Announcements

1)

Exam 1 Review

- Asymptotics
  - $f = O(g)\ or\ f \in O(g) \rightarrow f \leq c \cdot g$
  - $f = \Omega(g)\ or\ f \in \Omega(g) \rightarrow f \geq c \cdot g$
  - $f = \Theta(g)\ or\ f \in \Theta(g) \rightarrow f \leq c \cdot g\ \ and\ \ g \leq c \cdot f$
  - General rules: omit additive or multiplicative constants, higher order exponential dominates lower order, exp dom. poly., poly dom log.
- How long does some code take to run?

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed})$$

```
for i in range(n):
        for j in range(m):
                k=n*m
                while k>0:
                        print k
                        k=k-1
```

O(m^2 n^2)

```
for i in range (n):
        for j in range(i):
                print (i,j)
```

$$T(n) = \sum_{i=1}^{n} i = n * (n + 1)/2 = O(n^2)$$

- Three primary types of runtime analysis: best case (fastest runtime on any input), worst case (longest runtime on any input), average case (average runtime based on randomness of the data or algorithm choices)

    - What is worst case for mergeSort? What is best case for mergeSort? Both O(n log n). What is worst case for selection algorithm? O(n^2) What is best case for selection algorithm? O(n)
    - Ridiculous algorithm: BOGO sort. Randomly permute list with (n-1) swaps, check if it is sorted. Best case? O(n). Worst case and average case are the same. Geometrically distributed random variable with probability of success = 1/n!, with expected number of trials n!. But you have to swap (n-1) each iteration, so (n-1)*n!
- Dynamic Programming - solve an exponential number of problem in polynomial time by exploiting relationships among optimal subproblems.
    - Fibonacci, direct translation from mathematical definition
    - Edit distance, the subproblems are more subtle. What are subproblems? How do we move between subproblems? This is essentially the IH, going to the next step, or previous to this step. [Fig]
    - DPs on graphs, shortest paths, adding complexity: introduction to 'invariants' with Dijkstra's algorithm. Bellman-Ford, exponential number of paths from s to t, with negative edges, Dijkstra invariant no longer holds. Instead we add additional information to track problems of a certain complexity, namely, shortest s to t path using 'x' edges. Floyd-Warshall used shortest-paths going through a subset of nodes. [Fig]
- PROBLEM: Largest independent set in a tree.
    - INPUT: a binary tree G(V,E) where nodes have weights
    - OUTPUT: largest subset of nodes $I \in V$ such that no two nodes in the set share an edge
- Notice: each node defines a subtree
- Suppose that we know the maximum independent subset of all of the nodes below some target node i
- We have two choices for i, either it is in the MIS or it is not.
- Then, the MIS of node i is simply the maximum of the MIS of its two immediate children AND the MIS of its grandchildren plus the cost of node i
- **Divide and Conquer** - solve a problem by splitting up the work into smaller subproblems, solving the small problems, and cleverly combining the solutions

Assume n is a power of b (ignore rounding)

Size of subproblems decrease by b → reach base case $\log_b n$ levels

k^{th} level has a^k subprobs.

each of size n/b^k

total work done @ level k is

$$a^k O\left(\frac{n}{b^k}\right)^d = O(n^d)\left(\frac{a}{b^d}\right)^k$$

This work forms a geometric series from $k=0$ to $\log_b n$ w/ ratio $\frac{a}{b^d}$

① $\frac{a}{b^d} < 1$ series decreases and sum given by $O(n^d)$ ← first term

(2) $\frac{a}{b^d} > 1$ series increases

and sum given by last term

$$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b(n)}}{(b^{\log_b(n)})^d}\right) = a^{\log_b(n)}$$

$$= n^{\log_b a}$$

(3) $\frac{a}{b^d} = 1$ all terms $= O(n^d)$

and there are $\log(n)$ of them

- merge sort - partition, recurse, solve when small, merge back up
- binary search - dive into, guarantee solution is in piece you dive into
- selection problem (medians) - take extra time figuring out which set to dive into [Fig]
- FFT - complicated scheme for evaluation of polynomials and convolutions, divide the FFT matrix up to cleverly evaluate a polynomial at complex roots of unity. Useful for filtering, convolutions of 1D or 2D. Edge detection.
- T(n)=2T(n/2)+O(n) => 2^k * O(n/2^k)^1 = O(n)*1 how many levels? log(n)m O(n log n)
- T(n)=2T(n/2)+O(1) => 2^k * O(n/2^k)^0 = O(1)*2^k, increasing, how many leaves? O(n)
- T(n)=4T(n/2)+O(n) => 4^k * O(n/2^k) = O(n)*2^k, increasing, leaves? N. O(n^2)
- T(n)=T(n/2)+O(n^2) => O(n/2^k)^2 = O(n^2)*(1/2^2k), decreasing, top? O(n^2)
- Algorithm analysis technique: loop invariants, show that it holds **initially,** it is **maintained (**if true before loop, true after)**,** and it is meaningful at **termination.** Essentially proof by induction (base case, induction hypothesis, conclusions).

- Data structures -- maintaining invariants for a representation of data
    - Heaps, weight of parent > weight of child, left-to-right filling
        - Max-heapify: O(log n), assumes children are heaps, moves node down
        - Build-max-heap: O(n), max-heapifies internal nodes [Fig]
        - heapsort: O(n log n), removes root, replaces with bottom-right most node
        - insert, getMax, increaseKey (move up)
    - Stacks, queues, linked-lists, trees
    - Binary search trees: invariant, left child less than node, right child greater than node
    - Wont be on exam: Red-black trees add a color per node, maintain invariants that red node must have black nodes as children, all node to leaf paths must have the same number of black nodes
        - Tree rotations used to preserve invariants