Структуры данных

При решении любой задачи возникает необходимость работы с данными и выполнения операций над ними. Набор этих операций для каждой задачи, вообще говоря, свой. Однако, если некоторый набор операций часто используется при решении различных задач, то полезно придумать способ организации данных, позволяющий выполнять именно эти операции как можно эффективнее. После того, как такой способ придуман, при решении конкретной задачи можно считать, что у нас в наличии имеется «черный ящик» (его мы и будем называть структурой данных), про который известно, что в нем хранятся данные некоторого рода, и который умеет выполнять некоторые операции над этими данными. Это позволяет отвлечься от деталей и сосредоточиться на характерных особенностях задачи. Внутри этот «черный ящик» может быть реализован различным образом, при этом следует стремиться к как можно более эффективной (быстрой и экономично расходующей память) реализации. Этот материал посвящен описанию некоторых часто используемых на практике структур данных и способов эффективной реализации этих структур. Описываемые структуры часто сопровождаются примерами задач, в которых они могут быть применены.

п1) Простые структуры данных.

В этом пункте мы рассматриваем некоторые простые структуры данных. Материал разделен на три части: очереди и стеки, списки, структуры для представления графов. Сведения эти излагаются достаточно кратко и не сопровождаются примерами, так как множество различных областей теории алгоритмов (в частности, теория графов, которая, хотелось бы надеяться, будет излагаться на этих сборах) содержат значительное количество примеров применения этих структур, так что примеры, наверное, будут рассмотрены на последующих теоретических занятиях.

а) Очереди и стеки.

Иногда возникает необходимость обработки данных в некотором порядке. Часто встречаются следующие ситуации:
1) Данные нужно обрабатывать в порядке их получения.
2) Данные нужно обрабатывать в порядке, обратном порядку получения.

Структуру данных «очередь» (queue), удобно применять в первой ситуации. Эта структура поддерживает следующие операции:
QUEUE-INIT – инициализирует (создает пустую) очередь;
ENQUEUE (x) – добавляет в очередь объект x;
DEQUEUE – удаляет из очереди объект, который был добавлен раньше всех, и возвращает в качестве результата удаленный объект (предполагается, что очередь не пуста);
QUEUE-EMPTY – возвращает TRUE, если очередь пуста (не содержит данных), и FALSE – в противном случае.
Опишем реализацию очереди на базе массива. Будем использовать для хранения данных массив Q[1..N], где число N достаточно велико. При этом данные всегда будут храниться в некотором интервале из последовательных ячеек этого массива. Мы будем использовать переменные Head и Tail для указания границ этого интервала. Более точно, данные хранятся в ячейках с индексами от Tail+1 до Head (предполагается, что Tail < Head; если же Head = Tail, то очередь пуста). Соблюдается также следующее правило: чем позже был добавлен в очередь объект, тем большим будет индекс ячейки массива, в которую он помещается. Это означает, что объект, который был добавлен в очередь раньше всех – это Q[Tail+1].
Псевдокод операций, работаюших с очередью, может выглядеть следующим образом:

Operation QUEUE-INIT

```
  Head:= 0;
  Tail:= 0;
End;

Operation ENQUEUE (x)
  Head:= Head+1;
  Q[Head]:= x;
End;

Operation DEQUEUE
  Tail:= Tail+1;
  Return Q[Tail];
End;

Operation QUEUE-EMPTY
  If Tail < Head
    Then Return FALSE
    Else Return TRUE;
End;
```

Все операции над очередью при такой реализации работают за O(1), следовательно, такая реализация эффективна по времени. Однако, число N нужно выбирать достаточно большим (в зависимости от задачи), чтобы избежать «переполнения» очереди, то есть ситуации, когда Head > N. Это может приводить в некоторых задачах к неэффективному использованию памяти. Альтернативная реализация очереди может быть построена на базе списков (см. следующий раздел).

Структуру данных «стек» (stack) удобно применять во второй ситуации. Она во многом аналогична очереди, за исключением того, что из стека удаляется тот объект, который был добавлен позже всех (а не раньше всех, как в очереди). Итак, стек поддерживает следующие операции:
STACK-INIT – инициализирует стек;
PUSH (x) – добавляет в стек объект x;
POP – удаляет из стека объект, который был добавлен позже всех, и возвращает в качестве результата удаленный объект (предполагается, что стек не пуст);
STACK-EMPTY – возвращает TRUE, если стек пуст, и FALSE – в противном случае.
Стек будем реализовывать также на базе массива S[1..N]. Данные будем хранить в некотором интервале последовательных ячеек массива (более точно, в ячейках с индексами от 1 до Top). Top – переменная, которая содержит текущее количество объектов в стеке. Как и в случае очереди, соблюдается правило: если $i < j \leq$ Top, то объект S[i] был добавлен в стек раньше, чем объект S[j]. Это гарантирует, что объект S[Top] – тот объект, который был добавлен в стек позже всех.
Псевдокод операций над стеком может быть следующим:

```
Operation STACK-INIT
  Top:= 0;
End;


Operation PUSH (x)
  Top:= Top+1;
  S[Top]:= x;
End;
```

```
Operation POP
  Top:= Top-1;
  Return S[Top+1];
End;

Operation STACK-EMPTY
  If Top > 0
    Then Return FALSE
    Else Return TRUE;
End;
```

Реализованные операции, работают, очевидно, эффективно по времени – за O(1). Однако, использование памяти, как и в случае очереди, может оказаться в некоторых задачах неэффективным.

б) Списковые структуры

Список – это структура, в которой данные выписаны в некотором порядке. Однако, в отличие от массива, этот порядок определяется указателями, связывающими элементы списка в линейную цепочку. Обычно элемент списка представляет собой запись, содержащую ключ (идентификатор) хранящегося объекта, один или несколько указателей и необходимую информацию об объекте. Под ключом подразумевается какая-либо величина, идентифицирующая объект. К примеру, если мы храним информацию о городах, то ключом может быть название города.
Над данными поддерживаются следующие операции:
LIST-INIT – инициализирует список;
LIST-FIND (k) – возвращает TRUE, если в списке есть объект с ключом k, иначе возвращает FALSE;
LIST-INSERT (obj) – добавляет в список объект obj;
LIST-DELETE (x) – удаляет из списка элемент x.
Возможна различная связь данных в списке. Если каждый элемент списка содержит указатель на элемент, следующий непосредственно за ним, то получаемый список называют *односвязным*. Если в дополнение к этому каждый элемент списка содержит указатель на элемент, следующий непосредственно перед ним, то список называют *двусвязным.* Обычно у последнего элемента списка указатель на следующий элемент равен NIL. Но в некоторых списках удобно, чтобы этот указатель показывал на первый элемент списка. Таким образом, список из цепочки превращается в кольцо. Такие списки (односвязные и двусвязные) называют *кольцевыми*.
Опишем реализацию всех операций для двусвязного некольцевого списка. Каждый элемент списка будем хранить как запись, содержащую следующие поля: Key – ключ объекта, Data – дополнительная информация об объекте, Next – указатель на следующий элемент списка (Next = NIL у последнего элемента списка), Prev – указатель на предыдущий элемент списка (Prev = NIL у первого элемета списка). Будем всегда поддерживать указатель Head на первый элемент списка (если список пуст, то Head = NIL). Будем считать, что добавляемые объекты – записи, содержащие поля Key – ключ и Data – дополнительная информация.
Реализация операции LIST-INIT тривиальна:

```
Operation LIST-INIT
  Head:= NIL;
End;
```

Операция LIST-INIT, очевидно, выполняется за O(1).
Для реализации операции LIST-FIND нужно просмотреть все элементы списка в поисках нужного.

```
Operation LIST-FIND (k)
  X:= Head;
  While X <> NIL Do Begin
    If X^.Key = k
      Then Return TRUE;
    X:= X^.Next;
  End;
  Return FALSE;
End;
```

Операция LIST-FIND выполняется за O(n), где n – количество элементов в списке. Это означает, что список не является структурой, эффективно выполняющей поиск.
При добавлении элемента в список можно вставить его в начало списка, при этом нужно переписать некоторые указатели.

```
Operation LIST-INSERT (obj)
  New(X);
  X^.Key:= obj.Key;
  X^.Data:= obj.Data;
  X^.Next:= Head;
  X^.Prev:= NIL;
  If Head <> NIL
    Then  Head^.Prev:= X;
  Head:= X;
End;
```

Операция LIST-INSERT выполняется за O(1).
Наконец, при удалении элемента из списка нужно соединить указателями предыдущий и следующий за ним элементы. При этом нужно аккуратно обработать случай, когда элемент является первым или последним в списке.

```
Operation LIST-DELETE (x)
  If x^.Prev <> NIL
    Then x^.Prev^.Next:= x^.Next
    Else Head:= x^.Next;
  If x^.Next <> NIL
    Then x^.Next^.Prev:= x^.Prev;
  Dispose(x);
End;
```

Операция LIST-DELETE выполняется за O(1), однако для ее выполнения нужно иметь указатель на удаляемый элемент списка. Если этот указатель неизвестен, то удаляемый элемент нужно сначала найти, для чего придется просмотреть (в худшем случае) весь список, и удаление будет работать за O(n), где n – количество элементов в списке.

Упражнение 1. Опишите реализацию всех операций для других видов списков. Укажите достоинства и недостатки каждого из видов.
Упражнение 2. Реализуйте очередь и стек на базе списочной структуры. Укажите достоинства и недостатки такой реализации.

в) Способы хранения графов.

Вообще говоря, данный вопрос не принято относить к теме «структуры данных». Однако, в принципе, граф можно рассматривать как структуру данных, а любой способ его хранения – как реализацию этой структуры данных (мы так делать не будем). К тому же, в данном разделе мы увидим пример эффективного применения списков, описанных в предыдущем разделе.

Для начала проясним, что такое граф. Рассмотрим произвольный конечный непустой набор (множество) объектов V. Назовем эти объекты вершинами, а набор V – множеством вершин. Рассмотрим теперь некоторый конечный набор (множество) E, состоящий из пар вида (a, b), где $a \in V$ и $b \in V$ ($a \neq b$). Будем полагать, что все пары в E попарно различны. Эти пары называются ребрами, а набор E – множеством ребер. Двойка G = (V, E) называется (конечным) *ориентированным* графом. Можно наглядно представить понятие граф, если для каждой вершины нарисовать точку, и для каждого ребра (a, b) провести стрелку от точки, обозначающей вершину a, к точке, обозначающей вершину b. Нарисованная схема напоминает карту однонаправленных дорог, соединяющих некоторые города. И в самом деле, графы часто применяют для моделирования различных транспортных сетей. Однако, область применения графов этим не ограничивается. Если вместе с каждым ребром (a, b) граф содержит и ребро (b, a), то такой граф называют *неориентированным*. На рисунке ребра такого графа изображают не стрелками, а линиями.

Обычно вершины графа нумеруют последовательными натуральными числами от 1 до $|V|$. Простым способом хранения графа является *матрица смежности*. Матрицей смежности графа G называется таблица A[1..$|V|$, 1..$|V|$] такая, что A[i, j] = 1, если граф G содержит ребро из вершины с номером i в вершину с номером j и A[i, j] = 0 в противном случае. Заметим, что на главной диагонали этой матрицы расположены нули и, если граф G – неориентированный, то эта матрица симметрична относительно главной диагонали. Иногда рассматривают *взвешенные* графы, в которых каждому ребру (a, b) ставится в соответствие число w(a, b), называемое весом ребра. При хранении таких графов также можно использовать матрицу смежности, при этом A[i, j] = w, если граф G содержит ребро из вершины с номером i в вершину с номером j веса w, и A[i, j] = 0, если граф не содержит ребра из вершины с номером i в вершину с номером j. Достоинством (одним из немногих) матрицы смежности является возможность получения информации о наличии или отсутствии заданного ребра в графе за время O(1). Действительно, достаточно только проверить, чему равна соответствующая ячейка матрицы A.

Серьезным недостатком матрицы смежности является то, что ее хранение требует использования O($|V|^2$) байт памяти. В задачах часто встречаются разреженные графы, у которых величина $|E|$ гораздо меньше, чем $|V|^2$. Матрица смежности таких графов содержит очень много нулей. В таких случаях хорошим выбором будет использование *списков смежности*. Будем хранить граф G при помощи массива Adj[1..$|V|$]. При этом каждый его элемент Adj[i] представляет собой список всех ребер, выходящих из вершины с номером i. Такое представление графа расходует память экономнее: его использование требует всего $O(|V|+|E|)$ байт памяти, что гораздо меньше $O(|V|^2)$ для разреженных графов. Списки смежности также можно использовать для хранения взвешенных графов, при этом вес ребра хранится в элементе списка как дополнительная информация. Следует отметить, что при использовании списков смежности мы уже не можем определить за O(1) присутствие или отсутствие заданного ребра в графе. Теперь нам нужно просмотреть весь список ребер, выходящих из начальной вершины этого ребра, для чего (в худшем случае) может понадобиться $O(|V|)$ времени. Однако большинство алгоритмов на графах не работают отдельно с каждым ребром. Обычно некоторый процесс производится или со всеми ребрами графа, или со всеми ребрами, выходящими из заданной вершины. И в том, и в другом случае списки смежности оказываются более эффективными чем матрица смежности. Иногда рассматривают графы, в которых может быть несколько ребер, соединяющих одну и ту же пару вершин. Такие графы называют *мультиграфами*. Списки смежности позволяют хранить такие графы, тогда как хранение мультиграфов при помощи матрицы смежности затруднительно (если, вообще, возможно).

В заключение отметим, что возможны и другие представления графов и выбор конкретного представления зависит от постановки решаемой задачи. Списки смежности и матрица смежности, однако, являются достаточно универсальными способами хранения графов и используются чаще всего.

Упражнение 3. Предположим, что граф необходимо также модифицировать, например, добавлять и удалять ребра. Сравните с этой позиции матрицу смежности и списки смежности.
Упражнение 4. Предположим, что граф не нужно модифицировать. Предложите способ хранения графа, обладающий всеми описанными выше достоинствами списков смежности и дополнительно позволящий проверять наличие или отсутствие заданного ребра за O(logE).

n2) Приоритетные очереди.

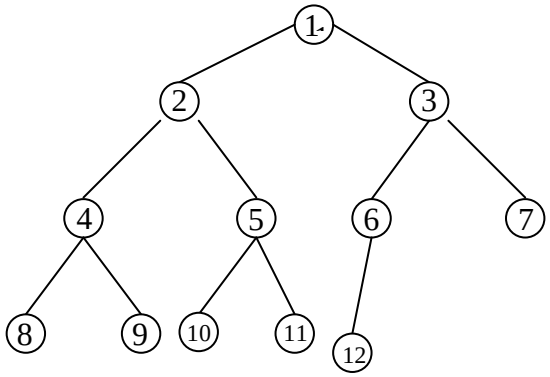
Иногда необходимо работать с динамически изменяющимся множеством объектов, среди которых часто нужно находить объект с минимальным ключом (будем полагать, что ключи – это числа). В этом случае может пригодиться структура данных, называемая “приоритетной очередью”. Более точно, приоритетная очередь – это структура, хранящая набор объектов и поддерживающая следующие операции:

INSERT (x) – добавляет в очередь новый объект x;
MINIMUM – возвращает объект с минимальным значением ключа (если таких несколько, то любой из них);
EXTRACT-MIN – удаляет из очереди объект с минимальным значением ключа (если таких несколько, то удаляется объект, возвращаемый операцией MINIMUM).

Этот пункт разделен на два раздела. В первом рассматривается реализация приоритетной очереди, называемая бинарной кучей. Эта реализация позволяет также удалять произвольные объекты из множества, а также менять значения ключей объектов (для выполнения этих операций, однако, необходимо знать информацию о позиции объекта, к которыму мы хотим их применить). Во втором разделе рассматривается два примера применения бинарной кучи.

а) Бинарная куча

Будем считать, что объекты хранятся в вершинах полного двоичного дерева (самый нижний уровень дерева заполнен, возможно, не полностью).



Пронумеруем вершины этого дерева слева направо сверху вниз (см. рисунок). Пусть N – количество вершин в дереве. Нетрудно видеть, что справедливы следующие свойства.
Свойство 1. Высота полного двоичного дерева из N вершин (то есть максимальное количество ребер на пути от корня к листьям) есть O(log N).
Свойство 2. Рассмотрим вершину полного двоичного дерева из N вершин, имеющую номер i. Если i = 1, то у вершины i нет отца. Если i > 1, то ее отец имеет номер i div 2. Если 2i < N, то у вершины i

есть два сына с номерами 2i и 2i+1. Если 2i = N, то единственный сын вершины i имеет номер 2i. Если 2i > N, то у вершины i нет сыновей.
Будем говорить что объекты, хранящиеся в дереве, образуют бинарную кучу, если ключ объекта, находящегося в любой вершине, всегда не превосходит ключей объектов в сыновьях этой вершины. Будем хранить бинарную кучу в массиве H. Элемент этого массива H[i] будет содержать объект, находящийся в вершине дерева с номером i.
Следующее очевидное свойство бинарной кучи является ключевым.
Свойство 3. В бинарной куче объект H[1] (или объект, хранящийся в корне дерева) имеет минимальное значение ключа из всех объектов.
Отсюда сразу же получается эффективная реализация операции MINIMUM, работающая за O(1).

```
Operation MINIMUM
  Return H[1];
End;
```

Реализация остальных операций не столь очевидна. Рассмотрим операцию INSERT. Сначала мы помещаем добавляемый объект x на самый нижний уровень дерева на первое свободное место. Если окажется, что ключ этого объекта больше (или равен) ключа его отца, то свойство кучи нигде не нарушено, и мы корректно добавили вершину в кучу. В противном случае, поменяем местами объект с его отцом. В результате вершина с добавляемым объектом «всплывает» на одну позицию вверх. Это «всплытие» продолжается до тех пор, пока ключ объекта не станет больше (или равен) ключа его отца или пока объект не «всплывет» до самого корня дерева. Время работы операции INSERT прямо пропорционально высоте дерева. Как следует из Свойства 1, оно равно O(log N).

```
Operation INSERT (x)
  N:= N+1;
  H[N]:= x;
  i:= N;
  While (i > 1) and (H[i].Key < H[i div 2].Key) Do Begin
    S:= H[i];
    H[i]:= H[i div 2];
    H[i div 2]:= S;
    i:= i div 2;
  End;
End;
```

Теперь рассмотрим операцию EXTRACT-MIN. Для ее реализации мы сначала перемещаем объект из листа с номером N в корень (при этом объект в корне затирается, так как его и нужно удалить). Ставший свободным при этом лист удаляется. Если теперь окажется, что ключ объекта в корне меньше (или равен) ключей объектов в его сыновьях (что очень маловероятно), то свойство кучи нигде не нарушено и удаление было проведено корректно. В противном случае, выберем сына корня с минимальным значением ключа и поменяем объект в корне с объектом в этом сыне. В результате объект, находившийся в корне, «спускается» на одну позицию вниз. Этот «спуск» продолжается до тех пор, пока объект не окажется в листе или его ключ не станет меньше (или равен) ключей объектов в его сыновьях. Операция выполняется за O(log N), так как время ее работы пропорционально высоте дерева.

```
Operation EXTRACT-MIN
  H[1]:= H[N];
  N:= N-1;
  i:= 1;
  While 2*i <= N Do Begin
```

```
    If (2*i = N) or (H[2*i].Key < H[2*i+1].Key)
      Then  Min:= 2*i
      Else Min:= 2*i+1;
    If H[i].Key <= H[Min].Key
      Then Break;
    S:= H[i];
    H[i]:= H[Min];
    H[Min]:= S;
    i:= Min;
  End;
End;
```

Итак, мы эффективно реализовали приоритетную очередь на базе бинарной кучи. Рассмотрим теперь еще две операции, которые можно эффективно выполнять в бинарной куче:

CHANGE-KEY (i, k) – установить ключ объекта H[i] равным k;
DELETE (i) – удалить объект H[i] из кучи.

Для реализации операции CHANGE-KEY рассмотрим три варианта. Если H[i].Key > k, то ключ вершины i уменьшается и может нарушиться свойство кучи в ее отце. Аналогичная ситуация возникала в операции INSERT. В этом случае действуем точно так же – производим «всплытие» вершины. Если H[i].Key = k, то ключ вершины i не меняется и ничего делать не надо. Если, наконец, H[i].Key < k, то ключ вершины i увеличивается и в ней может нарушиться свойство кучи. Аналогичная ситуация возникала в операции DELETE. В этом случае нужно произвести «спуск» вершины. Во всех трех случаях время работы будет ограничено величиной O(logN).

Упражнение 5. Напишите псевдокод операции CHANGE-KEY.

Операцию DELETE можно написать следующим образом:

```
Operation DELETE (i)
  CHANGE-KEY (i, H[1].Key-1);
  EXTRACT-MIN;
End;
```

После выполнения CHANGE-KEY объект, находившийся в H[i], будет иметь минимальное значение ключа среди всех объектов, после чего будет удален при помощи вызова EXTRACT-MIN. Время работы операции равно O(logN).

Упражнение 6. Пусть массив H содержит N объектов, но не является кучей. Опишите способ превратить его в кучу за линейное время O(N).

б) Примеры.

Пример 1. Рассмотрим следующую задачу HUMBLE. Предположим, что задано множество $P = \{p_1, p_2, \ldots, p_k\}$, где все p_i – попарно различные простые числа. Рассмотрим множество M, состоящее из тех чисел, в разложении которых на простые множители нет чисел, отличных от p_i. То есть в M входят все числа вида $p_1^{a1}p_2^{a2}\ldots p_k^{ak}$, $a_i \geq 0$. Необходимо вывести N минимальных чисел из множества M в порядке возрастания. Будем предполагать, что числа на входе и выходе этой задачи помещаются в некоторый стандартный целочисленный тип и все арифметические операции над ними выполняются за O(1). Опишем решение этой задачи, работающее за O(NlogN+KlogK), причем расходы памяти составят порядка O(N+K) байт.

Мы предполагаем здесь, что числа p_i отсортированы по возрастанию (в противном случае их можно отсортировать за O(KlogK)). Для решения будет использоваться приоритетная очередь, реализованная при помощи бинарной кучи. Каждый объект, хранящийся в очереди, будет соответствовать некоторому числу из множества M. Ключ объекта как раз и будет равен значению числа. Кроме этого, в объекте хранится дополнительная информация – номер максимального простого числа, входящего в разложение ключа объекта на простые множители. Итак, для каждого объекта obj, хранящегося в очереди, имеем: $p_{obj.Data}$ – максимальное число в разложении obj.Key на простые множители.

Наш алгоритм будет работать по шагам. На каждом шаге некоторый объект будет доставаться из кучи. Его ключ и будет очередным выводимым числом. Далее некоторые новые объекты, которые могут вывестись на следующих шагах, будут помещаться в кучу. Сейчас необходимо решить, какие именно объекты мы будем помещать в кучу на каждом шаге.

Самый простой вариант заключается в следующем: когда мы достаем из кучи объект с ключом X, то можно поместить в кучу объекты с ключами Xp_1, Xp_2, …, Xp_k. Этот вариант далек от идеала по двум причинам. Первая состоит в том, что объект с одним и тем же ключом может быть помещен в кучу несколько раз (например, объект с ключом p_1p_2 будет помещен в кучу, когда мы достаем из нее объект с ключом p_1 и объект с ключом p_2). Если бы даже этого не было, то возникает вторая проблема: во время работы алгоритма нам нужно сделать порядка O(NK) добавлений элементов в кучу, что недопустимо как по причинам, связанным со временем работы, так и по причинам, связанным с памятью.

Решим сначала проблему с повторами в куче. Для этого вспомним, что мы пока никак не используем дополнительную информацию, хранящуюся в объектах. Новое наше правило выглядит следующим образом: когда мы достаем из кучи объект с ключом X и дополнительной информацией t, то будем помещать в кучу объекты с ключами Xp_t, Xp_{t+1}, …, Xp_k. Теперь наличие одинаковых объектов в куче исключено и мы можем оценить сложность решения. Нетрудно видеть, что она равна O(NKlog(NK)), а расходы памяти составляют O(NK) байт.

Для улучшения решения, неплохо бы придумать такое правило добавления новых объектов в кучу, при котором на каждом шаге добавляется некоторое постоянное (не зависящее от N или K) количество объектов. Рассмотрим следующие две унарные операции A и B над объектами. В результате действия операции A из объекта obj получается объект с ключом $obj.Key \times p_{obj.Data}$ и дополнительной информацией obj.Data. В результате действия операции A из объекта obj получается объект с ключом $obj.Key / p_{obj.Data} \times p_{obj.Data+1}$ и дополнительной информацией obj.Data+1 (предполагается, что obj.Data < k). Наш алгоритм основан на следующих свойствах введенных операций.

Свойство 1. Начиная с объекта с ключом p_1 с помощью применения конечного числа операций A и B в некотором порядке можно получить объект с ключом, равным какому угодно числу из множества M.

Свойство 2. В результате действия операций A и B ключ объекта увеличивается.

Свойство 3. Любой объект X (кроме объекта с ключом p_1) может быть получен из некоторого объекта Y применением операции A или B. При этом, если X получен из Y применением операции A, то не существует объекта Z такого, что X получается из Z применением операции B. Аналогично, если X получен из Y применением операции B, то не существует объекта Z такого, что X получается из Z применением операции A. (Ключи всех рассматриваемых объектов принадлежат множеству M).

Окончательный вариант используемого правила таков: доставая из кучи объект с ключом X, будем помещать в нее объекты, получаемые из X применением операций A и B. При этом свойство 1 гарантирует, что каждое число из M будет когда-нибудь выведено. Свойство 2 гарантирует, что числа будут выводиться в порядке возрастания. Наконец, свойство 3 исключает возможность появления повторов в приоритетной очереди.

Псевдокод полученного алгоритма выглядит следующим образом:

Algorithm HUMBLE

```
  INIT-HEAP;
  obj.Key:= p[1];
  obj.Data:= 1;
  INSERT (obj);
  For i:= 1 to N Do Begin
    minobj:= MINIMUM;
    EXTRACT-MIN;
    OUTPUT(minobj.Key);
    obj.Key:= minobj.Key*p[minobj.Data];
    obj.Data:= minobj.Data;
    INSERT (obj);
    If minobj.Data < N
      Then Begin
        obj.Key:= (minobj.Key div p[minobj.Data])*p[minobj.Data+1];
        obj.Data:= minobj.Data+1;
        INSERT (obj);
      End;
  End;
End;
```

Нетрудно видеть, что время работы полученного алгоритма (с учетом времени на сортировку массива p), равно O(NlogN+KlogK). Так как общее количество добавлений в кучу не превышает 2N, то для нее необходимо O(N) байт памяти, и общие расходы памяти составят O(N+K) байт.
Упражнение 7. Проведите строгое доказательство свойств 1-3 и объясните подробнее, почему их выполнение гарантирует корректность работы приведенного алгоритма.

В примере 1 мы использовали только те операции, которые допустимы в приоритетной очереди. Но, как говорилось в предыдущем разделе, в бинарной куче можно эффективно выполнять еще и операции CHANGE-KEY и DELETE. Следующий пример иллюстрирует полезность операции CHANGE-KEY.

Пример 2. Рассмотрим следующую задачу THERACE. На координатной прямой в точках с координатами X_1, X_2, …, X_N расположены N точек. Для простоты будем предполагать, что $X_1 < X_2 < \ldots < X_N$. В момент времени 0 каждая точка, имеющая координату X_i, начинает двигаться вправо с постоянной скоростью $V_i > 0$. В некоторые моменты времени одна из точек может “обогнать” другую. Будем считать, что никакие два «обгона» не происходят одновременно. Упорядочим все обгоны по возрастанию момента времени, в которые они происходят. Вывести информацию о первых K произошедших обгонах (предполагается, что K не превосходит общего числа обгонов). Для каждого обгона вывести, какая точка обгоняет какую. Мы рассмотрим решение этой задачи, имеющее сложность O((N+K)logN) и использующее порядка O(N) байт памяти.
Для упрощения дальнейших рассуждений введем фиктивную точку с номером N+1, которая находится где-то очень далеко справа и движется очень быстро, и точку с номером 0, которая находится где-то очень далеко слева и движется очень медленно. Следующая простая функция TIME (i, j) выдает момент времени, когда i-я точка обгоняет j-ю (предполагается, что $1 \le i, j \le N+1$). Если такого не может произойти никогда, то возвращается специальное значение INFINITY (бесконечность). Мы считаем, что бесконечность больше любого конечного числа.

```
Function TIME (i, j)
  If (i = N+1) or (j = N+1)
    Then Return INFINITY
    Else
    If (X[i] < X[j]) and (V[i] > V[j])
```

```
    Then Return (X[j]-X[i])/(V[i]-V[j])
    Else
     If (X[j] < X[i]) and (V[j] > V[i])
      Then Return (X[i]-X[j])/(V[j]-V[i])
      Else Return INFINITY;
End;
```

Наш алгоритм будет работать с бинарной кучей. В каждый момент времени будет находиться ровно N объектов. Каждый объект соответствует некоторой точке. Для каждого объекта мы храним следующую дополнительную информацию: поле Id, равное номеру точки, которой соответствует объект; поле Next, равное номеру точки, которая в рассматриваемый момент времени является ближайшей справа от точки Id; поле Prev, равное номеру точки, которая в рассматриваемый момент времени является ближайшей слева от точки Id (поля Next и Prev могут принимать фиктивные значения 0 и N+1). Что касается ключа объекта, то он равен моменту времени, когда точка Id обгонит точку Next (возможно, что ключ равен INFINITY). Кроме кучи мы будем также поддерживать массив Pos[1..N], обладающий следующим свойством: H[Pos[i]].Id = i. То есть, Pos[i] хранит местоположение в куче того объекта, поле Id у которого равно i. Здесь мы существенно используем то, что значение поля Id – целое число от 1 до N, и никакие два объекта не могут иметь одинаковое значение этого поля.

Упражнение 8. Покажите, как можно модифицировать операции для работы с кучей, так, чтобы ассимптотическое время работы этих операций осталось неизменным, и кроме этого всегда поддерживались правильные значения элементов массива Pos.

Основная идея нашего алгоритма очень проста: когда точка с номером i обгоняет точку с номером j, то чуть-чуть раньше этого момента эти две точки были соседями. Следовательно, для того, чтобы получить информацию об очередном обгоне, необходимо взять из кучи объект obj с минимальным ключом. Из него мы узнаем, что на очередном обгоне точка с номером obj.Id обгоняет точку с номером obj.Next. После этого обгона расположение точек на прямой немного изменится и нужно модифицировать некоторые поля некоторых объектов в куче и переходить к следующему обгону. Так мы делаем K раз. Рассмотрим псевдокод получившегося алгоритма.

```
Algorithm THERACE
  INIT-HEAP;
  For i:= 1 to N Do Begin
    obj.Id:= i;
    obj.Next:= i+1;
    obj.Prev:= i-1;
    obj.Key:= TIME (obj.Id, obj.Next);
    INSERT (obj);
  End;
  For i:= 1 to K Do Begin
    minobj:= MINIMUM;
    OUTPUT (minobj.Id, minobj.Next);
    H[Pos[minobj.Id]].Prev:= minobj.Next;
    H[Pos[minobj.Id]].Next:= H[Pos[minobj.Next]].Next;
    H[Pos[minobj.Next]].Prev:= minobj.Prev;
    H[Pos[minobj.Next]].Next:= minobj.Id;
    If H[Pos[minobj.Id]].Next <> N+1
      Then H[Pos[H[Pos[minobj.Id]].Next]].Prev:= minobj.Id;
    If minobj.Prev <> 0
      Then Begin
```

```
      H[Pos[minobj.Prev]].Next:= minobj.Next;
      CHANGE-KEY (Pos[minobj.Prev], TIME (H[Pos[minobj.Prev]].Id, H[Pos[minobj.Prev].Next]);
    End;
  CHANGE-KEY (Pos[minobj.Id], TIME (H[Pos[minobj.Id]].Id, H[Pos[minobj.Id]].Next);
  CHANGE-KEY (Pos[minobj.Next], TIME (H[Pos[minobj.Next]].Id, H[Pos[minobj.Next]].Next);
 End;
End;
```

Очевидно, что время работы приведенного алгоритма составляет O((N+K)logN) и мы используем порядка O(N) байт памяти.

Упражнение 9. Предложите алгоритм, считающий общее число обгонов, которые произойдут, за время O(NlogN). Возможно, Вам помогут идеи из пункта 4, но есть и алгоритм, не использующий этих идей, а основанный на подходе «разделяй и властвуй».

n3) Система непересекающихся множеств.

В некоторых задачах необходимо хранить разбиение какого-то набора объектов на непересекающиеся множества и уметь эффективно выполнять следующие операции:

FINDSET (x) – возвращает идентификатор множества, содержащего объект x (под идентификатором мы понимаем уникальное число определяющее множество);
UNION (a, b) – объединяет множества с идентификаторами a и b в одно (предполагается, что a≠b).

Структуру данных, умеющую выполнять такие операции, назовем *системой непересекающихся множеств*. Далее мы предполагаем, что объекты пронумерованы от 1 до N и операция FINDSET вместо объекта x получает на вход его номер, так что система непересекающихся множеств непосредственно с объектами работать не будет. В дальнейшем, мы отождествляем объекты с их номерами. Будем считать, что в начальный момент времени каждый объект находится в отдельном множестве. В связи с этим мы также рассматриваем еще одну операцию, которая применяется только один раз и инициализирует систему непересекающихся множеств:

INIT (N) – инициализировать систему непересекающихся множеств, поместив в нее набор из N непересекающихся множеств – по одному множеству для каждого объекта.

Мы рассматриваем разные реализации системы непересекающихся множеств. В некоторых реализациях время работы каждой операции может быть достаточно точно оценено. В других реализациях мы будем рассматривать оценку такого рода: чему равно время выполнения произвольного набора из M операций (FINDSET и UNION), после того как она была инициализирована вызовом INIT (N)?

Пункт разделен на 4 раздела. Первые три из них посвящены реализации системы непересекающихся множеств на основе массивов, списков и леса. В последнем разделе рассматривается пример задачи, эффективно решаемой с помощью такой структуры данных.

а) Реализация на основе массива.

Эта реализация является идейно очень простой, однако очень неэффективной по времени. Идея заключается в том, чтобы хранить некоторый массив Pr[1..N], в котором для каждого объекта i хранится значение представителя множества, в котором этот объект находится. Итак, Pr[i] – представитель множества, содержащего объект i. Реализация всех операций при этом очевидна и приводится без комментариев.

```
Operation INIT (N)
  For i:= 1 to N Do
    Pr[i]:= i;
End;

Operation FINDSET (x)
  Return Pr[x];
End;

Operation UNION (a, b)
  For i:= 1 to N Do
    If Pr[i] = a
      Then Pr[i]:= b;
End;
```

Оценим время работы каждой из операций. Операция INIT работает за O(N), операция FINDSET – за O(1), операция UNION – за O(N). Серьезный недостаток – долгое время работы операции UNION (то, что INIT работает долго – не страшно, так как она вызывается всего один раз). Так как общее количество операций UNION не превосходит N-1 (после каждой из них общее количество множеств уменьшается), то получаем также следующий результат: после того, как был произведен вызов INIT (N), суммарное время работы любой последовательности из M операций не превосходит $O(N^2+M)$. Вывод таков: полученная реализация крайне неэффективна, однако для небольших значений N и M она хороша вследствие своей простоты.

б) Реализация на основе списков

Более эффективная реализация получится, если хранить объекты (их номера) каждого множества в некотором списке. Опишем ее более подробно. Для хранения списков мы используем массив Next[1..N], смысл которого таков: если объект i является последним в своем списке, то Next[i] = 0, иначе Next[i] равен номеру объекта, который является следующим в списке за i. Как и раньше, мы сохраняем массив Pr, хранящий представителей. Теперь мы добавляем еще одно условие: представителем каждого множества является объект, находящийся в начале списка, которым представлено это множество. Кроме этого, мы используем массив L[1..N], где L[i] – длина (количество элементов) в списке, содержащем объект i. Наконец, нам понадобится массив Tail[1..N], в котором Tail[i] равен последнему элементу списка, содержащего элемент i. Однако мы заботимся о правильности значений L[i] и Tail[i], только если i – представитель какого-нибудь множества.
Перейдем к реализации операций. Операция INIT просто помещает каждый объект в отдельный список и работает за O(N).

```
Operation INIT (N)
  For i:= 1 to N Do Begin
    Next[i]:= 0;
    Pr[i]:= i;
    Tail[i]:= i;
    L[i]:= 1;
  End;
End;
```

Операция FINDSET остается прежней и работает за O(1).

```
Operation FINDSET (x)
  Return Pr[x];
End;
```

Для реализации операции UNION нужно присоединить хвост одного из списков к голове второго, далее везде во втором списке поменять представителей, и, наконец, изменить длину первого списка. Так как время работы операции получается пропорциональной длине второго списка, то имеет смысл в качестве первого списка выбирать более длинный из двух объединяемых, а в качестве второго – менее длинный. Как мы увидим дальше, именно эта идея поможет нам добиться выигрыша.

```
Operation UNION (a, b)
  If L[a] < L[b]
    Then Begin
      s:= a;
      a:= b;
      b:= s;
    End;
  Next[Tail[a]]:= b;
  Tail[a]:= Tail[b];
  L[a]:= L[a]+L[b];
  x:= b;
  While x <> 0 Do Begin
    Pr[x]:= a;
    x:= Next[x];
  End;
End;
```

Как уже говорилось, время работы операции операции UNION есть O(min{L[a], L[b]}) и в худшем случае оно составляет O(N). Однако, как показывает следующая теорема, в среднем это время невелико (порядка O(logN)).

Теорема 1. Если система непересекающихся множеств реализована на основе списков, то после вызова INIT (N) суммарное время работы любой последовательности из M операций не превосходит O(NlogN+M).

Доказательство. Суммарное время работы всех операций FINDSET из этих M, очевидно, не превосходит O(M). Покажем, что суммарное время работы всех операций UNION не превзойдет O(NlogN). Так как, кроме времени, затрачиваемого на переписывание представителей, операция UNION работает за константное время, то достаточно показать, что значение Pr[i] может меняться не более чем O(logN) раз для каждого i. Рассмотрим момент, когда значение Pr[i] изменяется. Тогда объект i переходит из множества, состоящего из L[b] элементов в множество, в котором L[a]+L[b] элементов. Так как $L[a]+L[b] \geq 2L[b]$, то количество элементов в множестве, содержащем элемент i, увеличилось как минимум вдвое. Так как в множестве не может быть более N элементов, то значение Pr[i] не может изменяться более чем O(logN) раз. □

Как видим, полученная реализация достаточно эффективна. Дополнительным достоинством списочной реализации заключается то, что мы имеем возможность находить все элементы в заданном множестве за O(n), где n – количество элементов в этом множестве. В случае реализации на основе массива или леса (см. следующий раздел), для этого понадобилось бы O(N) времени.

в) Реализация на основе леса.

Еще более эффективная реализация получается, если хранить каждое элементы каждого множества в виде дерева. Тогда вся система непересекающихся множеств представляет собой набор деревьев – лес. Для хранения деревьев будем использовать массив P, смысл которого таков: если объект i является корнем какого-нибудь дерева, то P[i] = i, иначе P[i] содержит номер объекта, являющегося отцом объекта i. Представителем множества мы будем считать корень дерева, содержащего это множество.

Для реализации операции FINDSET нам необходимо подняться от объекта x до корня дерева, в котором он находится и возвратить найденное значение. Для реализации операции UNION необходимо провести ребро от корня одного из объединяемых деревьев к корню другого. Однако, если объединять деревья «как попало», то никакой выгоды от такой реализации мы не получим – деревья могут выродиться в цепочки и операция FINDSET будет работать очень долго. Логично, что нам следует стремиться минимизировать высоту получаемого при объединении дерева. Для этого мы применяем следующую эвристику (*объединение по рангам*). Введем массив рангов Rank[1..N]. Если i – корень некоторого дерева, то мы будем интерпретировать Rank[i] как оценку сверху для высоты дерева с корнем в i. Эта оценка была бы точной, если бы не еще одна эвристика, которую мы рассмотрим позже. Теперь операция UNION проводится следующим образом: если ранги объединяемых деревьев не равны, то мы проводим ребро от дерева с меньшим рангом к дереву с большим рангом и не меняем значения рангов; если же они равны, то мы проводим ребро произвольным образом, и увеличиваем ранг корня дерева, к которому мы провели ребро на 1. Таким образом, на данном этапе реализации Rank[i] равен высоте дерева с корнем i.

Объединение по рангам является полезной эвристикой, но, применяя только нее, мы получаем реализацию с производительностью, приблизительно совпадающей со списочной, то есть не добиваемся выигрыша. Рассмотрим теперь еще одну эвристику (*сжатие путей*). Смысл ее прост: во время выполнения операции FINDSET, после того как мы поднялись от объекта до корня дерева, содержащего объект, у всех объектов на пройденном пути мы можем поменять отца, и в качестве нового отца установить корень дерева. Это позволить ускорить работу последующих вызовов FINDSET для этих объектов. Приведем теперь реализацию всех операций.

```
Operation INIT (N)
  For i:= 1 to N Do Begin
    P[i]:= i;
    Rank[i]:= 0;
  End;
End;

Operation FINDSET (x)
  If x <> P[x]
    Then x:= FINDSET (P[x]);
  Return P[x];
End;

Operation UNION (a, b)
  If Rank[a] < Rank[b]
    Then P[a]:= b
    Else Begin
      P[b]:= a;
      If Rank[a] = Rank[b]
        Then Rank[a]:= Rank[a]+1;
    End;
End;
```

Очевидно, что время работы операции INIT есть O(N), а время работы операции UNION есть O(1). Что же касается времени работы операции FINDSET, то в худшем случае оно может оказаться равным даже O(N), но как показывает следующая теорема (которую мы приводим без доказательства), в среднем оно также невелико.

Теорема 2. Если система непересекающихся множеств реализована на основе списков, то после вызова INIT (N) суммарное время работы любой последовательности из M операций не превосходит O(Mlog*N), где log*N равен минимальному количеству раз взятия двоичного логарифма от числа N для получения числа, меньшего 1.

Следует заметить, что log*N - очень медленно растущая величина. В частности, $\log^*N \leq 5$ для всех $N < 2^{65536}$. Так что, в принципе, можно считать, что операция FINDSET в данной реализации выполняется в среднем за время, равное константе. Оценка в теореме 2 не является точной и может быть улучшена и, кроме того, доказано, что (при некоторых достаточно естественных предположениях) описанная реализация системы непересекающихся множеств является самой эффективной из всех возможных.

Упражнение 10 (*). Доказать теорему 2.

г) Пример.

Рассмотрим простой пример задачи, которую можно эффективно решать с помощью системы непересекающихся множеств. Задача BRIDGES, которую мы будем решать, звучит следующим образом. Предположим, что некоторая страна состоит из V островов, пронумерованных от 1 до V. Правительство страны решило соединить острова мостами, так чтобы с любого острова по мостам можно было попасть на любой. Для этого был разработан план строительства. По плану нужно было построить E мостов. Первый мост, который будет построен, соединит острова A_1 и B_1, второй мост – острова A_2 и B_2, и т.д. Гарантируется, что после того, как будут построены все E мостов, можно будет попасть с любого острова на любой, пользуясь мостами. Однако возможно, что это событие произойдет и раньше. Необходимо найти такое минимальное i, что после постройки моста между островами A_i и B_i можно будет попасть с любого острова на любой.

Для решения мы будем использовать систему непересекающихся множеств (способ реализации которой потом уточним). Каждое хранимое множество соответствует некоторому набору островов, такому, что в этом наборе можно пройти с любого острова на любой, пользуясь мостами. Изначально каждый остров находится в отдельном множестве. Когда мы рассматриваем очередной мост, то возможны два варианта. Если он соединяет два острова, находящиеся в одном множестве, то мы ничего не делаем. Иначе мы объединяем два множества, в которых находятся соединяемые мостом острова, в одно. Как только останется только одно множество, мы сможем пройти с любого острова на любой. Реализация алгоритма проста.

```
Algorithm BRIDGES
  INIT (V);
  X:= N; i:= 0;
  While X > 1 Do Begin
    i:= i+1;
    If FINDSET (A[i]) <> FINDSET (B[i]) Then Begin
      UNION (FINDSET (A[i]), FINDSET (B[i]));
      X:= X-1;
    End;
  End;
  OUTPUT (i);
End;
```

Сложность полученного решения оценим в зависимости от реализации системы непересекающихся множеств. Если использовать реализацию на основе массива, то она равна $O(V^2+E)$, на основе списков – $O(V\log V+E)$, на основе леса – $O(E\log^* V)$. Возникает вопрос: какую реализацию лучше выбрать? Как ни странно, в разных случаях лучшим выбором может оказаться каждая из реализаций. А именно, если величина E никак не ограничена сверху (кроме, естественно, очевидного ограничения $E \le V^2$), то реализации на основе массивов и списков будут иметь сложность $O(V^2)$, а реализация на основе леса – $O(V^2\log^* V)$. В данном случае нужно выбирать реализацию на основе массива или списков, и вследствие простоты лучше выбрать реализацию на основе массива. Пусть теперь известно, что величина E есть $o(V^2)$, но $\Omega(V\log V)$. Тогда реализация на основе массива по-прежнему имеет сложность $O(V^2)$, тогда как сложность списковой реализации есть $O(E)$, а реализации на основе леса – $O(E\log^* V)$. Здесь оптимальным выбором будет уже списковая реализация. Наконец, если известно, что E есть $o(V\log V)$, то, как нетрудно убедиться, оптимальной будет реализация системы непересекающихся множеств на основе леса.

n4) Структуры с одиночной модификацией.

В этом пункте мы рассматриваем структуры следующего типа. Предположим, что имеется таблица A[1..N], элементами которой, для простоты, будут числа. Наши структуры будут выполнять операции двух типов:

1) Модифицировать значение ячейки таблицы («модификация» означает изменение значения, смысл этого слова в каждой структуре будет уточняться).
2) Найти какую-нибудь числовую характеристику заданного интервала ячеек таблицы (этой характеристикой, к примеру, может быть сумма ячеек или минимальной значение ячейки на интервале; характеристика, опять же, для каждой структуры – своя).

Мы бы хотели, чтобы каждая из этих двух операций выполнялась за время $O(\log N)$.
Мы рассмотрим две структуры такого типа. Одна из них умеет находить сумму на интервале и модифицировать ячейки как угодно (уменьшать или увеличивать), вторая умеет находить минимум на интервале и уменьшать значения ячеек. Далее мы показываем, как модифицировать первую структуру для выполнения запросов в двумерной таблице (при этом интервал переходит в прямоугольник и время работы операций увеличивается до $O((\log N)^2)$). Наконец, в последнем разделе рассматриваются два примера применения описанных идей. Один из них показывает, что введенные структуры могут оказаться полезными для ускорения алгоритмов динамического программирования. Во втором из них разрабатывается еще одна структура описанного типа (основанная на аналогичных идеях), что показывает, что введенные идеи являются достаточно общими. Отметим, что названия структур, которые мы здесь используем (а также структур из пункта 5), не являются общеупотребительными.

а) Сумматор. Под сумматором мы будем понимать структуру с одиночной модификацией, умеющую выполнять следующие две операции:

MODIFY (pos, value) – модифицировать значение ячейки A[pos], увеличив его на value (то есть присвоить A[pos] значение A[pos]+value; value может иметь произвольный знак).
FINDSUM (l, r) – найти сумму значений ячеек в интервале индексов от l до r, то есть величину A[l]+A[l+1]+…+A[r-1]+A[r] (предполагается, что $l < r$).

К этим двум операциям естественно добавить еще одну, задающую начальные значения в таблице:

INIT (N) – заполнить таблицу A[1..N] нулями.

Упражнение 11. Предложите (идейно) простую реализацию сумматора, в которой операция MODIFY выполняется за O(1), а операция FINDSUM – за O(Sqrt(N)).

Перейдем теперь к реализации сумматора. Нам понадобятся некоторые дополнительные обозначения. Рассмотрим произвольное натуральное число X. Переведем его в двоичную запись: $X = (a_k a_{k-1} \ldots a_0)_2$. Будем идти по этой записи справа налево, пока не встретим единицу. Пусть это будет a_t. Таким образом, $a_t = 1$ и $a_i = 0$ для $i < t$. Пусть теперь NEXT (x) – это число, равное $X + 2^t$ (мы также будем говорить, что NEXT (x) получается из x добавлением крайнего справа бита), а PREV (x) – это число, равное $X - 2^t$ (мы будем говорить, что PREV (x) получается из x удалением крайнего справа бита). Величины NEXT (x) и PREV (x) будут очень важными во всех наших структурах данных, поэтому мы хотим уметь вычислять их быстро – за O(1). Ниже приведен один из вариантов такого вычисления.

```
Function PREV (x)
  Return x and (x-1)
End;

Function NEXT (x)
  Return (x shl 1) – (x and (x-1))
End;
```

Упражнение 12. Покажите, что приведенные алгоритмы корректно вычисляют значения NEXT (x) и PREV (x).

Как ни странно, в нашей реализации сумматора значения элементов таблицы A храниться не будут. Вместо этого мы будем использовать таблицу S[1..N]. Смысл элемента S[i] таков: S[i] равен сумме элементов таблицы A в интервале индексов от PREV (i) + 1 до i, то есть S[i] = A[PREV (i) + 1] + A[PREV (i) + 2] + … + A[i-1] + A[i]. В частности, если i – нечетно, то S[i] = A[i]. Операция INIT, как обычно, тривиальна. Время ее работы составляет O(N).

```
Operation INIT (N)
  For i:= 1 to N Do
    S[i]:= 0;
End;
```

Разберемся теперь, как выполнять операцию FINDSUM. Сначала мы рассмотрим частный случай, когда l = 1. Идея состоит в следующем. Рассмотрим значение S[r] – в нем хранится сумма ячеек от (Prev(r)+1) – ой до r – ой. Если разделить отрезок от 1 до r на две части: от 1 до PREV (r) и от PREV (r) + 1 до r, то получаем, что сумма значений на отрезке от 1 до r равна FINDSUM (1, PREV (r)) + S[r]. Таким образом, в данном случае мы можем воспользоваться рекурсией. Пусть теперь l > 1. Тогда для нахождения суммы значений на отрезке от l до r можно использовать то, что эта сумма равна разности FINDSUM (1, r) – FINDSUM (1, l-1), а случай l = 1 мы уже рассмотрели. Таким образом, получаем следующую реализацию операции FINDSUM.

```
Operation FINDSUM (l, r)
  If l > 1
    Then Return FINDSUM (1, r) – FINDSUM (1, l-1)
    Else
      If r > 0
        Then Return FINDSUM (1, PREV (r)) + S[r]
        Else Return 0;
End;
```

Не проводя пока временного анализа, перепишем операцию FINDSUM в более понятном нерекурсивном виде.

```
Operation FINDSUM (l, r)
  Sum:= 0;
  x:= r;
  While x > 0 Do Begin
    Sum:= Sum + S[x];
    x:= PREV (x);
  End;
  x:= l-1;
  While x > 0 Do Begin
    Sum:= Sum – S[x];
    x:= PREV (x);
  End;
  Return Sum;
End;
```

Время работы операции теперь оценить несложно. Во время работы каждого из циклов While у переменной x на каждой итерации количество единиц в двоичной записи уменьшается на 1. Так как перед каждым из циклов ее значение было не больше N, то и количество единиц в ее двоичной записи не превосходило O(logN). Итак, общее время работы операции FINDSUM есть O(logN).
Нам осталось только описать, как реализовать операцию MODIFY. Общая идея тут такова: нужно найти все такие x, что позиция pos лежит на отрезке [PREV (x) + 1, x] и изменить значения S[x] на value. Следующая теорема описывает все числа x, обладающие нужным нам свойством.

Теорема 3. Все числа x такие, что pos лежит на отрезке [PREV (x) + 1, x] можно построить следующим образом: $x_0 = pos$; $x_k = NEXT(x_{k-1})$, $k \geq 1$.
Доказательство. Понятно, что x_0 – минимальное число, обладающее нужным нам свойством. Далее заметим следующее, пусть pos лежит на отрезке [PREV (x_k) + 1, x_k] и крайний справа бит в двоичной записи x_k находится на позиции. Если рассмотреть произвольное x из отрезка [x_k+1, NEXT (x_k) –1], то у числа x крайний справа бит будет находиться на позиции, меньшей t, а значит PREV (x) ≥ x_k. Так как pos ≤ x_k, то pos не может находиться на отрезке [PREV (x) +1, x]. С другой стороны, если рассмотреть x_{k+1} = NEXT (x_k), то нетрудно видеть, что PREV (x_{k+1}) ≤ PREV (x_k) и поэтому pos будет находиться на отрезке [PREV (x_{k+1}) +1, x_{k+1}]. Таким образом, x_{k+1} – минимальное число, большее x_k, обладающее нужным нам свойством. ∎

Основываясь на доказанной теореме, нетрудно написать реализацию операции MODIFY.

```
Operation MODIFY (pos, value)
  x:= pos;
  While x <= N Do Begin
    S[x]:= S[x]+value;
    x:= NEXT (x);
  End;
End;
```

Временной анализ операции MODIFY также несложен. В цикле While позиция крайнего справа бита в числе x на каждой итерации увеличивается. Следовательно, через O(logN) операций x гарантированно станет больше, чем N. Таким образом, время работы операции MODIFY составляет O(logN).

Упражнение 13. Предположим, что в таблице A находятся некоторые числа, которые не меняются и необходимо эффективно выполнять в ней операцию FINDSUM. Предложите простой способ для

выполнения этой операции за O(1). Вам разрешается провести предварительную подготовку к выполнению запросов FINDSUM (например, вычислить какую-нибудь дополнительную информацию). Время на предварительную подготовку не должно превышать O(N).

б) Минимизатор. Под минимизатором будем понимать структуру данных, выполняющую две операции:

MODIFY (pos, value) – установить значение A[pos] равным min{A[pos], value}.
FINDMIN (l, r) – найти минимальное из значений в таблице с индексами от l до r, то есть min{A[l], A[l+1], …, A[r-1], A[r]}.

Сюда опять же добавляется инициализация таблицы:

INIT (N) – установить значения в таблице A[1..N], равными бесконечности.

Минимизатор является менее гибкой структурой, чем сумматор, так как в минимизаторе мы не можем увеличить значение ячейки таблицы. При разработке минимизатора мы могли бы поступить точно так же, как при разработке сумматора (с заменой суммы на минимум). Однако проблема заключается в том, что умение выполнять операцию FINDMIN (l, r) для l = 1 не позволяет нам выполнять эту операцию для произвольного l. Поэтому нам придется немного усложнить структуру.
Мы будем использовать три массива A[1..N], L[1..N], R[1..N]. Массив A будет содержать значения, находящиеся в таблице. Массив L будет аналогичен массиву S в сумматоре, а именно L[i] будет содержать минимальное из значений в таблице A на интервале индексов от PREV (i)+1 до i. Элементы же массива R будут следующими: значение R[i] равно минимальному из значений в таблице A на интервале индексов от i до NEXT (i) –1. В который уже раз мы замечаем, что написание операции INIT, работающей за O(N) тривиально.

```
Operation INIT (N)
  For i:= 1 to N Do Begin
    A[i]:= INFINITY;
    L[i]:= INFINITY;
    R[i]:= INFINITY;
  End;
End;
```

Операцию MODIFY реализовать также несложно. Обновление значения A[pos] очевидно. Обновление значений L и R аналогично обновлению значений S в сумматоре.

```
Operation MODIFY (pos, value)
  A[pos]:= min(A[pos], value);
  x:= pos;
  While x <= N Do Begin
    L[x]:= min(L[x], value);
    x:= NEXT (x);
  End;
  x:= pos;
  While x > 0 Do Begin
    R[x]:= min(R[x], value);
    x:= PREV (x);
  End;
End;
```

Рассуждая, как и при анализе операций сумматора, получаем, что время работы операции MODIFY составляет O(logN).
Реализация операции FINDMIN основывается на следующей теореме, доказательство которой несложно и оставляется на упражнение.

Теорема 4. Рассмотрим отрезок [l, r]. Пусть $x_0 = l$. Будем вычислять $x_k = NEXT(x_{k-1})$ и остановимся, как только очередное $x_t > r$. Пусть $y_0 = r$. Будем вычислять $y_k = PREV(y_k)$ и остановимся, как только очередное $y_p < l$. Тогда $x_{t-1} = y_{p-1}$.

Упражнение 14. Доказать теорему 4.

Теперь можно сделать следующее. Разобьем отрезок [l, r] на три части: $[l, x_{t-1}-1]$, $[x_{t-1}, y_{p-1}]$, $[y_{p-1}+1, r]$. Для нахождения минимума на первом отрезке будем использовать значения R, для нахождения минимума на третьем отрезке будем использовать значения L, второй отрезок состоит только из одного элемента, значение которого мы можем взять из таблицы A. Общий минимум на отрезке [l, r] будет равен минимуму из трех найденных минимумов.

```
Operation FINDMIN (l, r)
  Res:= INFINITY;
  x:= l;
  While NEXT (x) <= r Do Begin
    Res:= min(Res, R[x]);
    x:= NEXT (x);
  End;
  Res:= min(Res, A[x]);
  x:= r;
  While PREV (x) >= l Do Begin
    Res:= min(Res, L[x]);
    x:= PREV (x);
  End;
  Return Res;
End;
```

Нетрудно видеть, что время работы операции FINDMIN составляет O(logN).

в) Двумерный сумматор.

В этом пункте мы переносим идеи, заложенные в основе сумматора, на двумерный случай. В этом разделе, мы будем работать с двумерной таблицей A[1..N, 1..N] и выполнять в ней две операции:

MODIFY (x, y, value) – увеличить значение A[x, y] на value (знак value может быть любым).
FINDSUM (xl, yl, xr, yr) – найти сумму значений в прямогольной области ячеек таблицы A с левым верхним углом в ячейке A[xl, yl] и правым нижним углом в ячейке A[xr, yr].

Как обычно таблица инициализируется нулями при помощи операции INIT (N).
Как и в сумматоре, мы не храним значения в таблице A, а вместо этого используем таблицу S[1..N, 1..N], где S[i, j] равно сумме значений в прямоугольной области таблицы с левым верхним углом в ячейке S[PREV (i) + 1, PREV (j) + 1] и правым нижним углом в ячейке S[i, j]. Выполнение операции MODIFY аналогично одномерному случаю, но теперь уже появляется два вложенных цикла While, из-за чего временная оценка возрастает до $O((\log N)^2)$. Выполнение операции FINDSUM для частного случая, когда xl = 1 и yl = 1 также аналогично одномерному случаю, и

опять появляются два вложенных цикла While и временная оценка возрастает до $O((\log N)^2)$. Если обозначить значение, возвращаемое операцией FINDSUM (xl, yl, xr, yr) через Sum[xl, yl, xr, yr], то выполняется тождество Sum[xl, yl, xr, yr] = Sum[1, 1, xr, yr] – Sum[1, 1, xl-1, yr] – Sum[1, 1, xr, yl-1] + Sum[1, 1, xl-1, yl-1], на основе которого операция FINDSUM реализуется в общем случае. Операция INIT (N) заполняет значения S нулями и работает за $O(N^2)$. Более подробное обоснование всего вышесказанного оставляется на упражнение.

```
Operation INIT (N)
  For i:= 1 to N Do
    For j:= 1 to N Do
      S[i, j]:= 0;
End;

Operation MODIFY (x, y, value)
  i:= x;
  While i <= N Do Begin
    j:= y;
    While j <= N Do Begin
      S[i, j]:= S[i, j] + value;
      j:= NEXT (j);
    End;
    i:= NEXT (i);
  End;
End;

Operation FINDSUM (xl, yl, xr, yr)
  If (xl = 1) and (yl = 1)
    Then Begin
      i:= xl;
      While i > 0 Do Begin
        j:= yl;
        While j > 0 Do Begin
          Sum:= Sum + S[i, j];
          j:= PREV (j);
        End;
        i:= PREV (i);
      End;
      Return Sum;
    End
    Else Return FINDSUM (1, 1, xr, yr) + FINDSUM (1, 1, xl-1, yl-1) –
                FINDSUM (1, 1, xl-1, yr) – FINDSUM (1, 1, xr, yl-1);
End;
```

Упражнение 15. Обосновать правильность приведенной реализации двумерного сумматора.

г) Примеры.

Пример 1. Цель этого примера – показать, что рассмотренный тип структур с одиночной модификацией часто помогает ускорять алгоритмы, основанные на динамическом программировании. Мы рассматриваем классическую задачу LIS о наибольшей возрастающей подпоследовательности и приводим ее решение, работающее за O(NlogN). Хоть задача LIS хорошо изучена и известы другие ее решения (отличные от описываемого), имеющие такую же временную

оценку, но наш подход нетрудно перенести на многие другие задачи (естественно, далеко не на все).
Задача LIS заключается в следующем: задано N чисел X_1, X_2, …, X_N. Необходимо вычеркнуть минимальное количество чисел так, чтобы оставшиеся шли по возрастанию. Другими словами, нужно выделить из последовательности из N чисел подпоследовательность максимальной длины, в которой элементы идут по возрастанию. Мы будем искать только длину искомой подпоследовательности.
Без ограничения общности мы считаем, что X_i – целые числа и $1 \le X_i \le N$. Если это не так, то можно заменить каждое число X_i на количество чисел X_j таких, что $X_j \le X_i$.

Упражнение 16. Покажите, что такая замена не изменяет ответа в задаче. Опишите, как произвести подобную замену за O(NlogN).

Рассмотрим решение задачи LIS, основанное на динамическом программировании, работающее (при тривиальной реализации), за $O(N^2)$. Обозначим через F_i длину возрастающей подпоследовательности максимальной длины, последним элементом которой является X_i. Нетрудно видеть, что если мы знаем все значения F_i, то ответ на поставленную задачу равен $\max\{F_1, F_2, \ldots, F_N\}$. Далее, если мы знаем значения $F_1, F_2, \ldots, F_{i-1}$, то значение F_i можно вычислить следующим образом: $F_i = \max\{F_j \mid X_j < X_i, 1 \le j < i\}+1$ (здесь мы полагаем, что максимум по пустому множеству равен 0). Смысл этого соотношения заключается в следующем: мы пробуем в качестве предпоследнего элемента подпоследовательности, последним элементом которой является X_i, брать всевозможные X_j (естественно, что $j < i$ и $X_j < X_i$, иначе мы не получим подпоследовательность, в которой элементы возрастают). Очевидно, что максимальная длина возрастающей подпоследовательности, последним элементом которой является X_i, а предпоследним – X_j, равна F_j+1, поэтому наше соотношение, перебирая все возможные варианты для предпоследнего элемента и, выбирая максимальное значение из F_j+1, корректно вычисляет F_i. Однако описанный способ вычисления F_i работает за O(N) и нам понадобится порядка $O(N^2)$ операций для вычисления всех значений F_i.
Для ускорения решения задачи мы будем использовать максимизатор – структуру, находящую максимум на интервале, абсолютно аналогичную минимизатору (только инициализируемую не бесконечностями, а нулями). Опишем теперь, что мы будем хранить в таблице A. Элемент таблицы A[i] будет равен $\max\{F_j \mid X_j = i\}$ (мы берем максимум только по тем значениям F_j, которые уже вычислены). Очевидно, после того, как все значения F_i вычислены, ответ в задаче может быть получен как $\max\{A[1], A[2], \ldots, A[N]\}$. Кроме того, $F_i = \max\{F_j \mid X_j < X_i, 1 \le j < i\}+1 = \max\{A[1], A[2], \ldots, A[X_i-1]\}+1$ и для вычисления F_i мы можем использовать операцию FINDMAX. Алгоритм запишется следующим образом:

```
Algorithm LIS
  INIT (N);
  For i:= 1 to N Do Begin
    F[i]:= FINDMAX (1, X[i]-1);
    MODIFY (X[i], F[i]);
  End;
  OUTPUT (FINDMAX (1, N));
End;
```

Массив для хранения значений F в данном случае излишен и без него можно было бы обойтись. Сложность полученного алгоритма, очевидно, равна O(NlogN).

Упражнение 17. Модифицируйте алгоритм LIS (не меняя времени работы) так, чтобы он находил саму возрастающую подпоследовательность максимальной длины (а не только ее длину).

Пример 2. В этом примере мы рассматриваем, как можно написать систему управления свободными местами на автостоянке. Предположим, что на автостоянке есть N парковочных мест, расположенных на прямой слева направо и пронумерованных от 1 до N. Автомобиль может подъезжать к стоянке около некоторого парковочного места и двигаться вправо, пока не найдет свободное место, на которое он и паркуется (для простоты, будем полагать, что свободное место всегда найдется). Наша система будет уметь выполнять три операции:

INIT (N) – инициализирует систему (все парковочные места свободны);
ADDAUTO (pos) – операция возвращает номер места, на котором припаркуется автомобиль, который подъехал к позиции pos и обновляет информацию о занятости парковочных мест;
DELETEAUTO (pos) – операция обновляет информацию о занятости парковочных мест, помечая, что автомобиль, стоявший на месте pos уехал (то есть место pos стало свободным).

Мы хотим, чтобы операции ADDAUTO и DELETEAUTO работали за O(logN). Мы будем базироваться на идеях, заложенных в минимизаторе. Будем хранить три массива A[1..N], L[1..N], R[1..N]. Значение A[i] = 0, если i-е место свободно, и A[i] = 1, если i-е место занято. Значение L[i] будет равно количеству занятых мест на интервале [PREV(i)+1, i], а значение R[i] будет равно количеству занятых мест на интервале [i, NEXT (i)-1]. Для инициализации мы просто заполняем все массивы нулями.

```
Operation INIT (N)
  For i:= 1 to N Do Begin
    A[i]:= 0;
    L[i]:= 0;
    R[i]:= 0;
  End;
End;
```

Рассмотрим теперь вспомогательную операцию MODIFY (pos), которая освобождает место pos, если оно было занято и занимает его, если оно было свободно. Ее реализация абсолютно аналогична реализации операции MODIFY в минимизаторе.

```
Operation MODIFY (pos)
  If A[pos] = 1
    Then value:= -1
    Else value:= 1;
  A[pos]:= A[pos]+value;
  x:= pos;
  While x <= N Do Begin
    L[x]:= L[x]+value;
    x:= NEXT (x);
  End;
  x:= pos;
  While x > 0 Do Begin
    R[x]:= R[x]+value;
    x:= PREV (x);
  End;
End;
```

Как ни странно, но операция DELETEAUTO оказывается частным случаем операции MODIFY.

```
Operation DELETEAUTO (pos)
```

```
  MODIFY (pos);
End;
```

Нам остается только реализовать операцию ADDAUTO. Общая идея тут такова: сначала мы положим x = pos. Затем будем двигаться вправо (заменяя x на NEXT (x)), пока не окажется, что отрезок [x, x + NEXT (x) – 1] содержит свободное место. Если место x свободно, то мы нашли ответ. Иначе мы положим шаг движения h равным NEXT(x)-x. Далее мы будем уменьшать шаг движения (каждый раз в два раза), сохраняя следующий инвариант: место x занято, но на отрезке [x+1, x+h] есть свободное место. Когда шаг движения станет равным 1, значение x+1 и будет ответом. Нам останется только пометить с помощью MODIFY, что место x+1 теперь занято.

```
Operation ADDAUTO (pos)
  x:= pos;
  While R[x] = NEXT (x) – x Do
    x:= NEXT (x);
  If A[x] = 1 Then Begin
    h:= NEXT (x) – x;
    While h > 1 Do Begin
      h:= h shr 1;
      If L[x+h] = h
        Then x:= x+h;
    End;
    x:= x+1;
  End;
  MODIFY (x);
  Return x;
End;
```

Упражнение 18. Модифицируйте описанные алгоритмы для случая, когда автостоянка является кольцевой (то есть, если ехать вправо, за последним парковочным местом идет первое).

n5) Структуры с интервальной модификацией.

В предыдущем пункте мы рассмотрели несколько структур данных, в которых операция MODIFY изменяла значение ровно одного элемента таблицы A. В этом пункте мы захотим большего – потребуем, чтобы операция модификации изменяла значения целого интервала ячеек таблицы A.
Мы рассмотрим две структуры такого рода. Одна из них позволяет находить максимальное число на интервале ячеек и увеличивать (и уменьшать) значения ячеек заданного интервала на заданное число. Вторая структура позволяет находить количество нулевых ячеек на интервале ячеек таблицы (в предположении, что значения ячеек неотрицательны) и также увеличивать (и уменьшать) значения ячеек заданного интервала на заданное число. В заключение, мы иллюстрируем применение каждой из этих структур на примере.

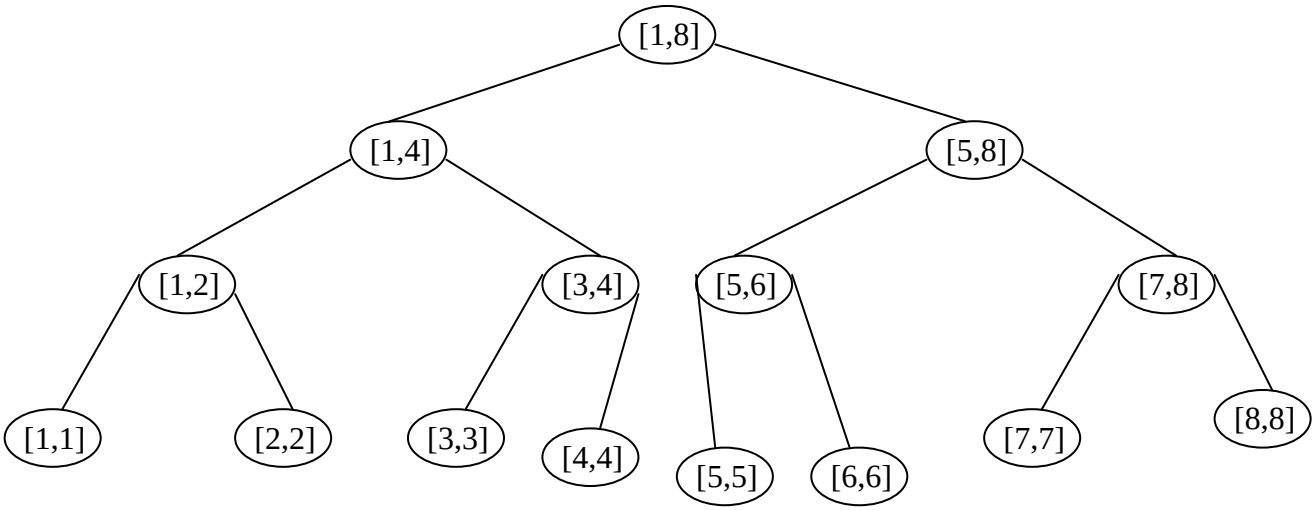
а) Дерево максимумов.

Дерево максимумов – это структура данных, которая позволяет выполнять следующие операции:

FINDMAX (l, r) – найти максимальное число на интервале ячеек таблицы от A[l] до A[r] (всюду дальше этот интервал обозначается через [l, r]).
MODIFY (l, r, value) – увеличить каждое из чисел на интервале ячеек таблицы от [l, r] на value (возможно, что value < 0).

К этим операциям добавляется инициализация таблицы A[1..N] нулями при помощи операции INIT (N).

Информацию о значениях таблицы A мы будем хранить в виде двоичного дерева. Каждая вершина дерева будет хранить информацию о некотором интервале ячеек таблицы A. Корень дерева будет соответствовать всей таблице, то есть интервалу [1, N]. Два сына корня соответствуют двум интервалам в два раза меньшей длины: [1, N/2] и [N/2+1, N] и т. д. Листьям дерева соответствуют интервалы самой маленькой длины – содержащие всего одну ячейку. Общее правило тут таково: если некоторой вершине соответствует интервал [l, r], то ее левому сыну соответствует интервал [l, (l+r) div 2], а ее правому сыну – интервал [(l+r) div 2+1, r]. Рисунок показывает, как будет выглядеть дерево при N = 8.



Опишем теперь, какую информацию мы будем хранить в каждой вершине дерева. Каждая вершина дерева будет представлять собой запись с 6-ю полями: l, r – левый и правый концы интервала, который ей соответствует; Left, Right – указатели на левого и правого сына вершины (равные NIL, если вершина является листом); Max, Add – еще две числовые величины, точный смысл которых мы проясним чуть позже (при инициализации они заполняются нулями). Естественно, что нам необходим для работы указатель на корень дерева, который мы будем хранить в переменной Root. Теперь мы можем представить операцию INIT, которая в отличие от предыдущих разделов, уже не так тривиальна. Операция INIT делает следующее: рекурсивно создает интересующее нас дерево и заполняет поля Max и Add нулями.

```
Operation INIT (N)
  Root:= CREATETREE (1, N);
End;

Function CREATETREE (l, r)
  New(X);
  X^.l:= l;
  X^.r:= r;
  X^.Max:= 0;
  X^.Add:= 0;
  If l < r
    Then Begin
      X^.Left:= CREATETREE (l, (l+r) div 2);
      X^.Right:= CREATETREE ((l+r) div 2+1, r);
    End
```

```
  Else Begin
    X^.Left:= NIL;
    X^.Right:= NIL;
  End;
  Return X;
End;
```

Вспомогательная рекурсивная функция CREATETREE строит дерево, корень которого соответствует интервалу [l, r], и возвращает указатель на корень построенного дерева. Для оценки сложности (а заодно и памяти, требуемой для хранения дерева) нам понадобится следующее простое свойство, которое нетрудно доказать по индукции.

Свойство 1. Если вершина дерева соответствует отрезку [l, r], то ее поддерево содержит 2(r-l)+1 вершину. В частности, все дерево состоит из 2N-1 вершины.

Отсюда сразу же получаем, что для хранения дерева нам потребуется O(N) байт памяти. Далее, так как в каждом вызове CREATETREE мы создаем за константное время новую вершину, то количество вызовов CREATETREE будет равно 2N-1, и сложность операции INIT оказывается равной O(N).

Теперь настало время описать поля Max и Add. Рассмотрим произвольную вершину дерева X, которая соответствует отрезку [l, r]. Пусть путь от корня дерева до X состоит из k+1 вершины X_1 (это корень), X_2, …, X_k, X. Мы будем все время поддерживать следующий инвариант: максимальное значение в ячейках таблицы на интервале [l, r] равно X_1.Add+X_2.Add+…+X_k.Add+X.Add+X.Max (мы далее обозначаем эту сумму через S(X)). Нетрудно видеть, что при инициализации этот инвариант выполняется, так как максимум на любом интервале равен 0.

Рассмотрим реализацию операции MODIFY. Она будет рекурсивной. Рекурсия будет начинаться с корня и углубляться внутрь дерева (но просматривать далеко не все его вершины). Будем передавать в рекурсивную процедуру следующие параметры: X – вершина, для которой мы вызываем рекурсию, l, r – концы интервала, который мы модифицируем, value – величина, на которую нужно увеличить значения элементов интервала. Будем сохранять следующее свойство: интервал [l, r] (который мы модифицируем) является подинтервалом (или совпадает) интервала [X^.l, X^.r] (который соответствует рассматриваемой вершине). Для самого верхнего вызова (когда X – корень) это свойство, очевидно выполняется. Рассмотрим теперь произвольный вызов. Возможны два варианта. Первый – интервал [l, r] совпадает с интервалом [X^.l, X^.r]. Ситуация здесь такова: если увеличить все значения интервала [l, r] на value, то и максимальное значение на интервале увеличится на value. Чтобы отметить это, мы могли бы увеличить значение X^.Max на value. Но это увеличение изменит только величину S(X), и оставит значение S неизменным для всех вершин из поддерева вершины X. Но все вершины из поддерева вершины X соответствуют подинтервалам интервала [l, r], следовательно, максимальное значение на каждом из этих интервалов также увеличится на value. Поэтому мы увеличиваем величину X^.Add на value, тем самым увеличивая значение S на value во всех вершинах из поддерева вершины X. Второй вариант – интервал [l, r] является строгим подинтервалом интервала [X^.l, X^.r]. Тогда мы поступаем следующим образом: рассматриваем интервал [l, r]∩[X^.Left^.l, X^.Left^.r] (если он не пуст) и запускаем рекурсию для него и левого сына вершины X, и, аналогично, рассматриваем интервал [l, r] ∩[X^.Right^.l, X^.Right^.r] (если он не пуст) и запускаем рекурсию для него и правого сына вершины X. После этого нужно проследить чтобы значение S(X) по-прежнему совпадало с максимальным значением на отрезке [X^.l, X^.r] (которое могло измениться). Это мы сделаем следующим образом: максимальное значение на отрезке [X.l, X.r] равно максимуму из максимальных значений на отрезках [X^.Left^.l, X^.Left^.r] и [X^.Right^.l, X^.Right^.r], которые в

свою очередь равны S(X^.Left) и S(X^.Right). Так как S(X^.Left)-S(X) = X^.Left^.Max+X^.Left^.Add – X^.Max и S(X^.Right)-S(X) = X^.Right^.Max+X^.Right^.Add-X^.Max, то, устанавливая X^.Max равным max(X^.Left^.Max+X^.Left^.Add, X^.Right^.Max+X^.Right^.Add), мы получим, что одно из значений S(X^.Left)-S(X) и S(X^.Right)-S(X) равно нулю, а второе – меньше или равно нуля. Тем самым, S(X) = max(S(X^.Left), S(X^.Right)), чего мы и добивались.

```
Operation MODIFY (l, r, value)
  RECMODIFY (Root, l, r, value)
End;

Procedure RECMODIFY (X, l, r, value)
  If (l = X^.l) and (r = X^.r)
    Then X^.Add:= X^.Add+value
    Else Begin
      If l <= X^.Left^.r
        Then RECMODIFY (X^.Left, l, min(X^.Left^.r, r), value);
      If r >= X^.Right^.l
        Then RECMODIFY (X^.Right, max(X^.Right^.l, l), r, value);
      X^.Max:= max(X^.Left^.Max+X^.Left^.Add, X^.Right^.Max+X^.Right^.Add);
    End;
End;
```

Самое удивительное заключается в том, что полученная реализация операции MODIFY имеет сложность O(logN), как показывает следующая теорема.

Теорема 5. При выполнении операции MODIFY общее количество вызовов RECMODIFY есть O(logN).

Доказательство. Для простоты рассмотрим частный случай, когда N является степенью двойки. В общем случае рассуждения аналогичны.

Мы будем использовать индукцию. Для начала докажем следующий результат: если был произведен вызов RECMODIFY(X, l, r, value) и X^.l = l, то общее количество вызовов RECMODIFY, которые произойдут, не превзойдет $2\log_2(X^.r-X^.l+1)+1$. Если X – лист, то утверждение, очевидно, верно. Иначе возможны два варианта. Первый состоит в том, что $r \le X^.Left^.r$ и рекурсия для правого поддерева не будет вызываться. Тогда общее количество вызовов, по индукции не превзойдет $2\log_2(X^.Left^.r-X^.Left^.l+1)+2 = 2(\log_2(X^.r-X^.l+1)-1)+2 = 2\log_2(X^.r-X^.l+1) \le 2\log_2(X^.r-X^.l+1)+1$. Второй вариант получаем, если $r > X^.Left^.r$. Но тогда вызов рекурсии для левого поддерева не повлечет новых рекурсивных вызовов (там совпадут интервалы [l, r] и [X^.l, X^.r]), и по индукции получаем, что общее количество вызовов не превзойдет $2\log_2(X^.Right^.r-X^.Right^.l+1)+3 \le 2\log_2(X^.r-X^.l+1)+1$.

Абсолютно аналогично получаем, что, если был произведен вызов RECMODIFY(X, l, r, value) и X^.r = r, то общее количество вызовов RECMODIFY, которые произойдут, также не превзойдет $2\log_2(X^.r-X^.l+1)$.

Теперь мы готовы рассмотреть общий случай. Докажем, что, если был произведен вызов RECMODIFY(X, l, r, value), то общее количество вызовов RECMODIFY, которые произойдут, не превзойдет $4\log_2(X^.r-X^.l+1)+1$. Если X – лист, то утверждение верно. Иначе возможны два варианта. Если отрезок [l, r] целиком попадает внутрь одного из отрезков [X^.Left^.l, X^.Left^.r] или [X^.Right^.l, X^.Right^.r], то произойдет только один рекурсивный вызов и по индукции будем иметь, что общее количество вызовов не превзойдет $4(\log_2(X^.r-X^.l+1)-1)+4 \le 4\log_2(X^.r-X^.l+1)+1$. Иначе при вызове RECMODIFY для левого поддерева выполнится X^.r = r, а при вызове RECMODIFY для правого поддерева выполнится X^.l = l и в силу доказанных утверждений, общее количество вызовов не превзойдет $2(2(\log_2(X^.r-X^.l+1)-1)+1)+1 \le 4\log_2(X^.r-X^.l+1)+1$.

Отсюда получаем, что при выполнении MODIFY после вызова RECMODIFY (Root, l, r, value) произойдет не более чем $4\log_2 N+3 = O(\log N)$ вызовов. ↑

Нам осталось рассмотреть только реализацию операции FINDMAX. Она также будет рекурсивной и идейно аналогичной MODIFY (даже еще проще). Мы передаем в рекурсию еще один дополнительный параметр SumAdd, равный сумме значений Add на пути от корня до вершины X. Тогда максимум на интервале [X^.l, X^.r] равен S(X) = SumAdd+X^.Max, чем мы и пользуемся.

```
Operation FINDMAX (l, r)
  Return RECFINDMAX (Root, l, r, Root^.Add);
End;

Function RECFINDMAX (X, l, r, SumAdd)
  If (l = X^.l) and (r = X^.r)
    Then Return SumAdd+X^.Max
    Else Begin
      Res:= -INFINITY;
      If l <= X^.Left^.r
        Then Res:= max(Res, RECFINDMAX (X^.Left, l, min(X^.Left^.r, r), SumAdd+X^.Left^.Add);
      If r >= X^.Right^.l
        Then Res:= max(Res, RECFINDMAX (X^.Right, max(X^.Right^.l, l), r, SumAdd+X^.Right^.Add);
      Return Res;
    End;
End;
```

Рассуждая точно так же, как при доказательстве Теоремы 5, получаем, что время работы FINDMAX есть O(logN). Таким образом, мы получили очень мощную структуру данных. В частности, она обладает всеми возможностями максимизатора (см. предыдущий пункт) и даже гораздо большими возможностями (к примеру, умеет уменьшать значения ячеек). При этом временные оценки для операций у нее точно такие же, как и у максимизатора. Возникает вопрос: зачем тогда нужно было разрабатывать максимизатор? Ответ заключается в константах, заключенных в понятии O. Хоть, ассимптотические оценки для используемых времени и памяти у максимизатора и дерева максимумов совпадают, но дерево максимумов использует в 4 раза (если убрать излишние значения l и r у каждой вершины, то в 8/3 раза) больше памяти и работает также в несколько раз медленнее, чем максимизатор. К тому же реализация всех операций для максимизатора (приведенная нами) занимает 668 байт, тогда как реализация операций для дерева максимумов занимает 1350 байт (в два раза больше). Поэтому не стоит использовать дерево максимумов там, где можно ограничиться максимизатором.

б) Дерево отрезков.

В этом разделе нас будет интересовать структура, обладающая следующими операциями:

MODIFY (l, r, value) – увеличить все значения на отрезке ячеек [l, r] на value (value может быть < 0, но после увеличения все значения должны остаться неотрицательными)
FINDCOVER (l, r) – найти количество ячеек из отрезка [l, r], равных 0.

Как обычно, мы инициализируем таблицу A нулями при помощи операции INIT (N).
Такая структура данных может быть интерпретирована как хранилище отрезков. При этом ячейка A[i] хранит количество отрезков, покрывающих клетку i. Операция MODIFY при value > 0 добавляет value новых отрезков [l, r], а при value < 0 удаляет value отрезков [l, r] (при этом не обязательно удалять те отрезки, которые мы когда-то добавляли, главное – не удалить отрезков

больше, чем их есть на самом деле). Операция FINDCOVER возвращает количество ячеек отрезка [l, r], не покрытых отрезками.
Мы реализуем эти две операции за O(logN). Все, что нам понадобится – немного усложнить дерево максимумов. Теперь в вершине дерева вместо поля Max появляется поле Min и S(X) уже равно минимальному значению на отрезке X. Тем самым дерево максимумов превращается в дерево минимумов. Далее мы добавляем еще один параметр Count, который равен количеству минимальных (то есть равных S(X)) элементов отрезка [X^.l, X^.r]. Пересчет этого параметра несложен. Операция FINDCOVER, реализованная аналогично FINDMAX в дереве максимумов, может существенно использовать этот параметр, а именно, в случае, когда отрезки [l, r] и [X^.l, X^.r] совпадают, то, если S(X) > 0, то на отрезке [X^.l, X^.r] нет нулевых элементов, а, если S(X) < 0, то на отрезке [X^.l, X^.r] ровно Count нулевых элементов. Мы приводим сразу полную реализацию операций для дерева отрезков, а более подробные объяснения оставляем на упражнения.

```
Operation INIT (N)
  Root:= CREATETREE (1, N);
End;

Function CREATETREE (l, r)
  New(X);
  X^.l:= l;
  X^.r:= r;
  X^.Min:= 0;
  X^.Add:= 0;
  X^.Count:= r-l+1;
  If l < r
    Then Begin
      X^.Left:= CREATETREE (l, (l+r) div 2);
      X^.Right:= CREATETREE ((l+r) div 2+1, r);
    End
    Else Begin
      X^.Left:= NIL;
      X^.Right:= NIL;
    End;
    Return X;
End;

Operation MODIFY (l, r, value)
  RECMODIFY (Root, l, r, value)
End;

Procedure RECMODIFY (X, l, r, value)
  If (l = X^.l) and (r = X^.r)
    Then X^.Add:= X^.Add+value
    Else Begin
      If l <= X^.Left^.r
        Then RECMODIFY (X^.Left, l, min(X^.Left^.r, r), value);
      If r >= X^.Right^.l
        Then RECMODIFY (X^.Right, max(X^.Right^.l, l), r, value);
      X^.Min:= min(X^.Left^.Min+X^.Left^.Add, X^.Right^.Min+X^.Right^.Add);
      X^.Count:= 0;
      If X^.Min = X^.Left^.Min+X^.Left^.Add
```

```
      Then X^.Count:= X^.Count+X^.Left^.Count;
    If X^.Min = X^.Right^.Min+X^.Right^.Add
      Then X^.Count:= X^.Count+X^.Right^.Count;
  End;
End;

Operation FINDCOVER (l, r)
 Return RECFINDCOVER (Root, l, r, Root^.Add);
End;

Function RECFINDCOVER (X, l, r, SumAdd)
 If (l = X^.l) and (r = X^.r)
  Then
    If X^.Min+SumAdd = 0
      Then Return X^.Count
      Else Return 0;
  Else Begin
    Res:= 0;
    If l <= X^.Left^.r
      Then Res:= Res + RECFINDCOVER (X^.Left, l, min(X^.Left^.r, r), SumAdd+X^.Left^.Add);
    If r >= X^.Right^.l
      Then Res:= Res + RECFINDCOVER (X^.Right, max(X^.Right^.l, l), r, SumAdd+X^.Right^.Add);
    Return Res;
  End;
End;
```

Упражнение 19. Объясните подробно, почему описанная реализация дерева отрезков работает правильно.

в) Примеры.

Пример 1. Рассмотрим следующую задачу RECTANGLES. На плоскости задано N прямоугольников. Координата левого нижнего угла i-го прямоугольника равна (XL_i, YL_i), а координаты правого нижнего угла – (XR_i, YR_i) ($XL_i < XR_i$, $YL_i < YR_i$). При этом эти координаты неотрицательны и достаточно невелики ($0 \le XL_i, XR_i, YL_i, YR_i \le C$). Необходимо вычислить площадь, покрываемую объединением всех этих прямоугольников. Мы опишем алгоритм, решающий эту задачу за O(N(logN+logC)+C), использующий O(N+C) байт памяти.

Мы применяем достаточно общую идею, которая состоит в следующем: рассмотрим вертикальную прямую, изначально находящуюся где-то далеко слева и начнем двигать ее вправо. В каждый момент времени мы будем хранить информацию о том, какие части прямой покрываются прямоугольниками, а какие – нет. Эту информацию нужно обновлять в двух случаях: когда прямая достигает левого края i-го прямоугольника, нужно добавить отрезок [YL_i, YR_i] на прямую, и наоборот, когда прямая достигает правого края i-го прямоугольника, нужно удалить отрезок [YL_i, YR_i] с прямой. Если на каком-то участке длины Δx состояние прямой не меняется, то площадь, покрываемая прямоугольниками на этом участке, равна $L\Delta x$, где L – длина части прямой, покрытой отрезками.

Состояние дел на прямой мы будем хранить, используя дерево отрезков. В ячейке A[i] будет храниться количество отрезков, покрывающих отрезок [i-1, i] прямой. Мы также используем массив Event событий, которые должны произойти во время движения прямой. Каждое событие характеризуется четырьмя параметрами: x – абсцисса, на которой происходит это событие; y_1, y_2 – координаты концов добавляемого на прямую (или удаляемого с прямой) отрезка; value – величина, которая равна 1, если отрезок добавляется, и равна –1, если отрезок удаляется. Переменная

EventCnt (равная, впрочем, 2N) обозначает общее количество событий. Как уже отмечалось выше, каждый прямоугольник порождает два события. У одного из них $x = XL_i$, $y_1 = YL_i$, $y_2 = YR_i$, value = 1, а у второго – $x = XR_i$, $y_1 = YL_i$, $y_2 = YR_i$, value = -1. Мы предполагаем, что массив событий уже отсортирован по полю x (такая сортировка может быть произведена за время O(NlogN)). Тогда алгоритм может быть записан следующим образом:

```
Algorithm RECTANGLES
  S:= 0;
  INIT (C);
  For i:= 1 to EventCnt-1 Do Begin
    MODIFY (Event[i].y1+1, Event[i].y2, Event[i].value);
    S:= S + (Event[i+1].x - Event[i].x) * (C – FINDCOVER (1, C));
  End;
  OUTPUT (S);
End;
```

Пример 2. Рассмотрим следующую задачу MAXSQUARE. Предположим, что имеется таблица T[1..C, 1..C]. Почти все элементы таблицы равны нулю, за исключением N ячеек $T[X_1, Y_1]$, $T[X_2, Y_2]$, …, $T[X_N, Y_N]$, которые равны 1. Далее мы считаем, что ячейка T[1,1] – левый нижний угол таблицы, а ячейка T[C, C] – правый верхний. Необходимо найти квадратную область таблицы со стороной в t ячеек, сумма элементов в которой максимально возможна (естественно, что $t \le C$). Уточним входные данные в этой задаче: это величины C, t, N, X_1, Y_1, …, X_N, Y_N (полностью значения в таблице не задаются, достаточно задать только ненулевые значения). В качестве ответа будем выводить сумму ячеек таблицы в найденной области. Мы описываем алгоритм решения задачи, время работы которого равно O((N+C)(logN+logC)), а расходы памяти составят O(N+C) байт.

Для решения мы используем вспомогательный массив списков List[1..C]. Ячейка List[x] содержит список всех Y_i, таких, что $X_i = x$ (то есть, список ординат всех единичных ячеек таблицы T с заданной абсциссой). Построить списки List несложно за время O(NlogN+C), и они занимают O(N+C) байт памяти. Идея решения такова: мы рассматриваем интервал абсцисс [x, x+t-1], начиная с x = 1 и постепенно увеличивая x. Кроме того мы поддерживаем таблицу A[1..C] (точнее, поддерживаем информацию о ней при помощи дерева максимумов), в которой элемент A[y] равен сумме ячеек в квадратной области со стороной длины t с левым нижним углом в ячейке T[x, y] (для текущего значения x). Если x фиксировано, то с помощью запроса FINDMAX (1, C) мы можем найти максимальную сумму ячеек во всех квадратных областях со стороной длины t и абсциссой левого нижнего угла, равной x. При увеличении x информацию в таблице A нужно обновлять, так как некоторые единичные ячейки выходят из рассматриваемого интервала, а некоторые, наоборот, входят. Для обновления мы используем операцию MODIFY. Алгоритм записывается следующим образом:

```
Algorithm MAXSQUARE
  INIT (C);
  For x:= 1 to t-1 Do Begin
    L:= List[x];
    While L <> NIL Do Begin
      MODIFY (max(1, L^.y-t+1), L^.y, 1);
      L:= L^.Next;
    End;
  End;
  Res:= 0;
  For x:= 1 to C-t+1 Do Begin
    L:= List[x+t-1];
```

```
    While L <> NIL Do Begin
      MODIFY (max(1, L^.y-t+1), L^.y, 1);
      L:= L^.Next;
    End;
    Res:= max(Res, FINDMAX (1, C));
    L:= List[x];
    While L <> NIL Do Begin
      MODIFY (max(1, L^.y-t+1), L^.y, -1);
      L:= L^.Next;
    End;
  End;
  OUTPUT (Res);
End;
```

Упражнение 20 (*). В эти лекции планировалось включить еще один раздел о решении задач LCA и RMQ за время (O(N), O(1)). Однако этот материал прямого отношения к структурам данных не имеет, и, к тому же, полученные лекции и так довольно велики. Поэтому этот материал оставляется (для желающих) на самостоятельное изучение. Вам поможет статья LCA Revisited, в которой описывается (самими авторами) алгоритм Фарака-Бендера и Колтона решения задачи LCA за (O(N), O(1)). Эту статью можно найти на сайте http://shade.msu.ru/~mab. Интересного Вам чтения!