Задача 3-1 (75 баллов). Известно, что каждая клетка организма человека содержит ДНК, которая представляет из себя цепочку нуклеотидов, кодируемых символами A, G, T, C, и, кроме того, что эти цепочки у близких родственников должны быть похожими.

Для определения того, принадлежат ли два заданных ДНК s_1 и s_2 близким родственникам, вам нужно решить следующую задачу. Необходимо найти такую позицию i в строке s_1 (позиции нумеруются с единицы), что, если приложить s_2 к s_1, начиная с позиции i, то, во-первых, каждому символу s_2 будет соответствовать какой-то символ s_1 (то есть s_2 не выйдет за пределы s_1), а во-вторых, количество соответствующих друг другу совпадающих символов в s_1 и s_2 будет максимально возможным. Другими словами, $i + length(s_2) - 1 \le length(s_1)$ и мощность множества $\{j : s_2[j] = s_1[i + j]\}$ — максимально возможная из всех таких i. Если таких позиций i несколько, найдите первую из них (с наименьшим номером).

Решение должно работать за $O(n \log n)$ в худшем случае.

На входе две непустые строки s_1 и s_2 длины не более 200 000, причем s_2 не длиннее, чем s_1. Выведите номер искомой позиции. Позиции нумеруются с единицы.

Пример входа	Пример выхода
AGTCAGTC GTC	2
AAGGTTCC TCAA	5

Решение.

Сначала мы найдем для каждой из букв A, G, C, T по отдельности и для каждой позиции i, такой что можно приложить s_2 к s_1 в позиции i, количество позиций j, таких что $s_2[j] = s_1[i + j]$ и равно соответствующей букве (A, G, C или T соответственно).

Имея эту информацию, можно перебрать все позиции i слева направо, сложить в них соответствующие значения для букв A, G, C и T и выбрать позицию с максимальной суммой, а из них — самую первую, за линейное время от длины s_1.

Итак, решаем подзадачу: есть две строки s_1 и s_2, состоящие из нулей и единиц (заменим все вхождения нужной буквы на 1, а все остальные буквы на 0 в обеих строках). Длину строки s_1 обозначим за n, а длину строки s_2 — за m. Нужно для каждой позиции i, такой что $i + m \le n$, найти количество позиций j, таких что $s_2[j] = s_1[i + j] = 1$. Можно было бы решать эту задачу полным перебором позиций i, а потом полным перебором позиций j, но это решение работает за $O(nm)$ и нам не подходит. Вместо этого представим себе, что каждая из строк s_1 и s_2 задает многочлен от одной переменной x, все коэффициенты которого — нули и единицы. Строка s_2 задает многочлен $s_2[0]+s_2[1]x+s_2[2]x^2+\cdots+s_2[m-1]x^{m-1}$, а строку s_1 мы развернем в обратную сторону, и она будет задавать многочлен $s_1[n-1] + s_1[n-2]x + \cdots + s_1[0]x^{n-1}$. Найдем произведение этих многочленов: оно будет равно $\sum\limits_{d=0}^{n+m-2} x^d \sum\limits_{j=\max(0,d-n+1)}^{\min(d,m-1)} s_2[j]s_1[n-1-d+j]$, соответственно, коэффициент при x^{n-1-i} этого многочлена при i от 0 до $n - m$ будет равен $\sum\limits_{j=0}^{m-1} s_2[j]s_1[i + j]$, что в свою очередь равно количеству позиций j, в которых $s_2[j] = s_1[i+j] = 1$. Именно эти числа мы и ищем, значит, если мы найдем произведение многочленов, то решим нашу подзадачу. Многочлены будем перемножать с помощью

быстрого преобразования Фурье, как рассказывалось на лекциях, это потребует выполнения $O(n \log n)$ операций.

Итого, общая сложность решения $O(n \log n)$ на умножение многочленов, которое нужно будет сделать 4 раза, по одному на каждую из букв A, G, C, T, и еще $O(n)$ на подготовку многочленов и на получение суммарного результата по каждой позиции — $O(n)$. Памяти нужно $O(n)$.

Задача 3-2 (70 баллов). На дороге в некоторых местах разбросаны золотые монеты. Для каждой монеты известно ее местоположение, которое задается одним целым числом — расстоянием в метрах от начальной отметки. Все монеты расположены правее начальной отметки. Али-баба бегает по дороге и собирает монеты, начиная делать это в момент времени 0. За одну секунду он пробегает ровно один метр. У каждой монеты есть крайний срок, до которого (включительно) ее нужно подобрать, иначе монета исчезнет. Али-баба должен собрать все монеты и сделать это за минимально возможное время. Он может стартовать в любой точке прямой, собирать монеты в произвольном порядке, но обязательно нужно успеть собрать все монеты и при этом минимизировать затраченное время.

В первой строке входа задано число n — количество монет ($1 \le n \le 1000$). В каждой из следующих строк — по 2 целых числа, первое из которых задает положение монеты, а второе — крайний срок в секундах, за который нужно успеть собрать эту монету. Координаты монеток от 0 до 1 000 000. Ограничения по времени также не превосходят 1 000 000. Выведите одно число — минимальное время, за которое Али-баба может собрать все монеты. Если Али-баба не может успеть собрать все монеты, выведите строку No solution.

Пример входа	Пример выхода
5 1 3 3 1 5 8 8 19 10 15	11
5 1 5 2 1 3 4 4 2 5 3	No solution

Решение.

Сделаем для начала несколько наблюдений. Пусть монеты упорядочены по координате слева направо (если нет, переупорядочим их). Рассмотрим оптимальный путь Али-Бабы. Очевидно, Али-Бабе стоит начать свой путь в какой-то из монет, иначе можно выкинуть весь отрезок пути от изначальной точки до первой собранной монеты, и путь станет строго лучше. Очевидно, что в этом пути не может оказаться в какой-то момент, что Али-Баба собрал i-ю монету и j-ю монету, а какую-то монету между ними — не собрал: действительно, он в какой-то момент между посещением монеты i и

монеты j должен был проходить мимо этой монеты, и мог мгновенно ее собрать. От этого путь не ухудшился бы. Далее, разобьем путь на отрезки времени, в конце каждого из которых Али-Баба собирает очередную монету. Из предыдущего замечания следует, что множество всех собранных в конце каждого такого отрезка времени монет представляет собой подотрезок множества всех монет от какой-то i-й монеты до какой-то j-й. Али-Баба не мог оказаться в конце этого промежутка времени где-то между i-й и j-й монетой: действительно, какую-то из монет i и j он собрал раньше, чем другую. Без ограничения общности будем считать, что i-ю он собрал до j-й. Но тогда в тот момент, когда он поднял j-ю монету он уже обязан был к тому же собрать и все монеты между i-й и j-й. Т.е. в момент, когда Али-Баба собирает новую монету, он обязательно стоит в каком-то из концов подотрезка собранных монет.

Если Али-Баба движется оптимальным образом, и мы знаем, что в какой-то момент времени он собрал все монеты от i-й до j-й и при этом еще стоит на i-й монете, то мы можем утверждать, что он проделал все это оптимальным образом: он собрал все монеты от i-й до j-й и оказался рядом с i-й монетой за кратчайшее возможное время, т.к. иначе можно было бы заменить его путь на лучший, и тогда весь путь Али-Бабы тоже оказался целиком лучше, что противоречит предположению об оптимальности. Соответственно, здесь мы видим оптимальность для подзадач, а значит можем воспользоваться динамическим программированием.

Решим задачу для каждой пары монет (i, j), а также для каждого выбора конца отрезка от i-й до j-й монеты, в котором оказался Али-Баба, собрав все монеты от i-й до j-й. Всего $2C_{n+1}^2$ возможных конфигураций. Для $i = j$ ответ очевиден: Али-Баба мог просто начать, стоя в монете номер i, и тогда бы он потратил 0 времени на то, чтобы все это проделать, меньше нельзя. Теперь пусть $i < j$ и, например, Али-Баба в конце оказался в позиции i. Как он мог сделать это оптимальным образом? Он собрал монету i последней. Значит, до этого он собрал все монеты от $i+1$-й до j-й. При этом он последней из них собрал либо $i + 1$-ю, либо j-ю. Пусть наш массив с ответами называется ans, и в $ans[i][j][0]$ лежит время на сбор всех монет от i-й до j-й, $i < j$, так, чтобы оказаться в левом конце отрезка — в i, а в $ans[i][j][1]$ — соответственно то же самое, но чтобы оказаться в правом конце отрезка — в j. Тогда получаем формулу для пересчета ответа:

- $ans[i][j][0] = \min(ans[i+1][j][0] + dist(coins[i], coins[i+1]), ans[i+1][j][1] + dist(coins[i], coins[j]))$
- $ans[i][j][1] = \min(ans[i][j-1][0] + dist(coins[i], coins[j]), ans[i][j-1][1] + dist(coins[j-1], coins[j]))$

Здесь $dist(coins[i], coins[j])$ — расстояние по прямой от i-й монеты до j-й.

Если для какой-то кобминации индексов оказывается, что $ans[i][j][0] > lifetime[i]$ ($lifetime[i]$ — время жизни i-й монеты) или $ans[i][j][1] > lifetime[j]$, то это означает, что Али-Бабе не удастся собрать все монеты от i-й до j-й вовремя и при этом оказаться около нужной монеты. Тогда мы записываем в соответсвующую ячейку массива бесконечность или какое-нибудь специальное значение. Если это значение затем участвует в суммировании и выборе минимума, то оно всегда ведет себя как бесконечнсть. Таким образом, если в конце мы получим, что оба ответа $ans[0][n-1][0]$ и $ans[0][n-1][1]$ — бесконечности, то ответ к задаче — No solution. Если же хоть одно из них конечно, то ответ к задаче — $\min(ans[0][n-1][0], ans[0][n-1][n-1])$.

Таким образом мы можем посчитать ответ сначала для отрезков вида (i, i), затем для отрезков вида $(i, i+1)$, затем для $(i, i+2)$ и т.д. Всего элементов в массиве $O(n^2)$, а каждый пересчет производится за $O(1)$, поэтому общая сложность алгоритма $O(n^2 \times 1) = O(n^2)$. Затраты памяти — также $O(n^2)$, которые используются для хранение массива ans.

Здесь мы опять же увидели оптимальность для подзадач. Здесь также, в отличие от первой задачи, присутствует второй важнейший признак для динамического программирование: большое перекрытие для подзадач. Мы могли бы решать эту задачу и без всякого динамического программирования, просто считать массив ans с помощью рекурсивной функции, которая бы вызывала себя и считала те же самые минимумы. Но в этом случае можно доказать, что решение получилось бы экспоненциальной сложности. Многие подзадачи с короткими подотрезками считались бы заново много раз в случае рекурсивного решения. Это и есть большое перекрытие по подзадачам. За счет динамического программирования мы избегаем повторного вычисления во всех этих подзадачах, и экономим ровно на этом. Эта экономия оказывается настолько существенной, что мы приходим от экспоненциального решения к полиномиальному, всего лишь второй степени. Важно помнить эти два основных признака динамического программирования: оптимальность для подзадач и большое перекрытие подзадач.

Задача 3-3 (75 баллов). Пете поручили написать менеджер памяти для новой стандартной библиотеки языка `H++`. В распоряжении у менеджера находится массив из N последовательных ячеек памяти, пронумерованных от 1 до N. Задача менеджера — обрабатывать запросы приложений на выделение и освобождение памяти. Запрос на выделение памяти имеет один параметр K. Такой запрос означает, что приложение просит выделить ему K последовательных ячеек памяти. Если в распоряжении менеджера есть хотя бы один свободный блок из K последовательных ячеек, то он обязан в ответ на запрос выделить такой блок. При этом наш менеджер выделяет память из самого длинного свободного блока, а если таких несколько, то из них он выбирает тот, у которого номер первой ячейки — наименьший. После этого выделенные ячейки становятся занятыми и не могут быть использованы для выделения памяти, пока не будут освобождены. Если блока из K последовательных свободных ячеек нет, то запрос отклоняется. Запрос на освобождение памяти имеет один параметр T. Такой запрос означает, что менеджер должен освободить память, выделенную ранее при обработке запроса с порядковым номером T. Запросы нумеруются, начиная с единицы. Гарантируется, что запрос с номером T — запрос на выделение, причем к нему еще не применялось освобождение памяти. Освобожденные ячейки могут снова быть использованы для выделения памяти. Если запрос с номером T был отклонен, то текущий запрос на освобождение памяти игнорируется. Требуется написать симуляцию менеджера памяти, удовлетворяющую приведенным критериям.

В первой строке входа два числа N и M — количество ячеек памяти и запросов соответственно ($1 \le N \le 2^{31} - 1$, $1 \le M \le 10^5$). Каждая из следующих M строк содержит по одному числу. $(i+1)$-я строка содержит положительное число K, если i-й запрос — запрос на выделение K ячеек памяти ($1 \le K \le N$), и отрицательное число $-T$, если i-й запрос — запрос на освобождение памяти, выделенной по запросу номер T ($1 \le T < i$).

Для каждого запроса на выделение памяти выведите в выход одно число на отдельной строке с результатом выполнения этого запроса. Если память была выделена,

выведите номер первой ячейки памяти в выделенном блоке, иначе выведите число -1.

Пример входа	Пример выхода
6 8 2 3 -1 3 3 -5 2 2	1 3 -1 -1 1 -1

Решение.

Для решения этой задачи нам понадобится две структуры данных: двоичная куча и двусвязный список. Будем хранить отрезки свободной памяти в куче таким образом, чтобы самый длинный свободный отрезок всегда оказывался в голове кучи. Также будем хранить все свободные и занятые отрезки памяти по порядку в двусвязном списке. Изначально есть единственный свободный отрезок от 1 до N, он лежит и в куче, и в списке. По мере проведения операций по выделению и освобождению памяти, части этого отрезка станут занятыми, а затем освободятся, поэтому может получиться несколько свободных и несколько занятых отрезков. В двусвязном списке все свободные и занятые отрезки идут по порядку, при этом нет двух подряд идущих свободных отрезков: если такая ситуация может произойти, мы просто объединяем соседние свободные отрезки. Все отрезки также должны помнить, на какой позиции в куче они лежат, чтобы их можно было удалять из кучи. Кроме того, заведем массив с результатами запросов, в котором в случае успешного запроса на выделение будет храниться указатель на выделенный отрезок памяти, иначе будет храниться какое-нибудь специальное значение, например `NULL`.

Рассмотрим реализации операций выделения и освобождения памяти.

Выделение памяти:

Находим самый длинный свободный отрезок в голове кучи. Сравниваем его длину с длиной запрашиваемого отрезка памяти. Если длина недостаточна, отклоняем запрос. Если длина свободного отрезка равна K — количеству запрашиваемой памяти — выделяем этот отрезок, удаляем из кучи, помечаем, как занятый. Если длина свободного отрезка более K, выделяем его начальный кусок длины K в отдельный отрезок, а оставшийся конец — в другой отрезок. Первый из этих отрезков — создается как новый занятый отрезок, а второй — получается сдвигом начала изначального наибольшего свободного отрезка вправо и уменьшением его длины. После этого уменьшенный начальный отрезок просеивается вниз по куче, а новый занятый — добавляется в двусвязный список сразу перед ним. Если удалось выделить память, сохраняем в массиве с результатами запросов указатель на выделенный отрезок.

Освобождение памяти:

Находим по указателю в массиве с результатами запросов занятый отрезок. Во-первых, он должен стать свободным. Во-вторых, посмотрим на его соседей в двусвязном списке. Их максимум двое: соседи слева и справа. Тогда если какие-то из них свобод-

ные, то их нужно "приклеить" к нашему отрезку и создать таким образом более длинный свободный отрезок. Их дальнейшие соседи уже не могут быть свободными, т.к. мы поддерживаем свойство, что в двусвязном списке со свободными и занятыми отрезками у свободного отрезка соседи — занятые отрезки. Итак, объединим наш отрезок со всеми свободными соседями (т.е. с 0, 1 или 2 соседями), затем пометим как свободный и добавим в кучу.

Таким образом, мы научились обрабатывать оба запроса. Какая же сложность у наших операций? Операции с двусвязным списком выполняются за константу времени, пометки на отрезках и изменения длин — тоже. Единственная структура данных, с которой происходят операции не за константное время,— куча. В ней операции делаются за $O(size)$, где $size$ — текущий размер кучи. Заметим, что если всего было сделано m запросов, то изначальный свободный отрезок не мог разбиться занятыми на более чем m свободных отрезков. Поэтому размер кучи всегда не более m, а значит все операции с кучей работают за $O(\log(m))$. Итого сложность задачи $O(m\log(m))$. Затраты памяти $O(m)$.

Задача 3-4 (75 баллов). (Задача о скользящей k-й статистике.) В первой строке входа — три целых числа n, m, k. Во второй строке n целых чисел, задающих массив чисел, по которому будут двигаться два указателя, l и r. Изначально оба указателя направлены на самый первый элемент массива. Есть две операции: L — сдвинуть указатель l на один элемент вправо и R — сдвинуть указатель r на один элемент вправо. В третьей строке входного файла — m символов R или L, без пробелов, в одну строку. Это порядок выполняемых операций. Гарантируется, что указатель l никогда не "обгоняет" указатель r. Гарантируется, что указатель r никогда не выйдет за пределы массива. При этом $1 \le n, k \le 100000$, $0 \le m \le 2n-2$. Все числа в массиве неотрицательные и не превосходят 10^9.

Выведите ровно m строк, в каждой — ровно по одному целому числу. После выполнения каждой из операций нужно вывести k-е в порядке возрастания число среди всех чисел от l до r включительно, либо -1, если всего чисел от l до r меньше, чем k.

Пример входа	Пример выхода
7 4 2 4 2 1 3 6 5 7 RRLL	4 2 2 -1
4 6 1 1 2 3 4 RLRLRL	1 2 2 3 3 4

Решение.

Заведем две кучи: `smallestKNumbers` и `otherNumbers`. В первой будем хранить, очевидно, k наименьших чисел из текущего отрезка от l до r (или меньше, если длина текущего отрезка меньше k), а во второй — все остальные числа из данного отрезка. При этом

`smallestKNumbers` будет хранить в голове максимальный элемент, а `otherNumbers` — минимальный. Таким образом, в каждый момент времени, если на отрезке от l до r есть хотя бы k элементов, то k-я порядковая статистика находится в голове `smallestKNumbers`, иначе k-я порядковая статистика не существует, и тогда размер `smallestKNumbers` меньше k.

Нам потребуются следующие операции от обеих куч:

- Добавить элемент в кучу,
- Узнать минимум в куче,
- Удалить данный элемент массива из той кучи, в которой он находится.

Две первые операции реализуются как в обычной куче, и они работают за логарифм от размера кучи. Для третьей операции потребуется дополнительно хранить для каждого элемента массива в какой куче он лежит и на какой позиции. Имея эту информацию, мы умеем удалять элементы из кучи также за логарифм от размера, это было рассказано на лекциях.

Как тогда выглядит наш алгоритм.

Изначально, когда $l = r = 1$, мы добавляем в `smallestKNumbers` первый элемент массива, а `otherNumbers` оставляем пустой. Далее обрабатываем операции сдвига.

Сдвиг r означает, что нужно добавить $r + 1$-й элемент массива в наш отрезок:

1. Добавляем $r + 1$-й элемент в `smallestKNumbers`.
2. Если размер `smallestKNumbers` становится больше k, то удаляем голову `smallestKNumbers` и переносим ее в `otherNumbers`.
3. Выводим k-ю порядковую статистику на основе `smallestKNumbers`.

Сдвиг l означает, что нужно удалить l-й элемент массива из нашего отрезка:

1. Мы знаем, в какой куче и на какой позиции находится l-й элемент массива — удаляем его из этой кучи по этой позиции.
2. Если после этой операции размер `smallestKNumbers` стал меньше k, а куча `otherNumbers` непуста, то удаляем голову `otherNumbers` (в ней лежит наименьший из остальных элементов, т.е. тот, который теперь должен оказаться в `smallestKNumbers`) и добавляем в `smallestKNumbers`.
3. Выводим k-ю порядковую статистику на основе `smallestKNumbers`.

Такое решение работает за $O(m \log n)$ времени, т.к. каждый запрос обрабатывает за $O(\log n)$. Памяти тратим линейное количество, т.к. нам нужно хранить не более n элементов отрезка в кучах в каждый момент времени, помимо основного массива, и про каждый из них нам нужна константа дополнительной информации.

Замечания по реализации.

Чтобы в каждый момент времени иметь корректную информацию о том, в какой куче и на какой позиции находится каждый элемент массива, необходимо для каждого элемента хранить дополнительную информацию, например, в такой структуре:

```
struct ArrayElementInHeap {
  int value;
  enum {NONE, SMALLEST_K_NUMBERS, OTHER_NUMBERS} whichHeapIn;
  int heapPosition;
};
```

Необходимо иметь параллельный исходному массив `arrayElements` из `ArrayElementInHeap` и хранить в кучах, например, указатели на элементы на элементы этого массива. При операциях с кучами может происходить одно из двух:

1. Два элемента одной кучи обмениваются позициями — нужно поменять и их `heapPosition` друг с другом. Эту операцию необходимо инкапсулировать в методе `smartSwap` кучи, т.к. она точно используется как минимум 3 раза в операциях `siftUp` и `siftDown`, а на самом деле ее можно использовать вместо "ручных" решений почти во всех методах обычной кучи.

2. Элемент добавляется в кучу — в этот момент необходимо правильно инициализировать `whichHeapIn` и `heapPosition`.

3. Элемент удаляется из кучи — в этот момент необходимо присвоить `whichHeapIn = NONE;`, а также присвоить `heapPosition` какое-нибудь специальное значение.

Как мы видим, здесь в некоторой степени нарушается инкапсуляция данных: две разных кучи имеют доступ к элементам одного и того же массива `arrayElements`, что чревато рассинхронизацией данных. Почему так произошло? Потому что на самом деле здесь необходим более умный дизайн. Вместо того, чтобы работать с массивом чисел и двумя кучами отдельно, нужно понять, что на самом деле от нас хотят. А от нас хотят всего лишь реализовать интерфейс очереди, у которой кроме стандартных операций есть еще операция "узнать k-ю порядковую статистику среди элементов в очереди".

Ну давайте реализуем такой интерфейс в виде класса `QueueWithKthOrderStatistics`, в который и добавим в качестве членов как обычную очередь `arrayElements` (в которой для каждого элемента, находящегося в очереди, хранится `ArrayElementInHeap`, указывающий, в какой куче и на какой позиции он находится), так и обе кучи `smallestKNumbers` и `otherNumbers`.

При этом сдвиг r вправо означает выполнение операции `Push` для такой очереди, при этом нужно добавить $r + 1$-й элемент исходного (внешнего) массива с числами в `arrayElements`, проинициализировать его по умолчанию, а затем произвести указанные выше действия с кучами. Сдвиг l влево означает операцию Pop для нашей кучи, которая представляет собой удаление первого элемента из `arrayElements` сначала указанным выше образом из соответствующей кучи, а потом вызовом `arrayElements.Pop();`. Операция `GetKthStatistics` нашей очереди будет смотреть на кучу `smallestKNumbers`, и если в ней есть k элементов, то возвращать ее голову, иначе ответит, что k-й порядковой статистики нет.

Т.к. теперь все данные инкапсулированы внутрь нашей собственной очереди, и все методы, имеющие к ним доступ, пишем мы сами, можно не беспокоиться за целостность данных, нужно только аккуратно реализовать операции $Push$ и Pop как указано.

Отметим, что нам не нужно делать очередь `arrayElements` на основе вектора размера n, мы можем всегда в ней хранить только те элементы, которые сейчас в очереди. Таким

образом, мы поддерживаем операции с очередью “онлайн”, т.е. если бы вместо исходного массива у нас был бы просто поток запросов извне, вида “добавить элемент”, “удалить элемент”, “узнать k-ю порядковую статистику”, то мы могли бы обрабатывать эти запросы с помощью нашего класса за гарантированную логарифмическую сложность от текущего количества сохраненных элементов.

Далее, есть еще одна маленькая проблема: нам нужны две кучи, одна из которых в голове хранит минимум, а другая — максимум, и мы, конечно же, не хотим из-за этого писать две разные кучи. На самом деле, если задуматься, эти две кучи ничем не отличаются, кроме функции сравнения. Поэтому код нужно написать для всей кучи один раз, при этом пользуясь для сравнения элементов некоторой специальной функцией сравнения, которой можно параметризовать кучу. Сделать это можно как минимум тремя способами (в порядке предпочтения):

1. Параметризовать шаблонный класс кучи функтором сравнения элементов, как принято в STL.

2. Передавать указатель на функцию сравнения элементов в конструктор кучи и запоминать.

3. Реализовать базовый класс кучи и породить от него два класса — кучу с минимумом в голове и кучу с максимумом в голове — в которых переопределен только виртуальный метод сравнения элементов кучи.

.

Первый способ наиболее предпочтителен, поскольку привычен и много раз опробован в STL. Третий способ наименее предпочтителен, поскольку наиболее затратен как в смысле написания кода и количества классов в системе классов, так и с точки зрения производительности.

Главное, что ни в коем случае нельзя писать две эти кучи с помощью copy-paste, т.к. дублирование кода — самое главное зло, с которым необходимо бороться.