Things to maybe put on your cheatsheet

April 22, 2014

1 Old material

1.1 Geometric Series

\[ \sum_{k=0}^{n-1} ar^k = a \frac{1-r^n}{1-r} \]

1.2 Sorting

- Sorting using pairwise comparison: \( O(n \ast \log(n)) \)
- Sorting groupable things (colors, numbers) is faster (Counting Sort, linear)
- Sorting numbers with \( k \) digits on average via Radix Sort: \( O(kN) \)

A “stable” sort does not re-order things that already occur in sorted order.

1.3 Creating data structures and algorithms

- Always, always write your invariants! They help you see mistakes early, and they help us grade leniently.
- “Efficient” implies “advantage”, faster than a dumb brute force solution. For example, storing a list of numbers some clever way allows efficient queries if they are faster than doing the same query on an unordered list.

1.4 Proofs

To prove an algorithm, demonstrate:

- termination (it does not go on forever)
- safety (the result it produces is always correct).

If the proof is non-trivial, use one of the following proof techniques to construct a well-argued proof (proof by example, or proof by long paragraph of rambling text are not proofs):
• **contradiction:** Assume the thing you are profing is false, and show that it leads to obviously false conclusions. This is usually the way to go. Example: prove there is no largest integer. Suppose that this is false, and there exists some largest integer $k$. But $k + 1 > k$ is also an integer, so $k$ cannot be largest. But $k$ is defined as the largest integer! Therefore the initial statement must be false, and there is no largest integer.

• **induction:** In the case of a recursive algorithm, prove the base case (when it does not make a recursive call), then prove the inductive case (often by contradiction, by assuming the recursive call does the right thing and extending this to show the entire algorithm does the right thing).

### 1.5 Master theorem

$T(n) = aT(n/b) + f(n)$

- $f(x) = O(n^{\log_{b}a}) \rightarrow T(n) = \Theta(n^{\log_{b}a})$ - polynomially more root at leaves.
- $f(x) = \Theta(n^{\log_{b}a}) \rightarrow T(n) = \Theta(n^{\log_{b}a} \log n)$ - equal asymptotic complexity at each row in recursion tree.
- $f(x) = \Omega(n^{\log_{b}a}) \rightarrow T(n) = \Theta(f(n))$ - polynomially more work at root.

### 1.6 Data structures

We know a lot of good data structures. Use them!

#### 1.6.1 Unordered array. (use only if no cleverness needed)

- O(1) insert (amortized, due to table doubling)
- O(N) find
- O(find + 1) delete

#### 1.6.2 Sorted array. (use if you have all data upfront)

- O(log N) insert
- O(log N) find (binary search)
- O(N) delete
1.6.3 Binary Min/Max Heap (use if you see “min” or “max”)

- O(n) to build
- O(log n) insert
- O(find + log n) delete
- O(log n) extract max/min
- O(1) find max/min
- O(log n) increase/decrease key
- Stored in an array by $left(i) = 2i$, $right(i) = 2i + 1$ as a balanced binary tree.

1.6.4 Binary Search Tree (BST) with AVL (use if need to store sorted dynamic data)

- O(n * log n) to build
- O(log n) height, O(n) leaves
- O(log n) for search, insert, delete
- Can be augmented (keep track of extra information at each node) to implement additional queries efficiently.

1.6.5 Dictionary (use with unordered data)

- O(1) amortized cost for insertion, deletion, search (due to table doubling).

1.7 Hashing

When analyzing expectations for hashing, use uniform hashing assumption: $h(k)$ is uniform random over $[0..m-1]$. $h(i, k)$ is equally likely to be any sequence over $[0..m - 1]$.

1.7.1 Chaining

Dictionary := array of $M$ linked lists (“chains”). Insert element with hash $h(k)$ into list with index $(h(k)\%M)$. Loading factor $\alpha := \frac{N}{M}$, where $N$ is number of elements in dictionary. Expected length of each “chain” is $\alpha$
1.7.2 Open addressing (technically new material!)

Using an auxiliary hash function $h(k)$, define a new function for a sequence of hashes $h(k, i)$:

- Linear probing: $h(k, i) = h(k) + i$ - forms clusters of entries in table
- Double hashing: $h(k, i) = h_1(k) + i \cdot h_2(k)$ - much better!

Dictionary := table with $M$ entries. Insert key $k$ into the first available index $h(i, k)$.

- Danger! When deleting an item, we can’t simply remove it from the table - that would make elements in its chain appear deleted also. We must instead mark it as “deleted”, and allow future insertions to reclaim this slot.
- Danger! When inserting, always search first, as we would want to replace an existing value, if applicable.

1.8 Rolling hash

A “rolling” hash function is a hash function over a set of elements where computing the next (using `append()`, `skip()`) set of elements is efficient, as it reuses some of the previously done work. Example:

- $H(x_0, x_1, \ldots x_{i-1}) = x_0 \cdot a^{i-1} + x_1 \cdot a^{i-2} + \ldots x_{i-1} \cdot a^0$ - still takes linear time with $i$

- But if we have $H(x_0, x_1, \ldots x_{i-1})$, and want to compute $H(x_1, x_2, \ldots x_i)$, we can do so in $O(1)$ by $(H(x_0, x_1, \ldots x_{i-1}) - x_0 \cdot a^{i-1}) \cdot a + x_i$ (remove $x_0$’s contribution to $H$, and add $x_i$’th)

2 New Material

2.1 Numerics

$d$-digit numbers ($d$ “correct” digits when discussing precision).

2.1.1 Addition

Use grade school method (add one digit at a time, keep track of carry) - $O(d)$ complexity.
2.1.2 Multiplication

Express \( A \ast B \) as \((A_{hi} \ast b^{d/2} \ast A_{low}) \ast (B_{hi} \ast b^{d/2} \ast B_{low})\), where \( b \) is the base. Use Karatsuba trick to perform \( 3 \frac{d}{2} \) digit multiplications to obtain \((A_{hi}B_{hi} \ast b^{d} + A_{hi}B_{low} + A_{low}B_{hi} \ast b^{d/2} + A_{low}B_{low})\). \( T(d) = 3 \ast T(d/2) + O(d) \rightarrow \Theta(d \log^2(2)) \).

Other methods exist, but multiplication is not linear, approximately \( O(d^{1.5}) \) ish.

2.1.3 Newton’s Method

Use Newton’s method for division, roots, etc.

1. Think of a function with a root at the desired value. For example, if calculating \( \sqrt{a} \), use \( f(x) = x^2 - a \). This function evaluates to 0 (has a root at) \( x = \sqrt{a} \).

2. Write an expression for \( f' \), the derivative of \( f \). In this case, \( f' = 2x \)

3. Newton’s method refines a guess for an answer by assuming the function is somewhat linear in the neighborhood of the answer. The size of this neighborhood depends on \( f(x) \) in complicated ways. Some functions converge always (sqrt does), but most require a good initial guess, and are not guaranteed to converge. Given an initial guess \( x_0 \), we can iteratively “refine” our guess by: \( x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \). In the neighborhood of the true root (the answer), newton’s method converges quadratically (quadratically decreases the error, doubles the number of correct digits).

2.1.4 Floating point

“Dynamic range” - want to represent small numbers precisely, but also be able to represent very large numbers. Digits of precision != digits in general (1,000,000,000 has only one digit of precision, really). Floating point is a way to write the number in scientific notation!

\[
\text{float} = \{ \text{exponent } e, \text{ significand } s \} = s \ast b^e
\]

for some base \( b \) (we often use \( b = 2 \) or 10).

2.2 Graphs!

Graphs are logically a pair \( G = (V,E) \), and sometimes also weights \( W \), where \( V \) = set of “vertices”, \( E \) is a set of “edges” \((v_i,v_j) \in E\), where \( v_i,v_j \in V \). If applicable, \( W \) defines the “weight” of every edge by \( W(e) \), usually a real number.

Graphs can be represented as an adjacency list (list of edges, good for graphs with relatively few edges), or as a \( V \times V \) matrix \( G \), where an edge \( (\text{row} \rightarrow \text{col}) \) is defined if \( G[\text{row}][\text{col}] \) is defined (great for graphs with a LOT of edges).

\( \text{DAG} = \text{(D)irected - edges have arrows - (A)cyclic - no path from any vertex back to itself - (G)raph} \)
2.2.1 BFS

traverse the nodes reachable from $s$ starting at some node $s \in V$ by creating a
tree of edges. Invariant: When adding a node $v$ into the BFS tree induced by
$s$, where $v$ is $k$ edges away from $s$, all nodes currently in the BFS tree are $\leq k$
edges away from $s$. A BFS traversal takes $O(E)$ time (because only exploring
connected subset of $s$, so $E$ varies between $V$ and $V * V$).

2.2.2 DFS

traverse the entire graph by visiting a node, and exploring its entire adjacency
in depth-first order before moving on to the next unexplored node. Creates a
“forest of trees”. A DFS traversal of a graph sees several types of edges:

- tree edge - edges by which DFS discovers new nodes
- back edge - edges that connect a node to its already discovered ancestor
- forward edge - edges that connect a node to its already discovered descen-
dant. These do not occur for undirected graphs.
- cross edge - edges that connect a node to an already discovered node that
  is not an ancestor or descendant (either nodes in different paths on a tree,
or different trees altogether). These do not occur for undirected graphs.

A DFS traversal takes $O(V + E)$ time.

2.2.3 Topological sort

A topological ordering of a graph $G$ is an ordering of its vertices $[v_0, v_1, .., v_n]$ in
such a way that all the edges point from left to right (all edges $(v_i, v_j) \in E$ are
such that $i < j$). A valid topo. ordering is obtained by reversing the order of
recursive calls in a DFS traversal. ($O(V + E)$ time).

2.3 Shortest Path Finding

2.3.1 Unweighted graphs

Use BFS. Path in the BFS tree from root $s$ to any node $t$ is the shortest path
in $G$. $O(E)time$

2.3.2 Vague template for all shortest path algorithms

Repeat until done: Pick edge $u \rightarrow v \in E$: relax $v.d$ using $(u \rightarrow v)$

Relax compares the “previous best” distance $v.d$ with newfound path $u.d +
weight(u \rightarrow v)$, and updates the path if it is indeed shorter. This takes $O(1)$
time.
2.3.3 Weighted DAGs

Shortest paths occur in topological order, so go through all the vertices once in topological order, and for each vertex, relax all of its outbound edges. $O(V+E)$.

2.3.4 Weighted graphs with no negative edges

Use Dijkstra’s - greedy algorithm similar to BFS, where shorter paths are explored first. Use a priority queue sorted by current node.d values to prioritize shortest paths. $O(V \cdot \log V + E \cdot \log V)$ if using a binary heap to implement the priority queue.

2.3.5 Non-DAG graphs with negative edges

Use Bellman Ford!

Repeat $V - 1$ times: Relax all edges.

This takes $O(V \cdot E)$ time, and can be used to detect negative cycles (Negative cycle exists along path → no shortest path exists)

2.3.6 Longest path

This is very hard in general, but special cases are doable Longest path on DAG = Shortest path on DAG with edge weights multiplied by $(-1)$