**Module 4 Notes – Greedy Algorithms**

* A **greedy algorithm** works by making the decision that seems the “best” or “most optimal” at any moment, it never reconsiders this decision, whatever situation may arise later. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Generally used for optimization problems.
  + these types of algorithms don’t always yield optimal solutions
  + At every step, the procedure chooses the best candidate, without worry of the future and it never changes its mind. Once a candidate is included in the solution, it is there for good; once a candidate is excluded, it is never considered again
* Elements of a **greedy strategy:**
  + **Greedy-choice property –** A global optimal solution can be arrived at by making locally optimal (greedy) choices
  + **Optimal substructure –** an optimal solution to the problem contains within it, optimal solutions to subproblems
* These algorithms also typically consist of:
  + a set of **candidate solutions**
  + a function that checks if the candidates are feasible
  + **selection function** indicating at a given time which is the most **promising candidate**
  + **objective function** giving the value of a solution; (this is the function we are trying to optimize)
* **Greedy vs Dynamic Programming**
  + Both are methods for solving optimization problems
  + **greedy** **algos** are usually more efficient than **DP** solutions
  + However, often you need to use **DP** since the optimal solution can’t be guaranteed by a **greedy algo**
  + **DP** provides efficient solutions for some problems for which a brute force approach would be very slow
* **GM (greedy method)** is used for optimization problem, ones in which a maximum or minimum answer is required (i.e. max number of tasks done in a limited amount of time, minimum number of coins used to make a certain amount of change)
  + some examples of **GMs** are:
    - Knapsack
    - Coin Change
    - Data Compression
      * Huffman coding
    - Scheduling
      * activity selection
      * task scheduling/deadlines
    - Graph Algos
      * breath first search
      * Djikstra’s
      * Minimum spanning trees
* **0/1 Knapsack**
  + Given a knapsack with weight W > 0
  + A set (S) of n items with weights wi > 0 and benefits bi > 0 for i = 1 … n
  + S = { (item1, w1, b1), (item2, w2, b2), … { (itemn, wn, bn),
  + Find a subset of the items which does not exceed the weight (W) of the knapsack and maximized benefits
* **Activity Selection**
  + Sorted by finish time
  + slide #16 of “Scheduling” lecture
* **Huffman Code**
  + These are the problem where you have frequencies (i.e. the letter A’s frequency is .35, B is .12, etc.)
  + The twol symbols/elements with the smallest frequencies must be at the bottom of the optimal tree, as children of the lowest internal node
  + This is a good sign that we have to use a bottom-up manner to build optimal code
  + This approach is based on a **greedy** approach
  + See lecture titles “Huffman Codes”