CS 341 Review Sheet

Jim Zhang

Winter 2017

Contents

1 Preliminaries

1.1 Big-$O, \Omega, o, \omega, \Theta$ notation

Know the definitions.

$$T(n) \in O(f(n)) \iff \exists c, n_0 > 0 \mid \forall n \geq n_0, T(n) \leq cf(n)$$
$$T(n) \in \Omega(f(n)) \iff \exists c, n_0 > 0 \mid \forall n \geq n_0, T(n) \geq cf(n)$$
$$T(n) \in o(f(n)) \iff \forall c > 0 \exists n_0 \mid \forall n \geq n_0, T(n) \leq cf(n)$$
$$T(n) \in \omega(f(n)) \iff \forall c > 0 \exists n_0 \mid \forall n \geq n_0, T(n) \geq cf(n)$$
$$T(n) \in \Theta(f(n)) \iff T(n) \in O(f(n)) \cup \Omega(f(n))$$

Also, know this trick.

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = \begin{cases} 0 & \Rightarrow T(n) \in o(f(n)) \\ \infty & \Rightarrow T(n) \in \omega(f(n)) \\ c & \Rightarrow T(n) \in \Theta(f(n)) \end{cases}$$

1.2 Math facts

1. Sum of integers
2. Sum of squares
3. Sum of cubes
4. Sum of i^{th} powers is $O(n^{i+1})$
5. Sum of base-c powers up to $n-1$
6. $\sum_{i=1}^{n} i^{-1} \in \Theta(\log n)$
7. $\log(n!) \in O(n \log n)$

2 Divide and conquer

Break the big problem into smaller problems, solve the smaller problems, and use the solutions to solve the big problem.

Generally, it is easy to prove correctness by induction, but tricky to show runtime due to recurrence. We use three techniques to solve these recurrences.

1. **Guess-and-check:** Conjure an upper bound from the void and prove it by induction.
2. **Recursion tree**: Draw a tree. Consider the total amount of work done on each level, and sum across all levels.
3. **Master theorem**: If $T(n) = aT(n/b) + O(n^d)$, then

$$T(n) \in \begin{cases} O(n^d \log n) & a = b^d \\ O(n^d) & a < b^d \\ O(n^{\log_b a}) & a > b^d \end{cases}$$

2.1 Merge sort

- Sort numbers by recursively merging two halves.
- $O(n \log n)$ time. Can check by any technique.

2.2 2D maxima

- Input: Set of 2D points.
- Output: Set of "maximal" points (no other point has greater x and y coordinate)
- **Naive**: Check each point to see if it is maximal. $O(n^2)$.
- **Divide and conquer**: Sort points by x and bisect into left and right. In right, maximal points are globally maximal. In left, only points which have higher y than the highest point on right (determine by a linear search) are globally maximal. Base case: A single point is globally maximal. $O(n \log n)$.
- **Smart**: Scan right to left, keeping track of highest y point. $O(n)$.

2.3 Closest pair

- Input: Set of 2D points.
- Output: Pair of points of smallest Euclidean distance.
- **Naive**: Check all pairs. $O(n^2)$.
- **Divide and conquer**: Sort points and bisect into left and right. The shortest pair either crosses the median line or doesn't. If it doesn't, we have the answer. Otherwise, the shortest pair must lie on a 2δ-wide strip centered at the median line, where δ is the shortest present on either half — note also that a $2\delta \times \delta$ size "sliding window" has at most 8 points. So $O(n \log^2 n)$. To bring this down, presort by y — this eliminates the $O(n \log n)$ work of sorting points by y in each recursion and brings the runtime down to $O(n \log n)$.

2.4 Integer multiplication

- Input: n-digit integers X, Y
- Output: XY
- Grade school: $O(n^2)$.
- Naive D+C: If $n = 2d$, then $XY = (10^d A + B)(10^d C + D) = 10^{2d} AC + 10^d (AD + BC) + BD$. Still $O(n^2)$.
- Karatsuba-Ofman: Find AC, BD, and $(A+B)(C+D)$. Then $AD + BC = (A+B)(C+D) - AC - BD$. $O(n^{1.6})$.
- Stephen-Cook: Generalize that to k pieces. $O(n^{1+\epsilon})$.
- Schönhage-Strassen: Magic. $O(n \log n \log \log n)$.

2.5 Matrix multiplication

- Input: $n \times n$ matrices A, B
- Output: $A \times B$
- Naive: $O(n^3)$.
- Naive D+C: Find each quadrant. Still $O(n^3)$.
- Strassen's: Turns out you only need to do 7 quadrant multiplications and you can find out the last one. $O(n^{2.82})$ with larger overhead.
- Schönhage: $O(n^{2.522})$.
- Coppersmith Winograd: $O(n^{2.496})$, which later actually turned out to be $O(n^{2.372})$.
- Could be $O(n^2)$ time! Nobody's proved it couldn't.

2.6 Median-of-medians

- Input: n distinct numbers.
- Output: i^{th} smallest number.
- Naive: Sort. $O(n \log n)$
- Quickselect: Use quicksort, but only recurse on the side that matters. $O(n \log n)$ average, but $O(n^2)$ worst.
- Median-of-medians: Partition elements into $n/5$ groups of 5, sort each group, and recursively select a pivot from the $n/5$ medians. Pivot is guaranteed to be good enough to knock the running time down to $O(n)$.

3 Greedy algorithms

Just do what seems best at each step. Generally it is easy to show runtime, but hard to show correctness. Two techninques to show correctness:

1. Greedy-stays-ahead: Prove by induction that at every step, the solution given by the greedy algorithm is at least as good as any other solution.
2. Exchange: Consider any solution. We can transform it into the greedy solution without worsening the solution at any step.

There is also a third technique: Consulting with Mephisto on Walpurgisnacht.

3.1 Activity selection

- Input: n time intervals (start and finish times).
- Output: Largest set of of non-overlapping intervals.
- **Greedy**: Repeatedly pick the earliest finishing interval. $O(n \log n)$ to sort the activities, $O(n)$ if they are presorted. Proof is an argument by Greedy Stays Ahead on the earliest time by which any schedule can have chosen n intervals.

3.2 Job scheduling

- Input: n jobs with lengths ℓ_i.
- Output: Schedule of jobs that minimizes sum of completion times.
- **Greedy**: Sort by ascending length. Proof by induction.

3.3 Job scheduling 2

- Input: n jobs with lengths ℓ_i and weight w_i.
- Output: Schedule of jobs that minimizes sum of weighted completion times.
- **Greedy**: Sort by ascending ℓ_i/w_i. Proof: By exchange.

3.4 Stable marriage

- Input: n men, n women, and each person's preference ranking of the other sex.
- Output: A stable matching — one where there are no lovers (man/woman pairs where each would prefer the other over their assigned partner in the matching)
- **Gale-Shapley**: Greedy, correct, and deadly elegant. Members of one sex iteratively propose to their top remaining choice, and the other sex goes with the best they can get. Proof in five parts:

  1. algorithm terminates
  2. algorithm returns a matching
  3. matching is stable
  4. returned matching is always the same
  5. returned matching is proposer-optimal and receiver-pessimal

  Interestingly, the proof that there is an optimal matching in the first place is the algorithm itself.

4 Dynamic programming

Solve the big problem in terms of slightly smaller problems by building up a table.

- **Showing correctness**: Consider an optimal solution S^* and do some fancy casework with it to get optimal solutions for the slightly-smaller subproblems. Then invert to get the DP recurrence.
- **Back-tracing**: Your DP array will contain the optimal "value" of the solution, but you won't immediately have the concrete solution. To get it, trace backward from your solution's optimal value, picking a correct choice based on your DP recurrence, until you get to the starting point. This always works.

4.1 Linear independent set

- Input: List of n integers $x_1, ..., x_n$
- Output: Maximum set of integers such that no two are adjacent in the list
- **Naive**: Brute force. $O(2^n)$. Literally awful.
- **Greedy**: Pick highest remaining non-conflicting integer. Oh wait, that's wrong.
- **D+C**: I mean, you could try, but you shouldn't.
- **DP**: The optimal solution for the first n integers is the better of
  - The optimal solution for the first $n-1$ integers
  - The optimal solution for the first $n-2$ integers, plus we add the n^{th} integer

  which we can write as

  $$A[i] = \max(A[i-1], A[i-2] + x_i)$$

  and has runtime $O(n)$.

4.2 Sequence alignment

- Input: Strings $X = x_1, ..., x_m$, $Y = y_1, ..., y_n$, penalties $\alpha(i, j) \geq 0$, and the gap penalty δ.
- Output: Minimum penalty of any alignment of X and Y.
- **DP**: The optimal solution for $x_1, ..., x_m$ and $y_1, ..., y_n$ is the best of
  - The optimal solution for $x_1, ..., x_{m-1}$ and $y_1, ..., y_{n-1}$, plus the mismatch penalty $\alpha(x_m, y_n)$
  - The optimal solution for $x_1, ..., x_{m-1}$ and $y_1, ..., y_n$, plus the gap penalty δ
  - The optimal solution for $x_1, ..., x_m$ and $y_1, ..., y_{n+1}$, plus the gap penalty δ

  which we can write as

  $$A[m][n] = \min \begin{cases} A[m-1][n-1] + \alpha(x_m, y_n) \\ A[m-1][n] + \delta \\ A[m][n-1] + \delta) \end{cases}$$

  and has runtime $O(n^2)$.

4.3 Optimal parenthesization

- Input: Dimensions of n rectangular matrices, where A_i is $d_{i-1} \times d_i$.
- Output: The minimum-cost parenthesization to multiply $A_1 ... A_n$, where $(p \times q) \times (q \times r)$ costs pqr.
- **DP**: For a subproblem with n matrices, try all $n-1$ possible places to multiple the left half and right half. So

  $$OPT_{i,j} = \min_{k=0,...,j-i-1} OPT_{i,k} + OPT_{k+1,j}$$

  This takes $O(n^3)$ time. You also have to be careful with the order in which you fill in your DP table so that you never rely on unsolved subproblems (in this case, solve smaller subproblems first).

5 Graph algorithms

Graphs are fun. List of things that are things:

- Directed vs. undirected
- Weighted vs. unweighted
- in-deg, out-deg, deg
- Adjacency matrix vs. adjacency list
- Breadth-first search
- Depth-first search
- Search trees

5.1 Unweighted single source shortest paths

Breadth-first search.

5.2 2-coloring

Breadth-first search. Alternate colors for each layer.

5.3 Undirected connected components

Breadth-first search, propagating labels. Can also use DFS.

5.4 Topological sort

- Input: DAG
- Output: Vertices sorted such that each vertex appears after vertices reachable from it
- **Visual**: Keep popping sinks and adding to front of result
- **DFS**: Keep track of "finishing time" of each vertex (order in which the vertices run out of children). Order by descending finishing time. $O(n)$.

5.5 Strongly connected components

- Input: G: DAG
- Output: Strongly connected components of G
- **Kosaraju's**: First, topologically sort G with edges reversed. Then, the largest finishing times correspond to the smallest finishing times in G^{SCC}. This is the "magical order" for DFSing on G to get G^{SCC}. Proof: God no. $O(n)$.

5.6 Minimum spanning tree

- Input: Undirected graph with edge weights.
- Output: Minimum spanning tree of that graph.
- **Prim's**: Start at an arbitrary node and greedily expand the tree with the cheapest edge. $O(m \log n)$ with a priority queue.
- **Kruskal's**: Greedily pick the cheapest edge that does not result in a cycle. $O(m \log n)$ with union-find (see the Grand Grimoire).

5.7 Single-source shortest paths

- Input: Directed graph with edge weights, vertex v
- Output: Shortest paths from v to all other vertices
- **DAG DP**: If the graph is a DAG, you can DP. Use topological sort to get subproblem order. $O(n+m)$.
- **Dijkstra's**: If the graph has no negative edges, you can use Dijkstra's. It's just BFS, but your "breadth" is the weight of the path so far, and you use a priority queue. $O(m \log n)$.
- **Bellman-Ford**: Relax all edges n times. $O(mn)$. Can do negative edges and detect negative cycles.

5.8 All-pairs shortest paths

- Input: Directed graph with edge weights
- Output: Shortest paths between all pairs of vertices
- **Floyd-Warshall**: Try to use every possible node as an intermediary between every possible pair of nodes. $O(n^3)$

6 Intractible problems

6.1 Decision problems

A yes-no version of an optimization problem. For example, Knapsack-Optimization is "what is the maximum value we can put into the knapsack?". Knapsack-Decision is "can we put value $\geq k$ into the knapsack?". If you can solve the decision problem in T time, you can use it to solve the optimization problem in $T \log(b)$ time.

6.2 Complexity classes

- **P**: Polynomial-time solvable problems
- **NP**: Polynomial-time verifiable problems, given a polynomial-size certificate
- **NP-complete**: Problems that are as hard as any other problem in NP
  - To show, show in NP, then reduce to a known NP-complete problem

6.3 Reduction

6.3.1 Polynomial time reduction

$A \leq_P B$ ("A reduces to B") means that you can solve A using a solution to B by converting instances and solutions in polynomial time.

6.3.2 Turing reduction

$A \leq_P^T B$ ("A turing reduces to B") means that you can solve A using a solution to B as a subroutine.

6.4 NP-complete problems

6.4.1 SAT

- Input: Boolean expression
- Output: Whether it is satisfiable by any assignment of truth values
- Proven to be NP-complete just from the definition of NP-complete, by a real smart guy

6.4.2 3-CNF-SAT

- Input: n clauses or'ed together, each clause having 3 literals anded together

- Output: Whether it is satisfiable by any assignment of truth values
- Proven to be NP-complete by reduction from SAT (3-CNF-SAT can be used to solve SAT)

6.4.3 Independent set

- Input: Graph, and an integer k
- Output: Whether or not there is a subset of at least k vertices that don't share an edge
- Proven to be NP-complete by reduction from 3-CNF-SAT (Independent Set can be used to solve 3-CNF-SAT)

6.4.4 Vertex cover

- Input: Graph, and an integer k
- Output: Whether or not there is a subset of at most k vertices incident to every edge
- Independent Set reduces to it

6.4.5 Clique

- Input: Graph, and an integer k
- Output: Whether or not there is a complete subgraph of k vertices
- Independent Set reduces to it

6.4.6 Hamiltonian Cycle

6.4.7 Travelling Salesman

6.4.8 Subset Sum

6.4.9 0-1 Knapsack

6.5 Unsolvable problems

6.5.1 Halting problem

- Input: Program, and an input for it.
- Output: true iff the program halts on that input, false otherwise
- Is impossible in general. Proof: If it were possible, then you could write a program that halts iff it doesn't halt.

6.5.2 Halting-all

- Input: Program
- Output: true iff the program halts on every possible input
- Is impossible. Proof: If you can solve this, you can solve Halting (reduction)

6.5.3 Post correspondence problem

- Input: Two equal-sized lists of words.
- Output: true iff there is some sequence of indices such that concatenating the words in the first list at those indices has the same result as concatenating the words in the second list at those indices.
- Is impossible. Proof: By authority.