Asgn3 Design Doc

Elliott Jensen

October 2021

1 Introduction

This program compares the efficiency of insert, heap, shell and quick sort. It includes a test file called "sorting.c" which allows the user to interact with the program using getopt, to specify the seed, array size and number of elements to display and allows the user to specify which sorting algorithms they want to run.

2 Test File

This file includes main and calls all of the functions of the program. It uses getopt to parse through the the command line and finds which sorting algorithms to run and allows the user to enter values for seed, array size and number of elements to display.

```
main(argc,argc):
    test_heap = false;
    test_insert = false;
    ...

    seed = 13371453
    size = 100
    elements = 100
    getopt
    switch (opt):
    case 'a':
        test_heap = true;
        test_insert = true;
        test_shell = true;
        test_quick = true;
        break
    ...
```

Then the array is generated

```
srandom(seed);
for i in range(size):
    array[i] = random()
```

After generating the array there are a series of conditionals that activate different tests depending on which cases were specified.

```
if test_insert:
    insertion_sort(stats, array, elements)
    print("Insertion Sort" elements "elements," moves "moves," compares, "compares"
for i in range(elements):
    print(array[i])
    if i % 5:
        print("\n")
... the same for each sorting algorithm
```

3 Insert Sort

This file includes the insert sorting algorithm. It compares items in the array that are next to each other and swaps them if they are not in the right order.

```
def  insertion_sort(A: list):
    for i in range(1, len(A)):
        j = i
        temp = A[i]
        while j > 0 and  temp < A[j - 1]:
            A[j] = A[j - 1]
            j  -= 18
        A[j] = temp
```

*Taken from the Asgn3 directions section 2

4 Shell Sort

This file includes the shell sorting algorithm. Rather than only comparing elements that are next to each other like in insert sort, this program dynamically changes the gap between its comparisons. It relies on a separate gap function in order to dynamically control how far apart the comparisons are.

```
 from  math  import  log
 def  gaps(n: int):
    for i in range(int(log(3 + 2 * n) / log(3)), 0,  -1):
        yield  (3**i - 1) // 2
def  shell_sort(A: list):
    for  gap in gaps(len(A)):
        for i in range(gap , len(A)):
```

```
            j = i
            temp = A[i]
            while j >= gap  and  temp < A[j - gap]:
                A[j] = A[j - gap]
                j -= gap
                A[j] = temp
```

*Taken from Asgn3 directions section 3

5 Heap Sort

This file includes the heap sorting algorithm. The heap sort algorithm is separated into 4 functions: heap sort which handles the overall sorting, build heap which organizes the data into a heap, fix heap, which then organizes the heap so that the biggest number are on the top and the smallest numbers are on the bottom, and max child, which returns the larger of two children.

```
def  build_heap(A: list , first: int , last: int):
for  father  in  range(last // 2, first  - 1,  -1):
     fix_heap(A, father , last)
def  heap_sort(A: list):
first = 1
    last = len(A)
    build_heap(A, first , last)
    for  leaf in  range(last , first ,  -1):
     A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
        fix_heap(A, first , leaf - 1)
 def  max_child(A: list , first: int , last: int):
  left = 2 * first
    right = left + 14
    if right  <= last  and A[right - 1] > A[left - 1]:
     return  right
return  left
def  fix_heap(A: list , first: int , last: int):
found = False
    mother = first
    great = max_child(A, mother , last)
    while  mother  <= last // 2 and  not  found:
     if A[mother  - 1] < A[great  - 1]:
         A[mother  - 1], A[great - 1] = A[great - 1], A[mother  - 1]
         mother = great
         great = max_child(A, mother , last)
        else:
         found = True
```

*code taken from Asgn3 directions, section 4

6 Quick Sort

This file includes the quick sort algorithm. This sorting algorithm relies on the partition function in order to continuously cut the array in half in order to make its comparisons.

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

*code taken from Asgn3 directions section 5

7 Stats

This file includes the functions that allow the program to compare the number of moves and comparisons that each sorting algorithm is makeing. In all of the above code every movement of variables and comparison will rely on the code below

```
class stats():
stats.moves = 0
    stats.comparisons = 0
def cmp(Stats, x, y):
if x < y:
     return -1
    elif x == y:
     return 0
    else:
     return 1
def move(Stats, x):
stats.moves += 1
    return x
def swap(Stats, x, y):
swap_variable = x
```

```
    x = y
    y = swap_variable
def reset(Stats):
stats.moves = 0
    stats.comparisons = 0
```