**Задача «sortByFastest»:**

На вход получаем неупорядоченные наборы объектов (чисел), перегруженные по операциям сравнения (>,>=,==,<,<=), после чего требуется их сортировать. Для решения мы можем использовать множество алгоритмов сортировок, но среднее время работы каждого алгоритма зависит от:

1) Технических характеристик машины.

2) Длины неупорядоченного набора.

В следствие этого, знание зависимости временной сложности всякого алгоритма от тех.характеристик машины и длины неупорядоченного набора, может позволить автоматически выбирать самую быструю сортировку. Для этого достаточно снабдить каждый алгоритм собственной теоретической функцией, предсказывающей время работы конкретного алгоритма от пункта 1 и 2.

**Теоретическая функция предсказания времени работы:**

В академической среде часто используют следующую формальную модель временной сложности, для предсказания времени работы – выражаем через некоторую константу разовое время выполнения последовательного блока кода, а через n время повторений этого блока от длины n входного набора данных. Это модель несколько упрощающая реальное поведение компьютера, но обладающее достаточной прикладной ценностью, чтобы расчитывать лучшее, среднее и худшее время работы алгоритма. Конечная формула выглядит навроде time(n) = C1 \* (n-1) + C2 \* (n – 1) \* n)

**Методология эмпирического замера времени:**

Для дальнейшей работы важно точно фиксировать время работы некоторого отрезка кода, для чего работу того многократно воспроизводят с единым набором входным данных и фиксируют время.

Важна и корректная обработка множественных замеров – несмотря на крайне детерминированную работу компьютера, время замера каждый раз будет сильно отличаться, во многом из-за специфики работы ОС (возможности приостанавливать работу нашей программы на время ради прочих процессов). По этой причине наиболее точное время среди множественных замеров не среднее, а наименьшее – такой подход позволяет получать более точный и абсолютный, и сравнительный с другими алгоритмами замер на более малой выборке тестов.

**Учет пункта 1 (сложности алгоритма от длины набора «n»):**

Функции наихудшего времени работы известных алгоритмов сортировок зачастую известны. Но в данной программе я буду рассчитывать формулы для среднего времени работы отталкиваясь от эмпирических данных по генерации случайных наборов чисел разной длины n. Формулы будут записываться в методы, переопределяющие виртуальный метод getTime.

**Учет пункта 2 (тех.характеристик комьютера в теоретической функции алгоритма):**

В каждую сортировку будем включать счетчики по однородным блокам кода (циклам), чтобы фиксировать число циклов и конечное время работы всякого алгоритма от частного неупорядоченного набора. Константы «C» тем самым будут изначально иметь завышенное значение (погрешность единичного замера времени), но уменьшаться со временем при взятии более точного замера.

Для уточнения будем фиксировать вплоть до 2-ух циклов в сортировке, записывая их в систему линейных алгебраических уравнений в формате

**Замер1** {C1 \* + C2 \* = }

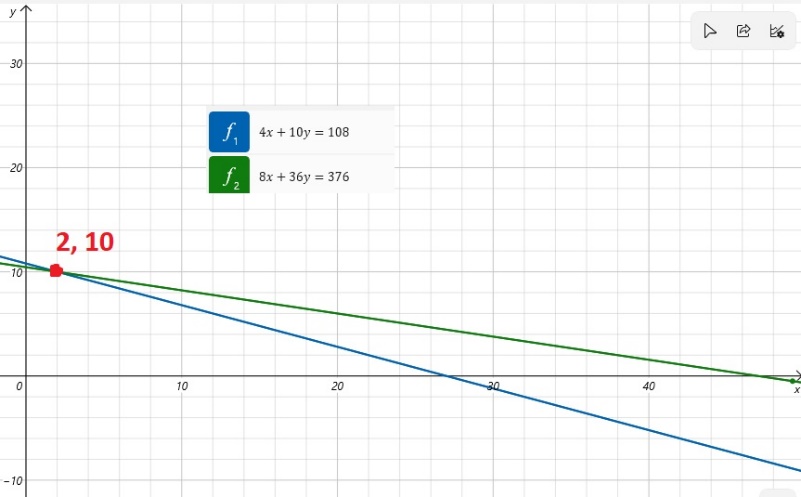
**Замер2** { C1 \* + C2 \* = }

*//одинаковые в системы лишь константы*

Методом Гаусса будем решать систему этих линейных уравнений для определения значения констант (зависимость времени от тех.характеристик компьютера).

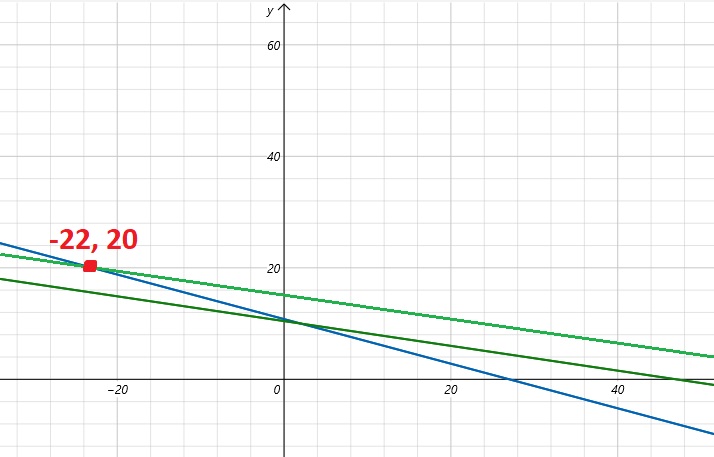
**Опускание строки наименьшей точности времени до строки наибольшей точности времени:**

Ожидаемое решение должно иметь константы большие, либо равные нулю. В таком случае мы получим графика навроде:



*//график идеальной точности*

Но реальные замеры времени будут иметь значение времени большее реального, чтобы будет поднимать и пропорционально увеличивать значения пересечения графика с осями. Пример:



*//эмпирическая параллельна реальной, но выше и правее ее в пересечении осей Х и Y (C1 и C2).*

Соответственно, так как мы знаем, что реальное значение пересечения лежит в положительно оси, при пересечении в отрицательной оси (2 или 4) мы можем гарантировано сдвигать одну из прямых до нулевого значения, гарантируя, что реальная кривая не будет больше этой. На этом принципе основан метод «narrowTimeAccuracy».

**Замена строки наименьшей точности времени новой строкой наибольшей точности времени:**

Если в функции уравнения разных скоростей **n,** то константы скорее всего будут строго больше нуля обе. Для этого нам нужно вставлять строку более точного времени в замен одной из текущих менее точного.

Критерий большей точности следующий – вставка новых строк будет приближать систему к реальной, если новая строка пересекает ось х или y в более малом значении (близком к началу системы координат), чем одна из существующих. В таком случае эта «существующая» заменяется на новую, иначе изменений не происходит.

Корректность этого подхода проверяется на тестовой функции, которая от известных констант генерирует точные строки, добавляя увеличивающий шум в их время, и убеждается, что с течением времени отбор строк приведет к реальным константам.

**Особенности сложности алгоритма «sortByFastest»:**

Помимо зависимости от длины неупорядоченного набора **n**, теперь время работы зависит от числа предшествующих вызовов **m** конкретной сортировки для замера ее времени.

Чем больше n, тем дольше в среднем выполняется конкретная сортировка, что очевидно. Но чем больше **m** (коррелирует со временем работы программы), тем реже нам приходится пересчитывать константы и пересортировывать множество сортировок.

Мы можем настраивать количество искусственных калибровок при инициализации (сами генерируем неупорядоченные наборы для сортировки). Далее программа переходит на прикладную работу, где сортирует реальные запросы, и калибрует константы по ним.

Издержки делятся на:

1. Подсчет циклов (не зависит от **m**). Имеет вид теоретической функции временной сложности сортировки, но значения констант гораздо меньше.
2. Пересчет констант (зависит от **m**). Имеет константное время выполнения для СЛАУ единого порядка (в нашем случае 2-ого).
3. Пересортировка алгоритмов сортировок O(n^2), где n это число самих алгоритмов. Частота срабатывания зависит от **m**.

По достижению определенного **m** число уточняющих калибровок прекратится на машине единой конфигурации, но все еще будет происходить подсчет циклов и проверка на большую точность замера.

**Слабые места sortByFastest и потенциальные улучшения:**

Существует проблема различной выборки замеров. Например, если входные неупорядоченные наборы имеют **n** в пределах от (1, 1000], и 1-ая сортировка срабатывает на интервале (1;100], а 2-ая на интервале (100;1000] после инициализации, то 1-ая будет использоваться в 10 раз реже и следовательно в 10 раз реже калиброваться. Для очень малого интервала это может привести к тому, что он будет вымещен более правым, но с другой стороны, длина такого интервала мала и можно свести это к погрешности конечного алгоритма.

Так же есть проблема теоретических функций единого порядка скорости – например, если функции обеих сортировок линейны, то набравшая меньшую константу на инициализации выкинет вторую, даже если на большей дистанции это было бы наоборот. Во много эта проблема того, что точка пересечения таких функций известна заранее и равна началу отсчета координат.

Теоретические функции так же имеют свою погрешность.

Возможные простые решения верхних проблем:

1. Тривиальное увеличение инициализирующий выборки для калибровки, так как алгоритмы там калибруются единой выборкой.
2. Можно выделить интервал малого **n** (например <1000), на котором нам позволительно калибровать даже потенциально плохие сортировки, так как преимущество алгоритма мы ожидаем на гораздо больших **n**. C достижением некоторой выборки всяким алгоритмом мы прекращаем его проверку и калибровку, полагая сложившиеся константы достаточно близкими к реальным, а выигрыш от избавления затрат на замеры большим, чем их продолжение с крайне редкими и малыми калибровками.
3. Более точные теоретические функции можно получать, если изначально калибровать их значениями ближе к границе интервалов (возможно разделение на правые и левые).