1. (0.75 points) For each of the following functions, determine whether it is $\mathcal{O}(n)$, $\Omega(n)$, or both. You do not need to provide proof or justification. Make sure you clearly indicate, for each one, whether you believe it is $\mathcal{O}(n)$, $\Omega(n)$, or both. If the graders cannot determine your answer, you will not get credit for it.
   Note that we are *not* asking you to provide the "best" $\mathcal{O}$-notation/Ω-notation; merely to describe it with one or both of the choices provided.

   $a(n) = n\sqrt{n}$ ☐ $\mathcal{O}(n)$ ☐ $\Omega(n)$
   $b(n) = \sqrt{n}$ ☐ $\mathcal{O}(n)$ ☐ $\Omega(n)$
   $c(n) = 2^{\log n}$ ☐ $\mathcal{O}(n)$ ☐ $\Omega(n)$

   $a(n)$ is $\Omega(n)$ but not $\mathcal{O}(n)$. $b(n)$ is $\mathcal{O}(n)$ but not $\Omega(n)$. $c(n)$ is both.

2. (1.25 points) Rank the following functions in order from smallest asymptotic running time to largest. Additionally, identify a pair of functions x, y where $x(n) = \Theta(y(n))$. You do not need to show your work.

   Your response to the first part should be exactly six characters. For example, if you believe the functions are currently listed in asymptotic order, your answer should be "abcdef" (without the quotes). For the second part, list exactly two characters; for example, if you think $a(n)$ and $b(n)$ have the property that $a(n)$ is $\Theta(b(n))$, then write "ab" or "ba" for the second part.

   (a) $a(n) = 2^{\log n}$
   (b) $b(n) = n^2 \log n$
   (c) $c(n) = n^2$
   (d) $d(n) = 10^{-100}\sqrt{n}$
   (e) $e(n) = n(\log n)^2$
   (f) $f(n) = 4^{\log n}$

   Note that $a(n) = n$ and $f(n) = n^2$. The order is *daecfb*, although c and f can be in either order, as they are Θ of one another.

3. (2 points) Suppose I have a Queue that currently contains n elements. I wish to remove from the queue the *most recently* added element to it. The only Queue functions you may call are enqueue, dequeue, front, size, and isEmpty; you may not access the private member data of the queue directly. State also the running time of your function in terms of n, the number of elements in the queue.

```
void removeMostRecentAdded (Queue & q)
{
```

   The most straight-forward way to solve this problem is to perform the following a number of times equal to one less than the number of elements in the queue:

   - Store the front element.
   - Dequeue the front element.
   - Enqueue what had been the front element

   After doing that, the front of the queue is the most recently added. We then dequeue that.

   This full process takes $\mathcal{O}(n)$.

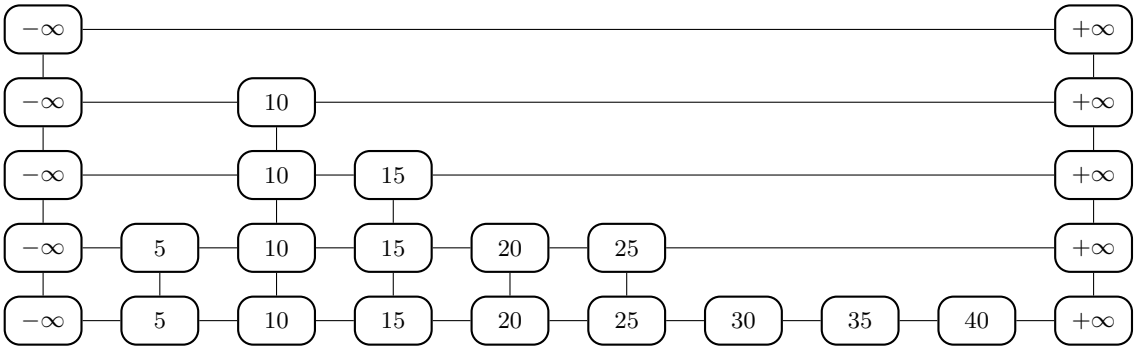
   Of course, if the queue has no elements, we can't do that, but "most recently added" isn't meaningful either in that case.

4. (1 point) Consider the skip list below. When this was created, it was empty (other than the $-\infty$ and $+\infty$ nodes). The elements were inserted, in some order, using the insertion operation that was discussed in lecture.

   (a) How many times did the coin land on heads? 8

   (b) How many total coin flips happened? 16

   Everything in this problem is as was defined in lecture. Course staff will not define terms from lecture for you for this problem. If you need to make an assumption, use the rule from the cover.



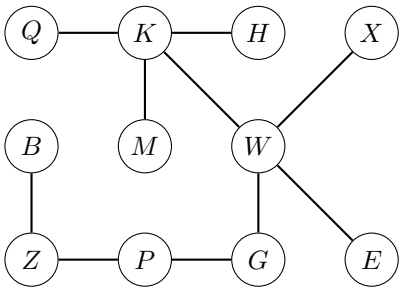
5. (3 points) Suppose you want to buy $n \geq 18$ cans of soda. The store only sells these in packs of five cans or six cans, and you must buy full packs of each (you cannot break up a five-pack and get two cans). Your goal is to get **exactly** n cans of soda, assuming there are as many five and six-packs available as you need. Use the variable numSmall to represent the number of 5-packs and the variable numLarge for the number of 6-packs.

   Finish the recursive function below to complete the ordering and return the counts by reference parameters. You may assume for this problem that there will be no overflow or underflow at any point in the problem and that stack space is not a concern. The code has been started for you and is part of a correct solution.

```
void buySoda (unsigned n, unsigned & numSmall, unsigned & numLarge)
{
  if( 20 == n ){
    numSmall = 4;   numLarge = 0;
  }else if (21 == n ){
    numSmall = 3; numLarge = 1;
  }else if (22 == n ){
      numSmall = 2; numLarge = 2;
  }
```

   Finish writing base cases until there are five. The else condition is to have a recursive call for $n - 5$ without the other two parameters, then increment numSmall, then return.

6. (2 points) Give a depth-first traversal of the following graph, starting at vertex G.



There are many correct answers for this.

7. (4 points) Consider the following definition for a node in a linked list:

```
struct Node
{
  Node(unsigned v, unsigned w, Node * n) : value(v), weight (w), next(n) {}
  unsigned value;
  unsigned weight;
  Node * next;
};
```

I want to select some of the nodes in the linked list. I have some value `maxWeight`; I need the sum of the "weight" of all nodes I select to be at most this value. I want to have the maximum possible total of the "value" field of the nodes chosen, subject to that constraint.

Complete the following function to achieve that. You need only return the total value achievable; you **do not** need to find the list itself. You may assume that the initial call gives a link to the front of the linked list and that no overflow or underflow will occur at any point in the run of a program.

This problem is substantially easier if you use recursion. For many students, this will be the hardest question on the test. You may wish to consider answering it last.

You will receive a score of zero on this problem if the body of your function is more than 15 statements long, uses any global variables, or contains any occurrence of the `C++` keywords **while**, **for**, **goto**, **static** or their equivalents (in other languages or in pseudo-code).

```
unsigned knapsack(Node * front, unsigned maxWeight)
{
```

If the list is empty, we return zero.

If the front element weighs more than maxWeight, we return knapSack of the rest of the list.

Otherwise, we compute two values:

- Knapsack of the rest of the list without changing maxWeight.
- Knapsack of the rest of the list, with maxWeight decreased by front's weight. We add front's value to this list.

Whichever of those two values is larger, we return.