FIT1045 Intro to Algorithms and Programming – Workshop 11

Objectives

The **objectives of this workshop** are:

- To implement a heap using a list.
- To investigate the behaviour of algorithms with respect to running time.

Useful Links:

For this workshop, you may find it useful to review some of the following concepts:

- Recursive Sorting (Lecture 15)
- Transform and Conquer (Lecture 18)
- Complexity (Lecture 19)

Task 0 (to be completed before class):

At this point in the unit you have been shown how to implement two iterative sorting algorithms:

- Insertion Sort (Lecture 7: Invariance in Problems)
- Selection Sort (Lecture 8: Greedy Approach)

And two recursive sorting algorithms:

- Merge Sort (Lecture 15: Recursive Sorting)
- Quicksort (Lecture 15: Recursive Sorting)

Create a Python script containing at least one iterative sorting algorithm and one recursive sorting algorithm. For example, it may include an `insertionSort` function and a `quickSort` function. You can obtain the code from the lectures, as long as you cite the original source. If you want a challenge, try re-implementing them yourself.

Have your Python script test the sorting functions by comparing the list they output with the sorted list produced by Python's built-in `sort` function. Your script should use at least 5 different input lists to test each sorting algorithm. It should output how many of the tests passed.

For example:

```
Insertion Sort: 5/5 tests passed.
Quicksort: 5/5 tests passed.
```

Note: Don't forget to make a new copy of each list before passing it to a sorting function. The sorting algorithms are designed to modify their input.

Note: You should be using a loop to do the tests. Don't copy your code for each new test.

Task 1:

Some information on heaps:

An "Introduction to the design and analysis of algorithms" by A. Levitin, 3rd Edition, Section 6.4 *Heaps and Heapsort*, describes how a list can be used to implement a heap.

The item at the top of the heap, the root node, is stored at position 0 in the list. The children (if there are any) of the item at position i in the list are at position $2 \times i + 1$ (the left child) and $2 \times i + 2$ (the right child).

For example, the (max-)heap in Figure 1 can be represented as the list $[8,4,7,2,3,1]$. Here the item at position 0 has the value 8, and has children at positions 1 and 2 with values 4 and 7 respectively.

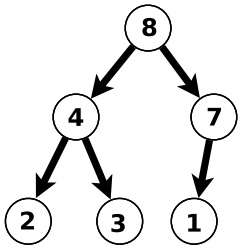

Figure 1: Heap

Part A:

Write a Python function `findChildren` that takes as input a list *heapList*, representing a heap, and a positive integer *position*. Your function should return **False** if the item at position *position* in *heapList* has no children, or if *position* is not a valid position in the heap, otherwise your function should print the item at *position* and its children, and then return **True**.

For example: If $heapList = [8,4,7,2,3,1]$ and position = 0, then the function returns **True** and prints:

```
Parent: 8
Left Child: 4
Right Child: 7
```

For example: If $heapList = [8,4,7,2,3,1]$ and position = 2, then the function returns **True** and prints:

```
Parent: 7
Left Child: 1
```

For example: If $heapList = [8,4,7,2,3,1]$ and position = 3, then the function returns **False**.

Task 2:

Useful material: In this task you will investigate the time taken by two different algorithms for computing x^n.

In Lecture 13, you were given the following Python code for computing x^n iteratively:

```
def power1(x, n):
    """computes x to the power of n"""

    value = 1
    for k in range(n):
        value *= x
    return value
```

and in Lecture 14, you were given the following Python code for computing x^n recursively:

```
def power2(x, n):
    """computes x to the power of n"""

    value = 1
```

```
    if n > 0:
        value = power2(x, n//2)
    if n % 2 == 0:
        value = value*value
    else:
        value = value*value*x
    return value
```

The following Python code randomly generates numbers x and n and then prints the time taken to compute x^n using each of the above functions.

```
import random
import timeit

# generate random integers for x and n
x = random.randrange(1, 100)
n = random.randrange(1, 100)

# start timing power1
start = timeit.default_timer()
power1(x,n)

# finish timing power1
end = timeit.default_timer()

print("Time taken by power1 was: {} seconds.".format(end-start))

# start timing power2
start = timeit.default_timer()
power2(x,n)

# finish timing power2
end = timeit.default_timer()

print("Time taken by power2 was: {} seconds.".format(end-start))
```

Part A:

Download the file "TaskA.py" containing this Python code from Moodle and run it. The output on one computer was:

```
Time taken by power1 was: 3.994200051238295e-05 seconds.
Time taken by power2 was: 8.54900099511724e-06 seconds.
```

but the timings may be different on your computer.

Part B:

Modify the program in Part A, so that it generates 100 random pairs of values for x and n and finds the time taken for each pair. Your program should output the number of times power1 was faster and the number of times power2 was faster.

As the values for x and n are randomly selected, different runs of the program may produce different results.

One example run produced:
Power1 faster: 37
Power2 faster: 63

Part C:

Modify your program from Part B, so that it outputs the worst and best cases in terms of running time for each of the power functions. Discuss the results with your peers.

For example: The output on one computer was:

```
Best for power1 was x=1, n=8 and time 2.201002644142136e-06 seconds.
Worst for power1 was x=99, n=28 and time 2.1292002202244475e-05 seconds.

Best for power2 was x=1, n=8 and time 3.058001311728731e-06 seconds.
Worst for power2 was x=56, n=28 and time 1.2235002941451967e-05 seconds.
```

Task 3:

For this task you will require your two sorting algorithms from Task 0.

Part A:

Write a function *timeSort* that creates a list of 10000 random numbers between 1 and 100, and prints how long it takes to sort the list using each of your sorting algorithms.

Part B:

Modify your program in Part B, so that it runs the function *timeSort* on 100 lists of 10000 numbers and, for each of your algorithms, outputs the list that takes the longest time to sort and the list that takes the least time.

Note: You will probably not get the same results for each run of 100 lists.

For example: For the `mergesort` algorithm, on one run your program might output (assuming you had a list of ten values instead) :

```
Shortest time taken was 2.4155000573955476e-05 seconds.
[44, 63, 39, 35, 73, 93, 98, 63, 16, 22]
Longest time taken was 8.260000322479755e-05 seconds.
[32, 30, 38, 16, 24, 39, 44, 28, 89, 61]
```

and on another run you might get:

```
Shortest time taken was 1.984200207516551e-05 seconds.
[83, 37, 48, 55, 19, 19, 71, 30, 58, 68]
Longest time taken was 8.31420038593933e-05 seconds.
[49, 74, 12, 41, 66, 34, 30, 24, 93, 55]
```

Another example: For the `quicksort` algorithm, on one run your program might output (assuming you had a list of ten values instead) :

```
Shortest time taken was 4.797999281436205e-06 seconds.
[99, 47, 23, 69, 23, 40, 66, 80, 37, 19]
Longest time taken was 7.48309976188466e-05 seconds.
[32, 38, 12, 58, 76, 45, 81, 67, 11, 94]
```

and on another run you might get:

```
Shortest time taken was 4.8009969759732485e-06 seconds.
[99, 90, 40, 57, 76, 26, 93, 19, 34, 69]
Longest time taken was 7.356700371019542e-05 seconds.
[11, 71, 26, 43, 97, 32, 81, 4, 55, 5]
```

For each of your sorting algorithms:

- What input do you think would produce the worst running time for your algorithm?
- What input do you think would produce the best running time for your algorithm?