<u>What I know *after* taking CS 31</u>

The document summarizes a subset of things which you should be knowing after taking CS 31.

1. **Worst Case Running Time.**
   - Computational problem Π has instances/inputs I; each input I has solution/output S.
   - An algorithm $\mathcal{A}$ for Π takes $I \in \Pi$ and returns its solution S.
   - Each instance $I \in \Pi$ has a notion of size $|I|$.
     Often, this is the number of bits required to describe I.
   - The running time of algorithm $\mathcal{A}$ on I is denoted as $T_{\mathcal{A}}(I)$.
   - The ***worst case running time*** of $\mathcal{A}$ as a function of size is defined to be

     $$T_{\mathcal{A}}(n) := \max_{I \in \Pi : |I| \leq n} T_{\mathcal{A}}(I)$$

2. **The Big-Oh Notation.**
   - Useful notation to tell the "big picture" without worrying about annoying details.
   - $g(n) \in O(f(n))$ if $\exists\ a, b > 0$ such that for all $n \geq b$, $g(n) \leq a \cdot f(n)$.
   - $g(n) \in \Omega(f(n))$ if $\exists\ a, b > 0$ such that for all $n \geq b$, $g(n) \geq a \cdot g(n)$.
   - $g(n) \in \Theta(f(n))$ if $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$.
   - Often the $\in$ is replaced by $=$; so we would say $T(n) = O(n^2)$ to imply $T(n) \in O(n^2)$.

3. **Divide and Conquer.**
   - Break a problem into two, recursively solve, combine solutions.
   - Often works for speeding up algorithms for which a not-so-bad naive solutions exist.
   - Problems seen: MERGE SORT, COUNTING INVERSIONS, MAXIMUM RANGE SUM, POLYNOMIAL MULTIPLICATION, CLOSEST PAIR OF POINTS, many others in the Psets.
   - Analysis Tool : Master Theorem.

4. **Dynamic Programming.**
   - Smart Recursion / Recursion with Memory.
   - Think of optimum solution; see if solution can be built by combining solutions of smaller subproblems.
   - Smaller subproblems should be "succinctly representable". The value should be defined by a "function" on not too many parameters. Function should have a recurrence relation.
   - 7-step way:
     - Definition of the function.
     - Base Cases.
     - Recurrence.

- Proof of Recurrence.
    - Pseudocode.
    - Recovery Pseudocode
    - Runtime and space.
- Problems Seen: SUBSET SUM, KNAPSACK, LONGEST COMMON SUBSEQUENCE, many others in the Psets.

5. **Randomized Algorithms.**

    - Algorithms which can toss independent coins.
    - Monte Carlo Algorithms : can be wrong with some teeny probability
    - Las Vegas Algorithms : can have random running times.
    - Problems Seen : CHECKING MATRIX MULTIPLICATION (Monte Carlo), QUICK SORT (Las Vegas)
    - Hashing. Universal Hash Functions. Using randomization to have low expected query times.
    - Perfect Hashing. Using randomization to pick collision-free hash functions fast.
    - Estimating Frequencies in Data Stream using Hashing : most modern thing seen in course!

6. **Depth First Search.**

    - Revisiting an old algorithm.
    - Lots of power in the first and last's returned.
    - Applications: CONNECTIVITY, CYCLE?, TOPOLOGICAL ORDER of DAGs, STRONGLY CONNECTED COMPONENTS, 2SAT. All *linear* time!
    - You should know how to implement this in any programming language.

7. **Breadth First Search.**

    - Shortest hop-length walks in $O(n+m)$ time.
    - Queue implementation of visited vertices.
    - Useful for checking BIPARTITE?.
    - You should know how to implement this in any programming language.

8. **Dijkstra.**

    - Clever generalization of BFS which works when graphs have positive cost edges.
    - Doesn't add everything in queue once distance label updated. Only the "smallest" such vertex added.
    - Runs in $O(m + n \log n)$ time using Fibonacci heaps. Or in $O(m \log n)$ time using usual heaps.
    - Can be used to find shortest length cycles (this was done in problem set).

- You should know how to implement this in any programming language.

9. **Bellman-Ford.**
   - In graphs with possibly negative cost edges, this algorithm either detects negative cost cycles, or figures out shortest paths.
   - Finds shortest cost walks whose lengths are bounded. In case of no negative cost cycles, shortest walks are shortest paths.
   - Dynamic program. Runs in $O(mn)$ time.
   - No one knows how to make it run faster.
   - All pairs shortest paths can be found in $O(n^3)$ time (this was done in a problem set.)
   - You should know how to implement this in any programming language.

10. **Flows and Cuts.**
    - Max Flow : send as much "stuff" as possible from source to sink with no excesses in any internal node.
    - Min Cut : minimum capacity edges whose removal disconnects source and sink.
    - Maximum s,t-flow ***equals*** Minimum s,t-cut. Deepest fact uncovered in the course.
    - Residual Networks! A major idea.
    - Ford-Fulkerson Algorithm: Keep augmenting flow in the residual network.
    - Can make it faster* : (a) augment flow on max-capacity path, (b) augment flow on shortest length path.
    - Plenty of applications : BIPARITITE MATCHING, LOAD BALANCING, PROJECT SELECTION,. Minimum s,t-cut can find a "cheapest subset" among all subsets fast – very, very useful tool!

11. **Reductions and Hardness*.**
    - Decision Problems: Π, each instance has solution YES or NO.
    - Polytime Algorithm : running time less than some fixed polynomial of size of the instance.
    - $\Pi_A \preceq_{\text{poly}} \Pi_B$ if there is an efficient algorithm taking YES instances of Π_A to YES instances of Π_B, and vice-versa.
    - $\Pi_A \preceq_{\text{poly}} \Pi_B$: Π_A is "easier/no harder" than Π_B. If Π_B has a polytime algorithm, so does Π_A; if Π_A has no polytime algorithm, neither does Π_B.
    - **P**: class of all polynomial time ***solvable*** problems.
    - **NP**: class of all polynomial time ***verifiable*** problems.
    - **NP**-hard: Π is **NP**-hard if $\Pi' \preceq_{\text{poly}} \Pi$ for any $\Pi' \in$ **NP**.
    - SAT is an **NP**-hard problem.