Теоретическое домашнее задание
по алгоритмам и структурам данных
Задание №1

Автор: *Ряжских Дмитрий*

МФТИ 2024

1 Задача 1 (Зацикленный)

1.1 Описание алгоритма

Для решения этой задачи можно использовать следующим алгоритмом:

1. **Обнаружение цикла.**

   - Заведём два указателя
   - Изначально оба указателя указывают на начало списка
   - За одну итерацию алгоритма первый указатель двигается на 1 элемент, а второй на 2
   - Второй указатель обрабатывает каждый узел списка (т.е. он не может "перепрыгнуть"первый указатель)
   - Если в списке нет цикла, то алгоритм завершится, когда второй указатель достигнет конца списка ("уткнётся"в null или EndNode). В этом случае количество шагов, будет равно количеству узлов в списке, то есть O(n)
   - Если в списке есть цикл, указатели в конечном итоге всё равно встретятся. Это произойдет в худшем случае за n шагов, потому что на каждом шаге разница между ними уменьшается на один при движении по циклу

   Таким образом асимптотика поиска цикла O(n)

2. **Определение длины цикла.**

   Если цикл обнаружен, зафиксируем позицию второго указателя и продолжим двигать первый указатель на один шаг, подсчитывая количество шагов до тех пор, пока он снова не встретится с зафиксированным указателем. Это количество шагов будет длиной цикла. Максимум потребуется n итераций, значит, асимптотика O(n) по времени.

---

3. **Определение узла входа в цикл.**

   Узлом из цикла будет тот, на который указывает второй указатель. Асимптотика O(1) по времени.

1.2 Итог

Таким образом, алгоритм работает за линейное время O(n), так как все его этапы (определение наличия цикла, поиск узла цикла и определение длины цикла) выполняются за линейное количество шагов. Алгоритм использует только несколько указателей и не требует дополнительной памяти, пропорциональной размеру списка, поэтому его пространственная сложность — O(1).

2 Задача 2 (Палкой по гоблину)

2.1 Описание алгоритма

Для решения этой задачи можно использовать алгоритм бинарного поиска по числу ударов, необходимых для уничтожения всех гоблинов. Основная идея заключается в том, чтобы проверить, возможно ли уничтожить всех гоблинов за k ударов и использовать бинпоиск для определения минимального k, при котором это возможно.

1. **Описание бинпоиска.**

   - Минимальное число ударов (left), которое понадобится, — это 0
   - Максимальное число ударов (right) можно установить как $\lceil \frac{\max h_i}{\max(p,q)} \rceil$, где $\max h_i$ — максимальное здоровье среди всех гоблинов, чтобы гарантировать, что мы можем перебить самого сильного гоблина
   - Определим функцию проверки (только ради краткости записи назовём её f), которая проверяет, можно ли перебить всех гоблинов за k ударов

- Пока разница между right и left больше 1 Повторяем все следующие шаги:
- $mid = \frac{left+right}{2}$
- Если f(k) возвращает true, значит k ударов достаточно, и right устанавливаем равным mid
- Если f(k) возвращает false, значит k ударов недостаточно, и left устанавливаем равным mid

2. **Описание функции f.**

2.1.1 Если p > q

- Заводим счётчик count_headshots = 0
- Для каждого гоблина считаем можно ли его убить только ударами, при которых на него не направляется посох. $remains\_hp_i = h_i$ - q * k
- Если $remains\_hp_i > 0$, то считаем сколько ударов посохом по нему нужно нанести, чтобы убить. $count\_headshots \mathrel{+}= \lceil \frac{remains\_hp_i}{p-q} \rceil$
- Если после итерации по всем гоблинам count_headshots > k, то нельзя всех убить за k выстрелов, следовательно f вернёт false, в противном случае - true

2.1.2 Если p ≤ q

- Заводим счётчик count_headshots = k
- Для каждого гоблина считаем можно ли его убить только ударами, при которых на него не направляется посох. $remains\_hp_i = h_i$ - q * k
- Если $remains\_hp_i > 0$, то сразу возвращаем false - сменив пассивный удар на прицеленный здоровье уже не уменьшится
- Если $remains\_hp_i < 0$ и p < q, попробуем заменить пассивные удары на активные, оставив гоблина мёртвым. $count\_headshots \mathrel{-}= \lfloor \frac{|remains\_hp_i|}{q-p} \rfloor$

- Если после итерации по всем гоблинам count_headshots > 0 и при этом p $<$ q, то нельзя всех убить за k выстрелов, следовательно f вернёт false, в противном случае - true

2.2 Оценка сложности.

Количество итераций бинарного поиска — это $O(log(right - left)) = O(log(right))$ (т.к. $left = 0$) $= O(log\lceil\frac{\max h_i}{\max(p,q)}\rceil) = O(log(\max h_i))$ (т.к. p и q константы)

Для каждого значения k, которое проверяется в процессе бинарного поиска, нам нужно пройтись по всем N гоблинам и проверить, достаточно ли k ударов, чтобы здоровье каждого из них стало меньше или равно нулю. Это занимает O(N) операций для каждого шага бинарного поиска

Итак, каждая проверка за O(N) выполняется за $O(log(\max h_i))$ итераций бинарного поиска. Следовательно, общая временная сложность будет: $O(Nlog(\max h_i))$

3 Задача 3 (Прибавляй, арифметизируй!)

3.1 Описание алгоритма

Заведём два дополнительных массива для хранения разности прогрессии (diff[N+1]) и начала прогрессии (delta[N+1]). После каждого запроса будем их обновлять, а в конце пересчитаем все элементы исходного массива, опираясь на дополнительные, и выведем его.

1. **Описание обработки запроса.**

   - Добавляем b в $delta[l]$, чтобы обозначить начало прогрессии
   - Добавляем d в $diff[l]$, чтобы прогрессия начала нарастать с l

- В $delta[r+1]$ вычитаем $b + (r-l) * d$, чтобы погасить действие прогрессии после r
- В $diff[r+1]$ вычитаем d, чтобы убрать наращивание после r.

2. **Описание конечного подсчёта.** (Накопление изменений) После обработки всех запросов нужно накопить значения в исходном массиве, используя $delta$ и $diff$. Для этого:

   - Заведём накопительные переменную $cur\_delta = 0$ и $cur\_diff = 0$
   - Проходим по массиву и накапливаем значения $cur\_delta \mathrel{+}= delta[i]$ $cur\_diff \mathrel{+}= diff[i]$
   - Обновляем элементы списка: $a[i] \mathrel{+}= cur\_delta$
   - Обновляем текущую разность $cur\_delta \mathrel{+}= cur\_diff$

   Теперь осталось вывести обновлённый исходный массив.

3.2 Оценка сложности.

Каждый запрос выполняется за O(1), следовательно все запросы выполнятся за O(Q). Обновление значений в конце происходит за O(N) т.к. на обработку каждого элемента уходит O(1). Таким образом, итоговая временная сложность O(Q + N)

Алгоритм использует O(N) дополнительной памяти, т.к. все вспомогательные массивы $\approx$ такой же длины как исходный, а количество дополнительных массивов константное.

4 Задача 4 (В поисках сдвига)

4.1 Описание алгоритма

4.1.1 Часть 1 (Элементы различны)

Для поиска k за O(logN), можно использовать бинарный поиск

- Установим два указателя: $left = 0$ и $right = N - 1$.
- Пока $left < right$ Вычислим средний индекс $mid = \lfloor \frac{left+right}{2} \rfloor$.
- Если $a[mid] > a[right]$, это значит, что mid находится в неотсортированной части массива, поэтому точка разрыва должна быть справа от mid, то есть $left = mid + 1$.
- Иначе это значит, что правая часть от mid до $right$ отсортирована, и точка разрыва должна быть слева от mid, точка разрыва находится в левой части или на mid, поэтому $right = mid$.
- После завершения цикла $left$ указывает на индекс минимального элемента, а количество сдвигов$k = left$.

Логарифмическая сложность достигается, поскольку на каждой итерации размер рассматриваемого диапазона уменьшается вдвое.

4.1.2 Часть 2 (Некоторые элементы неразличимы)

Рассмотрим массив, в котором все элементы одинаковы, например, $a = [1, 1, ..., 1]$. Здесь невозможно определить место разрыва, так как массив выглядит одинаково независимо от числа сдвигов k. Чтобы определить k, необходимо в худшем случае проверить все элементы, что требует $O(n)$ операций. Поэтому нижняя оценка времени выполнения любого алгоритма для поиска k в массиве с возможными одинаковыми элементами — это $\Omega(n)$.

5 Задача 5 (В поисках подогорода)

5.1 Описание алгоритма

5.1.1 массив высот

Построим массив высот (height[N][M], где N × M - размер огорода), который будет представлять количество подряд идущих строк с помидорами, начиная с текущей строки. Начнем с первой строки и будем обновлять height для каждой следующей строки, как в гистограмме

$$\begin{cases} heights[i][j] = garden[i][j], \quad i = 0 \\ heights[i][j] = heights[i-1][j] + 1 \quad if \quad garden[i][j] = 1 \quad else \quad 0, \quad i \neq 0 \end{cases}$$

Где garden - массив, представляющий огород из условия, в котором 1 - сектор с помидором, а 0 - сектор с огурцами.

5.1.2 подсчёт ответа

Теперь для каждой строки определим максимальный прямоугольник (таковым будем считать прямоугольник с наибольшей площадью), нижняя сторона которого лежит на этой строке.

- Инициализируем пустой стек
- Пока стек не пуст и текущая высота меньше или равна высоте элемента на вершине стека, забираем элемент из стека и вычисляем площадь прямоугольника с высотой равной высоте удаленного элемента
- Если стек пуст, ширина равна текущему индексу, иначе ширина равна разнице между текущим индексом и индексом элемента на вершине стека минус один
- Берём максимум из имеющегося ответа и полученным при итерации

- Помещаем текущий индекс в стек
- После того как прошли все элементы строки, обрабатываем оставшиеся элементы в стеке аналогичным образом

5.2 Итог

Проход по каждой строке занимает O(M), а обработка стека также выполняется за O(M), так как каждый элемент обрабатывается максимум дважды. Строк N, следовательно алгоритм работает за O(N*M) по времени и за O(N*M) дополнительной памяти.

6 Задача 6 (Деамортизация)

6.1 Описание алгоритма

Заведём три выделенных участка памяти (статических массива) длины N/2, N и 2N и назовём их previous, current и next соответственно и три указателя которые будут работать в соответствующих массивах: prev, cur, nxt.

- get - элементы динамического массива в i-й позиции считываются из current
- push - при добавлении элемента cur сдвигается на 1 вправо и записывает новый элемент в ячейку памяти на которую указывает. nxt сдвигается на 2 вправо, записывая в посещённые ячейки значения из ячеек массива current, стоящие в местах 2*(cur-N/2) и 2*(cur-N/2) + 1
- pop - при удалении элемента prev сдвигается на 1 вправо, записывая в посещённую ячейку значение из ячейки массива current, стоящее на месте (N-cur), после чего cur сдвигается на 1 влево

- allocate - когда cur=N-1 и вызывается add происходят следующие действия при непосредственным вызовом add: next становится current, бывший current становится previous, а бывший previous удаляется. Новое N равно старому удвоенному, а next - новый участок выделенной памяти длины 2 новых N

- deallocate - когда cur = N/2 - 1 и вызывается pop аналогично allocate: previous -> new MemoryBlock(N), current -> previous, next -> current, N -> N/2; delete old_next

6.2 Почему это работает

Чтобы подготовиться к будущему увеличению размера, указатель nxt сдвигается на 2 вправо, и значения из массива current копируются в массив next в удвоенные индексы. Таким образом, при заполнении массива current массив next будет уже заполнен половиной элементов, соответственно равных current

При удалении элемента указатель prev сдвигается вправо, чтобы показать на следующую позицию. Значение из массива current копируется в массив previous, что позволяет подготовиться к возможному уменьшению размера

Перекопирование происходит постепенно, по мере добавления элементов, так что в каждый момент времени выполняется только постоянное количество операций, чтобы поддерживать три массива

6.3 Оценка сложности

Подготовка массивов next и previous позволяет избежать больших затрат при увеличении размера. Так как перекопирование выполняется по частям, чистая стоимость каждой операции остается O(1).

В любой момент времени объем выделенной памяти ограничен сверху 3N, так как previous, current, и next имеют размеры N/2, N, и 2N соответственно.

При этом размер массива size всегда близок к N, что дает $capacity = \Theta(size)$. При увеличении и уменьшении размера массива поддерживается баланс между количеством элементов и выделенной памятью, так что нет перерасхода памяти.

7 Задача 7 (k-ичная куча куч)

7.1 формулы индексов

Индекс i-го ребенка j-го узла: Если мы используем нумерацию узлов начиная с 0, то для k-ичной кучи, где каждый узел имеет максимум k детей, индексы детей j-го узла вычисляются по следующей формуле:

$$i - j = k * j + i + 1$$

где i=0, 1, ..., k-1. Это предполагает, что дети каждого узла идут подряд после родителя.

$$i = \frac{i-1}{k}$$

(Здесь предполагается целочисленное деление)

7.2 оценка времени просеивания

При просеивании вниз в k-ичной куче узел сравнивается с k своими детьми на каждом уровне, чтобы найти минимального (или максимального, в зависимости от типа кучи). Поскольку глубина кучи составляет $\log_k N$ (где N — количество элементов в куче), время работы будет: O(k*logN)

При просеивании вверх узел поднимается по уровням кучи, сравниваясь с родителем на каждом уровне. Глубина кучи составляет $\log_k N$, поэтому время работы: O(logN)

7.3 оценка времени просеивания если дети в куче

Если k детей каждого узла организованы в виде бинарной кучи, то операции сравнения и поиска минимального (или максимального) среди детей занимают O(logk).

Просеивание вниз: На каждом уровне требуется O(logk) времени, чтобы найти минимального среди k детей. Поскольку глубина кучи равна $\log_k N$, итоговая сложность будет: O(logk*logN) = O(logN), т.к. k - заданная константа

Просеивание вверх: Это не изменяется, так как при просеивании вверх мы все равно сравниваем элемент только с его родителем, и время работы остается: O(logN)

Таким образом, использование бинарной кучи для хранения детей увеличивает время работы просеивания вниз, но просеивание вверх остается неизменным.

8 PS

Все задачи решались основываясь на том, что мы знаем асимптотику решения 'тривиального вида задачи', такого как 'обычный' бинпоиск или 'обычная' куча. Её же мы доказывали на лекции, а фактами с лекций по условию задания можно пользоваться бездоказательно.