SET-9. Задача A1. Анализ строковых сортировок

Посылки на codeforces:

- A1m: 321152528
- A1q: 321222267
- A1r: 321222093
- A1rq: 321222972

Ссылка на GitHub репозиторий: https://github.com/maklybae/algorithms/tree/main/set09/a1

Для замера производительности кода используется класс `SortTester`:

```
class SortTester {
 public:
  using Ms = std::chrono::microseconds;

  SortTester(int seed, int max_count, double nearly_factor) : generator_(seed,
max_count, nearly_factor) {
  }

  CmpSortTesterResult TestDefaultQuickSort(size_t size) {
    return TestDefault(QuickSort, size);
  }

  CmpSortTesterResult TestDefaultMergeSort(size_t size) {
    return TestDefault(MergeSort, size);
  }

  SortTesterResult TestMsdRadixSort(size_t size) {
    return Test(StringMsdRadixSort, size);
  }

  SortTesterResult TestStringQuickSort(size_t size) {
    return Test(StringQuickSort, size);
  }

  SortTesterResult TestStringMergeSort(size_t size) {
    return Test(StringMergeSort, size);
  }

  SortTesterResult TestStringComboSort(size_t size) {
    return Test(StringComboSort, size);
  }

 private:
  std::pair<long long, int> TestDefault(std::function<int(std::vector<std::string>&)>
sort_func,
                                        std::vector<std::string>& data) {
    auto start = std::chrono::high_resolution_clock::now();
    int cmp_count = sort_func(data);
    auto end = std::chrono::high_resolution_clock::now();
    long long time = std::chrono::duration_cast<Ms>(end - start).count();
    return {time, cmp_count};
  }

  CmpSortTesterResult TestDefault(std::function<int(std::vector<std::string>&)>
sort_func, size_t size) {
    CmpSortTesterResult result{};
    std::vector<std::string> random = generator_.Random(size);
    std::vector<std::string> reversed = generator_.Reversed(size);
    std::vector<std::string> nearly_sorted = generator_.NearlySorted(size);
    result.random = TestDefault(sort_func, random);
    result.reversed = TestDefault(sort_func, reversed);
    result.nearly_sorted = TestDefault(sort_func, nearly_sorted);
    return result;
```

```
  }

  long long Test(std::function<void(std::vector<std::string>&)> sort_func,
std::vector<std::string>& data) {
    auto start = std::chrono::high_resolution_clock::now();
    sort_func(data);
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<Ms>(end - start).count();
  }

  SortTesterResult Test(std::function<void(std::vector<std::string>&)> sort_func,
size_t size) {
    SortTesterResult result{};
    std::vector<std::string> random = generator_.Random(size);
    std::vector<std::string> reversed = generator_.Reversed(size);
    std::vector<std::string> nearly_sorted = generator_.NearlySorted(size);
    result.random_time = Test(sort_func, random);
    result.reversed_time = Test(sort_func, reversed);
    result.nearly_sorted_time = Test(sort_func, nearly_sorted);
    return result;
  }

  StringGenerator generator_;
};
```

`SortTester` использует класс `StringGenerator`, который генерирует случайные строки, строки, отсортированные в обратном порядке, и почти отсортированные строки.

```
class StringGenerator {
 public:
  StringGenerator(int seed, int max_count, double nearly_factor)
      : max_count_(max_count), random_engine_(seed), nearly_factor_(nearly_factor) {
    InitRandom();
    InitReversed();
    InitNearlySorted();
  }

  std::vector<std::string> Random(int count) const {
    std::vector<std::string> result;
    result.insert(result.end(), random_.begin(), std::next(random_.begin(), count));
    return result;
  }

  std::vector<std::string> Reversed(int count) const {
    std::vector<std::string> result;
    result.insert(result.end(), reversed_.begin(), std::next(reversed_.begin(),
count));
    return result;
  }

  std::vector<std::string> NearlySorted(int count) const {
    std::vector<std::string> result;
    result.insert(result.end(), nearly_sorted_.begin(),
std::next(nearly_sorted_.begin(), count));
    return result;
  }

 private:
  void InitRandom() {
    random_ = GenerateRandomStrings();
  }

  void InitReversed() {
    reversed_ = GenerateRandomStrings();
    std::sort(reversed_.begin(), reversed_.end(), std::greater<>());
  }

  void InitNearlySorted() {
    nearly_sorted_ = GenerateRandomStrings();
    std::sort(nearly_sorted_.begin(), nearly_sorted_.end());
```

```
    size_t swaps_count = static_cast<size_t>(nearly_factor_ * static_cast<double>
(max_count_));

    while (swaps_count > 0) {
      size_t i = random_engine_() % max_count_;
      size_t j = random_engine_() % max_count_;
      if (i != j) {
        std::swap(nearly_sorted_[i], nearly_sorted_[j]);
        --swaps_count;
      }
    }
  }

  std::string GenerateRandomString() {
    std::string str;
    for (size_t i = 0; i < kMinLength + random_engine_() % (kMaxLength - kMinLength +
1); ++i) {
      str += kAlphabet[random_engine_() % kAlphabet.size()];
    }
    return str;
  }

  std::vector<std::string> GenerateRandomStrings() {
    std::vector<std::string> result;
    result.reserve(max_count_);
    for (size_t i = 0; i < max_count_; ++i) {
      result.emplace_back(GenerateRandomString());
    }
    return result;
  }

  static constexpr std::string_view kAlphabet =
      "!@#%:;^&*()-0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
  static constexpr size_t kMinLength = 10;
  static constexpr size_t kMaxLength = 200;

  int max_count_ = 0;
  double nearly_factor_ = 0;
  std::vector<std::string> random_;
  std::vector<std::string> reversed_;
  std::vector<std::string> nearly_sorted_;
  std::mt19937 random_engine_;
};
```

Для начала необходимо получить результаты тестов.

```
mkdir -p cpp/bin \
    && cd cpp/bin \
    && cmake .. \
    && make
```

После сборки проекта можно запустить программу и получить результаты тестов:

```
./main > ../../results.csv
```

Наконец, можно приступить к анализу результатов.

In [6]:
```
import pandas as pd
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

plt.style.use("seaborn-v0_8-whitegrid")
```

In [28]:
```
df = pd.read_csv("results.csv", delim_whitespace=True)
df
```

```
/var/folders/7m/_s2dd8053g97sjss_c0260mc0000gn/T/ipykernel_47041/1929270259.py:1: FutureWarning:
The 'delim_whitespace' keyword in pd.read_csv is deprecated and will be removed in a future vers
ion. Use ``sep='\s+'`` instead
  df = pd.read_csv("results.csv", delim_whitespace=True)
```

Out[28]:

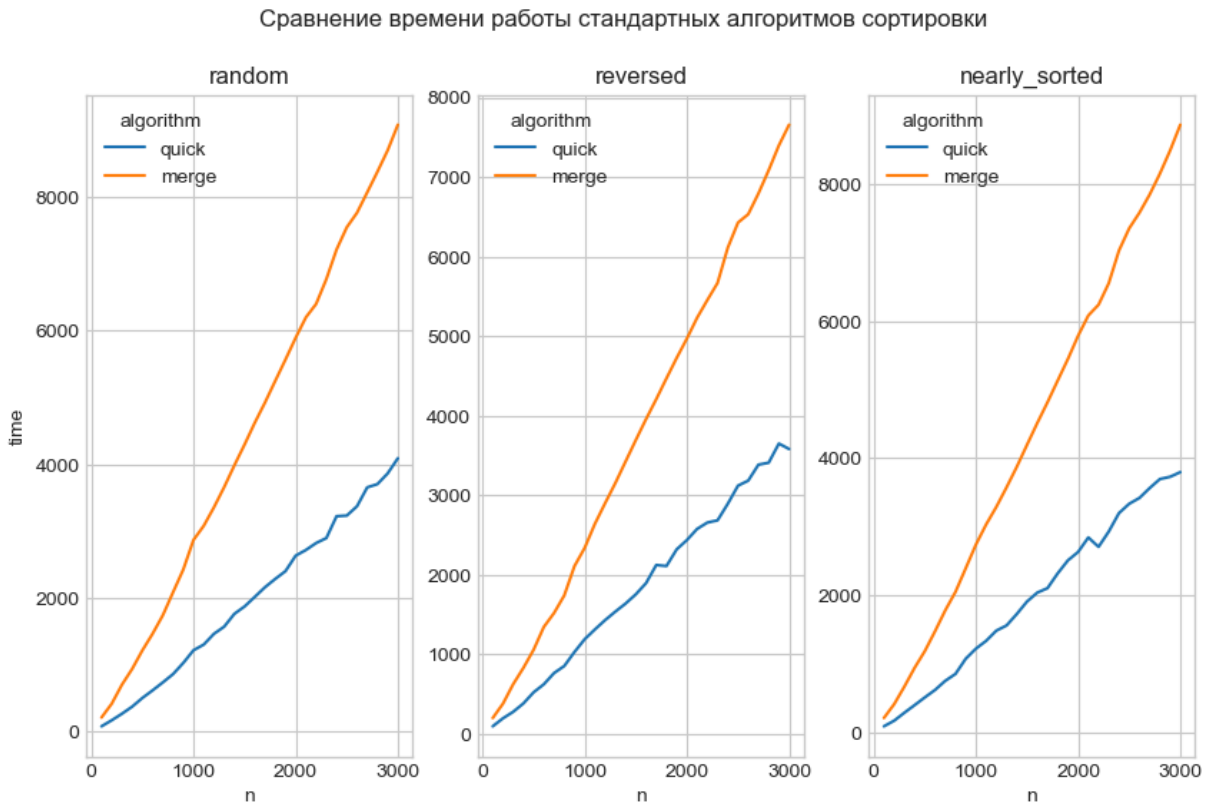
	n	algorithm	array_type	time	cmp
0	100	quick	random	77	534.0
1	100	quick	reversed	95	1033.0
2	100	quick	nearly_sorted	85	645.0
3	100	merge	random	209	70.0
4	100	merge	reversed	199	335.0
...	...	...	...	...	...
535	3000	string_merge	reversed	5460	NaN
536	3000	string_merge	nearly_sorted	6572	NaN
537	3000	string_combo	random	5397	NaN
538	3000	string_combo	reversed	5347	NaN
539	3000	string_combo	nearly_sorted	5375	NaN

540 rows × 5 columns

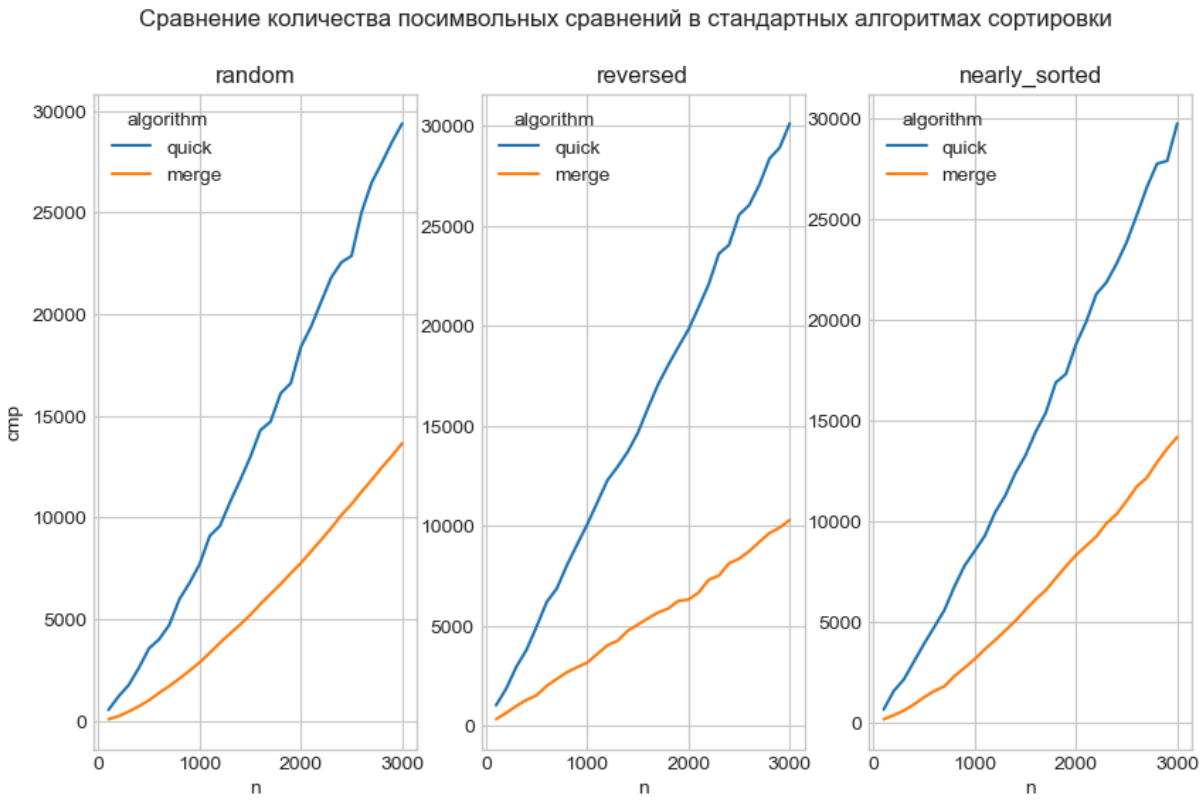
Построим графики сравнения времени и количества посимвольных сравнений для `QuickSort` и `MergeSort`.

In [41]:

```
fig, ax = plt.subplots(ncols=3, figsize=(10, 6))

for i, array_type in enumerate(["random", "reversed", "nearly_sorted"]):
    sns.lineplot(
        data=df[df["algorithm"].isin(["quick", "merge"]) & (df["array_type"] == array_type)],
        x="n",
        y="time",
        hue="algorithm",
        ax=ax[i],
    )
    ax[i].set_title(array_type)
    if i > 0:
        ax[i].set_ylabel("")

fig.suptitle("Сравнение времени работы стандартных алгоритмов сортировки");
```



In [42]:
```
fig, ax = plt.subplots(ncols=3, figsize=(10, 6))

for i, array_type in enumerate(["random", "reversed", "nearly_sorted"]):
    sns.lineplot(
        data=df[df["algorithm"].isin(["quick", "merge"]) & (df["array_type"] == array_type)],
        x="n",
        y="cmp",
        hue="algorithm",
        ax=ax[i],
    )
    ax[i].set_title(array_type)
    if i > 0:
        ax[i].set_ylabel("")

fig.suptitle("Сравнение количества посимвольных сравнений в стандартных алгоритмах сортировки")
```

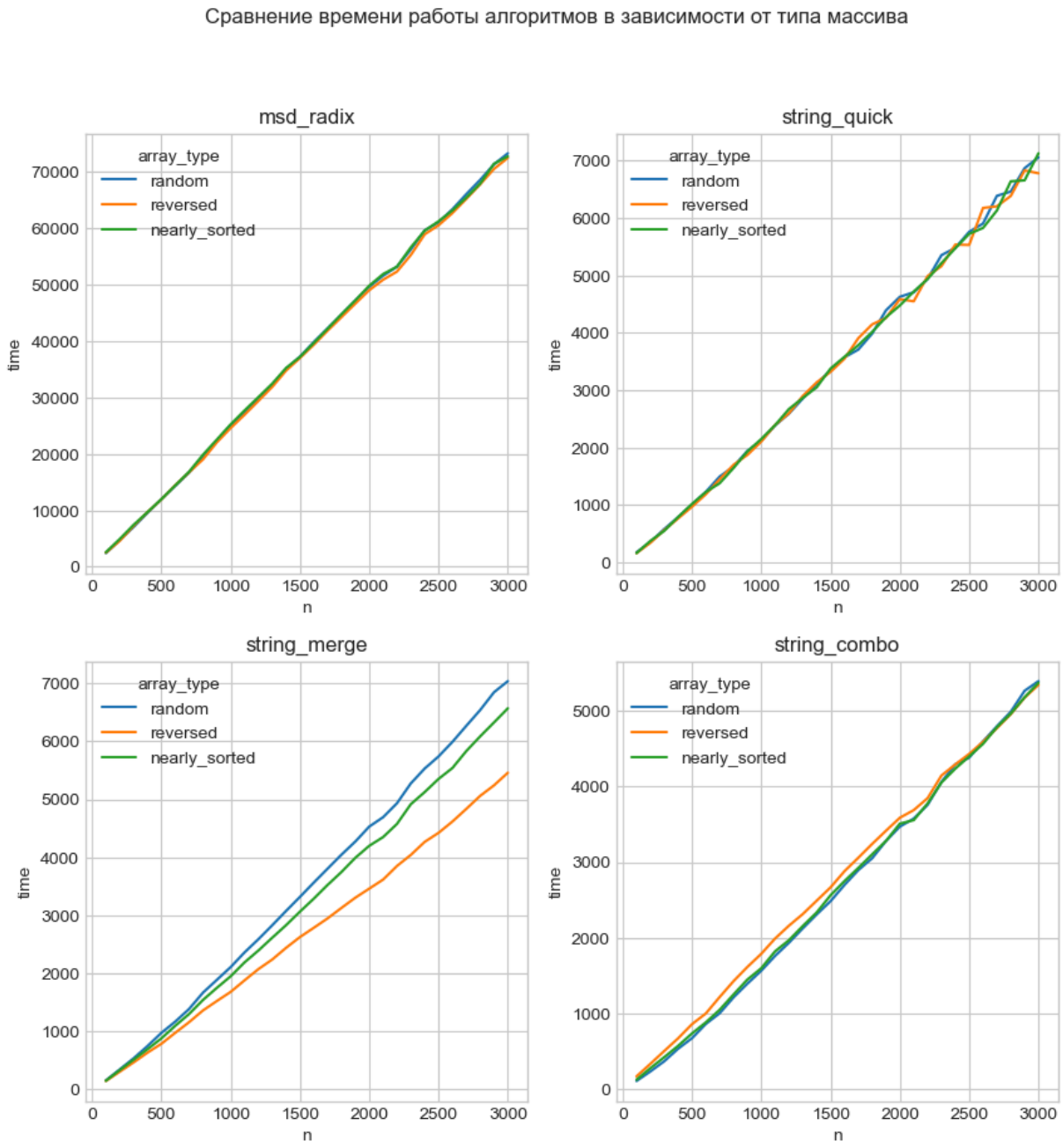


Заметим, что для каждого типа массивов `Quick Sort` работает быстрее, чем `MergeSort`, но при этом `Merge Sort` делает меньше посимвольных сравнений. Скорее всего, разница во времени работы возникает из-за специфики `Merge Sort` — разложить на разные массивы, уйти в рекурсию, а так как используются строки, то их перекладывание занимает много времени. Количество сравнений у `Merge Sort` меньше, так как на этапе `Merge` каждый из подмассивов отсортирован в правильном порядке и в этом случае требуется меньше сравнений.

Перейдем к рассмотрению отдельно адаптированных под строки алгоритмов сортировки.

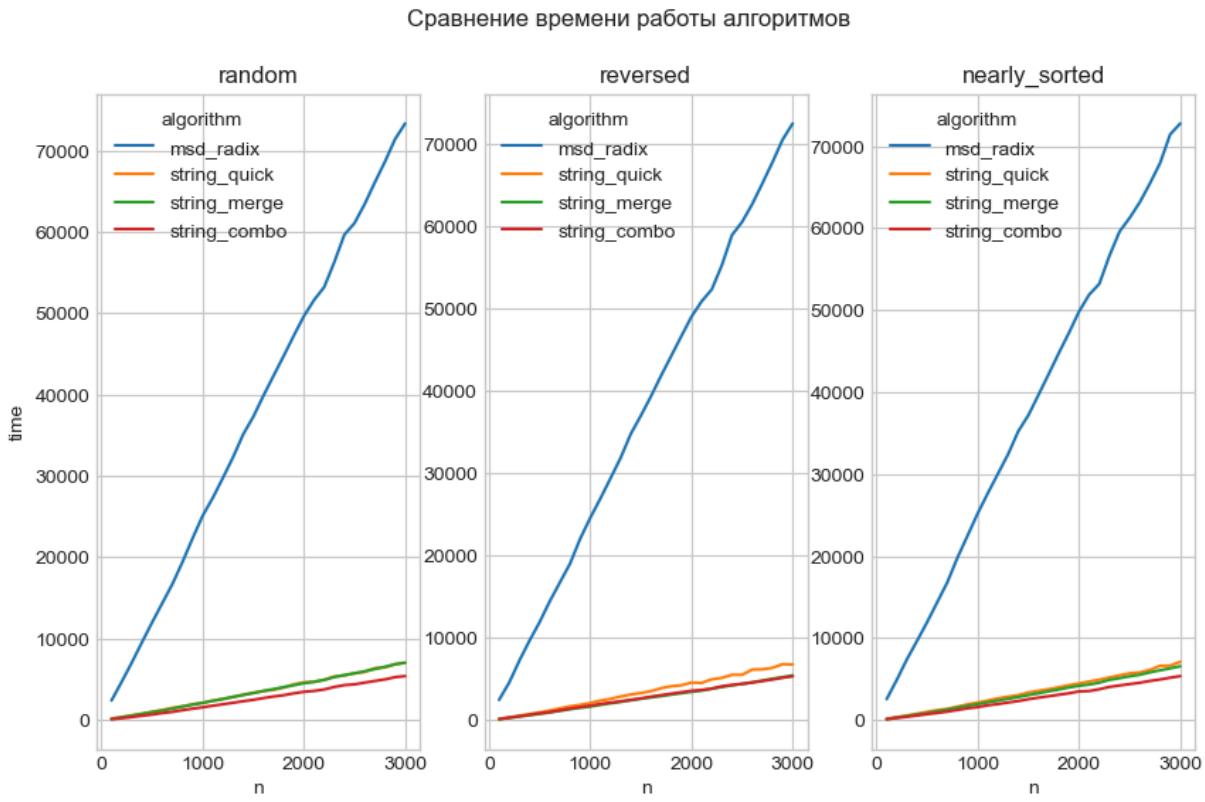
Построим график для времени работы в зависимости от количества элементов для разных типов массивов.

In [55]:
```
fig, ax = plt.subplots(ncols=2, nrows=2, figsize=(10, 10))

for i, algorithm in enumerate(["msd_radix", "string_quick", "string_merge", "string_combo"]):
    cur_ax = ax[i // 2, i % 2]
    sns.lineplot(
        data=df[df["algorithm"] == algorithm],
        x="n",
        y="time",
        hue="array_type",
        ax=cur_ax,
    )
    cur_ax.set_title(algorithm)

fig.suptitle("Сравнение времени работы алгоритмов в зависимости от типа массива");
```



Видим, что от изначального расположения строк в массиве мало что зависит. Лишь в `StringMergeSort` видна разница: сортировка полностью отсортированного в обратном порядке массива занимает меньше времени. Конечно, ведь на этапе `Merge` требуется сначала разложить элементы одного подмассива, затем другого, то есть никакого `Merge` по сути и нет.

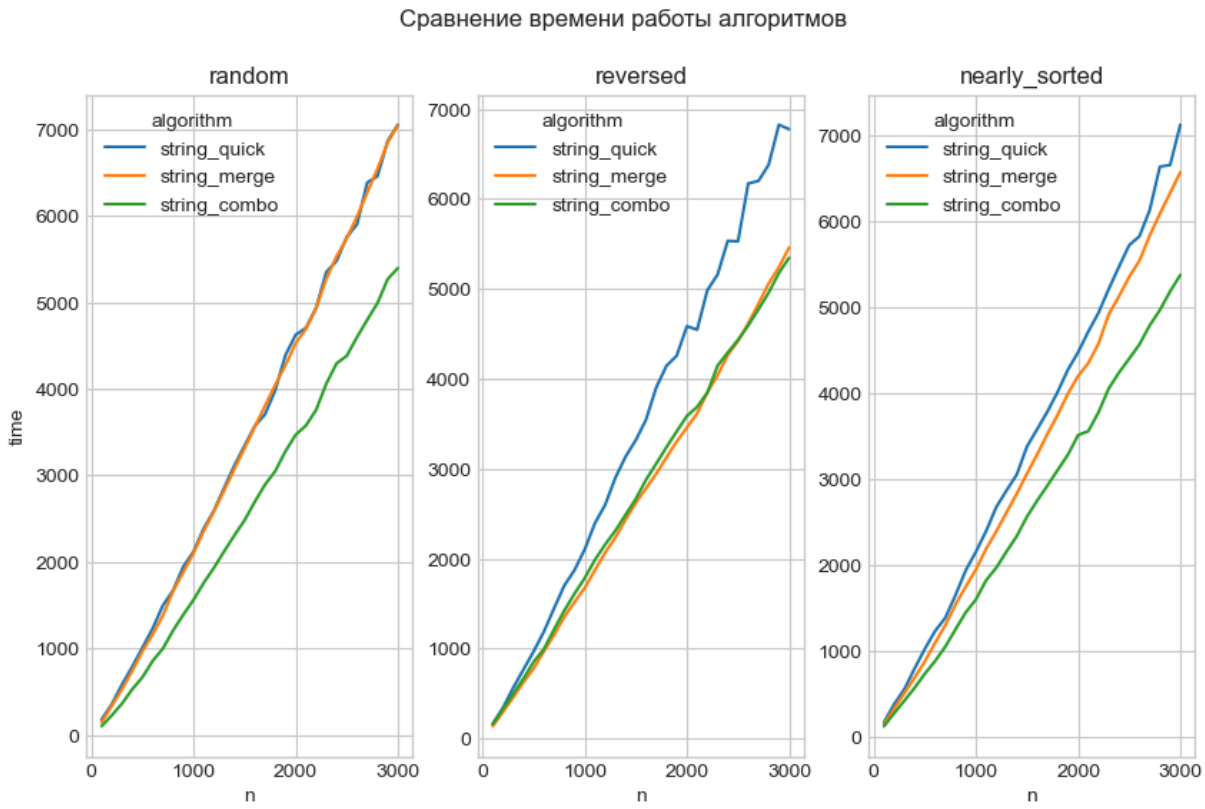
In [56]:
```
fig, ax = plt.subplots(ncols=3, figsize=(10, 6))

for i, array_type in enumerate(["random", "reversed", "nearly_sorted"]):
    sns.lineplot(
        data=df[df["algorithm"].isin(["msd_radix", "string_quick", "string_merge", "string_comb
        x="n",
        y="time",
        hue="algorithm",
        ax=ax[i],
    )
    ax[i].set_title(array_type)
    if i > 0:
        ax[i].set_ylabel("")

fig.suptitle("Сравнение времени работы алгоритмов");
```



`MsdRadixSort` сильно проигрывает остальным алгоритмам, при этом `StringComboSort` (`MsdRadixSort` вместе с `StringQuickSort` на малых фрагментах массива) держится на уровне с остальными алгоритмами.

Временно исключим проигравшего и рассмотрим остальные алгоритмы.

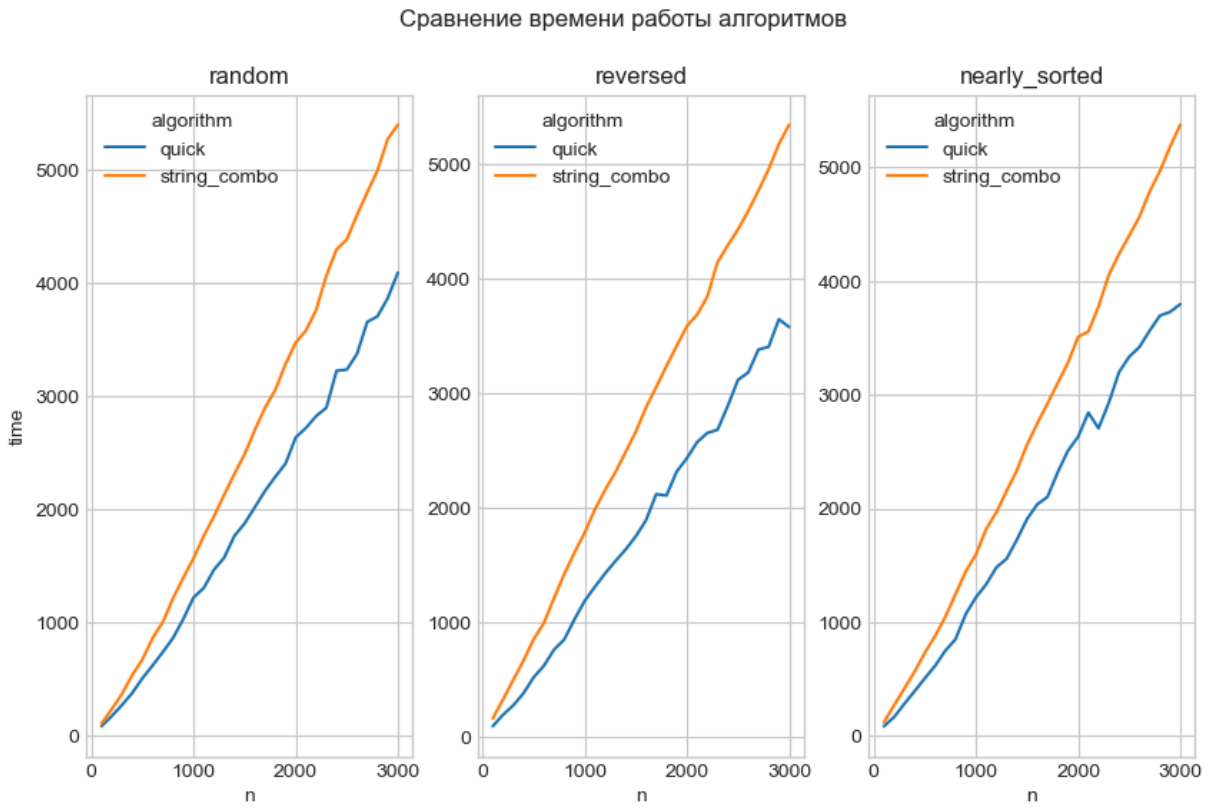
In [57]:
```
fig, ax = plt.subplots(ncols=3, figsize=(10, 6))

for i, array_type in enumerate(["random", "reversed", "nearly_sorted"]):
    sns.lineplot(
        data=df[df["algorithm"].isin(["string_quick", "string_merge", "string_combo"]) & (df["a
        x="n",
        y="time",
        hue="algorithm",
        ax=ax[i],
    )
    ax[i].set_title(array_type)
    if i > 0:
        ax[i].set_ylabel("")

fig.suptitle("Сравнение времени работы алгоритмов");
```



Комбинированная сортировка (`MsdRadixSort` + `StringQuickSort`) показывает лучший результат на всех типах массивов. Рейтинг остальных алгоритмов меняется в зависимости от типа массива.

В заключение, сравним двух победителей:

In [ ]:
```
fig, ax = plt.subplots(ncols=3, figsize=(10, 6))

for i, array_type in enumerate(["random", "reversed", "nearly_sorted"]):
    sns.lineplot(
        data=df[df["algorithm"].isin(["quick", "string_combo"]) & (df["array_type"] == array_ty
        x="n",
        y="time",
        hue="algorithm",
        ax=ax[i],
    )
    ax[i].set_title(array_type)
    if i > 0:
        ax[i].set_ylabel("")

fig.suptitle("Сравнение времени работы алгоритмов");
```



Замечаем, что обычный `QuickSort` работает быстрее, чем разработанная комбинированная сортировка. Скорее всего, это связано с тем, что строки массива слишком разные. Серьезные оптимизации конкретно строковых алгоритмов направлены на ускорение сортировки строк со схожими префиксами, в нашем же случае это не так. Получается, что весь `overhead` адаптированных алгоритмов вовсе не используется, а, между тем, "под капотом" этих алгоритмов сложные копирования/мувы строк, которые используют много вычислительных ресурсов.