COSC 201 Final Review - Feynman Liang

Asymptotic Analysis

Focuses on dominant asymptotic term (generally highest order polynomial of n), non-dominant terms annihilated during proof.
To prove, find c and n_0 which holds for:

- $O(g(n))$ - $\exists c > 0$ s.t. $\forall n > n_o$, $0 \leq f(n) \leq cg(n)$
- $\Theta(g(n))$ - $\exists c_1, c_2 > 0$ s.t. $\forall n > n_o$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
- $\Omega(g(n))$ - $\exists c > 0$ s.t. $\forall n > n_o$, $0 \leq cg(n) \leq f(n)$

Recurrence

Function which calls itself. Needs a non-recursive **base case**.
Pre/post conditions - truth at start/end of method **Assertions** - truth at point in execution **loop invariants** - truth at each iteration of loop

Master Theorem

Quick way to solve recurrences
$T(n) = aT(\frac{n}{b} + f(n))$ where $a \geq 1, b > 1$

- Case 1: If $f(n) \in O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$
- Case 2: If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$
- Case 3: If $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ **AND** $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and sufficiently large n, then $T(n) \in \Theta(n^{\log_b a})$

Ex: Straussen's matrix multiplication:

$$f(n) = \begin{cases} 1 & \text{if} n = 1 \\ 7f(\frac{n}{2}) + 18(\frac{n}{2})^2 & \text{if} n \neq 1 \end{cases}$$

This recurrence is case 1 ($18(\frac{n}{2})^2 \in O(n^{\log_2 7 - \epsilon})$ for $\epsilon \leq \log_2 7 - 2$), thus $f(n) \in O(n^{\log_2 7})$

Inductive Proof

To prove $P(k)$ is true for $\forall k \geq b$ using strong induction:

1. Prove $P(b)$ for some base case b
2. Assume $P(n)$ is true for $b \leq n \leq k$, show that this implies $P(k+1)$

Weak induction only assumes $P(k)$ to prove $P(k+1)$. This doesn't allow you to use any states past the $n-1$ state while proving P(n).

Divide and Conquer

Can be represented by a recurrence tree with number levels = depth of recurrence and children per node = number of sub-problems per recurrence.

Dynamic Programming

Solves optimization problems which exhibit **optimal substructure** (optimal sol'n depends on optimal solution of subproblems). To implement, define the optimal solution to the problem recursively (ex. best(a,b) = best(a,k) + best(k+1,b) for k=a..b).
Examples: maximum-subsequence, traveling salesman, rod cutting (V(0) = 0, V(k) = max(v(k-1)+p_i for 0<i≤k)

Memoization

Cache past calculations, improve performance on overlapping subproblems.

Solution reconstruction

Add a step to algorithm which keeps track of how optimal value was obtained by filling in array.

Data Structures/ADTs

Data structures are used to abstract lower level implementation details while allowing organization of data. ADTs provide operation on the data structures.

Arrays

- Insert - O(1) unsorted, O(n) sorted
- Delete - O(1) (O(n-1) if need to shift)
- Search - O(n) unsorted, O(log(n)) sorted

Priority Queue

Data structure which maintains sorted order. Typically implemented using min/max heap or sorted array.

Min/Max Heap

Tree structure which has property that $\forall$children $\leq$(max heap) or $\geq$(min heap) parent. Implies root node is max/min. To maintain heap property, must sift up (compare two children for one parent slot).
Sorted array can be used as min/max heap where every 2^k entries represent the kth level of the heap ([1,2,2,3,3,3,3,...]).
For binary (max 2 children) heap:

- insert - O(log n) - insert to empty on bottom and sift up
- getMin/Max - O(1) - get root element
- deleteMin/Max - O(log n) - delete top and sift up

Graphs

A graph contains n vertices connected by m edges. Degree of vertex is number of edges it has, max degree is denoted d. Paths can have weight (weighted) and direction (digraph).
Simple path is a path which does not visit a node twice, can prove shortest path is simple using induction on removing cycles in path.
Undirected max number of edges: $\binom{n}{k} = \frac{n(n-1)}{2}$, directed n^2

Type	allNeighbors	isNeighbor	insertEdge
Adj. Matrix	O(n)	O(1)	O(1)
Adj. List	O(d)	O(d)	O(1)
Edge List	O(m)	O(m)	O(1)
Vertex-Edge	O(max(m,d))	O(max(m,d))	O(m+n)

Adj. matrix has isNeighbor run time advantage, space $O(n^2)$. Adj. list has allNeighbors advantage, space $O(nd)$.

Set

Collection of elements with no redundancies. Maintaining set independence requires looking up the element during insertion to check redundancies. Hash table implementation $O(1)$ for all, array implementation:

Op	Unsorted	Sorted
insert	O(n)	O(n)
lookup	O(n)	O(log n)
delete	O(n/2)	O(n)

Hash Table

Can be used to implement map/set/dictionary. Contains M buckets and n elements. Uses a hash function to convert input to a bucket index.
Good hash functions have uniform distribution of hash values, computationally cheap. Typically consume set number of bits and performs bit operations to hash.
Without collisions, runtimes are:

- Insert - O(1)
- Delete - O(1)
- Lookup - O(1)

To resolve collisions, we can find a new bucket (**open addressing**) using different methods (h(k,i) returns another place to insert i if bucket h(k) is taken):

- Linear probing - $h(k,i) = (h(k) + i) \mod M$
- Quadratic probing - $h(k,i) = (h(k) + i^2) \mod M$
- Double hashing - $h(k,i) = (h(k) + i * h_2(k)) \mod M$

Problem with linear probing is if something collides once, it will always collide. Simpler probing faster but more likely to cause **clustering** - filled blocks which increase number of collisions. Large clusters at end of buckets effectively shrinks size of M.
Quadratic and double hash resolve by varying probed bucket dependent upon i or h_2.
Separate chaining is another collision resolution method where we just append to collided bucket (make linked list). Operations are $\Theta(n+M)$. Performance degrades more gracefully, don't need to resize when full.
Performance of hash map depends on **load factor** = n/m.

Problems and Algorithms

Sorting

Stable - Relative positions of elements don't change after sort
In place - Requires constant memory overhead

Insertion Sort

Construct sorted sub-array at start of array. Insert elements one by one and swap to maintain order. $O(n^2)$

Merge Sort

Divide and conquer. Recursively split array down middle until single elements, then merge all back together.
$T(n) = 2T(\frac{n}{2}) + cn \in \Theta(n \log n)$

Quick Sort

Pick partition element k, partition around k [$< k$,k,$> k$]. Recur on both sides. Expected $O(n \log n)$, worst $O(n^2)$ if partition is always max or min. Can randomize partition element to avoid worst case.

Selection of kth largest

Binary Search

Divide and conquer on sorted array. Guesses middle element and recurs on side dependent upon whether guess was $>$ or $<$ expected.
$T(n) = T(\frac{n}{2}) + c \in \Theta(\log n)$

Quick Select

Randomized quick sort, discards half which does not contain kth largest, $O(n^2)$ worst (always partition around min/max), $\Omega(n \log n)$ best.

Median of medians

Deterministic Quick Select. Partition into blocks of size 5, calculate medians of the n/5 blocks and take median of that to be new partition element. $T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n) \in O(n)$ time.

Matrix Algorithms

Matrix Multiplication

Dynamic programming solves optimal chain parenthesis by diagonally filling in a solution matrix.
Straussen's solves multiplication in $O(n^{\log_2 7})$.

Floyd-Warshall

Solves APSP/transitive closure in $O(n^3)$ instead of n^4 by moving k loop outside. Iterates over intermediate node k rather than path length.

```
T <- G
  for (k=1 to n)
    for (i=1 to n)
      for (j=1 to n)
        //Transitive Closure (Warshall algorithm)
        T[i,j] = T[i,j] || (T[i,k] && T[k,j])
        //All Pairs Shortest Paths (Floyd's algorithm)
        //iterate across the ks (all table lookups)
        M[i,j] = minof(T[i,j], T[i,k] + T[k,j])
```