**Лабораторные**

1. **Лабораторная: proc.loop.array\_inverter\_optimized**

**proc.loop.array\_inverter\_optimized**

В коде инвертирования массива

public class ArrayInverter {

public static void invert(int[] arr) {

for (int k = 0; k < arr.length / 2; k++) {

int tmp = arr[k];

arr[k] = arr[arr.length - k - 1];

arr[arr.length - k - 1] = tmp;

}

}

}

необходимо переписать цикл с инкремента счетчика от 0 до середины массива на цикл от середины массива до 0 (инкремент (k++) заменить на декремент (k--)).

1. **Лабораторная: proc.loop.array\_merger\_fixed**

**proc.loop.array\_merger\_fixed**

Необходимо корректно реализовать слияние сортированных массивов

public class Merger {

// wrong

public static int[] merge(int[] fst, int[] snd) {

int[] result = new int[fst.length + snd.length];

int fstIndex = 0;

int sndIndex = 0;

while (fstIndex + sndIndex != result.length) {

if (fst[fstIndex] < snd[sndIndex]) {

result[fstIndex + sndIndex] = fst[fstIndex++];

} else {

result[fstIndex + sndIndex] = snd[sndIndex++];

}

}

return result;

}

}

В данной заготовке метод merge реализован с ошибкой - он корректно сливает, но некорректно обрабатывает ситуацию, когда один из массивов полностью выбран.  
1. Исправьте ошибку.  
------------------  
Дополнительные улучшения:  
2. После того как обнаружится, что один из массивов "исчерпан" копируйте данные из второго массива в результат не в цикле, а одним вызовом System.arraycopy(...).  
3. Попробуйте минимизировать количество операций (сравнения, сумма, присвоение, ...) на один "сливаемый" элемент ("сравнения" намного дороже арифметических операций). Сейчас в коде 6 операций:  
- fstIndex + sndIndex  
- ... != result.length  
- fst[fstIndex] < snd[sndIndex]  
- fstIndex + sndIndex  
- fstIndex++  
- result[...] = fst[...]

1. **Лабораторная: proc.loop.bubble\_sort\_inverted**

**proc.loop.bubble\_sort\_inverted**

Алгоритм "сортировки пузырьком"

public class BubbleSorter {

public static void sort(int[] arr) {

for (int barrier = arr.length - 1; barrier >= 0; barrier--) {

for (int index = 0; index < barrier; index++) {

if (arr[index] > arr[index + 1]) {

int tmp = arr[index];

arr[index] = arr[index + 1];

arr[index + 1] = tmp;

}

}

}

}

}

допускает вариации и оптимизации. В данном случае приведен алгоритм, при котором на каждом проходе "всплывает" самый большой элемент. Перепишите алгоритм на такой, при котором "тонет" самый маленький элемент. Порядок сортировки должен сохраниться - по возрастанию. Элементы должны перебираться справа - налево. Всплывал - вправо, тонет - влево. Цикл не доходил до правого конца, теперь - не доходит до левого.  
В массиве из 6 элементов должны сравниваться (и в случае инверсии переставляться) пары  
4-5 3-4 2-3 1-2 0-1  
4-5 3-4 2-3 1-2  
4-5 3-4 2-3  
4-5 3-4  
4-5

1. **Лабораторная: proc.loop.selection\_sort\_optimized**

**proc.loop.selection\_sort\_optimized**

Переписать алгоритм сортировки выборками:

public class SelectionSorter {

public static void sort(int[] arr) {

for (int barrier = 0; barrier < arr.length - 1; barrier++) {

for (int index = barrier + 1; index < arr.length; index++) {

if (arr[barrier] > arr[index]) {

int tmp = arr[index];

arr[index] = arr[barrier];

arr[barrier] = tmp;

}

}

}

}

}

1. Убрать обмен элементами arr[barrier] c arr[index] каждый раз, когда находится очередной меньший элемент. Найти наименьший элемент во всем массиве - и обменять с ним.  
2. Убрать обращение во внутреннем цикле к элементу массива arr[barrier]. Вычитать значение ячейки массива в локальную переменную (за пределами внутреннего цикла) и использовать ее (во внутреннем цикле).  
В экспериментах пункт #1 ускорил сортировку в 2 раза. Пункт #2 ускорил сортировку еще на 10%-20%.

**5. Лабораторная: proc.loop.insertion\_sort\_optimized**

**proc.loop.insertion\_sort\_optimized**

Такая версия алгоритм сортировки вставками

public class InsertionSorter {

public static void sort(int[] arr) {

for (int k = 1; k < arr.length; k++) {

int newElement = arr[k];

int location = k - 1;

while (location >= 0 && arr[location] > newElement) {

arr[location + 1] = arr[location];

location--;

}

arr[location + 1] = newElement;

}

}

}

реализована не оптимально, так как внутренний цикл while  
- ищет позицию для вставки, перебирая последовательно элементы, при этом он  
- поэлементно "смещает" массив.  
В целях оптимизации перепишите алгоритм:  
1. Ищите позицию для вставки элемента бинарным поиском (Arrays.binarySearch(...)).  
2. Найдя позицию - смещайте всю часть массива за один вызов (System.arraycopy(...)).  
В экспериментах эти две оптимизации ускорили сортировку в 2.2-2.6 раза.

1. **Лабораторная: proc.loop.matrix\_mul**

**proc.loop.matrix\_mul**

Реализовать метод, умножающий две 2-мерные матрицы (не обязательно квадратные).  
Произведение двух двумерных матриц матриц A и B размером NxM и MxK- это матрица С размером NxK, где каждый элемент C[i][j] высчитывается по формуле:  
C[i][j] = A[i][0]\*B[0][j] + A[i][1]\*B[1][j] + ... + A[i][N-1]\*B[N-1][j]  
Пример:

|1 2| |3 4 5 6| |21 4 7 10|

|7 8| \* |9 0 1 2| = |93 28 43 58|

|3 4| |45 12 19 26|

так как  
1\*3 + 2\*9 = 21, 1\*4 + 2\*0 = 4, 1\*5 + 2\*1 = 7, 1\*6 + 2\*2 = 10  
7\*3 + 8\*9 = 93, 7\*4 + 8\*0 = 28, 7\*5 + 8\*1 = 43, 7\*6 + 8\*2 = 58  
3\*3 + 4\*9 = 45, 3\*4 + 4\*0 = 12, 3\*5 + 4\*1 = 19, 3\*6 + 4\*2 = 26  
1. Если один из аргументов равен null - throw new IllegalArgumentException().  
2. Если один из аргументов НЕ прямоугольная/квадратная матрица - throw new IllegalArgumentException().  
3. Если одна из матриц содержит null в качестве строки - throw new IllegalArgumentException().  
4. Если ширина первой матрицы не равна высоте второй матрицы - throw new IllegalArgumentException().

Подсказка: вам необходим тройной вложенный цикл.  
Ссылка: можете посмотреть [статью в вики: "Умножение матриц"](http://ru.wikipedia.org/wiki/%D0%A3%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BC%D0%B0%D1%82%D1%80%D0%B8%D1%86).