Algo Final

tyang27

May 2019

1 Complexity

- $O(f(n))$: $T(n) \leq cf(n)$
- $\Omega(f(n))$: $T(n) \geq cf(n)$
- $\Theta(f(n))$: $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

1.1 Summations using integral approximation

Taking integrals gives you a lower and upper bound. If divide by 0 error, take out the first value from the summation.

$$\int_0^n di \leq \sum_{i=1}^{n} \leq \int_1^{n+1} di$$

1.2 Tips and tricks

- log both sides
- $\log_a b = \log b / \log a$
- $\log ab = \log a * \log b$
- $n^a * n^b = n^{a+b}$

2 Recurrences

2.1 Induction

- Base case: $T(1) = c$
- Induction hypothesis: $T(n) = O(f(k))$ for $k < n$
- Induction step:
$$T(n) \leq cf(k)$$

2.2 Master's

- Format: $T(n) = aT(n/b) + cn^k$ and $T(1) = c$
- Conditions: $a \geq 1$, $b > 1$, $c > 0$, $k \geq 0$
- $T(n) = \Theta(n^k)$ if $a < b^k$
- $T(n) = \Theta(n^k \log(n))$ if $a = b^k$
- $T(n) = \Theta(n^{\log_b(a)})$ if $a > b^k$

2.3 Tree method

Work done at each level of the tree times number of leaves.

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

If $\frac{a}{b} + \frac{c}{d} < 1$, try $\Theta(f(n))$. Else if it equals 1, try $\Theta(n \log n)$.

2.4 Unrolling

Substitute $T(an - b)$ into $T(n)$, until you can generalize to i iterations. $T(X)$ to the base case, so $T(X) = 0$ and $X = 1$ or whatever is convenient. Plug the number of iterations back in.

3 Characteristic root

$$T(n) = T(n-1) + T(n-2)$$

Let $T(n) = c^n$. Exponentiate, rewrite as polynomial, and divide by the lowest degree,

$$c^n = c^{n-1} + c^{n-2}$$

$$c^n - c^{n-1} - c^{n-2} = 0$$

$$c^2 - c - 1 = 0$$

Using the roots, solve systems of equations to get coefficients or simply take the highest value,

$$T(i) = ar_1^i + br_2^i + \dots$$

Then,

$$T(n) = ar_1^n + br_2^n + \dots$$

4 Algorithms

4.1 Divide and conquer

4.1.1 Integer multiplication

4.1.2 Matrix multiplication

4.1.3 K-th order statistics using median of medians

- Divide the data into n/k groups of k elements.
- Find the medians of the n/k groups by comparing each one with each other in k choose 2 steps. Find the median of medians by recursing $T(n/k)$.
- Separate groups as higher or lower than median of medians in n.
- Recurse on the worst case, which is a portion p of the n/k full groups of k, and another portion $1 - p$ of the n/k partial groups of $k/2$.

$$T(n) = \frac{n}{k}\binom{k}{2} + T\left(\frac{n}{k}\right) + n + T\left(k\left[p * \frac{n}{k}\right] + \frac{k}{2}\left[(1-p) * \frac{n}{k}\right]\right)$$

4.2 Towers of Hanoi

Move $n - 1$ off recursively, then move the bottom one, then move the $n - 1$ back.

$$T(n) = 2T(n-1) + 1$$

5 Sorting

5.1 Sorting lower bound

Assume n distinct items. Then there are $n!$ permutations of these n data. We compare elements based on two possibilities/branches, the first element is greater or the second is. Then, $2^d \geq n!$ and we can solve for d. Using Stirling, $d \geq \log n! \geq n \log n$.

5.2 Stability of sorting

Stable if equal elements in input array stay in the same order of the sorted array.

5.3 Count sort - $O(n)$

Sort into buckets.

5.4 Radix sort

Sort the last column, then left one, until you run out of columns. $O(nk)$.

6 Dynamic Programming

- Algorithm takes exponential time and we want polynomial
- Algorithm has optimal substructure and repeating function calls. Meaning that answer is deterministic, same input will always give you the same output.
- Generally, define your subproblem's output, and what other output it depends on. Then, specify how to fill in the table.

6.1 Matrix chain product

Dp calculates operations from i to j, testing each split k. While growing your window between i, j,

$$c_{i,j} = \min_{i \geq k \geq j} \{m_{i,k} + m_{k+1,j} + p_{i-1}p_k p_j\}$$

6.2 Longest common subsequence

Dp calculates the longest common substring from i to j.

$$c_{i,j} = \max \begin{cases} c_{i-1,j} \\ c_{i,j-1} \\ 1 + c_{i-1,j-1} & x_i = y_j \end{cases}$$

6.3 Longest increasing subsequence

Do calculates the longest increasing subsequence from the beginning of the string to j.

$$c_j = \max_{0 \leq i < j} \begin{cases} c_i + 1 & x_i < x_j \\ 1 \end{cases}$$

7 Data structures

7.1 Heaps

- Property: Parent is larger than children
- Op: Heapify - convert array to tree, go left and up, bubbling parents down. Maintain loop invariant on subtrees. $O(\log n)$

- Op: Insert - insert node to maintain complete tree, then bubble up $O(\log n)$
- Op: Delete - swap with root, bubble new root down $O(\log n)$
- Op: Peek - look at root $O(1)$

7.2 Red black trees

- Property: Every node either black or red.
- Property: Root and NIL leafs are black.
- Property: If a node is red, then both its children are black.
- Property: All simple paths below a node contain the same # of blacks.
- Property: Max height is alternating reds and blacks $2\log n + 1$.
- Op: Insertion - go down binary search tree, insert as red. If parent is black, no problems. If parent is red, check uncle. If uncle is red, recolor (parent, uncle, gradnaprent), and recurse up to fix properties. If uncle is black, either do single or double rotation.
- Op: Deletion - go down binary search tree to find the node to delete and the node closest in value that can replace it, if replacement is red, or the removed node is red, black height property not violated, so just set the node to black. If replacement and removal nodes are black, set replacement to double black, which is worth two heights.
- Subop: Fix double black - if double black is root, change it to black. If double black's sibling is black and at least one nephew is red, rotate the sibling into the doubly black node. This may be a single or double rotation. If the sibling is black and both nephew are black, recolor the sibling, and give the double black problem to the parent. If sibling is red, rotate subtree into double black, and set it to red. Then, use the other case.
- Augment: Can keep track of size of a subtree by doing $node.size = left.size + right.size + 1$.

7.3 Hashing

7.3.1 Static

- The idea of a static dictionary is that we know how many elements are in our universe beforehand and what they are. We can search in $O(1)$, but we cannot insert.
- A set of functions G is 2-universal if the size of the set of functions that collide on distinct x and y into an array of R is $\leq |G|/r$.
- We can use this idea to create 2 layer hash table. Even though we use m^2 space for each secondary hash table, this is is surprisingly $O(n)$ space.
- First layer uses a hash function that maps elements to r bins. Then, another layer of r hash functions that map values into a secondary hash table of m^2. This only works because we know how many things will map into each index of the first layer.

7.3.2 Dynamic dictionary

- Different from static hashing, since we can insert.
- Linear probing is the simplest.

8 Techniques

8.1 Probability

$$E(X) = \sum_{x \in X} x * p(x)$$

Useful for quick sort analysis and randomized algorithms.

8.2 Amortization

$$\textit{steps} + \textit{potential} < k * \textit{operations}$$

An operation is sort of like a function call, whereas steps are the number of moves/cost related to the operation. Potential is user-defined and represents the highest possible number of steps after a move. The idea is that we use potential to soften the blow of a sudden jump in cost. We look at change in potential to prove complexity.

8.3 Greedy

- Take the locally optimal choice
- May not necessarily be optimal, e.g. in superstring problem, given a list of substrings, maximizing overlap between the substrings does not allow you to find the shortest superstring that contains all the substrings.

8.3.1 Huffman

- Goal is to encode a message with the smallest number of bits, given the frequencies of symbols.
- Combine the two smallest frequencies. Create a new node that represents the two that is the sum of the joined nodes.

9 Union Find

- Represents disjoint sets
- Two operations – union, and find.
- To find, naive approach is to have a pointer from each node to a parent that represents the set.
- To union, naive approach is to append one list to another. Complexity will be the number of nodes with parent pointers changed.
- Weighted-union heuristic - Append shorter depth set to the larger depth set. The lower bound remains the same, but on an aggregate sequence of operations, $O(m + n \log n)$, since for a pointer to change, it must be part of the smaller set, so the number of changes required grows in powers of 2.
- Union by rank - similar to weighted-union heuristic, but using rank. Rather than using size, we can use an upper bound on height before compression.
- Prop: If rank is r, then at least 2^r vertices by induction, so the $r \leq \log n$.
- Path compression - in a find, we go up normally up to the representative. Then, on tail recursion, update all nodes to point to the representative, so that future lookups will be faster.

9.1 Applications

- Offline minimum - we have inserts and extract min before the algorithm even starts, and we want to see what the mins would be. We basically create a bunch of disjoint sets. Then, iterate through the values from smallest to largest. Use the union find representatives to figure out where the closest extract-min is.
- Connected components of a graph

10 Graphs

- Directed graph (digraph)
- m edges
- n vertices
- Undirected graph iff $(i, j) \in E \implies (j, i \in E)$

10.0.1 DFS - $O(n + m)$

- Algorithm not described.
- Tree edge - When we visit a node for the first time, the edge we arrived on is a tree edge.
- Back edge - If v has been visited, and v is an ancestor of u, then (u, v) is a back edge and represents a loop.
- Forward edge - If v has been visited, and v is a descendent of u, (u, v) is a forward edge. This represents two paths running into each other.
- Cross edge - If v has been visited, and v is neither an ancestor or descendent of u, the (u, v) is a cross edge. This represents a connection to another part of the graph.
- Cyclic - cyclic iff DFS of G has a back edge, since cyclic means that $\exists u, v$ s.t. $u \rightsquigarrow v$, $v \rightsquigarrow u$.

10.0.2 Topological sort - $O(n + m)$

- Assumes acyclic graph.
- Do a DFS on G. List out vertices from right to left based on last visit basis.
- This is useful because we can tell which nodes have no incoming edges, and which ones have no outgoing edges. It also tell us about dependencies.

10.0.3 Strongly connected

- Two vertices are strongly connected iff $\exists u, v$ s.t. $u \rightsquigarrow v$, $v \rightsquigarrow u$
- Graph is strongly connected if all pairs of vertices are strongly connected.
- To find SCC, do DFS and remember the last visit times. Highlight the edges that go from larger last visit time to smaller. Then, find the transpose of the graph by reversing the edges. Perform a DFS on the transpose, starting with the largest finish times to smallest. Highlight the edges that go from larger finish time to smaller. If highlighted in both, then there must be a cycle, so we have a SCC.

10.0.4 BFS

- Algorithm not described.

10.1 Single source shortest paths

10.1.1 Dijkstra - $O((m + n) \log n)$

- Assume positive weights.
- Create a priority queue of closest neighbors. For the next closest neighbor u, can we reach u's neighbors v faster by going through u?
- By triangle inequality, we can construct a dp algorithm. $O(m \log n)$
- Correctness by loop invariant.

10.1.2 Bellman-Ford - $O(nm)$

- No assumption on the weights.
- $N - 1$ relaxations over every edge. If we can reach a nodes u, v in d_u and d_v, can we get to v faster by going through (u, v)?
- If Nth iteration continues to relax the weight, must be a negative cycle.
- This algorithm generalizes to all-pairs in Floyd-Warshall.

10.2 All pairs shortest path, three ways

10.2.1 Single source n times - $O(n((m+n)\log n))$

10.2.2 Matrix multiplication - $O(n^3)$

- Try all paths of k length, up to N.
- For each index (u,v), loop through intermediate node x and see if you can do $u \to x \to v$ faster than current $u \to v$.

10.2.3 Floyd-Warshall - $O(n^3)$

- N-1 relaxations, through one node x at a time.
- For each intermediate node x, loop through indices (u,v) and see if you can do $u \to x \to v$ faster than current $u \to v$.

10.3 Max flow

- Given a source s and a sink t, and a capacity of each edge $cap(e)$, we would like to maximize the flow/production/$|f|$ from source to sink.
$$|f| = \text{out of source} - \text{ into source} = \sum_{v\in V} f(s,u) - \sum_{v\in V} f(v,s)$$
- Property: The flow is bounded by 0 and the capacity.
$$0 \le \mathit{flow}(e) \le \mathit{capacity}(e)$$
- Property: At any vertex except the source and sink, flow is conserved, so that you can only flow out as much as you flow in. $\forall u \in V - \{s,t\}$,
$$\sum_{v\in V} f(v,u) = \sum_{v\in V} f(u,v)$$
- Residual graph - some edges represent capacity left, some edges represent capacity used up, indicating production.
- Property: At most double the number of edges going from a flow graph to residual graph
- Property: An augmentation of a flow by Δ obeys flow properties.
- Augmenting path - we augment all edges in a path, restricted by the bottleneck, the smallest edge.
- Partition the vertices into S,T and look at all the flows going from $S \to T$ minus the flows going from $T \to S$. The cut will represent production/flow through the cut and will be bounded by the capacity.
$$|f| = f(S,T) = \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) \le \mathit{cap}(S,T)$$
- Max flow min cut: 1) f is a max flow in G 2) no augmenting paths 3) $|f| = c(S,T)$
- Max flow min cut makes intuitive sense because the goal is to send as much flow through the network as possible, but this is limited by the smallest link. If you augment by the smallest link, then you maximize the amount that leftovers the other links can still send.

10.3.1 Ford-Fulkerson

- Start with no flow from source to sink.
- While there is an augmenting path capacity edge, find the min cut, and augment the flow by increasing the weight going towards the source and decreasing thew eight going towards the sink.

10.3.2 Edmond Karp - $O(nm^2)$

- Ford-Fulkerson may take too long, if you choose bad augmenting paths.
- Ford-Fulkerson, but rather than randomly choosing an augmenting path, choose the smallest edge in the shortest path.

11 P and NP

- P if there exists a polynomial time algorithm
- NP if there exists a polynomial verification algorithm or nondeterministic polynomial algorithm, $P \subseteq NP$
- NP complete if NP (by construction) and as hard as any other problem in NP (by reduction)
- $A \leq_p B$ - there exists a polynomial time mapping reduction from A to B such that 1) given an input in A, you can 2) transform it into an input in B such that 3) B gives the same answer as A. that is, if $x \in A, f(x) \in B$ and if $x \notin A, f(x) \notin B$ (or if $f(x) \in B, x \in A$)
- A no harder than B, so if you know that B is in P, then A must also be in P.
- B no easier than A, so if you know that A is NP, then B must also be NP.
- Reductions are transitive, so showing that one algorithm from NP is in P brings a lot more into P.

11.1 SAT problem

- Given a boolean expression, can it be satisfied? In other words, does there exist an input such that the overall output is true?
- Circuit-SAT is NP complete by comparing it to a general problem in NP-complete
- SAT is NP-complete because we can reduce Circuit-SAT to it. The intuition is to create a new variable for each gate. Then, this is basically the boolean logic version of Circuit-SAT, so it must also be NP-complete
- 3SAT is NP-complete because we can reduce SAT to it. The intuition is to create a new "hidden" variable for each logical operator, similar to what we did above, such that the new variable is true iff its inputs are true. Because this is a parse tree, we know that there can be at most two inputs, so two inputs plus another "hidden" variable is at most three inputs. If unary operation or not enough inputs to make 3, just add a dummy variable such that whatever it is, the output is still logically equivalent. This construction is correct because it satisfies 3SAT, and is logically equivalent to SAT.

11.2 k-clique problem

- Does there exist a k-clique in a graph?
- NP - given a set of vertices and a graph, you can check the number of vertices, that each vertex and edge exists and is valid
- $SAT \leq_p kclique$
- NP hard - construct a graph, where each clause forms a disjoint, unconnected set of nodes, choose one node from each clause to color. The only ones you cannot choose are the complement, since A and $\overline{A}$ cannot be true at the same time. If these colored nodes form a k-clique, then we know that this is satisfied. However, we know that SAT is NP-complete, so k-clique must also be NP-complete

11.3 Vertex cover

- A vertex cover is a subset of vertices from a graph such that it is connected to all the edges in the graph. The vertex cover problem is finding the minimum vertex cover, the fewest nodes required.
- NP - just check if all edges can be covered
- NP hard - We can reduce clique to vertex cover, since the vertex cover of the complement of a graph tells you which nodes are not connected to everyone else in the clique.

11.4 Hamiltonian cycle problem

- Hamiltonian simple is a simple cycle going through every vertex
- NP - given a list of vertices, we can check that $(v_i, v_{i+1}) \in E$, $v_1, v_n \in E$. In addition, $v_1, \ldots, v_n$ are distinct, where n is the number of vertices
- NP hard - NTS, $vertexcover \leq_p hamiltoniancycle$. That is, every input from
- Given a generic graph G, we can construct another graph G', such that G has a hamiltonian cycle iff G' has a vertex cover.