**Study Guide**

**Algorithmic Analysis:**

* Big-Oh Notation
* Properties of Growth-rate functions
* Common Growth Rates

**Divide and Conquer (Chapter 4):**

* Major Components
  + Divide step: partition the input into smaller parts and solve those subproblems
  + Conquer: recursively solve each subproblem
  + Combine: merge the solutions of the subproblems
* Recursion
  + T(n) = aT(n/b) + f(n)
    - a >= 1 is number of subproblems
    - b > 1 is factor by which the problem is divided
    - f(n) is the cost off dividing and combining (and any non-recursive work)
* Substitution Method: a way to prove time bounds on the recurrence T(n)
* Recursion-Tree Method: sum the work level by level
* Master Theorem Method: direct formula-based approach for solving many standard recurrences

**Priority Queues & Heaps (Chapter 6):**

* Operations and run times
  + FIND-MAX or FIND-MIN in O(1)
  + INSERT in O(log n)
  + EXTRACT-MAX or EXTRACT-MIN in O(log n)
  + INCREASE-KEY or DECREASE-KEY is O(n)
  + Initialized with BUILD-HEAP which is O(n)
* Advantages
  + Linear (O(n)) time to build a heap from an unordered list
  + Logarithmic time to insert/extract at root or find
  + Efficient insert and removal at the root node
  + Can build heap in place (no extra memory)
  + Parent-child relationship calculated by indexing (no pointers needed)
* Disadvantages
  + Searching for arbitrary elements not at root is linear, which means updating arbitrary elements is linear also
  + Doesn’t guarantee that equal node values maintain any specific order (if any satellite information is tied to it)
* Definition for validity
  + Max heap: each node value is at least as large as the values of its children nodes (less than or equal to for min heap)
  + A binary heap is a complete binary tree (all levels filled except maybe the last one)

**Sorting Exercises:**

* Insertion Sort
  1. Consider 1st element sorted
  2. Take next element and compare it to the elements in the sorted section
  3. Shift it left as needed until it is in the right spot
  4. Repeat
* Mergesort
  1. If only looking at 1 element, then done
  2. Divide array into halves
  3. Sort each half with recursive calls
  4. Merge sorted halves into one array
* Heapsort
  1. Build a heap from the input array (O(n))
  2. Extract/swap root with last element in heap
  3. Decrement heap size by 1 and re-heapify
  4. Repeat
* Quicksort
  1. If only looking at 1 or 0 elements, then done
  2. Choose a pivot point P
  3. Rearrange all elements so those less than P are on one side, and those greater than P are on the other.
  4. Put the pivot between the two partitions
  5. Recursively sort the left and right partitions
* Bucket Sort
  1. Create several “buckets”
  2. Distribute each element of the array into the appropriate bucket
  3. Sort each bucket
  4. Concatenate the sorted buckets in order
* Radix Sort
  1. Identify the number of digits and base/radix
  2. Sort the array by the last significant digit using a stable sort method
  3. Repeat for each significant digit until reaching the most significant
  4. Array is fully sorted after final pass
* Linear Sort

**Dynamic Data Structures:**

* Stacks & Queues
* Linked lists
* Binary Search Trees
* Operations and runtimes for all above

**Hashing:**

* Hashing Basics
  + Hash table – stores elements in table and uses hashing to determine each element’s index
  + Hash function – maps keys to an index in the hash table with a goal of uniformly distributing them across the table
  + Collision – a key is mapped to an index that already has a key in it
  + Separate Chaining – each table holds a linked list of key-value pairs (LL points to other memory location)
    - Easy to implement
    - Extra space overhead for the LL
  + Uniform Hash Function – a hash function is ideally uniform if it spreads keys evenly among the buckets in the hash table and minimizes collisions
  + Open Addressing
    - all elements are stored directly in the hash table (no separate chaining)
    - If a collision occurs, probe for another open spot
      * Linear Probing: h(k,i) = (h’(k) + i) mod m
      * Quadratic Probing: h(k,i) = (h’(k) + c1\*i + c2\*i2) mod m
      * Double Hashing: h(k,i) = (h1(k) + i\*h2(k)) mod m
        + h2(k) = 1 + (k mod m’) if not specified??
        + reduces clustering issues
  + Primary clustering – phenomenon in Linear Probing where long runs of occupied slots tend to get longer, increasing the average search time
  + Secondary Clustering – phenomenon in quadratic probing where different initial hash values can lead to same probe sequence beyond first collision (thus, resulting in infinite collision search sequence)
  + Load factor alpha – alpha = n / m
    - n = # of stored elements
    - m = # of slots in table
    - for separate chaining, average list length is roughly alpha
    - for open addressing, performance degrades significantly as alpha gets near one
* When to use a hash table and when not to
  + Use when you need fast average-case inserts, lookups, deletions (O(1))
  + You can design or use a good hash function with few collisions
  + Data size is large, but direct access array indexing isn’t feasible or keys are too large)
  + Don’t use when:
* Average case efficiency for hash operations
  + inserts, lookups, deletions all (O(1 + alpha))
* Properties of ideal hash function
  + Uniform distribution
  + Minimal collisions
  + Fast computation to convert key to hash value
  + Low risk of clustering
* How to be careful about the constants in hash functions
  + Poor hash functions that ignore high and low order bits
  + Table sizes should be prime numbers
  + Use prime-based moduli for the constants in linear or double hashing to avoid systematic collisions
* Which strategy to use and when (separate chaining, double hashing, etc.)
  + Separate chaining: easy to implement and memory is not too constrained
  + Open addressing:
    - Saves space
    - More sensitive to load factor
    - Double hashing best avoids clustering at cost of a more complex hash function pair
* Worst case efficiency for hash operations
  + Worst-case is O(n) for basic operations and can happen due to:
    - Too many keys collide in the same slot (chaining) or you need to probe the entire table (open addressing)

**Dynamic Programming:**

* Definition & Necessary Conditions
* Elements of Dynamic Programming
  + Optimal Substructure
  + Overlapping Subproblems
* Applications
  + Rod Cutting
  + Knapsack
  + Longest Common Subsequence

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Worst-Case** | **Average-Case** | **Best-Case** |
| Heapsort | O(n log n) | O(n log n) | O(n log n) |
| Insertion Sort | O(n2) | O(n2) | O(n) (already sorted) |
| MergeSort | O(n log n) | O(n log n) | O(n log n) |
| Quicksort | O(n2) (bad pivot) | O(n log n) | O(n log n) |
| Bucket Sort | O(n2) | O(n) | O(n) |
| Radix Sort | O(d \* n) (d = # digits) |  |  |
| Linear/Counting Sort |  |  |  |

Practice Problems

4.3-1

4.3-2

4.3-3

4.4-1

4.4-2

4.4-3

4.4-4