Assignment 4 – Sets and Sorting

Johann Lee

CSE 13S – Fall 2023

Purpose

The purpose of this program is to apply two different functions. The first of the functions is known as a set of Set functions. These will be used in conjunction with getopt() to record command-line options. These command-lines will then lead into the second set of functions which are implementing Sorting algorithms. These sorting algorithms will order arrays of integers which are fed to the program based on srandom() seeds. After these operations are finished based on the command-line the output will change. This will be discussed further in "How to Use the Program", however, simply stated the output will state which sort was used, the number of elements, the number of moves, the number of compares, and based on a command-line inputs it will print elements.

How to Use the Program

How to use the program will require to first make it using our Makefile. This is as simple as just inputting "make" into our console which will automatically generate an executable for us named "sorting". To execute this we are required to input "./sorting" and an assortment of flags. To figure out these flags we can use "./sorting" with no flags or "./sorting -H" which will print out a Synopsis, Usage, and options printout to help utilize the program. The input area is which sorting will be used which are represented by -a, -h, -b, -s, -q, and -i. These are labeled as using all sorts, using heap sort, using Batcher sort, using shell sort, using quick sort, and using insertion sort all concerning the order mentioned in the flags. The next set of inputs is -n which specifies the number of array elements (default being 100). Next is -p (defaulted at 100) which specifies the number of elements to print after sorting is finished. If this is set to 0 then no elements will be printed and only a summary of each sorting operation will be given. Finally, is -r (defaulted to 13371453) which specifies the seed that will be used to generate all of our random numbers to be sorted. All these flags are able to be used in order to specify which sorts are used on a specified number of array elements on a specified seed. This is then printed with a specified amount of number of elements in the form of a summary of each sorting method which mentions the the given numbers above as well as a block of elements that are printed out underneath using a block 5 columns.

Program Design

Functions

This section will explore the different functions that are utilized throughout the program but will not include any Sort methods. These will be explored in the Algorithms section of this report as it seems more appropriate. The following functions are included within the Sets category.

- Set set_empty(void)
  - This is the first of the Set functions and is used to return a set which is empty or in other words, all bits are set to 0.

```
set = 1111 1111
set = set & 0000 0000
return set
```

- Set set_insert(Set s, int x)
  - This is a function that inserts int x into given set s. In terms of our code it means that the set returned has the bit corresponding to x set to 1.

```
create set = 0 (initialize at 0)
set = 1 << x (shift 1 x times in set)
return s OR set (make sure that x position on set is
                 1 whether it was or wasn't already and
                 returns it)
```

- Set set_remove(Set s, int x)
  - This function is a contrast of the function before in which it subtracts int x into a given set s. This means that set s should be returned with the position x set to 0.

```
create set = 0 (initialize at 0)
set = not (1 << x) (makes set 1 and 0 at x)
return s AND set
```

- bool set_member(Set s, int x)
  - Returns a bool showing that value x is in set s.

```
create set = 0
set = 1 << x
if s AND set is true
    return true
else
    return false
```

- Set set_union(Set s, Set t)
  - This function returns a union of Set s and Set t. This means that if it returns a set that is ticked to 1 on any number that Set s or Set t is set to 1.

```
create set = 0
set = s OR t
return set
```

- Set set_intersect(Set s, Set t)
  - Returns an intersection of two sets meaning that that returned set is only ticked to 1 if both set s and set to contain it.

```
create set = 0
set = s AND t
return set
```

- Set set_difference(Set s, Set t)
  - Returns the difference between two sets meaning that in this case, it returns whatever elements are in Set s that are not in Set t.

```
create set = 0
set = s AND (NOT t)
return set
```

- Set set_complement(Set s)
  - Returns the complement of a given set. This means that it inverts the set so that it no longer contains any elements that were in the original set and only has elements that are in the universal set.

    ```
    create set = 0
    set = NOT s
    return set
    ```

The following section will include functions from the Statistics functions list. These functions all revolve around this struct:

```
provided by Darrel Long, asgn4.pdf page 11

typedef struct {
    uint64_t moves;
    uint64_t comparisons;
} Stats;
```

- int cmp(Stats *stats, int x, int y)
  - This function compares int x and int y and increments the comparisons field given in the struct. It returns -1 if x is less than y, 0 if x is equal to y, and 1 if x is greater than y.

    ```
    increment comparison in stats by 1

    if x > y
        return 1
    else if x == y
        return 0
    else
        return -1
    ```

- int move(Stats *stats, int x)
  - This function ”moves” x by increasing the moves field in stats and returning x.

    ```
    increment move in stats by 1

    return x
    ```

- void swap(Stats *stats, int *x, int *y)
  - This function swaps the elements pointed to by x and y and at the same time, it increments the move field in stats by 3 to reflect a swap using a temporary variable.

    ```
    increment move in stats by 3

    int temp = pointer x
    pointer x = pointer y
    pointer y = temp
    ```

- void reset(Stats *stats)
  - This rests stats, setting both moves and comparisons to 0. It should be used once a sort algorithm is done and a new one is going to be used.

    ```
    moves in stats = 0
    comparisons in stats = 0
    ```

- void print_stats(Stats *stats, const char *algorithm_name, int num_elements)
  - Prints the statistics for stats, listing the specific algorithm name and the number of elements. This will be used in generating output.

    ```
    print alg name, # elements
    ```

Algorithms

This section dive into the both the function and pseudo code of the sorting algorithms. These functions are given in Python which will be shown here and then a C pseudo-code implementation will be under each sort along with a description of how it works.

- Insertion Sort

  ```
  Provided by Darrell Long, asgn4.pdf page 4

  def insertion_sort(A: list):
      for k in range(1, len(A)):
          j = k
          temp = A[k]
          while j >= 1 and temp < A[j - 1]:
              A[j] = A[j - 1]
              j -= 1
          A[j] = temp
  ```

  Insertion Sort is a method in which elements are considered one at a time placing each one in its correct and ordered position. If we assume an array of size n with elements A[0] through A[n-1] for each k in 1 ¡= k ¡= n-1, Insertion sort compares k with each preceding element in descending order until its position is found.
  We can see this in Python where k will go through every value from 1 all the way to the position A[len(A) - 1] where len is the length of array A. We first put A[k] into a temporary variable and then begin comparing it top to down from its original position. This continues until it is placed in a position where its left neighbor is lower in value.

  ```
  def insertion_sort(Stats *stats, int *arr, int length)
      create int k, temp, j

      for k = 1, k < length, k++
          j = k
          temp = kth of array

          move++ (kth moved to temporary)

          while j >= 1 and temp < arr[j - 1]
              comparison++ (comparison between array)
              move++ (array moves elements)
              arr[j] = arr[j - 1]
  ```

```
                j--;

            arr[j] = temp
        }
```

- Shell Sort

```
Provided by Darrell Long, asgn4.pdf page 5

def shell_sort(A: list):
    for gap in gaps:
        for k in range(gap, len(A)):
            j = k
            temp = A[k]
            while j >= gap and temp < A[j - gap]:
                A[j] = A[j - gap]
                j -= gap
            A[j] = temp
```

Shell sort is a variation of insertion sort, however, it compares elements that are far apart from each other. This is the reason for the gap variable. As sorting continues the gap between items being compared decreases. Therefore, the shell starts with elements that are very far away.

```
def shell_sort(Stats *stats, int *A, int n)
    length = length of A
    int k, temp, k

    for int gap in gaps (given)
        for k = gap, k < length, k++
            j = k
            temp = A[k]
            while j >= gap and temp < A[j - gap]
                A[j] = A[j - gap]
                j -= gap
            A[j] = temp
```

- Quick sort

```
Provided by Darrell Long, asgn4.pdf page 8

def partition(A: list, lo: int, hi: int):
    i = lo - 1
    for j in range(lo, hi):
        if A[j] < A[hi]:
            i += 1

            A[i], A[j], = A[j], A[i]

    i += 1

    A[i], A[hi], = A[hi], A[i]

    return i
```

```
def quick_sorter(A: list, lo: int, hi: int):
    if lo < hi:
        p = partition(A, lo, hi)
        quick_sorter(A, lo, p - 1)
        quick_sorter(A, p + 1, hi)

    def quick_srt(A: list):
        quick_sorter(A, 0, len(A) - 1)
```

Quick Sort is the most commonly used algorithm for sorting. It is the fastest when using comparisons. It partitions arrays into two sub-arrays by selecting an element from the array and designating it as a pivot. Elements in the array that are less than the pivot go to the left sub-array and greater than or equal to the pivot go to the right sub-array. Quicksort is an in-place algorithm, meaning it does not allocate additional memory for its sub-arrays. Instead, it utilizes a subroutine called partition(). It applies itself recursively on the partitioned arrays, thereby sorting each array partition containing at least one element.

- Batcher's Odd-Even Merge Sort

```
def comparator(A: list, x: int, y: int):
      if A[x] > A[y]:
          # Swap A[x] and A[y]
          A[x], A[y] = A[y], A[x]

def batcher_sort(A: list):
      if len(A) == 0:
        return
      n = len(A)
      t = n.bit_length()
      p = 1 << (t - 1)

      while p > 0:
        q = 1 << (t - 1)
        r=0
        d=p

        while d > 0:
            for i in range(0, n - d):
                if (i & p) == r:
                    comparator(A, i, i + d)
            d=q-p
            q >>= 1
            r=p
        p >>= 1
```

Odd-even merge sort is like a sorting network. It is like a fixed number of wires, one for each input to sort, and is connected using comparators. Comparators compare the values traveling along the two wires they connect and swap the values if they're out of order. Since it is a sorting network it is limited to inputs that are powers of 2.

Results

References