10.7 What makes something NP?

- have looked at a lot of problems in class P
  - and handful in NP
  - => problems that are **solvable** in polynomial time by a non-deterministic algorithm
- can describe sorting as follows
  - 1) nondeterministically output the elements of a list
  - 2) check to see if $s_i < s_{i+1}$ for $i = [0, N-1)$
  - describes a two step nondeterministic process
    - 1st takes N steps (to output N elements)
    - 2nd takes N-1 comparisons
  - fits definition of an NP problem => sorting is in NP as well as P
  - we can do this with any algorithm
    - all algorithms in P are also in NP
  - for problems just in NP,
    - there is no known deterministic polynomial time algorithm to solve it
    - P subset of NP
    - but there are problems in NP not in P

- heart of difference b/t P and NP
  - large number of combinations that we must necessarily exempting for NP only problems
  - slightly more complex
    - e.g., 30 numbers (or 30 distinct cities): 30! combinations
      - only 1 is sorted list or shortest path
      - sorting algorithms can sort in as little as 150 comparisons
    - best we can do for shortest path is:
      - to examine all possible paths and see which is shortest
      - do not have an algorithm that, with a few ops can successfully eliminate a significant number of combinations from consideration...
        - instead, must look at all of them.
        - if we examine 1 billion paths per second
          - 30 cities and 30! combinations would take 840 billion centuries to check!

- we do have “heuristic” algorithms that approximate the optimal answer
  - no way of knowing what the optimal value is
  - optimal answer not guaranteed

- What puts a problem in class NP (only)?
  - have an extremely large number of possibilities for the optimal answer
  - *and* -> we do not have an efficient deterministic algorithm for sifting through them

- P = NP?
  - might seem ridiculous to question if the problems in P are the same class as problems in NP
    - example of sorting shows that P is subset of NP
previous example (shortest path) only shows that we do not yet have a deterministic polynomial algorithm that solves traveling salesman

- and other problems in NP
- does not mean there is no such algorithm
- **still an open question!!**

**10.8 Testing Possible Solutions**
- description of class NP state that these problems had a solution that included
  - 1) a non-deterministic first step, and
  - 2) a deterministic second step that check the solution in poly time
- check proposed solutions for job scheduling and graph coloring problems in poly time

○ **Job Scheduling**
  - recall that job scheduling problems gives a set of jobs that need to be done
    - each job has
      - a time it takes to complete
      - a deadline
      - and a penalty
    - jobs are specified as a 4-tuple: ( n, t, d, p )
      - n: job number
      - t: runtime
      - d: deadline
      - p: penalty
    - e.g., { (1,3,5,2), (2,5,7,4), (3,1,5,3), (4,6,9,1), (5,2,7,4)}
  - decision problem specifies some value P
  - wants to know if there is an ordering of jobs that can be done with penalty <= P
  - optimization problem:
    - smallest penalty for any ordering of jobs
  - will consider decision problem:.
    - solving decision problem with decreasing P until no more => solves optimization problem

  - say total penalty for all jobs is X
    - ask -> is there an order with penalty P=X?
    - if yes:
      - try again with P = X-1
    - keep decreasing until a value of Y gives us a NO for the decision problem
      - optimal answer is then Y + 1

○ following algorithm will test 1 potential solution for the decision problem

```plaintext
isPenaltyLess( list, N, limit )
T = 0
F = 0
J = 1
```
while (J <= N and P <= limit)
    T = T + list[J].t
    if list[J].d < T
        P = P + list[J].p
        J = J + 1

if P <= limit
    return yes
else
    return no

- requirement of NP
  - must be able to check proposed solution in poly time
  - analyze time complexity of above algorithm:
    - O(N) complexity => clearly poly time
    - (length of list)

- 10.8.2 Graph Coloring
  - problem: how to assign colors (represented as integers) to nodes of a graph such that no two adjacent nodes (connected by a single edge) have the same color
  - decision version of problem:
    - if graph can be colored with C colors or less
  - optimization problem:
    - determine smallest number of colors

- non-deterministic step:
  - will produce a proposed solution
  - list of nodes with assigned colors
  - responsible for deciding how many colors to use
  - (not something we need to check)

- Algorithm to check if proposed colors are valid way to color graph
  - algorithm uses adjacency list to hold graph
    - graph[j] is jth node of graph
    - graph[j].edgeCount is number of edges
    - graph[j].edge is an array with nodes adjacent to node j

`isValidColoring( graph, N, colors )`
// graph: the adjacency graph
// N: number of nodes in graph
// colors: array of colors assigned to each node

for j = 1 to N
    for k = 1 to graph[j].edgeCount
        if colors[j] == colors[ graph[j].edge[k] ]
            return no
algorithm checks the colors in graph provided
  - goes through each node
  - checks adjacent nodes for same color
    - if same, returns no
    - $O(N^2)$ at worse, clearly poly time