CS 315 - Day 17, Priority Queue and Applications

Previously we studied the Heap data structure, the Max Heap property, and then worked through creating a Heap, fixing a Heap, and using a Heap to sort.

Today, we will use the code provided with the textbook to sort, and then we will explore applications using Heaps.

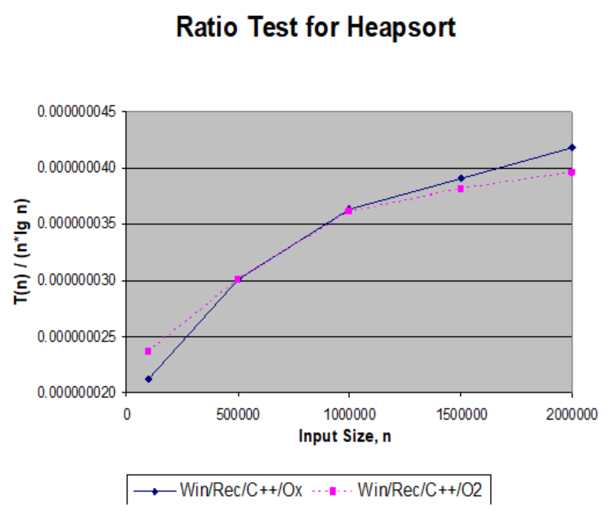
Run the Jupyter notebook `heapsortPoCkey.ipynb`, which includes the code implementing heapsort. Note that you will need to run all the execution groups to define the support functions before you can run the execution group with the sample sort.

Using the previous sorts as examples, update your code to time sorts over a range of N values, and confirm that the sort is in fact $O(n \cdot \lg n)$.

So, do the ratio's actually converge as N gets larger?

If not, what might explain that?

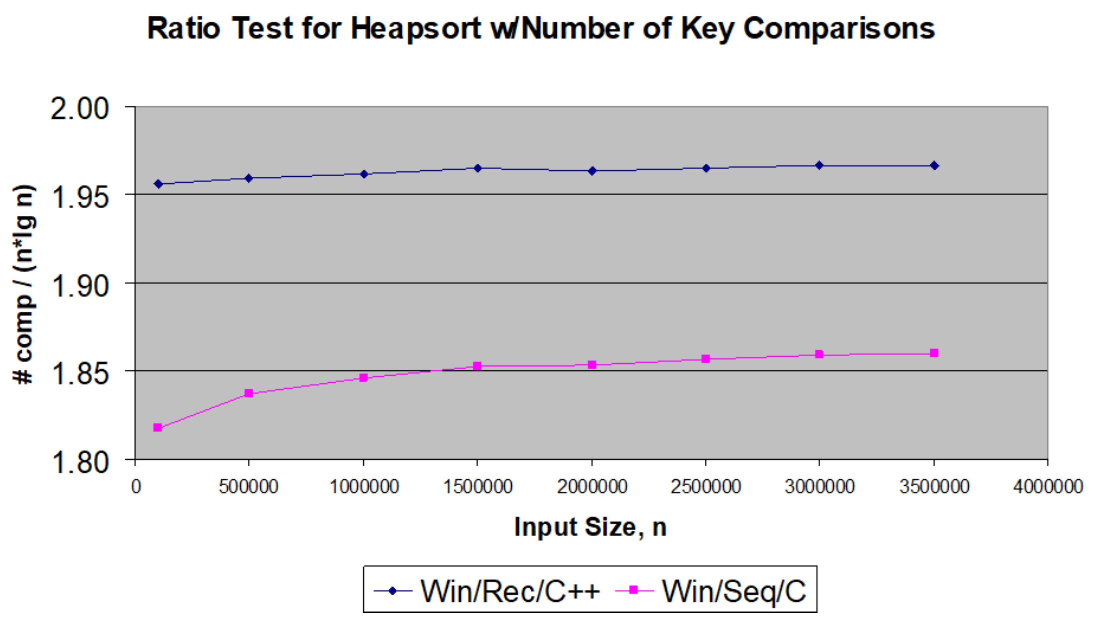
Someone you know has studied this as well: https://jcsites.juniata.edu/faculty/kruse/misc/LocalMAA2007.ppt



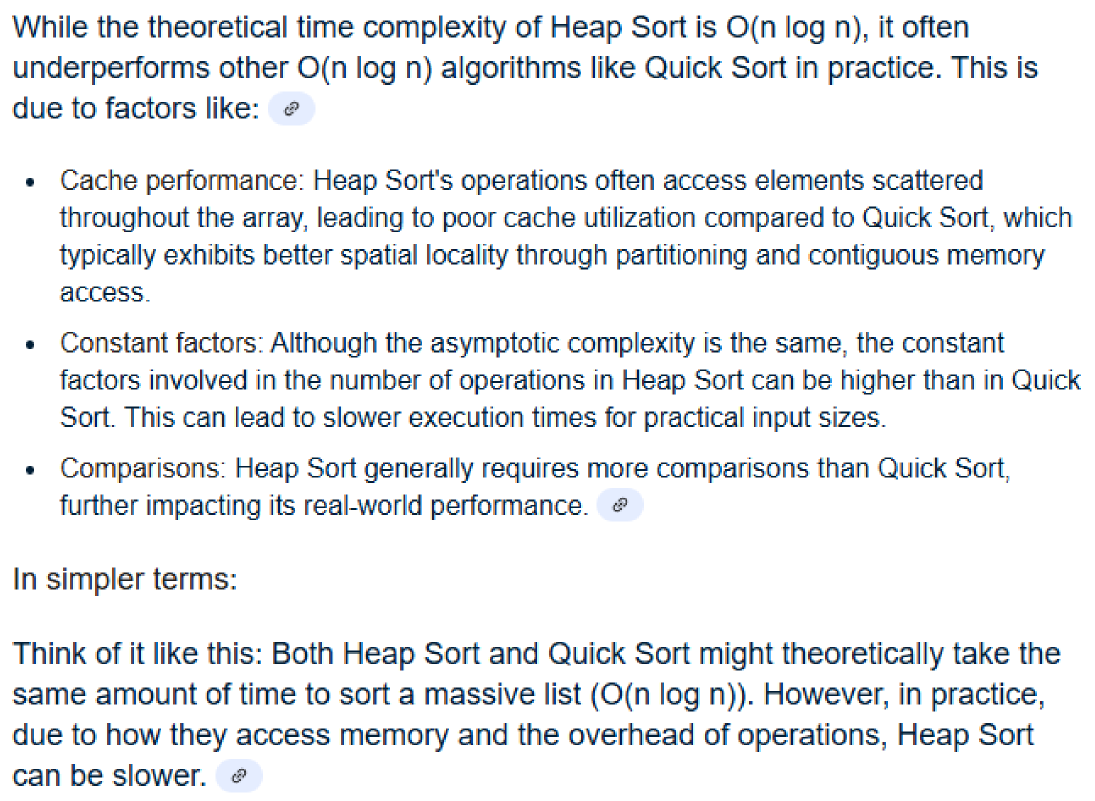
Your ratio test results probably mirror the results in the graph below. The ratios don't converge, they get larger as N gets larger.

But, the sort is working correctly, and the algorithm is implemented following the pseudocode, so there is a mystery why the actual timings are not $O(n \cdot \lg n)$.

But if we count how many times the most frequently executed line of code actually runs, which is the comparison (`if` statement) in Max-Heapify, we can determine if it executes on the order $O(n \cdot \lg n)$.



What does google have to say about this?



While the theoretical time complexity of Heap Sort is O(n log n), it often underperforms other O(n log n) algorithms like Quick Sort in practice. This is due to factors like:

- Cache performance: Heap Sort's operations often access elements scattered throughout the array, leading to poor cache utilization compared to Quick Sort, which typically exhibits better spatial locality through partitioning and contiguous memory access.
- Constant factors: Although the asymptotic complexity is the same, the constant factors involved in the number of operations in Heap Sort can be higher than in Quick Sort. This can lead to slower execution times for practical input sizes.
- Comparisons: Heap Sort generally requires more comparisons than Quick Sort, further impacting its real-world performance.

In simpler terms:

Think of it like this: Both Heap Sort and Quick Sort might theoretically take the same amount of time to sort a massive list (O(n log n)). However, in practice, due to how they access memory and the overhead of operations, Heap Sort can be slower.

https://www.google.com/search?q=why+is+heapsort+not+n+log+n+in+practice&rlz=1C1GCEU_enUS1150US1150&oq=why+is+heapsort+not+n+log+n+in+practice&gs_lcrp=EgZjaHJvbWUyCwgAEEUYChg5GKABMgkIARAhGAoYoAEyCQgCECEYChigATIHCAMQIRiPAjIHCAQQIRiPAtIBCTEyMzA0ajBqN6gCCLACAfEFqMr0G281VyY&sourceid=chrome&ie=UTF-8

We next consider a ***Priority Queue***. This data structure:

- Maintains a set S of elements, each associated with a key value.
- Easily tracks the maximum.
- Can remove the maximum using `Extract-Max`, which involves:
  - Removing the top of the heap.
  - Moving the last element in the heap to the top.
  - Calling `Max-Heapify` starting at the top.
- Other required functions for a Priority Queue are:
  - `Increase-Key`
  - `Heap-Insert`

There are many applications using Priority Queues, we will explore one that is inspired by the Constitution of the U.S., overseen by the U.S. Congress, and implemented by the U.S. Census Bureau.

But sadly, "***The Census Bureau uses a sorted array, stored in a single column of an Excel spreadsheet, which is recalculated from scratch at every iteration. You (should have) learned a more efficient implementation in your undergraduate data structures class.***" https://jeffe.cs.illinois.edu/teaching/algorithms/book/00-intro.pdf, pg 8-10

Guess what, that sounds like us…

Let’s start with a simple example to understand the process.

Consider three states, with populations:

State	***Population***
Maryland	6
Pennsylvania	9
Rhode Island	5

Six (6) representatives need to be allocated for these three states. ***Apportionment*** is finding the right proportion for each state’s number of representatives, based on the state’s population.

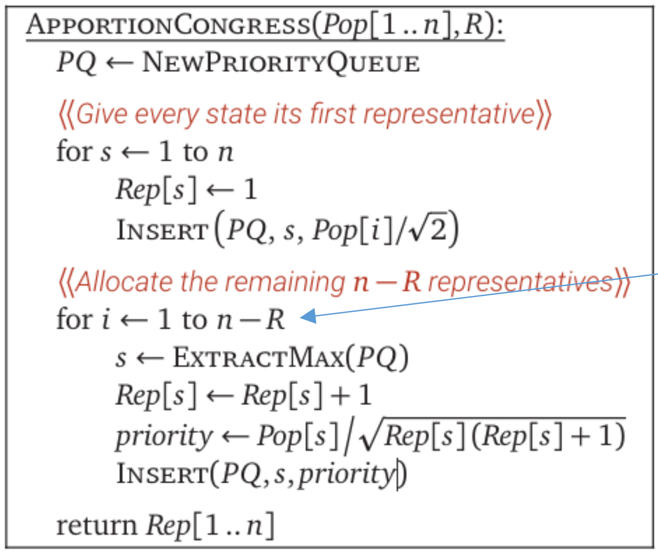
If there is a total population of 20, the first thought would be to divide the number of representatives by the total population, which would give $\frac{6}{20} = 0.3$ representatives per person.

State	***Population***	***Number of Reps***
Maryland	6	$6 \cdot 0.3 = 1.8$
Pennsylvania	9	$9 \cdot 0.3 = 2.7$
Rhode Island	5	$5 \cdot 0.3 = 1.5$

So… great… the total number of representatives is 6, but how are we getting 2.7 representatives from Pennsylvania?

There is a long and interesting history on how the US has done apportionment, so that each state has at least one representative and none are allocated a fractional number of representatives. The current method is the Huntingon-Hill method. And it lends itself very well to using a Priority Queue (although in practice it’s actually an Excel spreadsheet!).

It depends on the mathematical concept of a Geometric mean, https://www.investopedia.com/terms/g/geometricmean.asp . In our case, we calculate $\sqrt{number\_of\_reps * (number\_of\_reps + 1)}$, and use this result to decide whether to round the fractional number of representatives up or down.



```
ApportionCongress($Pop[1..n], R$):
    $PQ \leftarrow$ NewPriorityQueue
    ⟨⟨Give every state its first representative⟩⟩
    for $s \leftarrow 1$ to $n$
        $Rep[s] \leftarrow 1$
        Insert$(PQ, s, Pop[i]/\sqrt{2})$
    ⟨⟨Allocate the remaining $n - R$ representatives⟩⟩
    for $i \leftarrow 1$ to $n - R$
        $s \leftarrow$ ExtractMax$(PQ)$
        $Rep[s] \leftarrow Rep[s] + 1$
        $priority \leftarrow Pop[s] \big/ \sqrt{Rep[s](Rep[s]+1)}$
        Insert$(PQ, s, priority)$
    return $Rep[1..n]$
```

Figure 0.5. The Huntington-Hill apportionment algorithm

Note that it should be $R - n$, there are more representatives than states

State	***Population***	***Initial number of Reps***	***Geometric Mean***
Maryland	6	1	$\frac{6}{\sqrt{1 \cdot 2}} = 4.24$
Pennsylvania	9	1	$\frac{9}{\sqrt{1 \cdot 2}} = 6.36$
Rhode Island	5	1	$\frac{5}{\sqrt{1 \cdot 2}} = 3.54$

So initially each of the three states gets one representative, leaving the last three to be distributed.

The largest geometric mean is Pennsylvania's, 6.36, so Pennsylvania is awarded another representative, and its geometric mean is recalculated based on its new number of representatives:

State	***Population***	***Number of Reps***	***Geometric Mean***
Maryland	6	1	$\frac{6}{\sqrt{1 \cdot 2}} = 4.24$
Pennsylvania	9	2	$\frac{9}{\sqrt{2 \cdot 3}} = 3.67$
Rhode Island	5	1	$\frac{5}{\sqrt{1 \cdot 2}} = 3.54$

The largest geometric mean is now Maryland's, 4.24, so Maryland is awarded another representative, and its geometric mean is recalculated based on its new number of representatives:

State	***Population***	***Number of Reps***	***Geometric Mean***
Maryland	6	2	$\frac{6}{\sqrt{2 \cdot 3}} = 2.45$
Pennsylvania	9	2	$\frac{9}{\sqrt{2 \cdot 3}} = 3.67$
Rhode Island	5	1	$\frac{5}{\sqrt{1 \cdot 2}} = 3.54$

Finally we have one last representative to apportion, the largest geometric mean is Pennsylvania's, 3.67, so Pennsylvania is awarded the last representative (we recalculate the geometric means, although they aren't needed anymore:

State	***Population***	***Number of Reps***	***Geometric Mean***
Maryland	6	2	$\frac{6}{\sqrt{2 \cdot 3}} = 2.45$
Pennsylvania	9	3	$\frac{9}{\sqrt{3 \cdot 4}} = 2.60$
Rhode Island	5	1	$\frac{5}{\sqrt{1 \cdot 2}} = 3.54$

Run the `HeapHHApportionment-inclass.ipynb` code, and compare the output to the example above. In an assignment, you will be asked to calculate the number of representatives for each US state.

```python
import math
if __name__ == "__main__":
  import numpy as np

  # Enter the total number of Representatives
  R = 6

  # A list with the population of each state, given in the same order
  # (alphabetically is the most natural) as the list with the state names.
  statePop = [6,9,5]
  stateAbbrev = ["MD","PA", "RI"]

  # Set up the list containing the number of representatives for
  # each state, in the same state order as above.
  n = len(statePop)
  stateRep = [1]*n

  # Define the Max-Priority Queue.
  pq1 = MaxHeapPriorityQueue(KeyObject.get_key, KeyObject.set_key)

  # Fill the Max-Priority Queue with the state name as the element
  # and the associated geometric mean as the key value.
  # Initially, each state has 1 representative apportioned, so the
  # initial geometric mean will be the population/( sqrt(1+2) ).
  for i in range(n):
    pq1.insert(KeyObject( stateAbbrev[i],  statePop[i]/math.sqrt(2)  ))

  # The Huntington-Hill algorithm for apportionment.
  # Loop for as long as there are representatives left to apportion.
  for i in range(R-n):
    # Extract the state at the top of the priority queue.
    s =pq1.extract_max()

    # Convert the string of the state name to the location of the state in
    # the list with the number of representatives.
    stateLoc = stateAbbrev.index(s.string)

    # Increment this state's number of representatives.
    stateRep[ stateLoc ] += 1

    # Recalculate the new key value, which uses the geometric mean.
    priority =
statePop[stateLoc]/math.sqrt(stateRep[stateLoc]*(stateRep[stateLoc]+1))

    # Insert this state back into the Max-Priority Queue with its
    # updated key.
    pq1.insert( KeyObject( stateAbbrev[stateLoc] , priority) )

  # Once done with the loop, output the list of representatives.
  print(stateRep)
```