Задача 1

Полностью объяснить задачу из второго задания № 7. Под «полностью объяснить» подразумевается все — от свойств используемых структур данных до строчек кода.

Формат сдачи — предоставленный исходный код плюс устная беседа по нему.

Задача 2

Разобрать алгоритм шейкер-сортировки массива (см. ниже). Оценить асимптотическую сложность алгоритма.

Формат сдачи — названная асимптотическая сложность (как «О большое от чего-то») плюс устная беседа по алгоритму.

Задача 3

На базе любой удобной задачи второго задания реализовать структуру данных и интерфейс к ней (указаны ниже). Программа при запуске должна читать команды из файла commands.txt, выполнять их, печатать результат на экран и завершаться. Никаких интерактивных действий от пользователя не предполагается.

Структура данных: двухсвязный список

Тип данных: строка не более 15 символов

Состав инструкций в commands.txt:

INSERT A — добавить элемент со значением A, ничего не печатать. Если такой элемент уже есть — добавить еще одну копию.

FIND A — найти элемент со значением A. Если элемент найден, напечатать FOUND. Если не найден — напечатать NOT FOUND.

DELETE A — найти и удалить элемент со значением A. Если элемента нет — ничего не делать. Если элементов несколько — удалить одну из копий.

DELETE_ALL A — найти и удалить все вхождения элемента со значением A.

Пример:

Инструкции в файле:

INSERT alice

INSERT alice

INSERT alice

DELETE alice

FIND alice

DELETE_ALL alice

FIND alice

Вывод программы:

FOUND (так как две alice еще остались)

NOT FOUND (так как теперь все alice удалены)

Формат сдачи — предоставленный исходный код плюс устная беседа по нему.

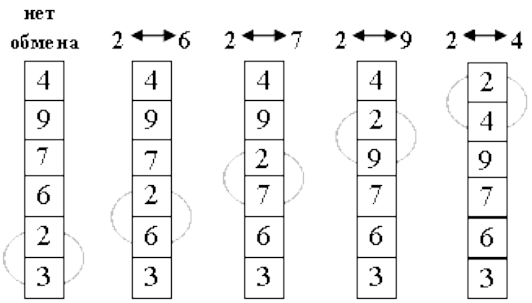
Шейкер-сортировка

Задача — отсортировать массив.

Предшественник шейкер-сортировка — сортировка методом пузырька. Логика ее работы следующая.

Расположим массив сверху вниз, от нулевого элемента - к последнему.

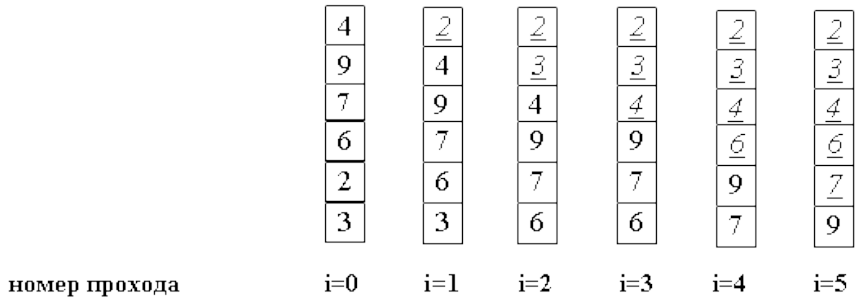
Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами.



Нулевой проход, сравниваемые пары выделены

После нулевого прохода по массиву "вверху" оказывается самый "легкий" элемент - отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом второй по величине элемент поднимается на правильную позицию...

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.



Пример реализации:

```
template<class T>
void bubbleSort(T a[], long size) {
  long i, j;
  T x;
  for( i=0; i < size; i++) {            // i - номер прохода
    for( j = size-1; j > i; j-- ) {     // внутренний цикл прохода
      if ( a[j-1] > a[j] ) {
        x=a[j-1]; a[j-1]=a[j]; a[j]=x;
      }
    }
```

```
  }
}
```

Метод сортировки пузырьком можно по-всякому улучшать. Чем мы сейчас и займемся.

Во-первых, рассмотрим ситуацию, когда на каком-либо из проходов не произошло ни одного обмена. Что это значит ?

Это значит, что все пары расположены в правильном порядке, так что массив уже отсортирован. И продолжать процесс не имеет смысла(особенно, если массив был отсортирован с самого начала !).

Итак, первое улучшение алгоритма заключается в запоминании, производился ли на данном проходе какой-либо обмен. Если нет - алгоритм заканчивает работу.

Процесс улучшения можно продолжить, если запоминать не только сам факт обмена, но и индекс последнего обмена k. Действительно: все пары соседих элементов с индексами, меньшими k, уже расположены в нужном порядке. Дальнейшие проходы можно заканчивать на индексе k, вместо того чтобы двигаться до установленной заранее верхней границы i.

Качественно другое улучшение алгоритма можно получить из следующего наблюдения. Хотя легкий пузырек снизу поднимется наверх за один проход, тяжелые пузырьки опускаются со минимальной скоростью: один шаг за итерацию. Так что массив 2 3 4 5 6 1 будет отсортирован за 1 проход, а сортировка последовательности 6 1 2 3 4 5 потребует 5 проходов.

Чтобы избежать подобного эффекта, можно менять направление следующих один за другим проходов. Получившийся алгоритм иногда называют "шейкер-сортировкой".

Пример реализации:

```
template<class T>
void shakerSort(T a[], long size) {
  long j, k = size-1;
  long lb=1, ub = size-1; // границы неотсортированной части массива
  T x;
  do {
       // проход снизу вверх
    for( j=ub; j>0; j-- ) {
      if ( a[j-1] > a[j] ) {
        x=a[j-1]; a[j-1]=a[j]; a[j]=x;
        k=j;
      }
    }
    lb = k+1;
    // проход сверху вниз
    for (j=1; j<=ub; j++) {
      if ( a[j-1] > a[j] ) {
```

```
        x=a[j-1]; a[j-1]=a[j]; a[j]=x;
        k=j;
      }
    }
    ub = k-1;
  } while ( lb < ub );
}
```

Насколько описанные изменения повлияли на эффективность метода?