Data Structures Lab

Sorting Everything Out
Assignment 4 Recap

● Assignments mostly graded
  ○ Should be returned later today
  ○ Less in-depth than previous grades
  ○ Email me if you want more detailed feedback

● Varied levels of completion
  ○ Full assignment
  ○ Only heap
  ○ Only huffman (no standalone heap)

● Don't worry about grade
  ○ Pass / Fail course
  ○ 380 points to pass
Assignment 4 Recap

- Code does not need to belong to a class
  - Classes are an integral part of Java
  - Not true for most languages

- In C++ functions can (and often should) be standalone
  - Just make sure compiler knows they exist
  - Write required functions earlier
  - Declare required functions earlier
Assignment 5 - Speedy Sorting

- Due tomorrow night
  - March 16

- Implement two sorting algorithms
  - Quicksort
  - Any other sort

- Compare running times on various inputs
  - Write up findings
  - Fastest run, slowest run, average run...
  - Draw some graphs
Assignment 5 - Speedy Sorting

- Extra Credit!
  - Implement more sorts for more points
  - Each extra sort is worth another 25 points (+100 max)

- Each additional sort must be in a different language
  - One you did not know prior to this course
  - Java isn't allowed
Assignment 5 - Speedy Sorting

- Sorting algorithms come in all shapes and sizes
  - HeapSort - $O(n \log n)$
  - MergeSort - $O(n \log n)$
  - QuickSort - $O(n^2)$
  - StoogeSort - $O(n^{2.7})$
  - BogoSort - $O(\infty)$

- Worst case runtime can differ from general runtime
  - How likely is the worst case?
  - What other overhead is introduced?

- In theory, theory is the same as practice
  - But in practice it isn't
Quickstart Implemented

- Divide and Conquer algorithm
  - Select a pivot
  - Sort elements around that pivot
  - Sort each half of the list recursively

- But how do we implement it?
How do we partition our list?

One approach:
- Construct two new lists
- Copy smaller elements into one
- Copy larger elements into the other
- Copy back to original list

That's a lot of work...
- Can't we just partition in place?
Quicksort Implemented

- Let's use a series of swaps
- Start by getting the pivot out of the way
  - Swap to the end
- Iterate through list
  - Initialize partition between "small" and "large" elements
  - Every time we find a "small" element, swap it forward
- Swap the pivot back into place
  - Swap to partition
Quicksort Implemented
Quicksort Implemented

9 4 8 3 5 2 1 6 7
Quicksort Implemented
Quicksort Implemented

9 4 8 3 7 2 1 6 5
Quicksort Implemented
Quicksort Implemented
Quicksort Implemented
Quicksort Implemented
Quicksort Implemented
Quicksort Implemented
Quicksort Implemented
Quicksort Implemented
Quicksort Implemented

- We can partition our list in place

- Now we just recurse on the two halves!
  - Make sure to include start/end indices in recursive call

- When do we stop?
  - Could recurse all the way down to 1 element
  - But quicksort gets unwieldy on small inputs

- Quicksort runs fast, but it has overhead
  - For small lists, insertion sort runs faster
  - Run some experiments!
Quicksort Implemented

- What pivot should we choose
  - The first element?
  - The middle element?
  - A random element?

- Pivot selection doesn't affect our asymptotic runtime
  - But it might affect our actual runtime
Quicksort Demonstrated
Quicksort Analyzed

- Divide and Conquer algorithm
  - Select a pivot
  - Sort elements around that pivot
  - Sort each half of the list recursively

- This algorithm sounds familiar...
  - Divide elements in two
  - Recursively sort each half
  - Combine sorted halves
Quicksort Analyzed

- Quicksort is awfully similar to Mergesort
- And Mergesort has better asymptotic runtime
  - QuickSort  - $O(n^2)$
  - MergeSort  - $O(n \log n)$
- Yet quicksort is used far more commonly than mergesort
  - Why is that?
Quicksort Analyzed

- Number of operations
  - Mergesort needs to merge elements after sorting
  - Quicksort merges as it partitions

- In-place sort
  - Mergesort needs to copy data at each recursive call
  - Quicksort just needs to swap

- Theoretical analysis is important
  - But there's no substitute for practical application
  - Algorithms perform differently in different environments
Lesser Known Sorting Algorithms

- Heapsort
- Gnome Sort
- Slowsort
- And many others...
HeapSort

- Heaps are ideally suited to sorting elements
  - Why?

- Use basic ordering properties
  - Insert all elements into a heap
  - Repeatedly extract minimum

- How long does that take?

- Any downsides?
HeapSort (without the heap)

- We don't actually need a Heap to implement HeapSort
  - Arrays are implicit heaps...

- Heapify!
  - Repeatedly "insert" elements
  - Repeat until entire array is in heap order
HeapSort (without the heap)

- A Heapified array isn't a sorted array
  - How can we fix that?
- Repeatedly extractMin
  - Each call reduces the size of our "heap" by one
  - Store removed elements at the end of our array
  - After all elements are extracted, array will be sorted
HeapSort (without the heap)

- Really neat sorting algorithm

- Efficient
  - Heapifying takes $O(n)$
  - Extracting takes $O(n \log n)$
  - Overall, $O(n \log n)$

- Sorts in place
  - No storage overhead!
Gnome Sort

- Also known as Stupid Sort

- A mix between insertion sort and bubble sort
  - Slowly builds a sorted list
  - Performs a series of swaps

- The algorithm
  - Look at the first two elements
  - If they are in the right order, step forward
  - If they are in the wrong order, swap and step back
  - Stop when you reach the end of the list

- Elements are swapped backwards into place
Gnome Sort

1 8 5 3
Gnome Sort
Gnome Sort

1 8 5 3
Gnome Sort

1 5 8 3
Gnome Sort
Gnome Sort
Gnome Sort
Gnome Sort
Gnome Sort
Gnome Sort

1 3 5 8
Gnome Sort
Gnome Sort
Gnome Sort

1 3 5 8
Gnome Sort

1 3 5 8
Gnome Sort

- Not the most efficient sort
  - But faster than insertion sort and bubble sort
  - And tiny to code

```c
void gnomesort(int n, int a[])
{int i = 0; while (i < n) { if (i == 0 || a[i-1] <= a[i]) i++; else {int t = a[i]; a[i] = a[i-1]; a[--i] = t;}}}
```
Slowsort

- We've seen **divide and conquer** algorithms...
  - Divide problem half
  - Conquer each piece recursively

- Slowsort uses a **multiply and surrender** philosophy
  - Replace problem with two slightly smaller problems
  - Multiply subproblems into sub-subproblems
  - Surrender solution when it can no longer be delayed
Slowsort

- To sort a list of elements:
  - Find the maximum
  - Sort the rest of the elements

- To find the maximum of a list of elements:
  - Recursively sort the first half
  - Recursively sort the second half
  - Return the larger of the two maxima

- We might be doing some extra work here...
  - Best case \( O(n^2) \)
  - Worst case ???
What have we learned?

● Unix Basics
  ○ Compilation
  ○ Execution
  ○ Both from the command line!

● C++ Classes
  ○ Constructors
  ○ Destructors
  ○ Member variables / functions
  ○ Header / Implementation files
What have we learned?

● Memory management
  ○ Object creation (What does "new" do again?)
  ○ Objection deletion (Because C++ won't do it for you)

● C++ under the surface
  ○ Pointers
  ○ References
  ○ Operator Overloading
  ○ Templates
  ○ Primitive Arrays
What have we learned?

- Coding in an unfamiliar language
  - Learning outside of lectures
  - Looking concepts up on your own
  - Very useful skill

- Beyond C++
  - Quite a few python/ruby/other submissions so far
  - Hopefully more for assignment 5
What have we learned?

● Putting theory into practice
  ○ Understanding a data structure is very different from implementing it

● Understanding underlying structures
  ○ How is a vector implemented?
  ○ How is a map implemented?

● The data structures you've implemented occur everywhere!
Class Evaluations

- I have a very one sided perspective on this class
  - Please share your side of things
  - Let me know what worked
  - Let me know what could have worked better

- Be honest
  - Don't worry about hurting my feelings

- Available through duckweb
That's all!

- Thanks for a great lab
- I hope you enjoyed it as much as I did