Реализация алгоритма IntroSort

ссылка на репозиторий:
https://github.com/AnnaAstashkina/A3_set3

id посылки: 293157049

код:

```
#include <iostream>
#include <vector>
#include <cmath>
#include <random>

int parent(int i) {
  return i / 2 - (i + 1) % 2;
}
int left(int i, int start) {
  return start + 2 * (i - start) + 1;
}
int right(int i, int start) {
  return start + 2 * (i - start) + 2;
}

void insertionSort(std::vector<int>& vect, int start, int end) {
  for (int i = start + 1; i <= end; ++i) {
    int key = vect[i];
    int j = i - 1;
    while (j >= start && vect[j] > key) {
      vect[j + 1] = vect[j];
      j--;
    }
    vect[j + 1] = key;
  }
}

void heapify(std::vector<int>& A, int i, int start, int end) {
  int j = i;
  int lef = left(i, start);
  int righ = right(i, start);
```

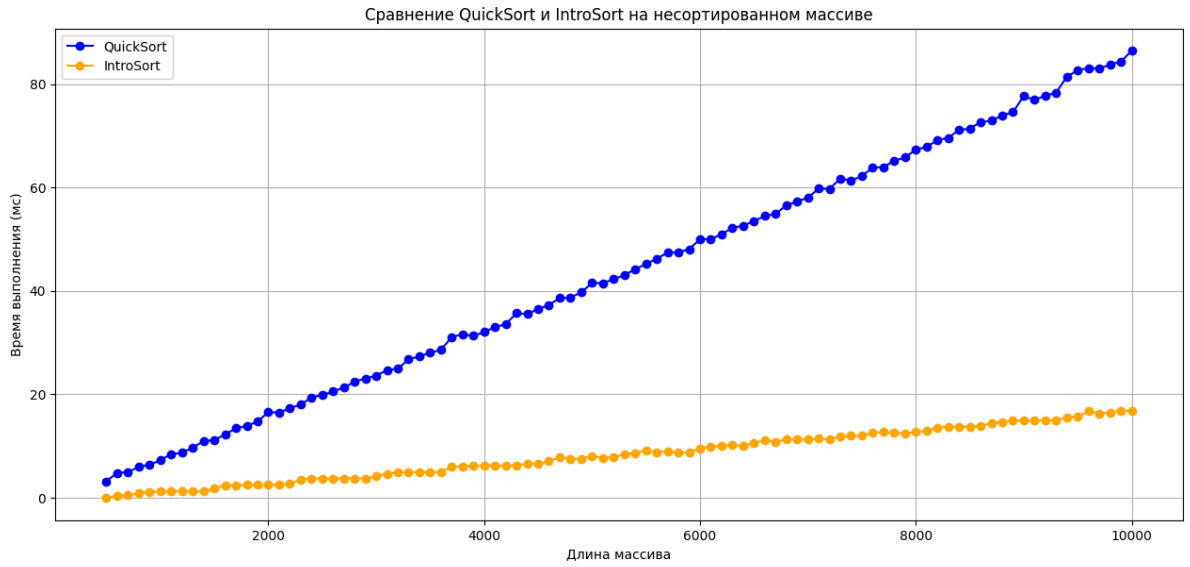
```
  if (lef <= end && A[lef] > A[j]) {
    j = lef;
  }
  if (righ <= end && A[righ] > A[j]) {
    j = righ;
  }
  if (j != i) {
    std::swap(A[j], A[i]);
    heapify(A, j, start, end);
  }
}

void buildMaxHeap(std::vector<int>& A, int start, int end) {
  int n = end - start + 1;
  for (int i = n / 2 - 1; i > -1; --i) {
    heapify(A, i + start, start, end);
  }
}

void heapSort(std::vector<int>& A, int start, int end) {
  buildMaxHeap(A, start, end);
  for (int i = end; i > start; --i) {
    std::swap(A[i], A[start]);
    heapify(A, start, start, i - 1);
  }
}

int partition(std::vector<int>& vect, int start, int end) {
  std::random_device rand_dev;
  std::mt19937 generator(rand_dev());
  std::uniform_int_distribution<> distr(start, end);
  int ind = distr(generator);
  std::swap(vect[ind], vect[end]);
  int pivot = vect[end];
  int i = start - 1;
  for (int j = start; j < end; ++j) {
    if (vect[j] <= pivot) {
      ++i;
      std::swap(vect[i], vect[j]);
    }
  }
  std::swap(vect[i + 1], vect[end]);
  return i + 1;
```

```
}

void quickSort(std::vector<int>& vect, int start, int end) {
  if (start >= end) {
    return;
  }
  int pivot = partition(vect, start, end);
  quickSort(vect, start, pivot - 1);
  quickSort(vect, pivot + 1, end);
}

void introsort(std::vector<int>& vect, int start, int end, int depth) {
  if (start >= end) {
    return;
  }
  if (end - start + 1 <= 15) {
    insertionSort(vect, start, end);
    return;
  }
  if (depth >= 2 * std::log2(vect.size())) {
    heapSort(vect, start, end);
    return;
  }
  int pivot = partition(vect, start, end);
  introsort(vect, start, pivot - 1, depth + 1);
  introsort(vect, pivot + 1, end, depth + 1);
}

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int n = 0;
  std::cin >> n;
  std::vector<int> vect(n);
  for (int i = 0; i < n; ++i) {
    std::cin >> vect[i];
  }
  introsort(vect, 0, n - 1, 0);
  for (int i = 0; i < n; ++i) {
    std::cout << vect[i] << " ";
  }
  return 0;
}
```

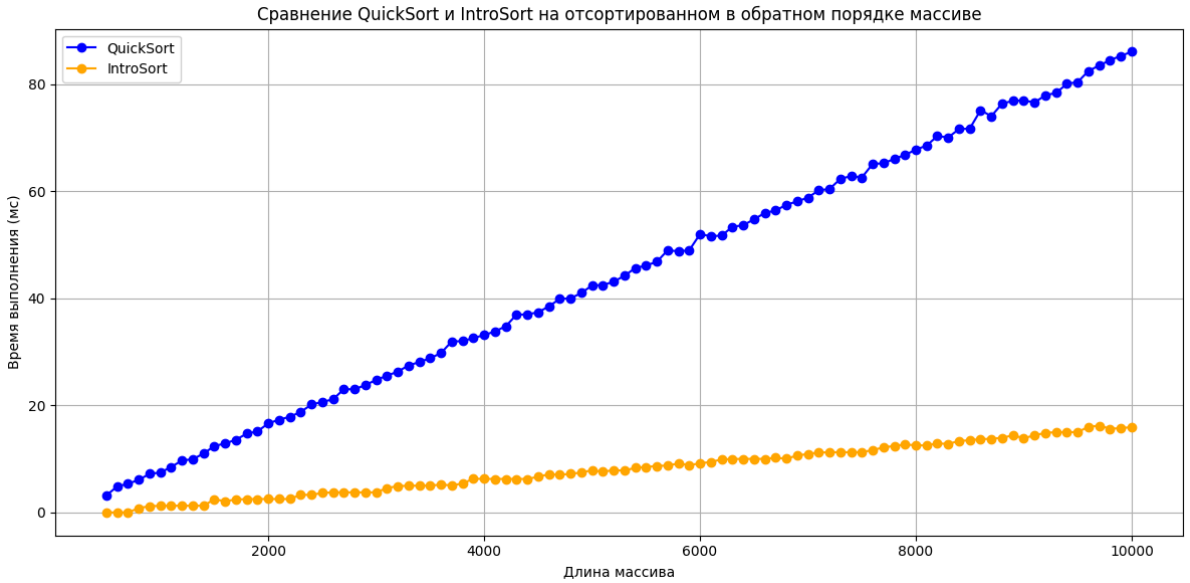
Графики и анализ

Графики для удобства анализа построены на одной картинке!

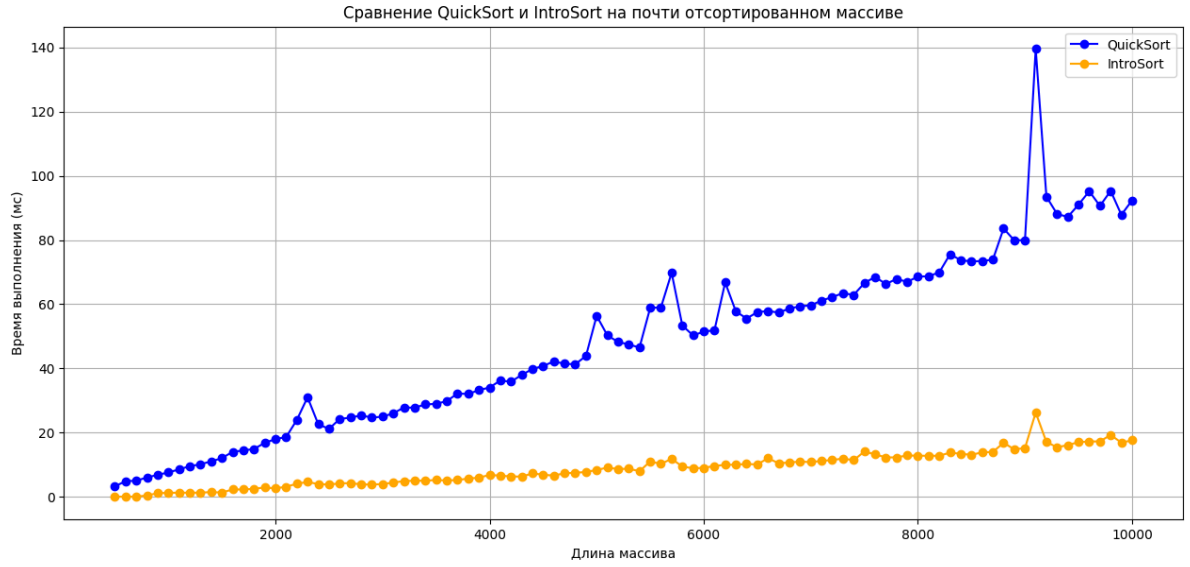
Графики времени выполнения (в мс) QuickSort и IntroSort для несортированного массива



Графики времени выполнения (в мс) QuickSort и IntroSort для сортированного в обратном порядке массива



Графики времени выполнения (в мс) QuickSort и IntroSort для частично-сортированного массива



Вывод про сравнения QuickSort и IntroSort

Из графиков видно, что на всех вариантах входных данных IntroSort работает значительно быстрее за счет своей гибридности, QuickSort же работает достаточно медленно на больших объемах входных данных вне зависимости от их вида.

График времени выполнения для QuickSort на разных входных данных

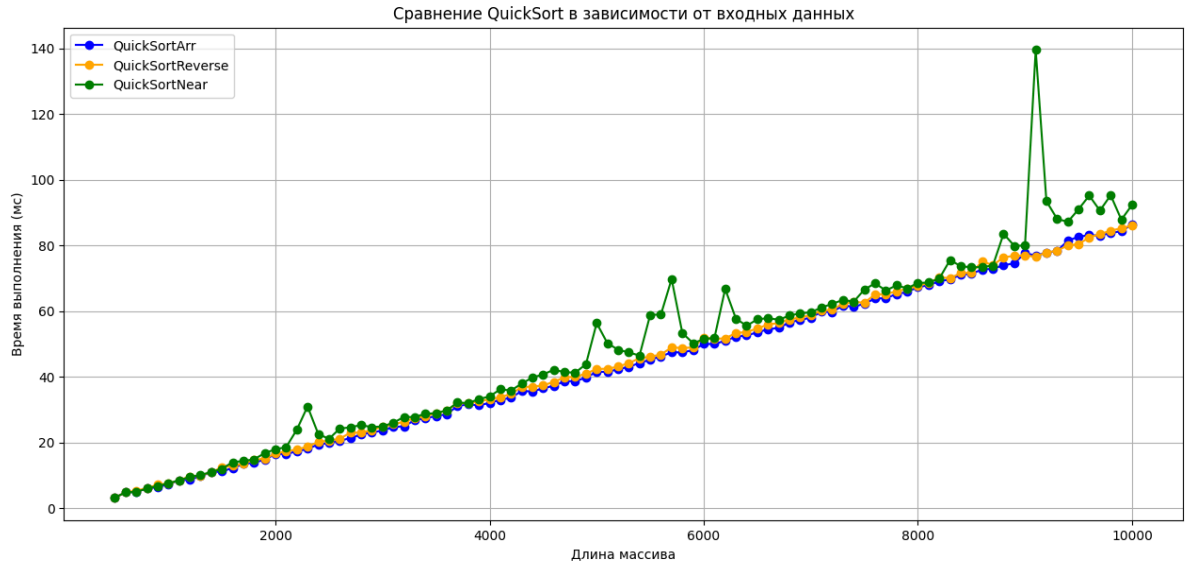


График времени выполнения для IntroSort на разных входных данных

Вывод про время работы QuickSort и IntroSort на разных входных данных

Из графиков видно, что разные входные данные не сильно влияют на время работу алгоритмов. Время увеличивается прямо пропорционально объему входных данных. Можно сделать вывод, что обоим алгоритмам не важна структура сортируемых массивов.

Общий код для получения результатов

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>
#include <cmath>

class ArrayGenerator {
 public:
  static std::vector<int> generateRandomArray() {
    std::random_device rand_dev;
    std::mt19937 generator(rand_dev());
    std::uniform_int_distribution<> distr(0, 6000);
    std::vector<int> arr(10000);
    for (int i = 0; i < 10000; ++i) {
```

```
    arr[i] = distr(generator);
  }
  return arr;
}

static std::vector<int> generateReverseSortedArray() {
  std::random_device rand_dev;
  std::mt19937 generator(rand_dev());
  std::uniform_int_distribution<> distr(0, 6000);
  std::vector<int> arr(10000);
  for (int i = 0; i < 10000; ++i) {
    arr[i] = distr(generator);
  }
  std::sort(arr.begin(), arr.end());
  std::reverse(arr.begin(), arr.end());
  return arr;
}

static std::vector<int> generateNearlySortedArray() {
  std::random_device rand_dev;
  std::mt19937 generator(rand_dev());
  std::uniform_int_distribution<> distr(0, 6000);
  std::vector<int> arr(10000);
  for (int i = 0; i < 10000; ++i) {
    arr[i] = distr(generator);
  }
  std::sort(arr.begin(), arr.end());
  for (int i = 0; i < 15; ++i) {
    std::uniform_int_distribution<> distrs(0, 500);
    int idx1 = distrs(generator);
    int idx2 = distrs(generator);
    std::swap(arr[idx1], arr[idx2]);
  }
  for (int i = 500; i <= 9900; i += 100) {
    std::uniform_int_distribution<> distrs(i, i + 100);
    for (int j = 0; j < 3; ++j) {
      int idx1 = distrs(generator);
      int idx2 = distrs(generator);
      std::swap(arr[idx1], arr[idx2]);
    }
  }
  return arr;
}
```

```
};

class SortTester {
 private:
  static int left(int i, int start) {
    return start + 2 * (i - start) + 1;
  }
  static int right(int i, int start) {
    return start + 2 * (i - start) + 2;
  }

  static void insertionSort(std::vector<int>& vect, int start, int end)
{
    for (int i = start + 1; i <= end; ++i) {
      int key = vect[i];
      int j = i - 1;
      while (j >= start && vect[j] > key) {
        vect[j + 1] = vect[j];
        j--;
      }
      vect[j + 1] = key;
    }
  }

  static void heapify(std::vector<int>& A, int i, int start, int end) {
    int j = i;
    int lef = left(i, start);
    int righ = right(i, start);
    if (lef <= end && A[lef] > A[j]) {
      j = lef;
    }
    if (righ <= end && A[righ] > A[j]) {
      j = righ;
    }
    if (j != i) {
      std::swap(A[j], A[i]);
      heapify(A, j, start, end);
    }
  }

  static void buildMaxHeap(std::vector<int>& A, int start, int end) {
    int n = end - start + 1;
    for (int i = n / 2 - 1; i > -1; --i) {
```

```
      heapify(A, i + start, start, end);
    }
  }

  static void heapSort(std::vector<int>& A, int start, int end) {
    buildMaxHeap(A, start, end);
    for (int i = end; i > start; --i) {
      std::swap(A[i], A[start]);
      heapify(A, start, start, i - 1);
    }
  }

  static int partition(std::vector<int>& vect, int start, int end) {
    std::random_device rand_dev;
    std::mt19937 generator(rand_dev());
    std::uniform_int_distribution<> distr(start, end);
    int ind = distr(generator);
    std::swap(vect[ind], vect[end]);
    int pivot = vect[end];
    int i = start - 1;
    for (int j = start; j < end; ++j) {
      if (vect[j] <= pivot) {
        ++i;
        std::swap(vect[i], vect[j]);
      }
    }
    std::swap(vect[i + 1], vect[end]);
    return i + 1;
  }

  static void quickSort(std::vector<int>& vect, int start, int end) {
    if (start >= end) {
      return;
    }
    int pivot = partition(vect, start, end);
    quickSort(vect, start, pivot - 1);
    quickSort(vect, pivot + 1, end);
  }

  static void introsort(std::vector<int>& vect, int start, int end, int
depth) {
    if (start >= end) {
      return;
```

```
    }
    if (end - start + 1 <= 15) {
      insertionSort(vect, start, end);
      return;
    }
    if (depth >= 2 * std::log2(vect.size())) {
      heapSort(vect, start, end);
      return;
    }
    int pivot = partition(vect, start, end);
    introsort(vect, start, pivot - 1, depth + 1);
    introsort(vect, pivot + 1, end, depth + 1);
  }

 public:
  static long long testQuickSort(std::vector<int>& arr) {
    auto start = std::chrono::high_resolution_clock::now();
    quickSort(arr, 0, arr.size() - 1);
    auto elapsed = std::chrono::high_resolution_clock::now() - start;
    return
std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
  }

  static long long testIntroSort(std::vector<int>& arr) {
    auto start = std::chrono::high_resolution_clock::now();
    introsort(arr, 0, arr.size() - 1, 0);
    auto elapsed = std::chrono::high_resolution_clock::now() - start;
    return
std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
  }
};

template <typename T>
void printVector(const std::vector<T>& vec) {
  for (int i = 0; i < vec.size(); ++i) {
    std::cout << vec[i] << ", ";
  }
  std::cout << std::endl << std::endl;
}

int main() {
  std::vector<int> randomArrParent =
ArrayGenerator::generateRandomArray();
```

```
  std::vector<int> reverseArrParent =
ArrayGenerator::generateReverseSortedArray();
  std::vector<int> nearlySortedArrParent =
ArrayGenerator::generateNearlySortedArray();
  double timeQuickArr;
  double timeIntrosortArr;
  double timeQuickReverse;
  double timeIntrosortReverse;
  double timeQuickNear;
  double timeIntrosortNear;
  std::vector<double> vecTimeQuickArr;
  std::vector<double> vecTimeIntrosortArr;
  std::vector<double> vecTimeQuickReverse;
  std::vector<double> vecTimeIntrosortReverse;
  std::vector<double> vecTimeQuickNear;
  std::vector<double> vecTimeIntrosortNear;
  std::vector<int> is;
  for (int i = 500; i <= 10000; i += 100) {
    is.push_back(i);
    for (int j = 0; j < 5; ++j) {
      std::vector<int> randomArr =
std::vector<int>(randomArrParent.begin(), randomArrParent.begin() + i);
      timeQuickArr += SortTester::testQuickSort(randomArr);
      randomArr = std::vector<int>(randomArrParent.begin(),
randomArrParent.begin() + i);
      timeIntrosortArr += SortTester::testIntroSort(randomArr);

      std::vector<int> reverseArr =
std::vector<int>(reverseArrParent.begin(), reverseArrParent.begin() +
i);
      timeQuickReverse += SortTester::testQuickSort(reverseArr);
      reverseArr = std::vector<int>(reverseArrParent.begin(),
reverseArrParent.begin() + i);
      timeIntrosortReverse += SortTester::testIntroSort(reverseArr);

      std::vector<int> nearlySortedArr =
          std::vector<int>(nearlySortedArrParent.begin(),
nearlySortedArrParent.begin() + i);
      timeQuickNear += SortTester::testQuickSort(nearlySortedArr);
      nearlySortedArr = std::vector<int>(nearlySortedArrParent.begin(),
nearlySortedArrParent.begin() + i);
      timeIntrosortNear += SortTester::testIntroSort(nearlySortedArr);
    }
```

```
    timeQuickArr /= 5.;
    timeIntrosortArr/=5.;
    timeQuickReverse /= 5.;
    timeIntrosortReverse /= 5.;
    timeQuickNear /= 5.;
    timeIntrosortNear /= 5.;

    vecTimeQuickArr.push_back(timeQuickArr);
    vecTimeIntrosortArr.push_back(timeIntrosortArr);
    vecTimeQuickReverse.push_back(timeQuickReverse);
    vecTimeIntrosortReverse.push_back(timeIntrosortReverse);
    vecTimeQuickNear.push_back(timeQuickNear);
    vecTimeIntrosortNear.push_back(timeIntrosortNear);
  }
  printVector(is);
  printVector(vecTimeQuickArr);
  printVector(vecTimeIntrosortArr);
  printVector(vecTimeQuickReverse);
  printVector(vecTimeIntrosortReverse);
  printVector(vecTimeQuickNear);
  printVector(vecTimeIntrosortNear);
}
```

Результаты, на основе которых строились графики

size: 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000, 2100, 2200, 2300, 2400, 2500, 2600, 2700, 2800, 2900, 3000, 3100, 3200, 3300, 3400, 3500, 3600, 3700, 3800, 3900, 4000, 4100, 4200, 4300, 4400, 4500, 4600, 4700, 4800, 4900, 5000, 5100, 5200, 5300, 5400, 5500, 5600, 5700, 5800, 5900, 6000, 6100, 6200, 6300, 6400, 6500, 6600, 6700, 6800, 6900, 7000, 7100, 7200, 7300, 7400, 7500, 7600, 7700, 7800, 7900, 8000, 8100, 8200, 8300, 8400, 8500, 8600, 8700, 8800, 8900, 9000, 9100, 9200, 9300, 9400, 9500, 9600, 9700, 9800, 9900, 10000

quick_arr: 3.2, 4.84, 4.968, 5.9936, 6.39872, 7.27974, 8.45595, 8.69119, 9.73824, 10.9476, 11.1895, 12.2379, 13.4476, 13.8895, 14.7779, 16.5556, 16.5111, 17.3022, 18.0604, 19.4121, 19.8824, 20.5765, 21.3153, 22.4631, 23.0926, 23.6185, 24.7237, 24.9447, 26.7889, 27.3578, 28.0716, 28.6143, 31.1229, 31.6246, 31.3249, 32.065, 33.013, 33.6026, 35.7205, 35.5441, 36.5088, 37.3018, 38.6604, 38.7321, 39.7464, 41.5493, 41.5099, 42.302, 43.0604, 44.2121, 45.2424, 46.2485, 47.4497, 47.4899, 48.098, 50.0196, 50.0039, 51.0008, 52.2002, 52.64, 53.528, 54.5056, 54.9011, 56.5802, 57.316, 58.0632, 59.8126, 59.7625, 61.7525, 61.3505, 62.2701, 63.854, 63.9708, 65.1942, 65.8388, 67.3678, 67.8736, 69.1747, 69.6349, 71.127, 71.4254, 72.6851, 72.937, 73.9874, 74.5975, 77.7195, 76.9439, 77.7888, 78.3578, 81.4716, 82.6943, 83.1389, 83.0278, 83.8056, 84.3611, 86.4722

intro_arr: 1.1659e-303, 0.4, 0.48, 0.896, 1.1792, 1.23584, 1.24717, 1.24943, 1.24989, 1.24998, 1.85, 2.37, 2.474, 2.4948, 2.49896, 2.49979, 2.49996, 2.69999, 3.54, 3.708, 3.7416, 3.74832, 3.74966, 3.74993, 3.74999, 4.15, 4.63, 4.926, 4.9852, 4.99704, 4.99941, 4.99988, 5.99998, 6, 6.2, 6.24, 6.248, 6.2496, 6.24992, 6.64998, 6.53, 7.106, 7.8212, 7.56424, 7.51285, 8.10257, 7.62051, 7.9241, 8.38482, 8.67696, 9.13539, 8.82708, 8.96542, 8.79308, 8.75862, 9.55172, 9.91034, 9.98207, 10.1964, 10.0393, 10.6079, 11.1216, 10.8243, 11.3649, 11.273, 11.2546, 11.4509, 11.2902, 11.858, 11.9716, 11.9943, 12.5989, 12.7198, 12.544, 12.5088, 12.7018, 12.9404, 13.5881, 13.7176, 13.7435, 13.7487, 13.9497, 14.3899, 14.678, 14.9356, 14.9871, 14.9974, 14.9995, 14.9999, 15.6, 15.72, 16.744, 16.3488, 16.4698, 16.894, 16.7788

quick_reverse: 3.2, 4.84, 5.368, 6.0736, 7.21472, 7.44294, 8.48859, 9.69772, 9.93954, 10.9879, 12.3976, 12.8795, 13.5759, 14.7152, 15.143, 16.6286, 17.3257, 17.8651, 18.773, 20.1546, 20.6309, 21.1262, 23.0252, 23.005, 23.801, 24.7602, 25.552, 26.3104, 27.4621, 28.0924, 28.8185, 29.7637, 31.9527, 31.9905, 32.5981, 33.1196, 33.8239, 34.7648, 36.953, 36.9906, 37.3981, 38.4796, 39.8959, 39.9792, 40.9958, 42.3992, 42.4798, 43.096, 44.2192, 45.6438, 46.1288, 46.8258, 48.9652, 48.793, 48.9586, 51.9917, 51.5983, 51.7197, 53.3439, 53.6688, 54.7338, 55.9468, 56.3894, 57.4779, 58.0956, 58.8191, 60.1638, 60.4328, 62.2866, 62.8573, 62.5715, 65.1143, 65.2229, 66.0446, 66.8089, 67.7618, 68.5524, 70.3105, 70.0621, 71.6124, 71.7225, 75.1445, 74.0289, 76.4058, 76.8812, 76.9762, 76.5952, 77.919, 78.3838, 80.0768, 80.4154, 82.4831, 83.4966, 84.4993, 85.2999, 86.06

intro_reverse: 8.50767e-315, 1.70153e-315, 3.40307e-316, 0.8, 1.16, 1.232, 1.2464, 1.24928, 1.24986, 1.24997, 2.44999, 2.09, 2.418, 2.4836, 2.49672, 2.49934, 2.49987, 2.49997, 3.29999, 3.26, 3.652, 3.7304, 3.74608, 3.74922, 3.74984, 3.74997, 4.54999, 4.91, 4.982, 4.9964, 4.99928, 5.19986, 5.03997, 5.40799, 6.2816, 6.25632, 6.25126, 6.25025, 6.25005, 6.25001, 6.65, 7.13, 7.026, 7.2052, 7.44104, 7.88821, 7.57764, 7.91553, 7.78311, 8.35662, 8.47132, 8.69426, 8.73885, 9.14777, 8.82955, 9.16591, 9.43318, 9.88664, 9.97733, 9.99547, 9.99909, 9.99982, 10.2, 10.04, 10.608, 10.9216, 11.1843, 11.2369, 11.2474, 11.2495, 11.2499, 11.65, 12.13, 12.426, 12.6852, 12.537, 12.5074, 12.9015, 12.7803, 13.3561, 13.4712, 13.6942, 13.7388, 13.9478, 14.3896, 13.8779, 14.3756, 14.8751, 14.975, 14.995, 14.999, 15.9998, 16.2, 15.64, 15.728, 15.9456

quick_near: 3.2, 4.84, 4.968, 5.9936, 6.79872, 7.55974, 8.51195, 9.50239, 10.1005, 11.0201, 12.004, 14.0008, 14.4002, 14.88, 16.776, 17.9552, 18.591, 23.9182, 30.9836, 22.5967, 21.1193, 24.2239, 24.6448, 25.329, 24.6658, 24.9332, 25.9866, 27.7973, 27.7595, 28.7519, 28.9504, 29.7901, 32.158, 32.0316, 33.2063, 34.0413, 36.2083, 35.8417, 37.9683, 39.7937, 40.7587, 42.1517, 41.4303, 41.2861, 43.8572, 56.3714, 50.2743, 48.2549, 47.451, 46.4902, 58.898, 58.9796, 69.7959, 53.3592, 50.2718, 51.4544, 51.8909, 66.7782, 57.7556, 55.5511, 57.5102, 57.902, 57.3804, 58.6761, 59.3352, 59.667, 61.1334, 62.2267, 63.4453, 62.8891, 66.5778, 68.5156, 66.3031, 67.8606, 66.9721, 68.5944, 68.7189, 69.9438, 75.5888, 73.7178, 73.3436, 73.4687, 73.8937, 83.5787, 79.9157, 79.9831, 139.597, 93.5193, 88.1039, 87.2208, 91.0442, 95.2088, 90.6418, 95.3284, 87.8657, 92.3731

intro_near: 7.41842e-309, 1.48368e-309, 2.96737e-310, 0.4, 1.08, 1.216, 1.2432, 1.24864, 1.24973, 1.44995, 1.28999, 2.258, 2.2516, 2.45032, 2.89006, 2.57801, 3.1156, 4.02312, 4.60462, 3.92092, 3.78418, 4.15684, 4.23137, 3.84627, 3.76925, 3.95385, 4.39077,

4.87815, 4.97563, 4.99513, 5.19903, 5.03981, 5.20796, 5.64159, 5.92832, 6.78566, 6.55713, 6.31143, 6.26229, 7.25246, 6.85049, 6.5701, 7.31402, 7.4628, 7.69256, 8.33851, 9.0677, 8.61354, 8.72271, 7.94454, 10.9889, 10.1978, 11.8396, 9.36791, 8.87358, 8.97472, 9.59494, 9.91899, 9.9838, 10.1968, 10.0394, 12.0079, 10.4016, 10.6803, 10.9361, 10.7872, 11.1574, 11.4315, 11.8863, 11.3773, 14.0755, 13.2151, 12.243, 12.2486, 12.8497, 12.7699, 12.754, 12.7508, 13.7502, 13.35, 13.07, 13.814, 13.9628, 16.7926, 14.7585, 15.1517, 26.2303, 17.0461, 15.4092, 16.0818, 17.0164, 17.2033, 17.2407, 19.2481, 16.8496, 17.5699