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 ≤ 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. Í

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.