1 You *Are* the Brute Squad[1] [SOME marks]

A "brute force" approach tries every possible solution to a problem to see if one works. To judge whether to usebrute force, we often need to know how many solutions there are.

Answer each of the following questions about counting solutions to problems. To receive partial credit, you *must* indicate your reasoning. You need not calculate numerical answers. Leave solutions as formulas that use any of: addition, subtraction, multiplication, division, factorial, and "choose" notation.

1. A "parity-checked" byte is 8 bits followed by an extra "parity" bit with the constraint that the parity bit is set so that the number of 1's in the 9 bits is even. The probability that a sequence of 9 bits is a parity-checked byte is the ratio of **the number of valid 9-bit parity-checked bytes** and **the number of 9-bit sequences**. How many of each are there?

2. A user's password is known to contain a `q`; two or fewer `t`s; and six digits 0–9, with no repeated digits. These can appear in any order. How many possible passwords are there?

3. To list all the anagrams of a word (rearrangements of the letters in the word that form another word), we can generate each rearrangement and check if that rearrangement is in a list of known words.

   For the word `baobabs`, how many *different* rearrangements are there?

4. Having found your soulmate, you know only that the person's e-mail address is of the form *`First.M.Last.###`*, where *First* is one of 5 possible nicknames, *M* is an uppercase letter, *Last* is one of 7 possible alternate spellings of a family name, and *###* is a sequence of three "non-decreasing" digits (e.g., it could be 011, 888, or 157, but not 527 or 766). How many e-mail addresses will you need to spam to reach your soulmate?

[1] Feel free to ignore the problems' names!

2 Little Big Problem [SOME marks]

Consider three alternate algorithms for checking if a binary tree of integer keys (with no duplicate keys) is a binary search tree. Each algorithm succeeds if all of its checks pass but fails otherwise.

A. For each node n: check against all nodes in n's left subtree to see that their keys are less than n's key. Check against all nodes in n's right subtree to see that their keys are greater than n's key. Recursively check the node's children.

B. For each node n: If n has a left child, check that the left child's key is less than n's key. If n has a right child, check that the right child's key is greater than n's key. Recursively check n's children.

C. This one we define explicitly with pseudocode:

```
// Returns true if node is the root of a (possibly empty) binary search
// tree in which all keys are in the range [lo, hi], i.e., greater than
// or equal to lo and less than or equal to hi.
function isBstHelper(node, lo, hi):
  if isEmptyTree(node):
    return true

  // These two checks are the only ones that count in the function
  if node.key >= lo and node.key <= hi:
    leftIsBst  = isBstHelper(node.left,  lo, node.key)
    rightIsBst = isBstHelper(node.right, node.key, hi)
    return leftIsBst and rightIsBst

  return false

function isBst(node):
  // assume -infinity < i < infinity for all integers i.
  return isBstHelper(node, -infinity, +infinity)
```

Now answer the following questions:

1. For the Algorithm C, we can give a precondition on the relationship between `lo`, `hi`, and the ancestors of `node` for each call to `isBstHelper`. State the precondition in clear and simple terms and then prove that it holds in this code (i.e., the precondition holds on entry to `isBstHelper` for every call to `isBstHelper` created by this code). (Any proofs related to the third algorithm below can assume this precondition as needed.)

2. For each algorithm, for a valid binary search tree with n nodes, what shape of tree necessitates the greatest number of checks, and how many checks does the algorithm perform (as a function of n)? Give your answer exactly and as a clean asymptotic big-Θ bound. (Hey, it's a counting problem!)

3. One of these algorithms is incorrect. Identify which one is incorrect and show that it is incorrect by providing and explaining a small example where it gets an incorrect result. The algorithm can be incorrect by saying a non-BST is a BST ("unsound") or by saying a BST is a non-BST ("incomplete"). Your first example shows that the algorithm is either unsound or incomplete. Either give an additional example indicating the algorithm has the second (bad) property or briefly prove that it does not.

4. One of the correct algorithms is asymptotically slower than the other. Prove that, in the worst case, the number of checks performed by the slower algorithm is *not* big-O of the number of checks performed by the faster algorithm using the exact formulae you determined above.

5. Prove that each of the correct algorithms is, in fact, correct. (It may help to start by proving both alogirthms eventually call themselves on each node in the tree, if they don't "fail" a check first.)

6. What data structure could we use to (efficiently and simply) remove the recursion from the third algorithm but still perform the same checks *in the same order*? A stack, a queue, a priority queue, or a hash table? Briefly justify your answer.

7. Convert the third algorithm to a divide-and-conquer task-based parallel algorithm with a reasonable sequential cutoff (if possible), using OpenMP and fully realized C++ rather than pseudocode. You may assume that sequential versions of the functions `isBst` and `isBstHelper`—named `isBstSeql` and `isBstHelperSeql`—are available if you need them. (Do **NOT** assume that the program is already in a parallel region on entry to the `isBst` function.)

3 Polar Graph Explorers [SOME marks]

First, some definitions: A "path" in a graph is a sequence of nodes where each node in the sequence has an edge leading to the node after it. The "length of a path" is *one less than* the number of nodes in the sequence (i.e., the number of edges in the sequence). A "cycle" is a path that starts and ends with the same node. (For our purposes, the length of a cycle is just its length as a path.)

Now, imagine a directed graph with n nodes and no "self-loops"—no node with an edge to itself.

1. What's the largest number of cycles of length 1 that there could be in the graph?

2. What's the largest number of paths of length 1 that are *not* cycles that there could be in the graph?

3. What's the largest number of paths of length 1 that there could be in the graph?

4. What's the largest number of cycles of length 2 that there could be in the graph?

5. What's the largest number of paths of length 2 that are *not* cycles that there could be in the graph?

6. What's the largest number of paths of length 2 that there could be in the graph?

4 Short But…Just Short [SOME marks]

1. Which of the following is the **best** choice of sort for large sorting problems where *reliably* quick performance is critical?

   (a) Insertion Sort
   (b) Quick Sort
   (c) Selection Sort
   (d) Merge Sort

2. Which of the following is the **worst** choice of sort for large sorting problems where *reliably* quick performance is critical?

   (a) Insertion Sort
   (b) Quick Sort
   (c) Selection Sort
   (d) Merge Sort

3. Which of these traversals does a Breadth First Search of a binary tree correspond to?

   (a) Pre-order
   (b) In-order
   (c) Post-order
   (d) None of these

4. We distinguished between the concepts of "parallelism" and "concurrency". Label each of the following with the term that best describes the ccntext of the scenario:

   (a) Taking the first step to improve performance of quicksort on a multicore machine by indicating that its two recursive calls can proceed in parallel.

   Circle one: **PARALLELISM** **CONCURRENCY**

   (b) Rewriting a hash table data structure so that multiple threads can share an instance of the data structure (i.e., independently call its methods) without causing data races.

   Circle one: **PARALLELISM** **CONCURRENCY**

5. Briefly justify the divide-and-conquer, task-based approach to implementing parallel map (including use of a sequential cutoff) vs. detecting the number of concurrent threads a computer can run without context-switching and efficiently forking that many threads, each responsible for an equal share of the input.

6. According to Amdahl's Law, what is (the limit on) the greatest speedup possible using parallelism in an algorithm in which half of the work is inherently sequential?

7. Imagine that a sequential algorithm runs in $O(n)$ time. No way can be found to directly parallelize the algorithm. However, an alternate algorithm runs in $O(n \lg n)$ time and can be parallelized (without changing work) with span $O((\lg n)^3)$. Asymptotically, in terms of n, how many processors do we need before it's likely to be worth switching to the latter algorithm? Briefly justify your answer.

   $p \in \Omega($_____$)$

8. Why do we often discuss the machine architecture details of a computer when considering use of B+-trees and not when considering use of BSTs?

   (a) BSTs automatically adjust to these architectural details.

   (b) B+-trees are typically less efficient than BSTs and therefore require careful tuning to achieve reasonable performance.

   (c) B+-trees are designed to optimize the "constants" for performance on particular machine configurations.

   (d) BSTs store both keys and values at every node in the tree, while B+-trees store only keys at internal nodes and both keys and values at the leaves.

9. Can the longest path in a DAG be longer than $|V|$ divided by the maximum in-degree of a single node? If so, give an example. If not, prove that it cannot happen.

5 Heigh-Ho, Heigh-Ho [SOME marks]

1. Which of these two definitions best describes "work", and why is the worse choice inaccurate (but often useful)? (1) The amount of time it would take to execute the parallel algorithm on a device whose hardware can run only one thread at a time. (2) The amount of time the sequential version of the algorithm takes.

2. Which of these two definitions best describes "span", and why is the worse choice inaccurate (but often useful)? (1) The number of nodes in the longest path through the DAG representing the parallel computation. (2) The limit of the amount of time the computation takes as we measure its runtime on machines with increasing numbers of processors.

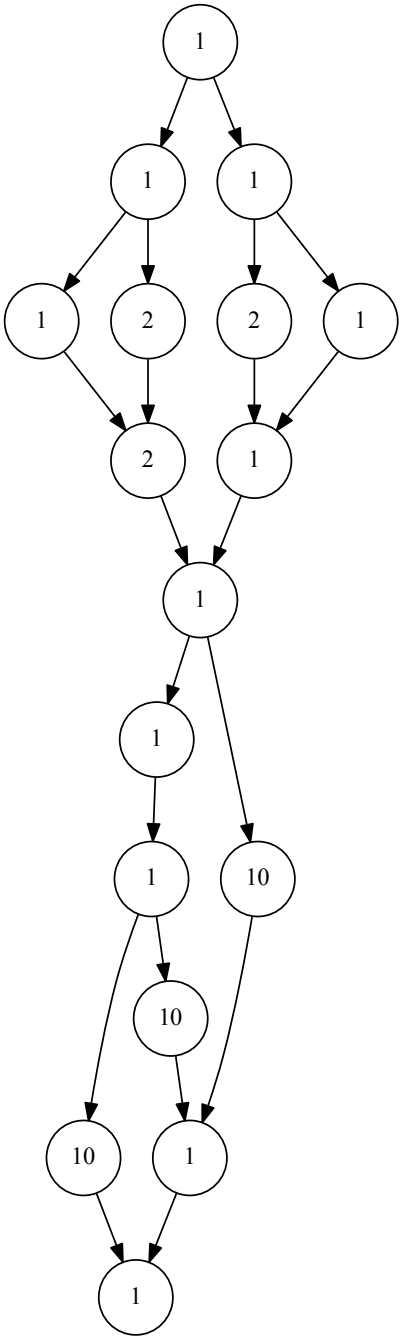
6 Analyze This [SOME marks]

Give a big-Θ time bound on the following code in terms of n:

```
int count = 0;
for (int i = 0; i < n*n; i++) {
  for (int j = i; j >= 1; j /= 2) {
    count++;
  }
  for (int k = n; k > count; k--) {
    count++;
  }
  count++;
}
int p = n*count;
for (int m = 0; m < p; m++) {
  count++;
}
```

7 Mumblety-DAG . . . er . . . [SOME marks]

Find the span of the following DAG describing a parallel computation where nodes are labelled with their runtimes. Explain your process!



8 Parallel Madlib Jumble [SOME marks]

Consider the following problem: Given a sorted array of n URLs, produce an output array of the frequency of each URL. Also produce the frequency of the most frequent URL.

Give an efficient plan to solve the problem by filling in the blanks and reordering the partially specified parallel map, reduce, prefix, and pack (i.e., filter) steps listed below. **Indicate your intended ordering of statements by writing 1^{st} before the first statement, 2^{nd} before the second, etc.**

Note: a pack "starting at" a particular index means the pack puts the first element that passes the filter at that index in the target array, the next element that passes at the following index, and so forth. (Using the constant 0 for the NUMBER gives a standard pack.)

Assume you have available the following pseudocode functions:

```
int isBoundary(pair<URL, int> array[], int len, int i)
  return i == len-1 || array[i].first != array[i+1].first

int getCount(pair<URL, int> p)
  return p.second

pair<URL, int> countLastURL(pair<URL, int> p1, pair<URL, int> p2):
  if (p1.first == p2.first)
    return pair<URL, int>(p1.first, p1.second + p2.second)
  else
    return p2

pair<URL, int> makePair(URL url)
  return pair<URL, int>(url, 1)  // pairs the url with the number 1.
```

You may fill in blanks as follows:

- a SOURCE or TARGET blank with any of: INPUT (input array); TEMP1, TEMP2, or TEMP3 (temporary arrays of the appropriate size); OUTPUT (output array of the appropriate size); or FREQ (the frequency of the most frequent URL).
- a FUNCTION blank with any of the functions provided above or `+`, `*`, `-`, `/`, or `max`.
- a NUMBER blank with any arithmetic expression using constants or the variables n or FREQ.

Parallel map using _____ (FUNCTION) from _____ (SOURCE) to _____ (TARGET).

Parallel map using _____ (FUNCTION) from _____ (SOURCE) to _____ (TARGET).

Parallel pack filtering on _____ (FUNCTION) from _____ (SOURCE) to _____ (TARGET) starting at index _____ (NUMBER).

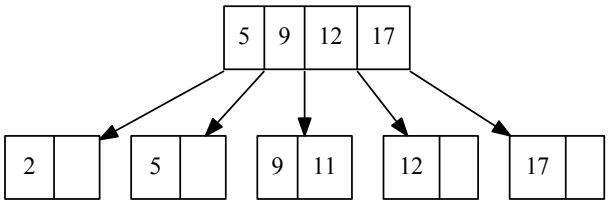
Parallel prefix using _____ (FUNCTION) from _____ (SOURCE) to _____ (TARGET).

Parallel reduce using _____ (FUNCTION) from _____ (SOURCE) to _____ (TARGET).

9 Groundhog Dayta Structures [SOME marks]

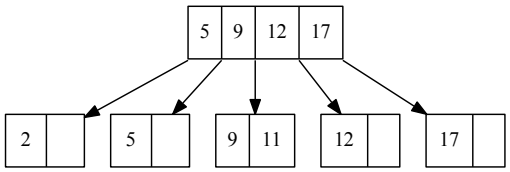
Each of the following problems starts with a picture of a data structure starting in a particular state. For each subproblem, assume the data structure starts in this state, and draw its state after the operation indicated. You may modify the given picture or draw a new one as you see fit, but make your answer clear.

1. **Groundhog B+-Tree**: draw the result of applying the indicated operations to the following B+-tree.

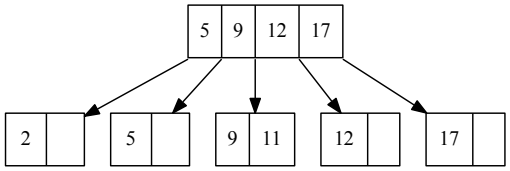


Assume: M=5 and L=2; we never perform "borrowing/lending" on insertion into a B+-tree; we always borrow if possible from immediate siblings on deletion, using the left-hand sibling if both have available items; when splitting a node with an odd number of items in two, we place the extra item into the left node; and a key at an internal node in a B+-tree is the smallest key that appears in the subtree to its right.

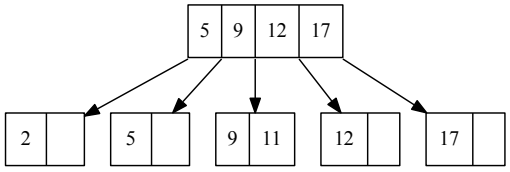
(a) INSERT 8



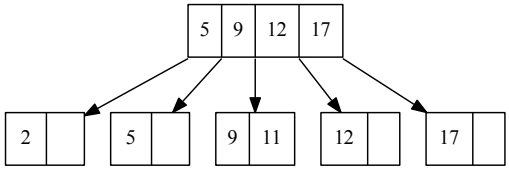
(b) INSERT 10



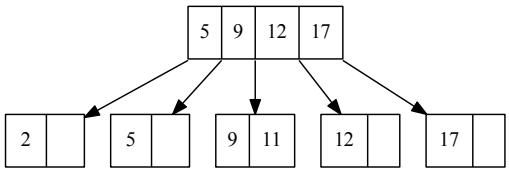
(c) INSERT 20



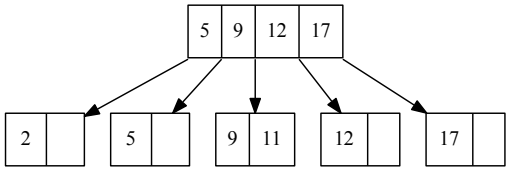
(d) DELETE 2



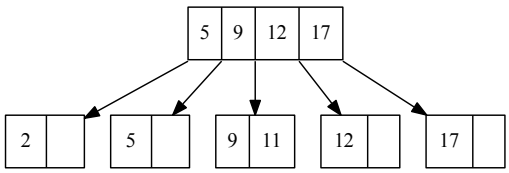
(e) DELETE 11



(f) DELETE 12



(g) In this part, show the **cumulative result** of applying the following **four** operations: DELETE 2, DELETE 5, DELETE 9, DELETE 11.



2. **Groundhog Hash Table**: draw the result of applying the indicated operations to the following hash table.

0	
1	1
2	16
3	3
4	
5	26
6	*T*

Assume: the hash function is simply "mod by table size"; the table uses quadratic probing; a "*T*" indicates a tombstone; and the table doubles in size and rehashes when an insertion fails.

Possibly useful notes: $7*0=0, 7*1=7, 7*2=14, 7*3=21, 7*4=28, 7*5=35, 7*6=42, 7*7=49, 7*8=56$.

(a) INSERT 0

0	
1	1
2	16
3	3
4	
5	26
6	*T*

(b) INSERT 10

0	
1	1
2	16
3	3
4	
5	26
6	*T*

(c) INSERT 20

0	
1	1
2	16
3	3
4	
5	26
6	*T*

(d) INSERT 30

0	
1	1
2	16
3	3
4	
5	26
6	*T*

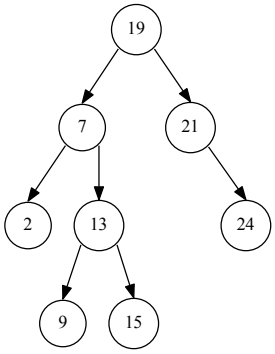
(e) INSERT 50

0	
1	1
2	16
3	3
4	
5	26
6	*T*

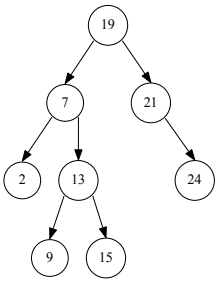
(f) DELETE 3

0	
1	1
2	16
3	3
4	
5	26
6	*T*

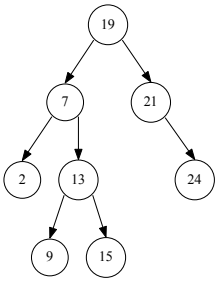
3. **Groundhog AVL Tree**: draw the result of applying the indicated operations to the following AVL tree.



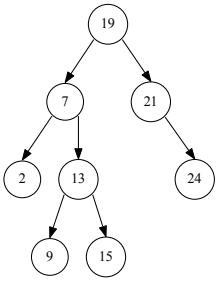
(a) INSERT 1



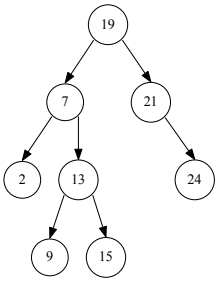
(b) INSERT 8



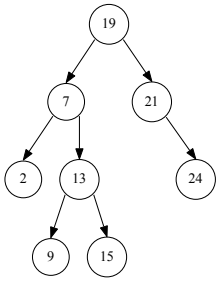
(c) INSERT 10



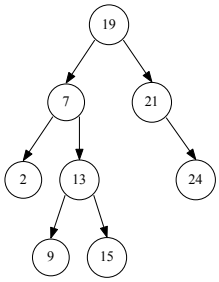
(d) INSERT 14



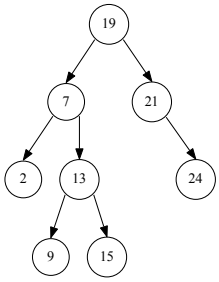
(e) INSERT 17



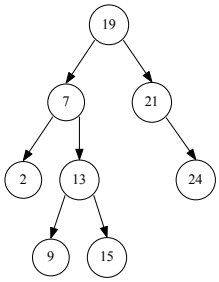
(f) INSERT 20



(g) INSERT 23



(h) INSERT 26

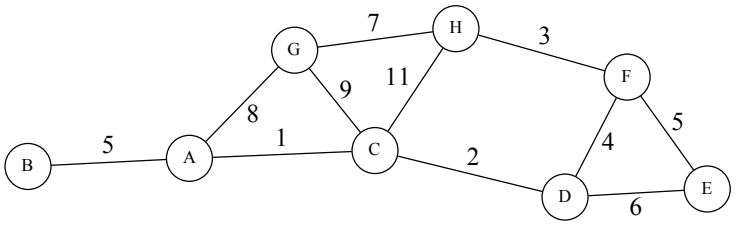


10 PARAheapsLEL [SOME marks]

Write an efficient parallelized (divide-and-conquer, task-based) version of `buildHeap` and analyze its work and span.

11 Computer Graph-ics [SOME marks]

Consider the following graph:

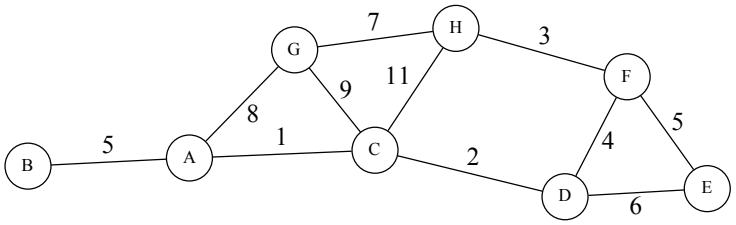


1. Write out an adjacency matrix representation of the graph.

2. Write out an adjacency list representation of the graph.

3. Complete this table showing the result of Dijkstra's shortest path algorithm called on node **A**. Fill in the cost of the shortest path to each node and the *order* in which the algorithm marks its path cost as "known" (i.e., label with 1^{st}, 2^{nd}, 3^{rd}, etc.). Node **A**'s cost has been labeled 0 and its order 1^{st} since it is the start node.

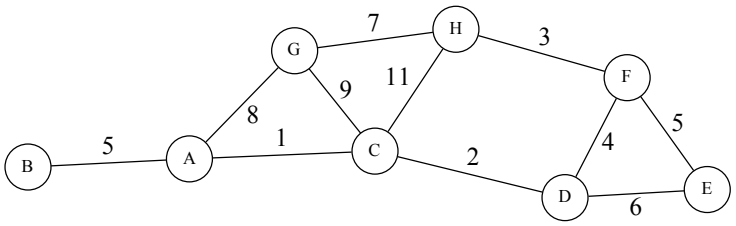
   We repeat the graph for convenience:



node	*cost*	*order*
A	0	1^{st}
B		
C		
D		
E		
F		
G		
H		

4. Complete this table showing the result of Kruskal's algorithm. For each node, indicate (with "Yes" or "No") whether it is in the minimum spanning tree found and the order the algorithm checks it (i.e., 1^{st}, 2^{nd}, 3^{rd}, etc.).

   We repeat the graph for convenience:



edge	*in MST?*	*order*
AB		
AC		
AG		
CD		
CG		
CH		
DE		
DF		
EF		
FH		
GH		

12 Gaming the System [SOME marks]

Penultimate Vivarium—a popular, massively multiplayer online role-playing game—accommodates enormous numbers of simultaneous "characters" who participate in a fictional world. However, these are a small fraction of the total number of characters. The complete pool of characters is stored in archival form on a distributed, parallel database... and is not our concern.

However, the current set of active players must be quickly accessible. Right now, there's a single master server that maps character's unique ID to an IP address where clients can fetch the character's information. Clever architecture makes it so that individual clients rarely need to access this machine, but it nonetheless sustains very high loads during North American daylight hours.

Because of the cyclical load and because deletions were causing unacceptable delays with the existing implementation of the map, newly active characters are inserted into the map as needed, but deletions (actually moves to the archive) of no-longer-active characters are done in a batch every few nights.

Each character's unique ID is a GUID. For our purposes, this is a 128-bit, unique (with sufficiently astronomically high probability that we won't worry about it), effectively random number.

1. (Assume for this and the next part that the server uses a *sequential* solution.) At a high level, propose both a good tree-based data structure and a good non-tree-based data structure to use for the map. Justify why your selections are good choices, and give at least two advantages of each one over the other. (If one strictly dominates the other, give two advantages of the better one and explain why the other style of solution cannot possibly be superior.)

2. How would tombstone-based deletion affect the company's deletion problem?

3. Muse intelligently on why PV should move to a parallel solution and how that would affect your recommendations.

13 This Is The Problem That Never Ends... [SOME marks]

Imagine we construct a linked list with the following code (assuming a `Node` constructor that takes an initial value and `next` pointer as arguments):

```
Node * tail   = new Node(4, NULL);
Node * third  = new Node(3, tail);
Node * second = new Node(2, third);
Node * head   = new Node(1, second);

tail->next = second;
```

1. Sketch what the result looks like. Omit the variables `tail`, `third`, and `second` but not `head`.

2. What would happen if we ran the following code on this linked list? Briefly explain your answer.

```
void printReverse(Node * n) {
  if (n != NULL) {
    printReverse(n->next);
    std::cout << n->data << std::endl;
  }
}
```

3. What would happen if we ran the following code on this linked list? Briefly explain your answer.

```
Node * last(Node * n) {
  if (n != NULL && n->next != NULL)
    return last(n->next);
  else
    return n;
}
```

4. Return to part 1.

14 Subsequence, Superparallel [SOME marks]

Consider the following problem: Given a sequence (array) of n numbers, find the largest "consecutive subsequence sum". A "consecutive subsequence" is a slice of the array starting at some index and running to another index with no breaks. So, $4, -2, 5, -15$ is a consecutive subsequence of $5, 7, 4, -2, 5, -15, 20$, but $4, 5, 20$ is not. The sum of a consecutive subsequence is simply the sum of all the elements in that slice of the array. (Note that the largest consecutive subsequence sum is always at least 0 because of the "empty subsequence".)

Now, consider the following (correct) divide-and-conquer algorithm for solving the problem:

```
int subSumHelper(int arr[], int lo, int hi) {
  if (hi <= lo) return 0;

  int mid = lo + (hi - lo) / 2;
  int lSubSum = subSumHelper(arr, lo, mid);
  int rSubSum = subSumHelper(arr, mid+1, hi);

  int bestLMidSubSum = 0;
  int lMidSubSum = 0;
  for (int i = mid - 1; i >= lo; i--) {
    lMidSubSum += arr[i];
    if (lMidSubSum > bestLMidSubSum)
      bestLMidSubSum = lMidSubSum;
  }

  int bestRMidSubSum = 0;
  int rMidSubSum = 0;
  for (int i = mid; i < hi; i++) {
    rMidSubSum += arr[i];
    if (rMidSubSum > bestRMidSubSum)
      bestRMidSubSum = rMidSubSum;
  }

  return max(bestLMidSubSum + bestRMidSubSum,
             max(lSubSum, rSubSum));
}

int subSum(int arr[], int n) {
  return subSumHelper(arr, 0, n);
}
```

1. Modify the sequential algorithm so that it makes parallel recursive calls with a reasonable sequential cutoff using OpenMP (and fully realized C++ rather than pseudocode). You may assume that the functions `subSumSeql` and `subSumSeqlHelper`—renamed versions of those above—are available if you need them. (Do **NOT** assume that the program is already in a parallel region on entry to the `subSum` function.)

2. Prove the sequential version of the algorithm correct. *Hint: make and prove an invariant for the first inner loop; then, just repeat a similar proof for the second. Only then prove the whole function correct by induction.*

3. Analyze the work and span for your parallelized version in terms of n (the length of the array).

4. We can also parallelize the `midSubSum` calculations using parallel prefix and parallel reduce:

   (a) Parallel prefix from INPUT using + over the range `mid` $-$ 1 down to `lo` to TEMP1.

   (b) Parallel reduce from TEMP1 using `max` over the range `mid` $-$ 1 to `lo` to `bestLMidSubSum`.

   (c) Parallel prefix from INPUT using + over the range `mid` to `hi` to TEMP1.

   (d) Parallel reduce from TEMP1 using `max` over the range `mid` to `hi` to `bestRMidSubSum`.

   Analyze the work and span of the algorithm using both parallel recursive calls and this parallelized calculation of the `midSubSums` in terms of n.

This page intentionally left blank.
If you put solutions here or anywhere other than the blank provided for each solution, you must *clearly* indicate which problem the solution goes with and also indicate where the solution is at the designated area for that problem's solution.

This page intentionally left blank.
If you put solutions here or anywhere other than the blank provided for each solution, you must *clearly* indicate which problem the solution goes with and also indicate where the solution is at the designated area for that problem's solution.

This page intentionally left blank.
If you put solutions here or anywhere other than the blank provided for each solution, you must *clearly* indicate which problem the solution goes with and also indicate where the solution is at the designated area for that problem's solution.