедуры BELLMAN-FORD. Вектор соответствует значениям d в BELLMAN-FORD — оценкам кратчайшего пути от источника до каждой вершины.

- Изначально вектор равен 0 для источника и 1 для всех остальных вершин, что соответствует значениям, установленным для d процедурой INITIALIZE-SINGLE-SOURCE.
- Каждое "умножение" текущего вектора на W ослабляет все ребра, как происходит в процедуре BELLMAN-FORD. То есть оценка расстояния в строке, например, расстояния до v, обновляется до меньшей оценки, если таковая имеется, за счет добавления некоторого значения $w(u, v)$ к текущей оценке расстояния до u.
- Ослабление/умножение выполняется $n-1$ раз.

Решение упражнения 23.2.4

С верхними индексами вычисление выглядит как

$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}.$$

Если бы с отброшенными верхними индексами процедура вычисляла и сохраняла d_{ik} или d_{kj} перед использованием этих значений для вычисления d_{ij}, то она могла бы вычислять одно из следующих величин:

$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k-1)}\right\},$$

$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k)}\right\},$$

$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k)}\right\}.$$

В любом из этих сценариев код вычисляет вес кратчайшего пути из i в j со всеми промежуточными вершинами в $\{1, 2, \ldots, k\}$. Если применять в вычислении $d_{ik}^{(k)}$, а не $d_{ik}^{(k-1)}$, то будет использоваться подпуть из i в k со всеми промежуточными вершинами в $\{1, 2, \ldots, k\}$. Но вершина k не может быть *промежуточной* вершиной на кратчайшем пути из i в k, т.к. в противном случае этот кратчайший пути содержал бы цикл. Таким образом, $d_{ik}^{(k)} = d_{ik}^{(k-1)}$. Аналогичные рассуждения применяются для того, чтобы показать, что $d_{kj}^{(k)} = d_{kj}^{(k-1)}$. Следовательно, мы можем опустить верхние индексы в вычислениях.

Решение упражнения 23.3.4

Дело в том, что он изменяет кратчайшие пути. Рассмотрим граф, в котором $V = \{s, x, y, z\}$ и есть 4 ребра: $w(s, x) = 2$, $w(x, y) = 2$, $w(s, y) = 5$ и $w(s, z) = -10$. Таким образом, для получения $\widehat{w}$ мы должны прибавить 10 к каждому весу. При w кратчайшим путем из s в y будет $s \to x \to y$ с весом 4, тогда как при $\widehat{w}$ кратчайшим окажется путь $s \to y$ с весом 15. (При $\widehat{w}$ путь $s \to x \to y$ имеет вес 24.) Проблема в том, что за счет простого добавления одной и той же величины к каждому ребру происходит штрафование путей с большим количеством ребер, даже если их веса являются низкими.

Решения для главы 24

Решение упражнения 24.2.11

Для любых двух вершин u и v в графе G мы можем определить транспортную сеть G_{uv}, состоящую из ориентированной версии G с $s = u$, $t = v$ и пропускными способностями всех ребер, равными 1. Поскольку транспортная сеть может не иметь антипараллельных ребер, для каждого ребра в G одно из ориентированных ребер в G_{uv} должно быть разбито на два ребра с добавлением новой вершины. Таким образом, G_{uv} имеет $|V| + |E|$ вершин и $3|E|$ ребер, т.е. $O(V+E)$ вершин и $O(E)$ ребер, как и требуется. Установите пропускные способности всех ребер G_{uv} в 1, чтобы количество ребер в G, пересекающих разрез, было равно пропускной способности разреза в G_{uv}. Пусть f_{uv} обозначает максимальный поток в G_{uv}.

Мы утверждаем, что реберная связность k равна $\min\{|f_{uv}| : v \in V - \{u\}\}$ для любой вершины $u \in V$. Далее мы покажем, что это утверждение верно. Предполагая верность утверждения, мы можем найти k следующим образом:

```
EDGE-CONNECTIVITY(G)
 k = 1
 Выбрать любую вершину u ∈ G.V
 for каждая вершина v ∈ G.V − {u}
     Настроить транспортную сеть G_uv, как было описано выше
     Найти максимальный поток f_uv в G_uv
     k = min{k, |f_uv|}
 return k
```

Утверждение следует из теоремы о максимальном потоке и минимальном разрезе и способа выбора пропускных способностей, который обеспечивает равенство пропускной способности разреза количеству пересекающих его ребер. Мы докажем, что $k = \min\{|f_{uv}| : v \in V - \{u\}\}$ для любой вершины $u \in V$, показав по отдельности, что k не меньше и не больше этого минимума.

- Доказательство того, что $k \geq \min\{|f_{uv}| : v \in V - \{u\}\}$.

  Пусть $m = \min\{|f_{uv}| : v \in V - \{u\}\}$. Предположим, что мы удаляем из графа G только $m - 1$ ребер. Согласно теореме о максимальном потоке и минимальном разрезе для любой вершины v мы имеем, что вершины u и v по-прежнему связаны. (Максимальный поток из u в v имеет величину не менее m, а потому любой разрез, отделяющий u от v, обладает пропускной способностью не менее m, откуда следует, что любой такой разрез пересекается не меньше, чем m ребрами. Таким образом, при удалении $m - 1$ ребер остается хотя бы одно ребро, которое пересекает разрез.) В итоге каждая вершина соединена с u, т.е. граф по-прежнему является связным. Итак, для того, чтобы граф перестал быть связным, потребуется удалить не менее m ребер, т.е. $k \geq \min\{|f_{uv}| : v \in V - \{u\}\}$.

- Доказательство того, что $k \leq \min\{|f_{uv}| : v \in V - \{u\}\}$.

  Рассмотрим вершину v с минимальной величиной $|f_{uv}|$. Согласно теореме о максимальном потоке и минимальном разрезе существует разрез с пропускной способностью $|f_{uv}|$, разделяющий вершины u и v. Поскольку пропускные способности всех ребер равны 1, данный разрез пересекают в точности $|f_{uv}|$ ребер. Если удалить эти ребра, тогда из u в v больше не будет пути, и граф перестанет быть связным. Следовательно, $k \leq \min\{|f_{uv}| : v \in V - \{u\}\}$.

- Таким образом, утверждение о том, что $k = \min\{|f_{uv}| : v \in V - \{u\}\}$, справедливо для любой вершины $u \in V$.

Решение упражнения 24.3.3

По определению, увеличивающий путь — это простой путь $s \rightsquigarrow t$ в остаточной сети G'_f. Поскольку в G нет ребер между вершинами в L и нет ребер между вершинами в R, то и в транспортной сети G' они отсутствуют, а потому их нет и в G'_f. Кроме того, единственные ребра, включающие s или t, соединяют s с L и R с t. Обратите внимание, что хотя ребра в G' могут идти только из L в R, ребра в G'_f также могут идти из R в L.

В итоге любой увеличивающий путь должен проходить следующим образом:

$$s \to L \to R \to \ldots \to L \to R \to t,$$

пересекая вперед и назад границу между L и R не более того количества раз, которое он может сделать, не используя вершину дважды. Он содержит s, t и равное количество различных вершин из L и R — всего не более $2 + 2 \cdot \min(|L|, |R|)$ вершин. Таким образом, длина увеличивающегося пути (т.е. количество его ребер) ограничена сверху величиной $2 \cdot \min(|L|, |R|) + 1$.

Решение задачи 24.4

а) Просто выполните одну итерацию алгоритма Форда–Фалкерсона. Ребро (u, v) в E с увеличенной пропускной способностью гарантирует, что ребро (u, v) находится в остаточной сети. Поэтому необходимо провести поиск увеличивающего пути и обновить поток, если путь найден.

Время

$O(V + E) = O(E)$ за счет нахождения увеличивающего пути либо с помощью поиска в глубину, либо с помощью поиска в ширину.

Чтобы увидеть, что требуется только одна итерация, рассмотрим отдельно случаи, когда (u, v) является или не является ребром, пересекающим минимальный разрез. Если ребро (u, v) не пересекает минимальный разрез, то увеличение его пропускной способности не изменяет пропускную способность любого минимального разреза и, следовательно, значение максимального потока не меняется. Если ребро (u, v) пересекает минимальный разрез, то увеличение его пропускной способности на 1 увеличивает пропускную способность этого минимального разреза на 1 и, следовательно, возможно величину максимального потока на 1. В этом случае либо нет увеличивающе-

го пути (тогда был бы какой-то другой минимальный разрез, который ребро (u, v) не пересекает), либо увеличивающий путь увеличивает поток на 1. В любом случае достаточно одной итерации алгоритма Форда-Фалкерсона.

б) Пусть f — максимальный поток до уменьшения $c(u, v)$.

Если $f(u, v) < c(u, v)$, то ничего не нужно делать.

Если $f(u, v) = c(u, v)$, тогда понадобится обновить максимальный поток. Поскольку $c(u, v)$ — целое число, которое уменьшается, оно должно быть не меньше 1, так что $f(u, v) = c(u, v) \geq 1$.

Определим $f'(x, y) = f(x, y)$ для всех $x, y \in V$ за исключением того, что $f'(u, v) = f(u, v) - 1$. Хотя f' удовлетворяет всем ограничениям пропускной способности, даже после уменьшения $c(u, v)$ это недопустимый поток, т.к. он нарушает закон сохранения потока в u (если только не соблюдается $u = s$) и в v (если только не соблюдается $v = t$). В f' имеется на одну единицу больше потока, входящего в u, чем исходящего из u, и на одну единицу больше потока, исходящего из v, чем входящего в v.

Идея заключается в том, чтобы попытаться перенаправить эту единицу потока так, чтобы она выходила из u и входила в v по другому пути. Если это невозможно, тогда потребуется уменьшить поток из s в u и из v в t на одну единицу.

Найдем увеличивающий путь из u в v (*примечание*: не из s в t).

- Если такой путь есть, тогда увеличить поток по этому пути.
- Если такого пути нет, тогда уменьшить поток из s в u, увеличив поток из u в s. То есть необходимо найти увеличивающий путь $u \rightsquigarrow s$ в G_f и увеличить поток по этому пути на 1. (Такой путь определенно есть, потому что имеется поток из s в u.) Аналогично нужно уменьшить поток из v в t, найдя увеличивающий путь $t \rightsquigarrow v$ в G_f и увеличив поток по этому пути на 1.

Время

$O(V + E) = O(E)$ за счет нахождения путей либо с помощью поиска в глубину, либо с помощью пои0ска в ширину.