­Университет ИТМО

Факультет программной инженерии и компьютерной техники

**Лабораторная работа №3**

по «Алгоритмам и структурам данных»

Базовые задачи / Timus

Выполнил:

Студент группы P32081

Васильченко Роман

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2023

# Yandex

#### Задача I «Машинки»

Пояснение: Данный алгоритм решает задачу, определяя минимальное количество операций, которые нужно выполнить, чтобы удовлетворить потребности ребенка в машинках. Алгоритм рассматривает последовательность машинок, которые нужно вытащить с полки, и постепенно помещает их на пол. При этом, если на полу уже находится максимальное число машинок, одну из них нужно убрать на полку. Чтобы определить, какую машинку убрать на полку, используется очередь с приоритетами, в которой хранятся индексы машинок и их позиция в последовательности. Машинки, которые уже находятся на полке, не добавляются в очередь, а хранятся в множестве для быстрого доступа. После того, как машинка была помещена на пол, её позиция в списке обновляется, и, если больше не нужно вытаскивать эту машинку, её позиция становится INT\_MAX, что гарантирует, что она останется на полу до конца последовательности. Алгоритм является оптимальным, потому что мы максимизируем количество машинок на полу, а значит минимизируем количество операций по их перемещению между полкой и полом. Также мы максимально быстро убираем машинки с пола, которые больше не нужны, и добавляем новые машинки на пол.

Время:

* Чтение входных данных занимает O(N) времени, где N - количество операций.
* Заполнение структур данных (вектора и списков) занимает O(N) времени.
* Главный цикл занимает O(N logK) времени, где K - максимальное количество машинок на полу. Это происходит потому, что для каждой операции нужно проверить, находится ли машинка на полу или на полке (O(1)), добавить машинку на пол (O(logK)), удалить машинку с пола (O(logK)) и добавить новую машинку на полку (O(logK)).

Итого, асимптотическая сложность этого решения составляет O(N logK).

Память:

* Для вектора и списков используется O(N) памяти, так как они хранят информацию о порядке операций и машинках, участвующих в операциях.
* Для хранения машинок на полу используется O(K) памяти.
* Для приоритетной очереди используется O(K) памяти.
* Итоговое использование памяти равно O(N + K).

Таким образом, это решение занимает O(N logK) времени и использует O(N + K) памяти.

Код:

#include <bits/stdc++.h>

using namespace std;

int main() {

  int num\_toys, max\_floor\_toys, num\_operations, op\_counter = 0;

  cin >> num\_toys >> max\_floor\_toys >> num\_operations;

  vector<int> sequence(num\_operations);

  vector<list<int>> toy\_entries(num\_toys);

  for (int i = 0; i < num\_operations; i++) {

    cin >> sequence[i];

    toy\_entries[--sequence[i]].push\_back(i);

  }

  unordered\_set<int> floor\_toys;

  priority\_queue<pair<int, int>> pq\_toy\_ids;

  for (int i = 0; i < num\_operations; i++) {

    int curr\_toy = sequence[i];

    toy\_entries[curr\_toy].pop\_front();

    if (floor\_toys.find(curr\_toy) == floor\_toys.end()) {

      if (floor\_toys.size() >= max\_floor\_toys) {

        floor\_toys.erase(pq\_toy\_ids.top().second);

        pq\_toy\_ids.pop();

      }

      op\_counter++;

      floor\_toys.insert(curr\_toy);

    }

    if (toy\_entries[curr\_toy].empty())

      pq\_toy\_ids.push({INT\_MAX, curr\_toy});

    else

      pq\_toy\_ids.push({toy\_entries[curr\_toy].front(), curr\_toy});

  }

  cout << op\_counter << endl;

  return 0;

}

#### Задача J «Гоблины и очереди»

Пояснение: Алгоритм использует структуру данных deque (двусторонняя очередь) для хранения гоблинов. При поступлении запроса на добавление гоблина в конец очереди, гоблин добавляется в правую часть очереди (right). При поступлении запроса на добавление привилегированного гоблина в середину очереди, он добавляется в левую часть очереди (left) с помощью функции push\_front(). При запросе на удаление первого гоблина из очереди, он удаляется из левой части очереди и его номер выводится на экран.

Чтобы сохранить порядок гоблинов в очереди, левая и правая части очереди должны содержать одинаковое количество элементов. Поэтому после добавления гоблина в правую часть очереди, мы перемещаем гоблинов из правой части в левую часть, пока не будет достигнуто равенство длин двух частей. Аналогично, после удаления гоблина из левой части очереди, мы перемещаем гоблинов из правой части в левую часть, пока длина левой части не станет на 1 меньше, чем длина правой части.

Алгоритм правильно обрабатывает все три типа запросов и поддерживает порядок гоблинов в очереди, поэтому он является верным решением задачи.

Время:

* Чтение входных данных занимает O(N) времени, где N - количество запросов.
* Операции добавления гоблинов в очередь (символы “+” и “\*”) занимают O(1) времени, так как добавление в конец или в начало дека имеет константную сложность.
* Операция удаления гоблина из очереди (символ “-”) занимает O(1) времени, так как удаление из начала дека также имеет константную сложность.
* Операции переноса гоблинов между левой и правой частью очереди имеют O(1) временную сложность.

Итоговая асимптотическая сложность кода составляет O(N).

Память:

* Для хранения элементов очереди используется дек (deque), который занимает O(N) памяти.
* Использование дополнительной памяти в коде минимально и не влияет на асимптотическую сложность.

Итоговое использование памяти - O(N).

Код:

#include <bits/stdc++.h>

using namespace std;

int main() {

  deque<int> left, right;

  int N, i;

  char goblinType;

  cin >> N;

  while (N--) {

    cin >> goblinType;

    if (goblinType == '+') {

      cin >> i;

      right.push\_back(i);

    }

    if (goblinType == '\*') {

      cin >> i;

      right.push\_front(i);

    }

    if (goblinType == '-') {

      cout << left.front() << "\n";

      left.pop\_front();

    }

    while (left.size() < right.size()) {

      left.push\_back(right.front());

      right.pop\_front();

    }

    cout << "LOL" << endl;

    while ((int)(left.size()) - 1 > (int)(right.size())) {

      right.push\_front(left.back());

      left.pop\_back();

    }

    for (int i = 0; i < left.size(); i++) {

      cout << left[i] << " ";

    }

    cout << endl;

    for (int i = 0; i < right.size(); i++) {

      cout << right[i] << " ";

    }

    cout << endl;

  }

  return 0;

}

#### Задача K «Менеджер памяти-1»

Пояснение: Это так называемый "бест-фит" алгоритм выделения памяти.

Алгоритм начинается с инициализации единственного свободного сегмента памяти, затем в цикле обрабатываются запросы на выделение и освобождение памяти.

При запросе на выделение памяти алгоритм проходит по массиву свободных сегментов и находит сегмент наиболее подходящего размера (т.е. не меньше запрашиваемого размера, но при этом наименее "избыточный"). Если такой сегмент найден, он разбивается на два - один для запрошенного размера, другой для оставшейся части. Запрошенный сегмент помечается как занятый, а оставшийся свободный сегмент возвращается в массив свободных сегментов.

При запросе на освобождение памяти алгоритм находит соответствующий сегмент по номеру запроса и пытается объединить его с соседними свободными сегментами, если таковые имеются. Если рядом нет свободных сегментов, то освобожденный сегмент помечается как свободный и добавляется в массив свободных сегментов.

В результате, после обработки всех запросов, все память будет разбита на занятые и свободные сегменты, и все запросы на выделение памяти будут обработаны корректно в соответствии с условиями задачи.

Время:

* Добавление блока в кучу или удаление из кучи занимает O(log n), где n - количество блоков в куче.
* Поиск максимального свободного блока занимает O(1), так как он всегда находится в корне кучи.
* Объединение свободных блоков занимает O(log n).
* Изменение размера свободного блока занимает O(log n).

Таким образом, общее время работы программы составляет O(n log n), где n - количество запросов на выделение и освобождение блоков памяти.

Память: Так как размер массива Структуры “MemorySegment” заранее определен, то память занимает всегда const, но если заменить на N, то тогда будет O(2 \* N) = O(N)

Код:

#include <bits/stdc++.h>

using namespace std;

struct MemorySegment {

  int index;

  int startPosition;

  int size;

  bool isAvailable;

  MemorySegment \*previous;

  MemorySegment \*next;

  MemorySegment(int index, int startPosition, int size, bool isAvailable,

                MemorySegment \*previous, MemorySegment \*next)

      : index(index), startPosition(startPosition), size(size),

        isAvailable(isAvailable), previous(previous), next(next) {

    if (previous) {

      previous->next = this;

    }

    if (next) {

      next->previous = this;

    }

  }

  void remove() {

    if (previous) {

      previous->next = next;

    }

    if (next) {

      next->previous = previous;

    }

  }

};

MemorySegment \*memoryRequests[100000];

MemorySegment \*freeMemorySegments[100000];

void swapSegments(int a, int b) {

  MemorySegment \*temp = freeMemorySegments[a];

  freeMemorySegments[a] = freeMemorySegments[b];

  freeMemorySegments[b] = temp;

  freeMemorySegments[a]->index = a;

  freeMemorySegments[b]->index = b;

}

bool compareSegments(MemorySegment \*a, MemorySegment \*b) {

  return a->size < b->size;

}

int getParent(int i) { return (i - 1) / 2; }

int getLeft(int i) { return 2 \* i + 1; }

int getRight(int i) { return 2 \* i + 2; }

void siftdown(MemorySegment \*\*heap, int n, int i) {

  int j = i;

  if (getLeft(i) < n && compareSegments(heap[j], heap[getLeft(i)])) {

    j = getLeft(i);

  }

  if (getRight(i) < n && compareSegments(heap[j], heap[getRight(i)])) {

    j = getRight(i);

  }

  if (i != j) {

    swapSegments(i, j);

    siftdown(heap, n, j);

  }

}

void siftup(MemorySegment \*\*heap, int i) {

  while (i > 0 && compareSegments(heap[getParent(i)], heap[i])) {

    swapSegments(i, getParent(i));

    i = getParent(i);

  }

}

void removeFromHeap(MemorySegment \*\*heap, int &n, int i) {

  swapSegments(i, n - 1);

  n--;

  if (i >= n) {

    return;

  }

  siftup(heap, i);

  siftdown(heap, n, i);

}

int main() {

  ifstream inputFile("input.txt");

  ofstream outputFile("output.txt");

  int memorySize, requestCount;

  inputFile >> memorySize >> requestCount;

  int currentFree = 0;

  MemorySegment firstSegment = {currentFree, 0,       memorySize,

                                true,        nullptr, nullptr};

  freeMemorySegments[currentFree++] = &firstSegment;

  for (int i = 0; i < requestCount; i++) {

    int memoryRequest;

    inputFile >> memoryRequest;

    if (memoryRequest > 0) {

      MemorySegment \*largestSegment = freeMemorySegments[0];

      if (!currentFree || largestSegment->size < memoryRequest) {

        memoryRequests[i] = nullptr;

        outputFile << -1 << endl;

      } else {

        memoryRequests[i] =

            new MemorySegment(-1, largestSegment->startPosition, memoryRequest,

                              false, largestSegment->previous, largestSegment);

        outputFile << memoryRequests[i]->startPosition + 1 << endl;

        largestSegment->startPosition += memoryRequest;

        largestSegment->size -= memoryRequest;

        if (largestSegment->size > 0) {

          siftdown(freeMemorySegments, currentFree, largestSegment->index);

        } else {

          largestSegment->remove();

          removeFromHeap(freeMemorySegments, currentFree, 0);

        }

      }

    } else {

      memoryRequest = -memoryRequest;

      MemorySegment \*segmentToFree = memoryRequests[memoryRequest - 1];

      if (segmentToFree == nullptr) {

        continue;

      }

      bool previousAvailable =

          segmentToFree->previous && segmentToFree->previous->isAvailable;

      bool nextAvailable =

          segmentToFree->next && segmentToFree->next->isAvailable;

      if (!previousAvailable && !nextAvailable) {

        segmentToFree->isAvailable = true;

        segmentToFree->index = currentFree;

        freeMemorySegments[currentFree] = segmentToFree;

        siftup(freeMemorySegments, currentFree++);

      } else if (!previousAvailable) {

        segmentToFree->next->startPosition = segmentToFree->startPosition;

        segmentToFree->next->size += segmentToFree->size;

        siftup(freeMemorySegments, segmentToFree->next->index);

        segmentToFree->remove();

      } else if (!nextAvailable) {

        segmentToFree->previous->size += segmentToFree->size;

        siftup(freeMemorySegments, segmentToFree->previous->index);

        segmentToFree->remove();

      } else {

        segmentToFree->previous->size +=

            segmentToFree->size + segmentToFree->next->size;

        siftup(freeMemorySegments, segmentToFree->previous->index);

        segmentToFree->remove();

        removeFromHeap(freeMemorySegments, currentFree,

                       segmentToFree->next->index);

        segmentToFree->next->remove();

      }

      memoryRequests[i] = nullptr;

    }

  }

  inputFile.close();

  outputFile.close();

}

#### Задача L «Минимум на отрезке»

Пояснение: Данный алгоритм решает задачу "Минимум на отрезке" с помощью структуры данных "Sparse Table". Последовательность чисел разбивается на блоки длины log(N), в каждом блоке хранится минимум. Затем используется рекурсивное дробление блоков до длины 1. В итоге, для каждого блока запоминается индекс минимального элемента. Для решения запроса на интервале [L, R] находим такие два блока, что первый целиком лежит в [L, R], а последний целиком лежит за пределами [L, R]. Тогда минимум на интервале [L, R] равен min(минимум в первом блоке, минимум во втором блоке).

Также алгоритм использует массив log2, который хранит значения функции log2(i) для i от 1 до N. Это позволяет быстро находить степень двойки, ближайшую к данному числу.

Алгоритм имеет сложность O(Nlog(N)) для предобработки и O(1) для запроса на интервале [L, R], что делает его эффективным для решения задачи "Минимум на отрезке".

Время: Реализованный алгоритм позволяет отвечать на запросы минимума в подотрезке за O(1) после O(n log n) предподсчета. Время работы данного алгоритма на поставленной задаче составляет O(n log k).

Память: Итоговая асимптотическая память алгоритма равна O(N log N), так как мы используем O(N log N) памяти для построения таблицы минимумов на отрезках с помощью Sparse Table и O(N) памяти для хранения входных данных и других переменных.

Код:

#include <algorithm>

#include <iostream>

#include <vector>

using namespace std;

const int MAXN = 150000;

const int MAXK = 10000;

int n, k;

int a[MAXN];

int log2[MAXN + 1];

int table[MAXN][20];

void build\_sparse\_table() {

  for (int i = 0; i < n; i++) {

    table[i][0] = a[i];

  }

  for (int j = 1; j <= log2[n]; j++) {

    for (int i = 0; i + (1 << j) <= n; i++) {

      table[i][j] = min(table[i][j - 1], table[i + (1 << (j - 1))][j - 1]);

    }

  }

}

int query\_sparse\_table(int l, int r) {

  int j = log2[r - l + 1];

  return min(table[l][j], table[r - (1 << j) + 1][j]);

}

int main() {

  ios\_base::sync\_with\_stdio(false);

  cin.tie(nullptr);

  cin >> n >> k;

  for (int i = 0; i < n; i++) {

    cin >> a[i];

  }

  log2[1] = 0;

  for (int i = 2; i <= n; i++) {

    log2[i] = log2[i / 2] + 1;

  }

  build\_sparse\_table();

  for (int i = 0; i <= n - k; i++) {

    cout << query\_sparse\_table(i, i + k - 1) << " ";

  }

  cout << endl;

  return 0;

}

# Timus

#### Задача №1494 «Монобильярд»

Пояснение: Алгоритм считывает количество шаров и последовательность номеров закатанных шаров, которую вводит пользователь. Затем программа проходит по этой последовательности, и для каждого номера шара происходит следующее:

Если номер текущего шара больше, чем любой шар, который был закатан ранее, то в стек добавляются все пропущенные номера шаров между максимальным номером и номером текущего шара.

Если номер текущего шара меньше или равен номеру на вершине стека, то этот шар удаляется из стека. Если номер текущего шара не совпадает с номером на вершине стека, то программа выводит "Cheater" и завершает работу.

Если последовательность номеров шаров была пройдена полностью без ошибок, то программа выводит "Not a proof".

Время: O(N), где N - количество бильярдных шаров. Данный алгоритм требует только одного прохода по данным, т.е. выполняется за линейное время.

Память: O(N), так как мы используем стек, который может содержать до N элементов.

Код:

#include <bits/stdc++.h>

using namespace std;

int main() {

  int numBalls, highestBall = 0;

  cin >> numBalls;

  stack<int> ballStack;

  for (int i = 0; i < numBalls; i++) {

    int currentBall;

    cin >> currentBall;

    if (currentBall > highestBall) {

      for (int j = highestBall + 1; j < currentBall; j++) {

        ballStack.push(j);

      }

      highestBall = currentBall;

    } else {

      if (currentBall == ballStack.top()) {

        ballStack.pop();

      } else {

        cout << "Cheater" << endl;

        return 0;

      }

    }

  }

  cout << "Not a proof" << endl;

  return 0;

}

#### Задача № 1628 «Белые полосы»

Пояснение: Программа решает задачу о нахождении количества белых полос в таблице. Каждая полоса - это максимальный по включению отрезок белых клеток, которые не пересекаются с неудачными днями Вась-Васи. На вход подаются размеры таблицы и координаты неудачных дней. Программа использует множества для хранения номеров строк и столбцов, в которых есть неудачные дни. Затем она находит белые полосы в каждой строке и столбце, а также количество пересечений одиночных элементов между строками и столбцами. Для этого используются вспомогательные функции countWhiteStripes и countSingleElementIntersections.

Время:

Асимптотическая сложность алгоритма составляет O(m \* log(n) + n \* log(m)), где m и n - размеры календаря, так как для каждой строки и столбца используется структура данных set, позволяющая быстро добавлять и искать элементы. В данном случае, количество неудачных дней k не влияет на асимптотику алгоритма.

Память:

Алгоритм использует дополнительную память для хранения наборов строк и столбцов, в которых есть неудачные дни, а также для временных наборов с единичными элементами. Размер каждого набора не превышает k, поэтому асимптотическая память составляет O(k).

Код:

#include <bits/stdc++.h>

using namespace std;

int countWhiteStripes(const set<int> &collection, int max\_size,

                      set<int> &single\_elements) {

  int prev = 0;

  int result = 0;

  for (int curr : collection) {

    if (curr - prev > 2)

      result++;

    else if (curr - prev == 2)

      single\_elements.insert(curr - 1);

    prev = curr;

  }

  if (max\_size - prev > 1)

    result++;

  else if (max\_size - prev == 1)

    single\_elements.insert(max\_size);

  return result;

}

int countSingleElementIntersections(const set<int> \*rows,

                                    const set<int> \*columns, int m, int n) {

  int intersections = 0;

  for (int i = 0; i < m; i++) {

    for (int singleton : rows[i]) {

      if (columns[singleton - 1].find(i + 1) != columns[singleton - 1].end())

        intersections++;

    }

  }

  return intersections;

}

int main() {

  int m, n, k, x, y;

  cin >> m >> n >> k;

  set<int> rows[m], columns[n];

  while (k--) {

    cin >> x >> y;

    rows[x - 1].insert(y);

    columns[y - 1].insert(x);

  }

  int result = 0;

  set<int> single\_elements;

  for (set<int> &row : rows) {

    result += countWhiteStripes(row, n, single\_elements);

    row = single\_elements;

    single\_elements.clear();

  }

  for (set<int> &column : columns) {

    result += countWhiteStripes(column, m, single\_elements);

    column = single\_elements;

    single\_elements.clear();

  }

  result += countSingleElementIntersections(rows, columns, m, n);

  cout << result;

  return 0;

}