Разбор задачи «Ожерелье»

Правильное решение

Задача всегда имеет решение. Достаточно с нужной стороны подогнать колечко N к колечку N-1, а дальше действовать рекурсивно, считая, что колечки N и N-1 склеены. Такое решение идейно простое, но не самое легкое для реализации, так как при протаскивании «склеенного колечка» необходимо делать сразу несколько перемещений.

Другое решение отдаленно напоминает метод сортировки пузырьком. Главное его преимущество заключается в том, что это решение очень легко придумать и запрограммировать. Кажется, что это решение не является верным, а представляет собой всего лишь некоторую эвристику, но доказать это не просто.

Решение заключается в том, что мы находим пару (x, y) соседних правильно упорядоченных колечек ($x < y$) таких, что их можно протащить друг через друга ($|y-x| > 1$), т.е. $y-x > 1$. Перетаскиваем колечки, ищем новую пару и так далее пока не получим полное упорядочивание колечек. Данное решение очень легко запрограммировать, но совершенно не понятно, почему оно всегда выдает правильный ответ.

Чтобы это понять, рассмотрим частный случай алгоритма: берем колечко с номером 1 и перемещаем его по часовой стрелке до тех пор, пока это возможно, т.е. пока не встретим колечко с номером 2. Затем аналогично перемещаем колечко с номером два, и так далее. В какой-то момент мы дойдем до последнего колечка. После этого повторяем проделанные операции заново и вновь пытаемся переместить колечко с номером 1. Интуитивно понятно, что в результате таких операций происходит постепенное упорядочивание колечек. Когда все колечки будут упорядочены по часовой стрелке, алгоритм завершит свое выполнение, так как ни одно из них невозможно будет переместить.

Докажем корректность работы алгоритма. Рассмотрим функцию $Dist(x, y)$, которая возвращает расстояние от колечка x до колечка y, если двигаться по часовой стрелке. Дополнительно рассмотрим сумму $S = 1 \cdot Dist(1, 2) + 2 \cdot Dist(2, 3) + \ldots + n \cdot Dist(n, 1)$. Очевидно, что при смене колечек x и y местами в сумме S изменяются только слагаемые $Dist(x, x+1)$ и $Dist(y, y+1)$. Первое слагаемое при такой транспозиции увеличивается на 1, а второе – уменьшается на 1, значит, а вся сумма изменяется на величину $\Delta S = x-y$. Учитывая, что $x < y$, получаем, что в результате перестановки колечек x и y сумма S уменьшается, а так как S все время остается положительным, то наш алгоритм рано или поздно закончит работу. Для завершения необходимо заметить, что в любой позиции, кроме правильного упорядочивания, существует «правильная» пара колечек, которые алгоритм должен поменять местами. Значит, после завершения работы алгоритма получится расстановка 1, 2, …, N.

Можно доказать, что оба алгоритма выдают минимально возможное количество перестановок и в случае расположения колечек в обратном порядке общее количество транспозиций будет $(N-2)(N-1)N/6$. Получается, что верное решение задачи имеет кубическое время выполнения и линейный расход по памяти, а максимальное количество перестановок получается на тесте 50, 49, …, 1 и равно 19600.

Эвристические решения и критерий их оценивания

Правильное решение набирает 100 баллов.

При небольших значениях N ($N \le 11$) верный ответ можно получить с помощью такого алгоритма: создаем граф, вершинами которого являются всевозможные конфигурации ожерелья. Затем при помощи поиска в ширину находим нужную последовательность транспозиций. Такое решение набирало 40-50 баллов.

Помимо этого оценивались решения, которые разбирали случаи маленьких значений N, прямой и обратный порядок расположения колечек и оценивалось примерно в 30 баллов. Решения, которые выдавали только 0 и –1 или проходили только тривиальные тесты на прямой порядок оценивались в 0 баллов.

Учитывая, что информация о номерах колечек хранится в массиве, можно было допустить ряд ошибок, связанных с тем, что массив – линейный объект, а ожерелье замкнуто и представляет циклическую структуру. Естественно такие решения проходили не все тесты и за счет ошибок реализации набирали меньшее количество баллов.

Разбор задачи «Менеджер памяти»

Разобьем всю память на отрезки, каждый из которых состоит либо из свободных, либо из занятых (выделенных в ответ на запросы) ячеек.
Будем некоторым образом хранить свободные и занятые отрезки. Важное ограничение, упомянутое в условии, состоит в том, что выделенный кусок памяти может находиться либо в начале памяти, либо непосредственно после занятого фрагмента памяти. В связи с этим потребуем, чтобы свободные отрезки не могли следовать друг за другом.
Нам необходимо уметь:

1) по заданному объёму памяти находить свободный отрезок, из которого данный объём можно выделить;
2) по номеру запроса находить занятый отрезок, выделенный этим отрезком;
3) по занятому отрезку находить свободные отрезки, находящиеся непосредственно перед и после него.

Поскольку по условию оптимальность не требуется, мы можем всегда выделять память из самого длинного имеющегося свободного отрезка.
Будем использовать следующие структуры данных:

1) двусвязный список свободных отрезков;
2) массив запросов, где индексом является номер запроса, а значением — выделенный при выполнении этого запроса отрезок (или некоторое специальное значение вроде «nil» или «NULL», если память выделена не была или запрос не был запросом на выделение);
3) кучу (heap) из указателей на свободные отрезки; куча строится по их длинам.

(Куча и двусвязный список должны содержать одни и те же свободные отрезки; этого можно добиться, используя указатели или классы Delphi и Visual Basic.)
Куча — это массив из N чисел $H_1, \ldots, H_N$, таких, что для любого i, $1 < i \le N$, выполняется условие $H_{[i/2]} \ge H_i$. Первый элемент в куче всегда не меньше остальных ее элементов, поэтому операция нахождения максимума в куче выполняется за время $O(1)$. Можно реализовать операции добавления элементов в кучу, удаления из нее и изменения элемента за время $O(\log N)$; как это сделать, описано, например, в книге Т. Кормена, Ч. Лейзерсона, Р. Ривеста «Алгоритмы: построение и анализ», М.. МЦНМО, 2000.
Обработка запроса на выделение памяти производится следующим образом. Рассмотрим самый длинный свободный отрезок A: с помощью кучи мы можем найти его в списке за $O(1)$. Если длины отрезка A недостаточно, чтобы вместить требуемый блок памяти, то такой блок невозможно разместить. Иначе мы размещаем данный блок, вставляя соответствующий занятый отрезок в список и укорачивая отрезок A.
Обработка запроса на освобождение памяти производится следующим образом. Мы удаляем соответствующий занятый отрезок A из списка, после чего разбираем 4 случая:

1. Непосредственно слева и справа от A не располагается свободного отрезка. В этом случае мы делаем A свободным отрезком и добавляем его в кучу.
2. Непосредственно слева от A не располагается свободного отрезка, непосредственно справа от A располагается свободный отрезок B. В этом случае мы удаляем отрезок A из списка и удлиняем отрезок B.
3. Непосредственно справа от A не располагается свободного отрезка, непосредственно слева от A располагается свободный отрезок. Действуем аналогично случаю 2.
4. Непосредственно слева от A располагается свободный отрезок B, непосредственно справа от A располагается свободный отрезок C. В этом случае мы удаляем отрезки A и C из списка и удлиняем отрезок B.

Разбор задачи «Казино»

Заметим, во-первых, что не любая стратегия игрока приводит к нужному результату. Например, если в примере, разбиравшемся в условии задачи (**rrrgggbbb** и т.д.), стоимость буквы **b** будет больше стоимости буквы **r**, то только один из путей приводит к оптимальному результату: три раза забрать буквы **gb**. После некоторого размышления можно понять, что всевозможные т.н. «жадные» решения — делать самый выгодный ход, делать самый длинный ход, делать самый левый ход, и т.п. — также в большинстве случаев неверны. Перебирать же все варианты, для каждого из них — опять все варианты, и так далее — оказывается слишком медленным подходом.

Для решения этой задачи применим динамическое программирование. В первой половине решения мы для каждого куска начальной последовательности фишек определим, можем ли мы забрать его целиком, не затрагивая при этом остальные фишки. Иными словами, для всех l и r найдем величину $a[l,r]$, равную 1, если фишки с номерами (то есть позициями в начальном расположении, считая слева) с l по r игрок может за несколько действий все забрать себе, и 0 — в противном случае. Затем по этим данным во второй половине решения мы восстановим ответ к задаче.

Во избежание путаницы будем называть последовательности, объявленные крупье (в отличие от начальной последовательности) *словами*.

Вторая половина решения проще первой, поэтому начнём с неё. Пусть нам известны величины $a[l,r]$. Пусть тогда $b[l,r]$ — это максимальная сумма денег, которую можно получить, играя только на отрезке с l-ой фишки по r-ую (то есть, не трогая остальных фишек). Ясно, что если $a[l,r]=1$, то $b[l,r]$ — это просто сумма стоимостей всех фишек с l-ой по r-ую. Если же $a[l,r]=0$, то уже не удастся забрать все фишки, соответственно, какая-то из фишек должна остаться — пусть это фишка с номером k. Тогда заметим, что задача распадается на две — для фишек слева от k-ой и для фишек справа от неё, т.е. что $b[l,r]=b[l,k-1]+b[k+1,r]$ (здесь и далее мы считаем, что для любого m $b[m,m-1]=0$). Но так как номер оставшейся фишки не фиксирован, то в действительности $b[l,r]$ равно максимуму этих величин по всем k:

$$b[l,r] = \max_{k=l,l+1,\ldots,r} (b[l,k-1] + b[k+1,r]) \qquad (*)$$

(напомним, что эта формула верна при $a[l,r]=0$). Теперь мы видим, что для нахождения величин $b[l,r]$ можно применить динамическое программирование: если мы будем перебирать r от 1 до длины начальной последовательности, а затем l от r до 1, то к моменту вычисления величины $b[l,r]$ все величины, которые необходимы для её нахождения по формуле (*), уже вычислены. Таким образом, зная величины $a[l,r]$ мы можем найти $b[l,r]$. Легко видеть, что на этом шаге требуется порядка L^3 действий (напомним, что L — это длина начальной последовательности).

Теперь вернёмся к первой половине решения, а именно — нахождению величин $a[l,r]$. Как узнать, можно ли забрать себе данную последовательность (в нашем случае — кусок начальной) целиком? Во-первых, возможно, эту последовательность можно разделить на две части, каждую из которых можно забрать себе целиком. То есть, если найдётся k такое, что $a[l,k]=1$ и $a[k+1,r]=1$, то и $a[l,r]=1$. Если же так сделать не удаётся, то можно заметить, что если мы всё-таки заберём всю последовательность целиком, то её первую и последнюю фишки мы заберём только на последнем шаге. Пусть на последнем шаге мы заберём некоторое слово S. Тогда задача сводится к тому, чтобы выкинуть из последовательности некоторые куски (можно считать, что для этих кусков величина a уже посчитана, поэтому мы знаем, какие из них можно выкинуть, а какие — нет), оставив первую и последнюю фишки на месте, так, чтобы осталось слово S.

Для решения этой задачи опять применим динамическое программирование. А именно, пусть $c[l,r,p,q]=1$, если можно из отрезка с l-ой фишки по r-ю можно так выкинуть несколько кусков, чтобы остались первые q фишек слова номер p, и при этом l-я и r-я фишки

остались на месте, и 0 — иначе (тогда нас интересует величина $c[l,r,p,length(p)]$, где $length(p)$ — длина слова номер p). Во-первых, ясно, что если первая фишка слова номер p не совпадает с l-ой фишкой начальной последовательности, или q-я — с r-ой, то $c[l,r,p,q]=0$. В противном случае, пусть (q-1)-я фишка слова получится из k-ой фишки последовательности. Тогда, во-первых, должно быть $c[l,k,p,q\text{-}1]=1$ (чтобы получить первые q-1 фишек), а, во-вторых, $a[k+1,r\text{-}1]=1$ (чтобы выкинуть кусок между (q-1)-ой фишкой и q-ой). Но так как число k не фиксировано, то получаем, что $c[l,r,p,q]=1$ тогда и только тогда, когда найдётся такое k. Тем самым получен алгоритм для подсчета c. Оценим время его работы. На подсчёт одного элемента массива c требуется порядка L действий (по количеству возможных k). Нам необходимо подсчитать элементы вида $c[l,r,p,length(p)]$. Из вышеприведённых рассуждений легко заметить, что для их подсчёта нам потребуется подсчёт только элементов вида $c[l,\ldots,\ldots,\ldots]$, а элементов такого вида порядка $L\cdot S$ (где S — сумма длин всех слов), так как для параметра r есть порядка L вариантов, а для пары параметров (p,q) — порядка S вариантов. Тем самым мы для любых l и r можем найти $a[l,r]$ за порядка $L\cdot L\cdot S= L^2\cdot S$ действий, а общее время работы получается порядка $L^2\cdot L^2\cdot S=L^4\cdot S$.

Однако легко заметить, что если подсчитать все величины $c[l,r,p,q]$ сначала, а потом лишь использовать найденные значения, то время работы будет меньше. Действительно, всего в массиве c $L^2\cdot S$ элементов, и на подсчёт каждого из них уйдёт порядка L действий, тем самым общее время работы будет порядка $L^3\cdot S$.