CSE 312 Operating Systems Spring 2020
Final Exam
Project Report

Ahmed Semih Özmekik
171044039

July 8, 2020

Abstract

In this report, I will describe the design choices and their motives in the designed virtual memory (with page tables), alongside with how the data is modeled and the design patterns that I have used in it, with many references to my C++ implementation. In addition to the technical explanations I mentioned, constants such as the optimal page size or best page replacement algorithm that we need to find as a result of the project will only be explained by demonstrating the tests, yet not all of them will be revealed in the report, since because adding all test outputs (which are very long) will reduce the readability of the report. And that's why, I explained the details of these tests in the demonstration video, where you can find the link below.

Demonstration Link: Click Here

1 Page Table

We need a page table data structure where we need to decide the fields of the entries it will contain.The page table should encapsulate the physical memory in it and make a page-replacement with the polymorphic algorithm, and if necessary, replace the physical memory between its entries. Let's examine the interface of the page-table we designed.[1]

Listing 1: PageTable class

```
1
2  class PageTable
3  {
4  public:
5      friend class PageReplAlgorithm;
6      friend class NRU;
7      friend class FIFO;
8      friend class SC;
9      friend class LRU;
10     friend class WSClock;
11
12     PageTable(unsigned int, unsigned int, unsigned int);
13     ~PageTable();
```

[1]Private fields are not shared here, and I will hide the private fields of classes throughout the entire report, as I will not go into the exact description of every code in the implementation.

```
14
15      /* address oriented in page-table */
16      bool isPresent(unsigned int) const;
17      void setModified(unsigned int address);
18
19      void print() const;
20
21      unsigned int get(unsigned int) const;
22      void set(unsigned int, unsigned int);
23
24      class Entry
25      {
26          friend class PageTable;
27
28      public:
29          Entry();
30
31          void setReferenced(bool);
32          void setModified(bool);
33          void setPresent(bool);
34
35          bool isReferenced() const;
36          bool isModified() const;
37          bool isPresent() const;
38
39          unsigned int getFrameNumber() const;
40      }
41  };
```

In the next sections, we will see the algorithm classes given by the page table in friendship in detail, but now, for a short explanation, we are aware of this data structure from the beginning: different algorithms perform different page-replacement. And the most important event of the page-table data structure is the page-replacement algorithm. After all, for the page-table, when a page frame is requested, selecting one of the page frames present (which is the target of the page-replacement algorithm) is to provide that desired page-frame from the disk.

Since the Page-replacement algorithms account for 80% of this job, I have defined them separately in a different header so that I can work cleaner with these algorithms. Since these classes needed to know all the details of the page-table, this friendship had to be given. This is the reason for this design decision.

This page-table class was generally used to model the page table. Apart from that, all the logical workflows were handled through other classes. The task of this class is simply

being a model by making it possible to find basic things like page-frame-number etc. for the given virtual memory indexes.

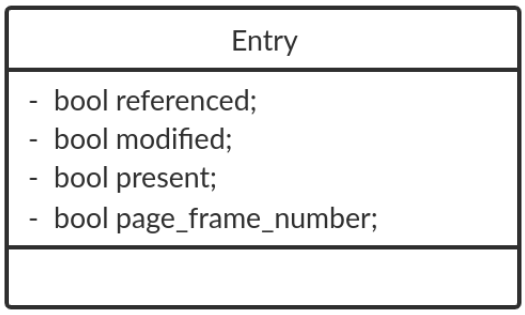

Figure 1: Page Table Entry

The caching and protection bits are not added to the fields of the entry, as they will not be required within the project. These fields are necessary and sufficient to satisfy the needs of all algorithms.

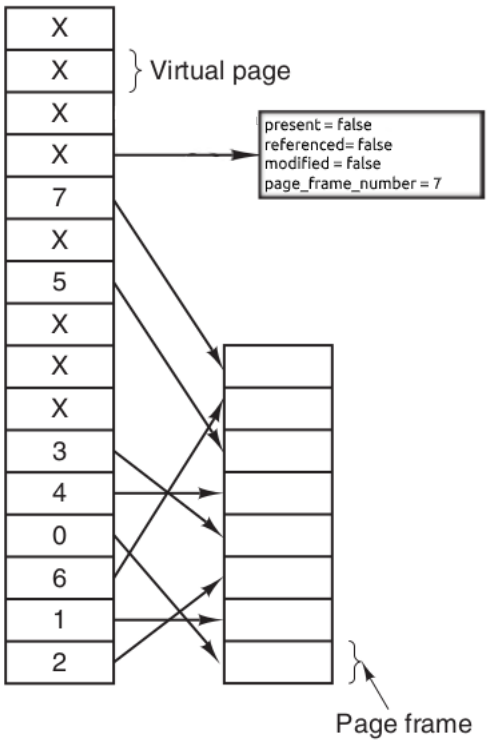

Figure 2: Page Table Entry in Page Table

2 Virtual Memory

As can be understood from the generality of the name, Virtual Memory class uses the page-table model mentioned above in logical forms, it creates a virtual-memory structure with all these data-structures. The hierarchy in implementation is similar to the MVC design pattern. Although it does not contain any view structure, it includes model and controller structures. While the **PageTable** class represents the model class, the two other classes we will see from now on represent the controller structure: **VirtualMemory** and **PageReplAlgorithm**.

2.1 Class: VirtualMemory

The only interface that we will use in our simulation is the interface of Virtual Memory class. This class is the concrete class of the virtual memory data structure we want.

Listing 2: VirtualMemory class

```
1  class VirtualMemory
2  {
3  public:
4      VirtualMemory(unsigned int, unsigned int, unsigned int,
5      std::string, std::string, unsigned int, std::string);
6      ~VirtualMemory();
7
8      void set(unsigned int index, int value, char *tName);
9      int get(unsigned int index, char *tName);
10     void fill(char *tName);
11     void setPartition(std::vector<char *>);
12     void resetPartition();
13     void stats() const;
14 };
```

This class is to encapsulate the working flow of the set and get functions[2] defined by their signature in the homework report. It achieves this by using the **PageTable** model class and the **PageReplAlgorithm** class. Although their headers are separated, we cannot think of these three classes independently.

We know how set and get functions work, these are the accessor and mutator of virtual memory. The functions we need to explain within our own design are the **setPartition** and **resetPartition** functions. Let's explain these.

[2]Defined as: "void set(unsigned int index, int value, char * tName)" and "int get(unsigned int index, char * tName)."

2.2 Allocation Policy Issues

It should not be forgotten that the homework has a global and local replacement selection. These functions emerged as a result of this difference. In the homework report, a design choice that was left to us, not explained under the local policy title, comes to the fore at this point:

> Local page replacement assumes some form of memory partitioning that determines how many pages are to be assigned to a given process or a group of processes. Most popular forms of partitioning are *fixed partitioning* and *balanced set algorithms* based on the working set model. The advantage of local page replacement is its scalability: each process can handle its page faults independently, leading to more consistent performance for that process. However global page replacement is more efficient on an overall system basis.[3]

After all, it was logical to choose the fixed-partitioning method, as we are designing a simulation and knowing how many different local sets we would run. That's why I chose this form.

In these functions, we define various local sets before starting to use set and get functions. We can see the use of this in the simulation code below. [4]

Listing 3: Simulation code

```
1  void PagingSimulation::simulate()
2  {
3      std::cout << "Starting simulation...\n";
4      std::chrono::steady_clock sc;
5      auto start = sc.now();
6
7      /* random filling */
8      std::cout << "Filling the array...\n";
9      memory_->setPartition({kFill});
10     memory_->fill(kFill);
11     memory_->resetPartition();
12
13     /* sorting quarters */
14     memory_->setPartition({kBubble, kQuick, kMerge, kIndex});
15
16     std::cout << "Sorting...\n";
17     sorter_threads_[0] = std::thread(&PagingSimulation::bubbleSort, this);
18     sorter_threads_[1] = std::thread(&PagingSimulation::quickSort, this);
```

[3] Bell, John. "Operating Systems Course Notes: Virtual Memory". University of Illinois at Chicago College of Engineering. Archived from the original on 23 September 2018. Retrieved 21 July 2017.

[4] We will discuss the simulation driver class further, but to explain the use of those functions I am giving a small peek for the sake of this section.

```
19      sorter_threads_[2] = std::thread(&PagingSimulation::mergeSort, this);
20      sorter_threads_[3] = std::thread(&PagingSimulation::indexSort, this);
21
22      /* wait for all quarters to finish */
23      for (size_t i = 0; i < THREAD_NUM; i++)
24          sorter_threads_[i].join();
25      memory_->resetPartition();
26
27      /* scan the array */
28      std::cout << "Scanning the array...\n";
29      memory_->setPartition({kCheck});
30      if (check())
31          std::cout << "Array is sorted!\n";
32      else
33          throw std::logic_error("Array is not sorted!\n");
34
35      /* print page table stats */
36      memory_->stats();
37
38      std::cout << "Simulation finished!\nElapsed time:\t";
39      auto end = sc.now();
40      auto time_span = static_cast<std::chrono::duration<double>>(end - start);
41      std::cout << time_span.count() << " secs" << std::endl;
42  }
```

Since we use string keys to separate local sets, it takes a string vector in **setPartition** to introduce local sets that will work together at once. For example, before making virtual memory fill, we state that the local set that will work in all virtual memory is "fill". An dit does partitioning, according to this information. (how many local sets are there)

After the fill process, for the 4 different sort threads to work, before it is started, it is reset to break the fixed partition remaining from the "fill" and then set again by giving 4 different string keys. As a result, virtual memory will give split the memory equal to 4. And each sorting thread will work on its own local set.

I think the method used here is an efficient method in the context of the project. Because; although the local set is defined, there will be a single local set used in virtual memory for the "fill" operation. This means there is no other process running in virtual memory. Since only "fill" is defined when creating the local set of "fill", all virtual memory is allocated to it. So even if local policy is preferred, since it is running all alone, it uses all the memory as it's local set, therefore it works like global. This goes for "check" too. After all, even when local policy is preferred, it makes sense to allocate all memory to them in terms of efficiency, as these two processes will run all alone in the virtual memory.

The only disadvantage of this method is that it is designed according to the workflow

where the simulation will be performed. I assumed that it will always be divided equally, the number of local sets to run at the same time must always equally divide the virtual memory size. But this is an insignificant rule because it is clear how simulation will work, therefore it won't effect us.

2.3 Algorithms

Algorithms constitute the bulk part of the project. Before we elaborate on how we implement all algorithms, let's explain our general structure.

Listing 4: PageReplAlgorithm abstract class

```
1
2  class PageReplAlgorithm
3  {
4  public:
5      PageReplAlgorithm(PageTable *pageTable, int *memory,
6      std::fstream *disc, bool allocPolicy);
7      virtual ~PageReplAlgorithm();
8
9      void replace(unsigned int);
10     virtual void recordGet(unsigned int, std::string);
11     virtual void recordSet(unsigned int, std::string);
12     virtual void recordNew(unsigned int);
13
14     struct Stats /* keeps the count for each field */
15     {
16         unsigned int read;
17         unsigned int write;
18         unsigned int page_miss;
19         unsigned int page_repl;
20         unsigned int disc_read;
21         unsigned int disc_write;
22     };
23     void printStats() const;
24
25     struct LocalReplacementInfo
26     {
27     public:
28         unsigned int lower_bound_, upper_bound_, local_free_index_;
29     };
30
31     virtual void addWorkingSet(std::string, unsigned int, unsigned int);
```

```
32      virtual void delWorkingSets();
33      int findIndex(std::string);
34      void writeFrame(unsigned int, unsigned int);
35      void readFrame(unsigned int);
36
37  protected:
38      virtual unsigned int find() = 0;
39      PageTable *page_table_;
40      int *memory_;
41      std::fstream *disc_;
42      bool local_;
43      std::string current_thread_;
44      std::string current_stat_;
45      unsigned int global_free_index_;
46
47      /* pages are stored for local page replacement */
48      std::map<std::string, LocalReplacementInfo> *threads_working_set_;
49      std::map<std::string, Stats> stats_;
50
51      bool inWorkingSet(unsigned int) const;
52  };
```

During this class, we benefited from all the beauty of polymorphism. For example, the **VirtualMemory** class defines algorithm instance as follows:

Listing 5: Initializing the algorithm instance

```
1  void VirtualMemory::initAlgorithm(std::string algorithmName)
2  {
3      if (kNRU == algorithmName)
4          algorithm_ = new NRU(page_table_, memory_, &disc_, policy_local_);
5      else if (kFIFO == algorithmName)
6          algorithm_ = new FIFO(page_table_, memory_, &disc_, policy_local_);
7      else if (kSC == algorithmName)
8          algorithm_ = new SC(page_table_, memory_, &disc_, policy_local_);
9      else if (kLRU == algorithmName)
10         algorithm_ = new LRU(page_table_, memory_, &disc_, policy_local_);
11     else if (kWSCLOCK == algorithmName)
12         algorithm_ = new WSClock(page_table_, memory_, &disc_, policy_local_);
13     else
14         throw std::logic_error("no such algorithm!");
15 }
```

Let's explain the important functions of this abstract class.

Every function that starts with "record" is defined as virtual. Because different algorithms keep track of different records and take action according to these different records. **recordGet** is a function defined for an algorithm to record what it needs to record when get function is called on the virtual memory.

For example, base class defined the **recordSet** function as:

Listing 6: C++ code using listings

```
1 void PageReplAlgorithm::recordGet(unsigned int index, std::string tName)
2 {
3     current_stat_ = current_thread_ = tName;
4     auto &entry = page_table_->getEntry(index);
5     entry.setReferenced(true);
6     stats_[current_stat_].read++;
7 }
```

When the get is done, the statistical data is updated as well as the referenced bit of the entry that is made is returned. On the other hand, the NRU derived class keeps its records as follows:

Listing 7: C++ code using listings

```
1 void NRU::recordGet(unsigned int index, std::string tName)
2 {
3     PageReplAlgorithm::recordGet(index, tName);
4     handleTimer();
5 }
```

Function **handleTimer()** refreshes the reference bits in every clock period, as it supposed to do in NRU algorithm.

They use the replace function of the base class of all derived classes. But the function defined as pure virtual is the **find** function.

Base abstract class finds an index and replaces that page, but does not know which index to change, so it's an abstract class. Other concrete algorithm classes need to write this **find** function with their different ways to find the index of the modular page. Let's briefly explain we implement them.

2.3.1 NRU

Listing 8: NRU class

```
1 class NRU : public PageReplAlgorithm
2 {
3 public:
4     NRU(PageTable *pageTable, int *memory, std::fstream *disc, bool allocPolicy
5
```

```
6       void recordGet(unsigned int, std::string);
7       void recordSet(unsigned int, std::string);
8
9   private:
10      unsigned int find();
11      void handleTimer();
12
13      unsigned int timer_;
14      static const unsigned int kClockPeriod;
15  };
```

The NRU algorithm was handled with a simple timer variable. A specific maximum clock period was determined. The timer variable is increased with each access operation, and when it reaches the period, the reference bits are cleared and refreshed. This is how we simulate the clock interrupt event in the real operating system.

2.3.2 FIFO

Listing 9: FIFO class

```
1   class FIFO : public PageReplAlgorithm
2   {
3   public:
4       FIFO(PageTable *pageTable, int *memory, std::fstream *disc, bool allocPolic
5
6       void recordNew(unsigned int);
7       void delWorkingSets();
8
9   protected:
10      virtual unsigned int find();
11      std::map<std::string, std::queue<unsigned int>> queues_;
12  };
```

The FIFO algorithm was implemented with a simple queue. Each time a new page is opened, this record is kept and queued. And the page to be replaced is the first page opened.

2.3.3 SC

Listing 10: SC class

```
1  class SC : public FIFO
2  {
3  public:
4      SC(PageTable *pageTable, int *memory, std::fstream *disc, bool allocPolicy
5
6  private:
7      unsigned int find();
8  };
```

Since the SC algorithm is a variant of the FIFO algorithm, it derives from it. Again, using the same queue structure at the bottom, when the page is popped from the queue, this time it gives another chance if it is referenced, and it is put back in order. If the reference bit is not set, it is being replaced.

2.3.4 LRU

Listing 11: LRU class

```
1  class LRU : public PageReplAlgorithm
2  {
3  public:
4      LRU(PageTable *pageTable, int *memory, std::fstream *disc, bool allocPolicy
5
6      void recordGet(unsigned int, std::string);
7      void recordSet(unsigned int, std::string);
8      void delWorkingSets();
9      void updateLists(unsigned int);
10
11 private:
12     unsigned int find();
13     std::map<std::string, std::vector<unsigned int>> lists_;
14 };
```

The linked-list structure in the LRU was implemented with **std::vector**. Here, too, the linked-list is updated on each access, and the page that is suitable for replacing is being the tail of the list, which is a page that is not accessed for a while.

2.3.5 WSClock

Listing 12: WSClock class

```
1  class WSClock : public PageReplAlgorithm
2  {
3  public:
4      WSClock(PageTable *pageTable, int *memory, std::fstream *disc, bool allocP
5
6      void recordGet(unsigned int, std::string);
7      void recordSet(unsigned int, std::string);
8      void delWorkingSets();
9      void updateLists(unsigned int);
10
11     struct WSClockEntry
12     {
13     public:
14         bool operator==(unsigned int _index) { return index == _index; };
15         unsigned int index;
16         std::chrono::steady_clock::time_point last_use;
17     };
18
19 private:
20     unsigned int find();
21
22     /* to simulate circular linked list implementation */
23     std::map<std::string, std::vector<WSClockEntry>> lists_;
24
25     static const double kTau;
26     static std::chrono::steady_clock clock_;
27     bool isIdeal(WSClockEntry &);
28 };
```

In order for each local set to have a different linked list structure, a map structure was defined with string keys of local sets. Within this structure, there are vectors that represent circular linked-lists. The τ constant was pulled to a suitable setting with some tests. By looking at the time of last use value of the page in the current index in the circular linked-list, it compared with the τ value and replaced accordingly.

2.4 Functions: Get and Set

By creating these basic structures in the algorithm and page-table, these were used in the virtual-memory encapsulating class:

Listing 13: Get and Set functions

```
1
2  int VirtualMemory::get(unsigned int index, char *tName)
3  {
4      print();
5      if (!page_table_->isPresent(index))
6      {
7          int physical_index = algorithm_->findIndex(tName);
8          if (physical_index != -1)
9          {
10             page_table_->set(index, physical_index);
11             algorithm_->recordNew(index);
12             algorithm_->readFrame(index);
13         }
14         else /* page-table is full. replace */
15             algorithm_->replace(index);
16     }
17
18     unsigned int address = page_table_->get(index);
19     algorithm_->recordGet(index, tName);
20     return memory_[address];
21 }
22
23 void VirtualMemory::set(unsigned int index, int value, char *tName)
24 {
25     print();
26     if (!page_table_->isPresent(index))
27     {
28         int physical_index = algorithm_->findIndex(tName);
29         if (physical_index != -1) /* empty slot is found */
30         {
31             page_table_->set(index, physical_index);
32             algorithm_->recordNew(index);
33         }
34         else /* page-table is full. replace */
35             algorithm_->replace(index);
36     }
```

```
37
38      unsigned int address = page_table_->get(index);
39      assert(address < physical_size_);
40      algorithm_->recordSet(index, tName);
41      memory_[address] = value;
42  }
```

3 Tests, Results and Graphs

Now all we have to do is call the simulation with the parameters we want. The expected outcomes in part 3 of the homework report will be included here. The implementations we will show here are presented in the demo video with detailed explanations.

3.1 Optimal Page Size (Part 3, First Program)

To find the optimal page frame size, as expected, we change the frame size in the loop and select the frame size of the lowest page replacement count as the most successful page frame size:

Listing 14: Optimal page size finding code

```
1  void PagingSimulation::findOptimalSize()
2  {
3      if (memory_ != nullptr)
4          delete memory_;
5      std::vector<Stats> sorters[THREAD_NUM];
6
7      startSorters(sorters, 12, "LRU");
8
9      for (size_t i = 0; i < THREAD_NUM; i++)
10     {
11         auto &sorter = sorters[i];
12         auto it = std::min_element(sorter.begin(), sorter.end(),
13          [](const Stats &s1, const Stats &s2)
14          { return s1.page_repl < s2.page_repl; });
15         unsigned int optimal_page_size =
16         std::pow(2, std::distance(sorter.begin(), it) + 2);
17         std::cout << "Optimal page size for " << QUARTER_NAMES[i]
18         << " is: " << optimal_page_size << std::endl;
19     }
20 }
```

Output result for physical memory holding 256 integers, virtual memory holding 4096 integers:

```
Optimal page size for bubble is: 64
Optimal page size for quick is: 64
Optimal page size for merge is: 64
Optimal page size for index is: 32
```

3.2 Optimal Algorithm (Part 3, Bonus 1)

In the same way, we decide to have the best algorithm in the loop. For all algorithms, the average of page replacement values was averaged in all statistics, and the lowest mean page replacement algorithm was selected.

Listing 15: Optimal algoritmh finding code

```
void PagingSimulation::findOptimalAlgorithm()
{
    if (memory_ != nullptr)
        delete memory_;

    std::vector<unsigned int> means[THREAD_NUM];
    for (auto &algorithm : ALGORITHM_NAMES)
    {

        std::vector<Stats> sorters[THREAD_NUM];

        startSorters(sorters, 8, algorithm);

        for (size_t i = 0; i < THREAD_NUM; i++)
        {
            auto &sorter = sorters[i];
            unsigned int mean = 0;

            std::for_each(sorter.begin(), sorter.end(),
            [&mean](const Stats &s) { mean += s.page_repl; });
            means[i].push_back(mean / sorter.size());
        }
    }

    for (size_t i = 0; i < THREAD_NUM; i++)
    {
        auto &mean = means[i];
        auto it = std::min_element(mean.begin(), mean.end());
```

```
29            std::string optimal_algorithm =
30            ALGORITHM_NAMES[std::distance(mean.begin(), it)];
31            std::cout << "Optimal algorithm " << QUARTER_NAMES[i]
32            << " is: " << optimal_algorithm << std::endl;
33        }
34    }
```

Output result for physical memory holding 256 integers, virtual memory holding 4096 integers:

```
1  Optimal algorithm bubble is: NRU
2  Optimal algorithm quick is: FIFO
3  Optimal algorithm merge is: LRU
4  Optimal algorithm index is: LRU
```

3.3 Optimal Algorithm (Part 3, Bonus 2)

In this section, the output of the code below was directed to the file to find the working sets in the LRU.

Listing 16: Dataset producing code for to draw graphs

```
1  void PagingSimulation::workingSetData()
2  {
3      if (memory_ != nullptr)
4          delete memory_;
5
6      unsigned int frame_size = 4;
7      unsigned int physical_num = std::pow(2, 8);
8      unsigned int virtual_num = std::pow(2, 10);
9
10     memory_ = new VirtualMemory(frame_size, physical_num, virtual_num, "LRU", '
11     memory_size_ = virtual_num * frame_size;
12
13     memory_->setPartition({kFill});
14     memory_->fill(kFill);
15     memory_->resetPartition();
16
17     memory_->setPartition({kBubble, kQuick, kMerge, kIndex});
18     sorter_threads_[0] = std::thread(&PagingSimulation::bubbleSort, this);
19     sorter_threads_[1] = std::thread(&PagingSimulation::quickSort, this);
20     sorter_threads_[2] = std::thread(&PagingSimulation::mergeSort, this);
21     sorter_threads_[3] = std::thread(&PagingSimulation::indexSort, this);
22
```

```
23      /* wait for all quarters to finish */
24      for (size_t i = 0; i < THREAD_NUM; i++)
25          sorter_threads_[i].join();
26      memory_->resetPartition();
27
28          delete memory_;
29      memory_ = nullptr;
30  }
```

The important thing here is to give 0 to the input, print period value. You can collect this private data only in this way. The outputs are in the file as follows (simulated to .csv format, space-seperated):

```
1  bubble 1
2  bubble 1
3  bubble 2
4  quick 1
5  bubble 2
6  merge 1
7  merge 1
8  ...
```

We see the name of the thread that has the first column, working set, and the second column shows the working set's size. With the python script I wrote below, this big data set was read and the graphic output was produced. Again, all these procedures shown in the demo.

Listing 17: Python script to produced graphs from dataset

```
1  import pandas as pd
2  import matplotlib.pyplot as plt
3  from scipy.interpolate import make_interp_spline, BSpline
4  import numpy as np
5
6  df = pd.read_csv('test.txt', sep=" ", dtype={
7                   'ws_size': 'int64'}, header=None, error_bad_lines=False)
8  df.columns = ["sorter", "ws_size"]
9
10 df['ws_size'] = df['ws_size'].astype('int64')
11 print(df.head(3))
12 sorters = [0, 0, 0, 0]
13 sorters[0] = df[df['sorter'] == 'bubble'].reset_index(drop=True).head(5000)
14 sorters[1] = df[df['sorter'] == 'quick'].reset_index(drop=True).head(5000)
15 sorters[2] = df[df['sorter'] == 'merge'].reset_index(drop=True).head(5000)
16 sorters[3] = df[df['sorter'] == 'index'].reset_index(drop=True).head(5000)
```

```
17
18
19 fig, axs = plt.subplots(2, 2)
20 axs[0, 0].set_title('Bubble Sort')
21 axs[0, 1].set_title('Quick Sort')
22 axs[1, 0].set_title('Merge Sort')
23 axs[1, 1].set_title('Index Sort')
24
25 indices = [(0, 0), (0, 1), (1, 0), (1, 1)]
26 colors = ['tab:orange', 'tab:green', 'tab:red', 'tab:blue']
27
28 for i in range(0, 4):
29     sorter = sorters[i]
30     sorter['k'] = pd.Series(list(range(len(df))))
31     x = sorter['k'].to_numpy()
32     y = sorter['ws_size'].to_numpy()
33
34     xnew = np.linspace(x.min(), x.max(), 50)
35     spl = make_interp_spline(x, y, k=3)
36     smooth = spl(xnew)
37
38     axs[indices[i]].set_xticklabels([])
39     axs[indices[i]].set_yticklabels([])
40     axs[indices[i]].set_xlabel('k')
41     axs[indices[i]].set_ylabel('w(k,t)')
42     axs[indices[i]].plot(xnew, smooth, colors[i])
43 plt.show()
```

Graphs are in the following pages.

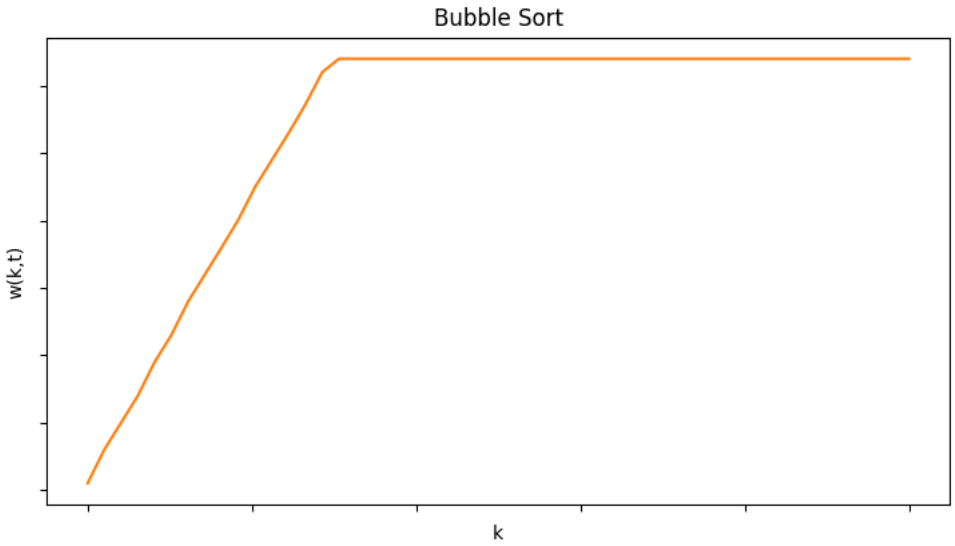

Figure 3: Working set graph bubble sort

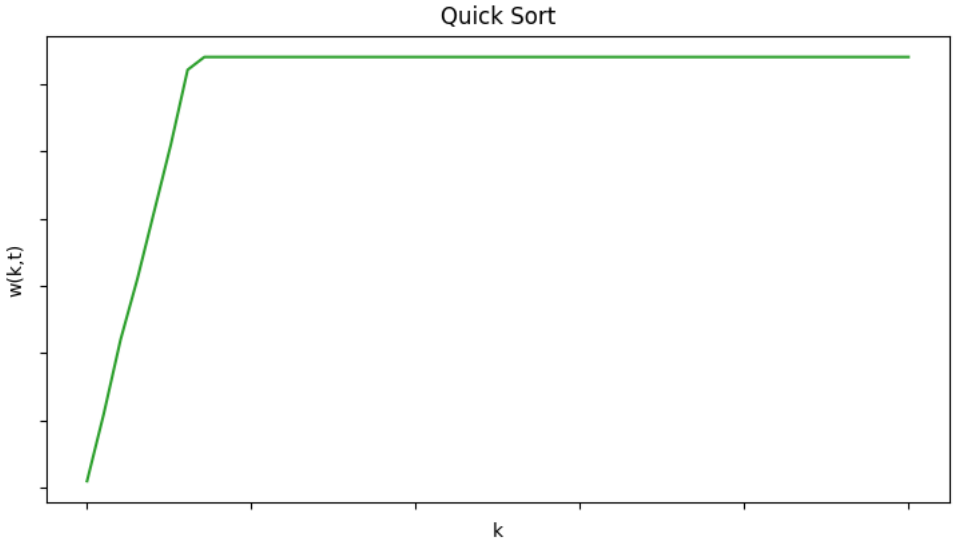

Figure 4: Working set graph quick sort

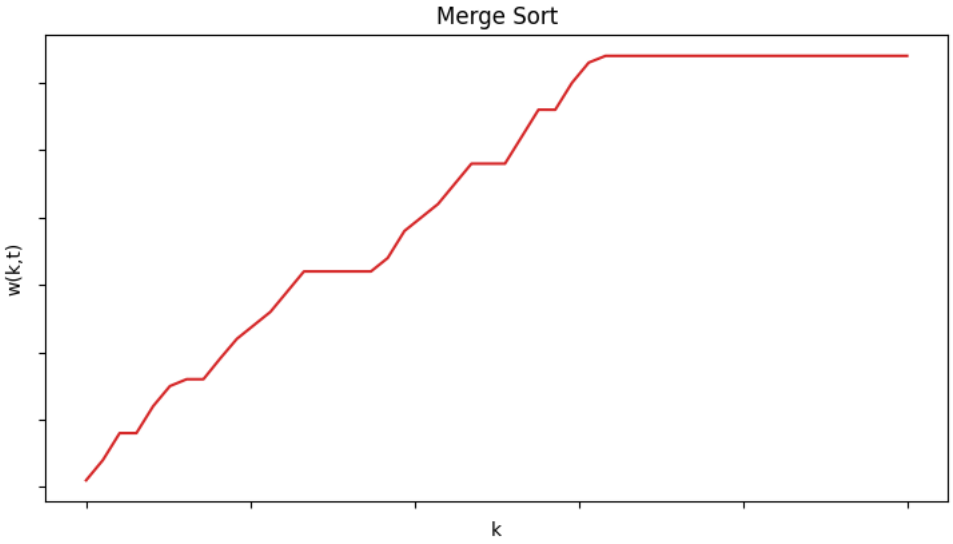

Figure 5: Working set graph merge sort

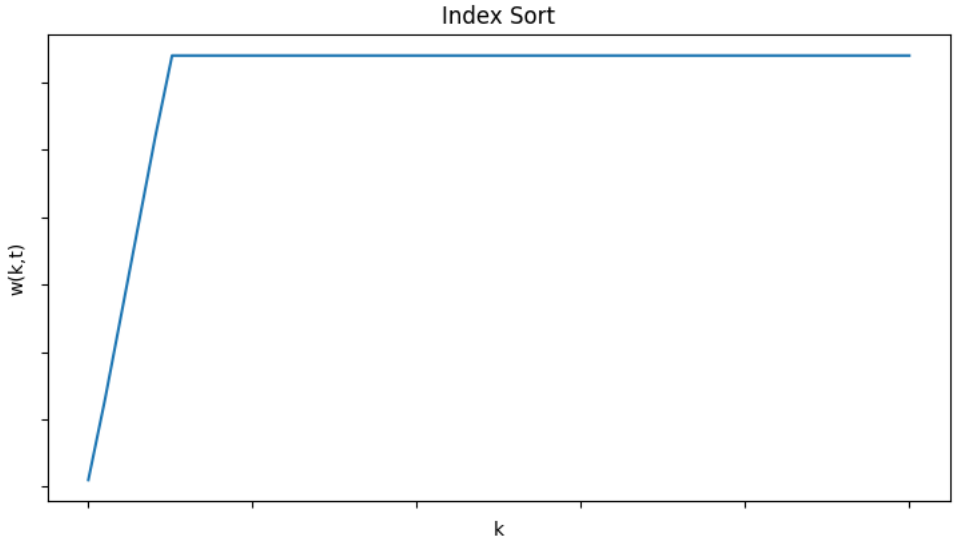

Figure 6: Working set graph for index sort