CS61B Review Session

Hongli (Bob) Zhao

Agenda

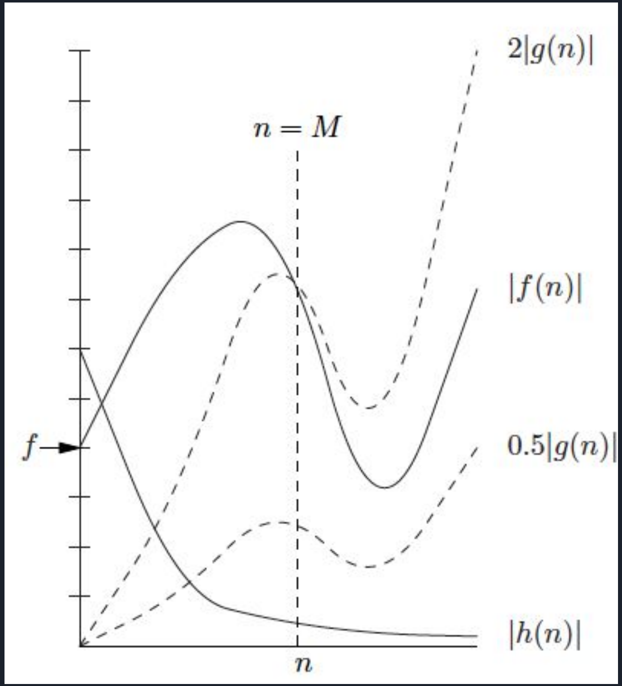
- Asymptotic analysis
  - Big O notation
  - Tips for Runtime Analysis
  - Example
- Sorting algorithms
  - Comparison sorting vs. count sorting
  - Runtime analysis
  - Stability
  - Tips for Exams

Asymptotics

Importance of Analyzing Time and Space Complexity

- Complexity is a measure of how "good" our program is
- Allows us to quantitatively and qualitatively scale up our programs
- Makes our development process maintainable by observing properties of input data

Big-O, Big-Omega and Theta



- We are interested in how the algorithm behaves in the **long run**
- Specify asymptotic bounds on families of functions:
  - O(f(x)) - the bounded above family
  - Ω(f(x)) - the bounded below family
  - ϴ(f(x)) - tight bound
    - ***Note***: "bounded above/below" does not specify absolute performance
    - Common "shapes": O(1), O(N), O(N^p), O(logN), O(NlogN), O(a^N), O(N^N)

General Rules of Thumb

- Do not make assumptions about the size of an input
  - *Demo*
- Only consider asymptotic behavior
  - Ignore lower order terms and constants
  - Ex. What is the relationship between $O(\log N^2)$ and $O(\log N^3)$?
- Test out different properties of data
  - Different inputs may yield different best case and worst case runtime
  - When upper bound and lower bound differ, there is no tight bound

Sorting Algorithms

- Main ideas
- Stability
- Practice test problems

Terminology

- Stability:

A sorting algorithm is stable if it preserves the original ordering of already sorted items

Ex. {2, 3, 1, 4a, 9, 4c, 7, 4b}

{1, 2, 3, 4a, 4c, 4b, 7, 9}

{1, 2, 3, 4a, 4b, 4c, 7, 9}

- Inversions:

Measures how disordered a sequence of items is

- Number of inversions:
  - The minimum number of pair swaps required to sort the list
  - Ex. {1, 3, 4, 2} has 2 inversions

Comparison sorts

- Does not rely on structure of data; only assumes an order exists
  - Arranges elements in order such that `arr[i] <= arr[i+1]` is true
- Cannot perform better than $\Omega(NlogN)$
  - Proof by ***Stirling's approximation***
- <u>Important sorts</u>: Insertion sort, Selection sort, Quick sort, Heap sort, Merge sort

Counting sorts

- What if we can do more than comparing?
  - range of data is limited (ex. Alphabet, bits, range of integers)
- Groups objects according to a certain criteria, and use the array structure to indicate ordering
- Can outperform $\Omega(NlogN)$
  - Has linear dependence on size of "dictionary"
- Important sorts: LSD sort, MSD sort

Insertion Sort

- In each loop, start from the leftmost unsorted entry, compare with the entry immediately to its left; Swap the two entries if arr[i+1] < arr[i]
- Repeat the first step until the entry to the left is not larger than this entry, or this entry has reached the left end of the array
- Repeat the previous two steps until all entries are sorted
- **Runtime:** Ω(N), O(N^2)
  - Best case achieved when list is nearly sorted
  - Worst case when list is in reverse order

{ 53, 26, 94, 18, 70 }

{ 26, 53, 94, 18, 70 }

{ 26, 53, 94, 18, 70 }

{ 26, 53, 18, 94, 70 }

{ 26, 18, 53, 94, 70 }

{ 18, 26, 53, 94, 70 }

{ 18, 26, 53, 70, 94 }

{ 18, 26, 53, 70, 94 }

Selection Sort

- Starting from the unsorted array, find the minimum value, and swap with the first value
- Starting from the unsorted (N-1) values, find the minimum value, and swap with the second value
- Repeat the process until all values are sorted
- **Runtime:** $O(N^2)$ in all cases:
  - O(N) for finding minimum value
  - O(N) for running selection sort on each entry

{ 53, 26, 94, 18, 70 }

{ 18, 26, 94, 53, 70 }

{ 18, 26, 94, 53, 70 }

{ 18, 26, 53, 94, 70 }

{ 18, 26, 53, 70, 94 }

{ 18, 26, 53, 70, 94 }

Heap Sort

Better than selection sort

- Assuming we are using a max-heap, starting from the unsorted array, heapify the array
- Swap smallest value with the root of the heap, pop the largest value and put at the back of the array
- Re-heapify the (N-1) unsorted array
- Repeat the previous steps until all values have been popped
- **Runtime**: O(NlogN):
    - Heapifying: O(NlogN)
    - Swap: O(1) * N = O(N)
    - Reheapify: O(logN) * N = O(NlogN)

{ 53, 26, 94, 18, 70 }

{ 94, 70, 53, 26, 18 }

{ 18, 70, 53, 26, 94 }

{ 18, 70, 53, 26 }{94}

{ 70, 26, 53, 18 }{94}

{ 18, 26, 53, 70 }{94}

{ 18, 26, 53 }{70, 94}

{ 53, 26, 18 }{70, 94}

{ 18, 26, 53 }{70, 94}

{18, 26}{ 53, 70, 94 }

Quick Sort

- Select a pivot to partition the array (usually the middle element)
- Smaller value goes left, large value goes right
- Repeat the first two steps until all sub-arrays cannot be partitioned anymore or have met a certain limit
- **Runtime**: Omega(NlogN), O(N^2)
    - Depends on specific *choice of pivot*!

{ 53, 26, 94, 18, 70 }

{ 53, 26 } 94 { 18, 70 }

{ 53, 26, 18, 70 } 94

{ 53, 26 } 18 { 70 } 94

18 { 53, 26, 70 } 94

18 { 53 } 26 { 70 } 94

18 26 { 53, 70 } 94

{ 18, 26, 53, 70, 94 }

Merge Sort

- Split: Recursively splits array into halves until further partitioning is impossible (singleton lists)
- Merge: From the bottom level, recursively build up the original sorted list
- **Runtime**: always O(NlogN)!
  - O(N): merging back every level
  - O(logN): number of levels

{ 53, 26, 94, 18, 70 }

{53, 26, 94} {18, 70}

{53, 26} {94} {18} {70}

{53} {26} {94} {18} {70}

{26, 53} {94} {18, 70}

{ 26, 53, 94 } {18, 70 }

{ 18, 26, 53, 70, 94 }

LSD / MSD Radix Sort

- Starting from the most significant / least significant digit, perform counting sort on the digit
- Repeat counting sort on the rest of the digits
  - LSD: from right to left
  - MSD: from left to right
- **Runtime**: O(B)
  - Each placement takes O(1) * B byte size of each entry

{1219, 2523, 1311, 4215, 3132}

{1311, 3132, 2523, 4215, 1219}

{1311, 4215, 1219, 2523, 3132}

{3132, 4215, 1219, 1311, 2523}

{1219, 1311, 2523, 3132, 4215}

Tips for Exam Problems

- **Pattern matching**
  - Heap sort has the greatest “shuffling” when starting out
  - Merge sort does not start sorting until all splitting has finished
  - Look for growing sorted sequence in selection and heap sort
  - insertion sort moves sequentially to the right
- **Algorithm implementations / Choosing algorithms**
  - Example facts:
    - When would we prefer insertion sort over merge sort?
    - How to implement Heap Sort using a min-heap?
- **Details about algorithms:**
  - How many inversions are there in {10,9,7,1,6}?
  - Will {1123, 1830, 1960, 1110, 1210, 1390} ever appear in the process of a LSD radix sort?

Summary

Algorithm	Best case	Worst case	Stability	Note	
Insertion	N	N^2	Yes	Performance depends on number of inversions	
Selection	N^2	N^2	depends	Has constant space. Can be made stable if use linked lists	
Heap	N	NlogN	No	Has the greatest “shuffling”	
Quick	NlogN	N^2	No	Runtime depends on choice of pivoting	
Merge	NlogN	NlogN	Yes	Can be highly paralellized	
LSD/MSD Radix	B	B	Yes	B = N*K (size of array * length of each item)	

Thank you!