1 Master Theorem

Given $T(1) = c$.
$T(n) = a * T\left(\frac{n}{b}\right) + d * f(n)$

1. $(n^{\log_b a})$ if $a \geq f(b)$
2. $(f(n))$ if $f(b) > a$
3. $(n^{\log_b a} * \log n)$ if $f(b) = a$

2 Big O

$f(x)$ is $O(g(x))$ => $\lim_{x\to\infty} \frac{f(x)}{g(x)} \leq c$ for some constant c
$f(x)$ is $\Omega(g(x))$ => $\lim_{x\to\infty} \frac{f(x)}{g(x)} \geq c$ for some constant c
$f(x)$ is $\Theta(g(x))$ => $\lim_{x\to\infty} \frac{f(x)}{g(x)} = c$ for some constant c

3 Random Facts

1. You can multiply in $\Theta(n^{\log_2 3})$.
2. Bubble sort is worst case $\Theta(n^2)$ and best case $\Theta(n)$ and average case $\Theta(n^2)$ - Stable, In place
3. Merge sort is $\Theta(n \log n)$ for all cases. - Stable, Not in place
4. Quick sort is worst case $\Theta(n^2)$ and best case $\Theta(n \log n)$. - Not stable, In place
5. You can comparison sort no faster than $O(n \log n)$
   (a) A decision tree for sorting has >= $n!$ leaves.
   (b) Minimum average depth of a tree with k leaves is $\Omega(\log k)$. So if a decision tree has $n!$ leaves, it must have $\log n!$ depth, which is $\Omega(n \log n)$.
6. Non Comparision Sorting
   (a) Bucket Sort/Bin Sort - hash your key to a bucket. Add it to the linked list for the hashed bucket. Now iterate through buckets, and print out each linked list in order. Requires a bounded input set. $\Theta(n + m)$ where m is the number of bins. - Stable, Not in place
   (b) Radix Sort - Bin Sort based on the least significant digit until you run out of digits. Requires a bounded input set. If we sort n numbers in base m and they are each k digits long, it takes $O(k(n + m))$ - Stable, Not in place Proof by induction – if digits are the same then they are already sorted O(wlogb(n+b)) we get to choose b for bn: O(wblogb) –make b as small as possible How long to sort n numbers in range [n2]? w = 2logn O(n) – 2 digits in base n

4 QuickSelect

This takes in an array, and finds the kth largest element in the array.

1. Split array into subarrays of size 5. Find the median of each subarray. This takes $O(n)$ comps.
2. Find the median of the medians of all the subarrays recursively. This median must be bigger than $\frac{3n}{10}$ elements and smaller than $\frac{n}{10}$ elements.
3. This tells us $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + n = \Theta(n)$

5 Graphs

1. A simple path is a path without cycles
2. A subgraph is strongly connected if you can get from all nodes in it to all other nodes.
3. A subgraph is weakly connected if its directed, but would be strongly connected if it was undirected.
4. Trees have $n - 1$ edges. Have at most $n - 1$ leaves. Has an average depth of $\log_2 n$

5.1 Red Black Trees

A binary tree such that:

1. Every node is red or black
2. Every leaf is black
3. The children of red nodes are black
4. Every simple path from a node to a descendent leaf contains the same number of black nodes

b(v) = number of black nodes from v to a leaf, not including v
Claim: The subtree rooted at v has at least 2b(v)-1internal nodes

5.2 Treaps

Generate heap-key h at random. Insert (k, h) into a treap – respects tree order for k, and heap order for h
Lemma: If there are no repeated values, the treap is unique
Proof: Trivial for 1 pair, but for more than one pair, take min heap key as root, and partition by tree key. Each subtree is unique by induction

5.3 LCA

Given rooted tree T, preprocess it to support queries
Query: LCA(u, v) return the deepest node that is an ancestor of u and v
Euler Tour: 2n - 1 – Visit each edge twice, way down and on the way up
A is the smallest depth between occurrence of I, J
Stuff before u — u — u and stuff below u — u — stuff after u and before v — v — — v —
A X Z X must contain the LCA, called w, of u and v. First occurrence of w is in A and the last occurrence is in Z. Everything between these occurrences are in ws subtree, so they are at least as deep as w. So w is the shallowest node in X.
RMQ & LCA - can be solved in $O(n)$ preprocess $O(1)$ query.

5.4 Level Ancestor

Query - LA(u, l) –¿ ancesotr of u with depth l
Pointers – point to ancestors with depths are a power of 2 less than your depth [O(nlogn), O(nlogn)]
Path decomposition (long path) – Double long Path – twice the size with ancestors above. Home path of node is unique path where node is in the bottom half of array.
Lemma: Let u be any node and let v be the top node in u's home path. Then height(v) $\geq$ 2height(u) or v is the root. Proof:Distance from v to bottom of this array is at least twice the distance of u to the bottom if —green array— = —red array—. If not equal then v is the root Prep: O(n) Query: O(logn) because height(root) n
Combine pointer and decomp to achieve [O(nlogn), O(1)]

5.5 Minimal Spanning Tree

Greedy Algorithm

1. Initialize all nodes to be infinite distance so far $O(n)$
2. Assume we have a MST so far.
3. Add the cheapest edge to the MST that adds a node we haven't seen before.
4. loop over steps 2-3 $O(n)$

$O(n^2)$ total

5.6 Cartesian Trees

Root - min element of A A[i]. Left subtree - Cartesian Tree on A[1] ... A[i-1] Right subtree - Cartesian Tree on A[i + 1] ... A[n] Constructed in O(n) time. Reduce RMQ to LCA put element of A into Cartesian Tree.

Kruskall's Algorithm

1. Sort edges by weight. $O(logn^m) = O(m \log n)$
2. Add an edge to MST if it connects two subgraphs that aren't already connected. $O(n \log n)$ for unions ($O(\log n$ and we do it n times) and $O(m)$ for finds (O(1) and we do it m times), so total for this step is $O(n \log n + m)$

Use Union Find for Step #2. Complexity $m \log n$, where m is the number of edges.

5.7 Cliques

Clique is a subgraph in a graph in which all nodes are connected to each other directly. Independent set is a subgraph in a graph in which no nodes are connected to each other at all.

MAX-CLIQUE - Iterate from i=n to 1 and call Clique(G, i) on it FIND-CLIQUE - remove nodes, and check to see if Clique still exists

5.8 DFS

1. Takes $O(n + m)$ time.
2. Finds connected components.
3. Tells you if you have a DAG (Directed Acyclic Graph) - If the DFS contains back edges, then you have a cycle.
4. Top sort - Number edges from n to 1 as you finish DFS at a vertex.
5. Strongly Connected Components - Top Sort, reverse edges, run DFS in order of Top Sort.

5.9 Van emde Boas Trees

Van emde Boas Trees - Does everything in $O(\log w)$ or $O(\log \log N)$ time.

Fusion Trees - Do everything in $O(\log_w n)$ time or $O(\frac{\log n}{\log w})$ time.

You can sort using these in $n*\min(\log w, \log_w n)$ time.

Both take $O(N)$ space.

A van emde boas tree that stores N things contains

1. min and max of its elements.
2. $\sqrt{N}$ van emde boas trees that each store $\sqrt{N}$ things - We will call these the buckets.
3. A summary tree stores $\sqrt{N}$ things. These things are the indexes of the above that are nonempty.

To insert/delete a number, you split up the higher and lower order bits of the input. You use the high order bits to determine which bucket to put the number in. We will then recurisvely insert/delete the lower-order bits into that bucket (tree), and update min/max.

To successor a number, you compare to min/max. If it's between min and max, you split the bits and index with high order bits into a bucket. Compare to min/max of the bucket. If it's between min/max, recursively call successor on bucket using low order bits. If it's less than min, return min. If it's greater than max, recursively call the summary widget using the high order bits to find the next non empty bucket and return its min.

The trick here is that if you make one recursive call on input of size $O(\sqrt{N}$, you get $O(\log \log N)$ time.

6 Identities

Basic identities.

6.1 Log Rules

1. $a = b^c <=> log_c(b)$
2. $a = b^{log_b(a)}$
3. $log(ab) = log(a) + log(b)$
4. $log_b(a) = log(a)/log(b)$
5. $log_b(1/a) = -log_b(a)$
6. $a^{log_b(c)} = c^{log_b(a)}$

6.2 Summation Rules

1. $\sum_{i=1}^{n} i = 1/2(k(k+1))$
2. $\sum_{i=1}^{n} i^2 = 1/6(k(k+1)(2k+1))$
3. $\sum_{i=1}^{n} i^3 = 1/4(n^2(n+1)^2))$
4. $\sum_{i=1}^{n} i * 2^i = (k-1) * 2^{k+1} + 2$

7 Min Degree Spanning Tree

7.1 Modifying the problem into a decision problem

Here is the modification to MDST:
Input: A graph G, and an integer k.
Output: Yes, if the graph has a spanning tree where each node has a degree of at most k.

7.2 Modified MDST is NP-Complete

We will prove that Hamiltonian Cycle reduces to the modified MDST.
Let's say we had a solution for the modified MDST problem.
We can now determine if G contains a Hamiltonian Cycle by checking to see if G has a spanning tree where each node has a degree at most 2. If G contains such a spanning tree, we know that G contains a Hamiltonian Path.
Since Hamiltonian Path can be reduced to Hamiltonian Cycle and vice versa Proof by Homework, we know that Modified MDST is NP-Complete.

7.3 Modified MDST provides a solution for MDST

First we will find the minimum degree d of a spanning tree in G, by calling modified MDST(2 . . . k).
Once we have the minimum degree, we will find the spanning tree T by the node removal technique discussed in class. Q.E.D