CPSC 221: Algorithms and Data Structures
PRACTICE Final Exam, 2012 April 21

**SAMPLE SOLUTION**

# 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?

   **Solution :** It turns out we have no choice over the parity bit in parity-checked bytes (b/c the number of *other* 1s dictates whether the parity bit is a 1… which is the point, actually).

   # of 9-bit sequences is $2^9$. (9 independent choices of 0 or 1.)

   # of 9-bit parity-checked bytes is $2^8$.

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?

   **Solution :** We're forced to use the `q`. We need to choose 0, 1, or 2 `t`s. Each choice makes a non-overlapping set of passwords; so, we'll add up the results.

   For now, let's look at only the case with 2 `t`s.

   Then, we need to select six digits to use of the ten available: $\binom{10}{6}$.

   Finally, we need to order the $1 + 2 + 6 = 9$ characters: 9!. But this counts all rearrangements of the 2 `t`s; so, we divide those out: $\frac{9!}{2!}$.

   All told, then, with 2 `t`s: $\binom{10}{6} \cdot \frac{9!}{2!}$.

   This generalizes nicely to the formula for $n$ `t`s, and we can just reuse that formula:

   $$\binom{10}{6}\left(\frac{7!}{0!} + \frac{8!}{1!} + \frac{9!}{2!}\right)$$

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?

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

**Solution :** Just 7! overcounts by a factor of the number of rearrangements of the `bs` and `as` there are. (We can also imagine "overcounting" the other letters by $1! = 1$ if you like.). So:

$$\frac{7!}{3!2!}$$

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?

**Solution :** Let's start with a slightly surprising formula:

$$5 \cdot 26 \cdot 7 \cdot \binom{12}{3}$$

You may have gotten this differently. Here's my strategy: the four parts are independent decisions, just multiply them together. The first three parts are simple. That leaves the ### part.

Allowing repeats causes some trouble. So, maybe we can "map this" into a world that disallows repeats. Indeed we can. We just add 1 to the second number and add 2 to the third number (where we'll allow the "digits" A and B for 11 and 12). This is a bijection: every original ### sequence maps to one and only one new ### sequence. So, there's the same number of both.

Now we're just choosing a sequence of three strictly increasing digits from the range 0–B. All permutations of three digits from this range are: $\frac{12!}{9!}$, but that overcounts the strictly increasing ones by a factor of 3!.

In other words, we want $\binom{12}{3}$.

## 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.)

   **Solution :** A good (and useful) precondition relating these might be: for all ancestors $a$ of node $n$ (not including $n$): $a$`.key` $\leq$ `lo` or $a$`.key` $\geq$ `hi`.

   We prove this by induction on the depth of recursion into `isBstHelper` (i.e., are we inside one call, two recursive calls, three recursive calls, etc.).

   **Base case**: In `isBst`, the call to `isBstHelper` passes `-infinity` for `lo` and `infinity` for `hi` and the root of the tree for the node. Such a node has no ancestors; so, the condition trivially holds.

   **Inductive Step**: Assume the precondition holds up to some call depth. If we make recursive calls at that depth, it must be because the code passes the check: `node.key >= lo and node.key <= hi`. Furthermore, the recursive calls will be on children of the current node, and so the only new ancestor will be the current node.

   In the first recursive call, we pass in `lo` unchanged but pass the node's key as `hi`. From the conditional above we know that key is no larger than `hi`. Thus, all the ancestors of the current node still pass the precondition: If they were less than or equal to `lo` before, they still are. If they were greater than or equal to `hi`, they're also greater than or equal to the current node's key, since `hi` is greater than or equal to it. Finally, the current node's key is obviously greater than or equal to itself (the new `hi` value). So, the precondition holds.

   The second recursive call behaves similarly, except that we leave `hi` alone and ensure that the new `lo`—which is the current node's key—is greater than or equal to the old `lo`.

   Thus, the precondition always holds.

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-$\Theta$ bound. (Hey, it's a counting problem!)

**Solution :** Algorithm A, checks each node against all of its descendants; so, we want the number of descendants to go down as slowly as possible. Thus, the worst case is a "stringy" (linked-list) tree. For a "stringy" tree with $n$ nodes, the algorithm performs $n-1$ checks (one against each of the $n-1$ descendants). So, in total, there are $(n-1)+(n-2)+(n-3)+\cdots+3+2+1 = \frac{n(n-1)}{2} \in \Theta(n^2)$ checks.

Algorithm B doesn't have a particular worst case because it performs exactly one check for each edge in the tree (checks each child against its parent). That's $n-1 \in \Theta(n)$ checks no matter what.

Algorithm C does exactly two checks at every node, including the root: $2n \in \Theta(n)$ total.

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.

**Solution :** Algorithm B is incorrect. A tree with 2 as the root, 3 as the right child, and 1 as its left child suffices to show the algorithm unsound. (It misses that 1 shouldn't be in the right subtree of 2.)

However, the algorithm *is* complete. That's because in any BST, *all* the keys in the left subtree of a node are less than that node's key; so, certainly the left child (if any) is less. Thus, a BST will always pass the "left child" check. Similarly, it will always pass the "right child" check. (No need for induction here; we just used a "without loss of generality"/"generalizing from the generic particular" proof.)

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.

**Solution :** Algorithm A is asymptotically slower than Algorithm C. Let's prove it by comparing the number of checks performed.

A performs $n(n-1)/2$ checks. C performs $2n$ checks. Thus, we want to prove $n(n-1)/2 \notin O(2n)$ (but $2n \in O(n(n-1)/2)$). (In CPSC 320, you'll learn that what we really want to prove is $n(n-1)/2 \in \omega(2n)$.)

Using the definition of big-$O$, we want: It is not the case that there's an $n_0$ and a $c$ such that for all $n > n_0$, $n(n-1)/2 \leq c2n$. In other words, for all $n_0$ and $c$, we can find an $n > n_0$ (a sufficiently large $n$) such that $n(n-1)/2 > c2n$.

So, let's do scratchwork. We'll set these two sides equal and solve for $n$ and then (probably, if it looks like it will work) point out that we can always pick a larger $n$ to prove our statement. Solving for $n$ for a little while then (SCRATCHWORK):

$$
\begin{aligned}
n(n-1)/2 &= c2n \\
(n-1)/2 &= 2c \\
n-1 &= 4c \\
n &= 4c+1
\end{aligned}
$$

(Note that we threw away the $n = 0$ solution by dividing by $n$, but that's clearly *not* the solution we want.)

So, when $n = 4c + 1$, the two sides are equal. Let's set $n$ to something even larger, say $\lceil 4c \rceil + 2$. (Note that $n$ is an integer but $c$ is a real number, thus the ceiling notation.) Technically, we also need to ensure that it's larger than $n_0$, but this part is always easy (as long as our function is non-decreasing). We just set $n = max(n_0 + 1, \lceil 4c \rceil + 2)$, and it's then guaranteed to be big enough for both requirements.

Now, we should be able to go back and get the result we needed:

$$\begin{aligned} n(n-1)/2 &= (\lceil 4c \rceil + 2)((\lceil 4c \rceil + 2) - 1)/2 \\ &> (4c+1)((4c+1) - 1)/2 \\ &= (4c+1)(4c)/2 \\ &= 8c^2 + 2c \\ &= 2c(4c+1) \\ &= 2cn \end{aligned}$$

QED, for any choice of $n_0$ and $c$, there's a sufficiently large $n$ such that $n(n-1)/2 > c2n$.

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.)

**Solution :** Let's prove that each correct algorithm (A and C) is "eventually" called recursively on each node (as long as all checks pass). I'll proceed by induction on the tree structure. Note that I assume we're invoking the algorithm on the root of the tree (or, for an empty tree, on "the empty tree" since it has no root).

Base Case: For an empty tree, the property is trivially true. For a tree with a single node, that node is the root, and we called the algorithm on it initially.

Inductive Step: Assume that, if called on the roots of two arbitrary trees—`left` and `right`—with all checks passing, the algorithm is eventually recursively called on all of those trees' nodes.

Now, we call it on a tree made of a root with `left` and `right` as its left and right subtrees. The code of C and the algorithm sketch of A show us that *if the checks pass*, we'll recursively call the algorithm on both `left` and `right`. In that case, we've already called the algorithm on the root, and we'll eventually call it (by our assumption) on all nodes in `left` and `right`.

QED

OK, now to show that the algorithms work. What makes a BST a BST?

First, it's a binary tree. Nothing about the algorithms checks that; so, we must have intended it as an assumption (although it would be easy to make something that was *not* a binary tree by, e.g., having one of the nodes' child pointers point at itself!).

Second, it's a binary *search* tree. Our definition of that is "for any node, all the keys in its left subtree are less than it while all the keys in its right subtree are greater than it".

Algorithm A's checks clearly enforce this property at a particular node. If any such check fails, the binary tree is not a BST because some node directly violates this property. If all such checks pass, we've already shown we'll eventually call the algorithm on every node in the tree; so, every node has the desired property, and the tree is a search tree!

How about Algorithm C?

Let's prove that if algorithm C returns true, then all nodes in the given tree are between the lo and hi passed in (and it returns false otherwise).

Base Case: On an empty tree, this property trivially holds (and the code does return true, as it should).

Inductive Step: Assume it's true for a pair of arbitrary trees, as described in the previous proof. Now, we call the algorithm on the root of a tree with these two trees as its subtrees. First, if the root's key doesn't lie between these values, the `if` statement ensures that we catch that fact and return false. If it does lie between these values, then we ensure that the left subtree is between lo and the node's key $k$, which we just said is between lo and hi. So, our assumption gives us that all nodes in the left subtree have keys between lo and $k$, but $k \leq$ hi; so, the left subtree has keys between lo and hi. The same reasoning (but with lo replaced rather than hi) applies to the right subtree.

QED, the Algorithm C returns true if and only if all nodes in the given tree are between the lo and hi passed in.

Now, let's say Algorithm C returns true on some binary tree. Then, all checks must have passed, which means Alg C was called recursively on every node (per our earlier proof). Consider an arbitrary node in the tree. When we call Alg C on its left subtree, we ensure that all of the nodes in that left subtree are between lo and the node's key. That means they're all $\leq$ the node's key. (Ignoring duplicates, which our definition of search trees themselves does, they're less than the node's key.) Similarly, we ensure everything in the right subtree has greater keys. Thus, only binary search trees "pass" Algorithm C.

Now, say we pass a valid BST to Algorithm C. Say (for contradiction) that it fails some check. Either the node's key is less than lo or greater than hi. Say it's less than lo. lo cannot be negative infinity (since all integers compare greater than -infinity). So, lo must be the key value of some ancestor we worked through on the way to this node. Since that ancestor's key value became lo, we must be in its right subtree (because only in the right recursive call is lo altered). From our ancestor's perspective then, it has a descendant in a right subtree whose key is less than its key. That violates the search tree property, which is a contradiction with this being a valid BST! Similarly, if we assume instead that the check failed because the node's key were greater than hi, we'd get a search tree violation from an ancestor in whose left subtree this node resides.

Thus, a binary tree "passes" Alg C iff it's a BST.

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.

   **Solution :** An explicit stack to simulate the (implicit) call stack.

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.)

   **Solution :** And now, flying without a net, here's my attempt without double-checking it through a compiler and test cases. Kind of like an exam. Expect errors. Note also that we asked an **UNFAIR** question. There's no way with a normal BST to get a good sequential cutoff. How do you know when there's only 1000 or so nodes in the current subtree? You can't know because any node "could be your last" (could be a leaf) until you check whether it has children (and grandchildren and. . . ).

   So, we cheated and went back and changed the question prompt so it gave us an out. Damn, but we're sneaky.

```
bool isBstHelper(Node * node, int lo, bool isNegInfinity, int hi, bool isPosInfinity) {
  // no sequential cutoff because there's no "right" way to do it.
  // a maybe-good-idea would be to count how many levels deep we
  // are (add 1 at each recursive call) and punt to sequential
  // after ~12-16 levels.  ("I think there is a world market for
  // maybe 10000 tasks." says TJ Watson.)
  if (node == NULL)
    return true;

  if ((isNegInfinity || node->key >= lo) &&
      (isPosInfinity || node->key <= hi)) {
    bool leftIsBst, rightIsBst;
#pragma omp task untied shared(leftIsBst)
    leftIsBst  = isBstHelper(node->left,  lo, isNegInfinity, node->key, false);
    rightIsBst = isBstHelper(node->right, node->key, false, hi, isPosInfinity)
#pragma omp taskwait
    return leftIsBst && rightIsBst;
  }

  return false;
}

bool isBst(Node * node) {
#pragma omp parallel
#pragma omp single
  return isBstHelper(node, 0, true, 0, true);
}
```

## 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?

   **Solution :** None; there's no self-loops.

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

   **Solution :** $n$ choices of starting point and $n-1$ choices to go next: $n \cdot (n-1)$.

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

   **Solution :** The sum of the previous two parts, since every path is either a cycle or not a cycle.

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

**Solution :** All the possible paths of length 1 with the first node tacked on to the end to make it a cycle: $n \cdot (n-1)$.

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

   **Solution :** Since there are no self-loops, the only way we could repeat a node is by going back to the first node... but that would be a cycle. So, we're choosing three distinct nodes: $n \cdot (n-1) \cdot (n-2)$.

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

   **Solution :** The sum of the previous two parts, as above.

## 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

   **Solution :** Insertion is usually slow (and inconsistent: quite fast in the best case!).

   Quick is usually fast but inconsistent (quite slow in the worst case!).

   Selection is slow.

   Merge is reliably fast.

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

   **Solution :** Either $O(n^2)$ sort is probably fine. I prefer insertion because it's $O(n^2)$ in the average case but much faster in some common cases which is sort of like being "unreliable".

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

**Solution :** None. All three of these correspond to a DFS. The only question is whether we process the "current node" *before* it's children, *between* its children, or *after* its children, respectively. A BFS can "move away" from a nodes children to a sibling and "come back" later.

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**

   **Solution :** PARALLELISM: clearly this is "improving performance by exploiting multiple processors".

   (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**

   **Solution :** CONCURRENCY: here the hash table is the shared resource in "managing simultaneous access to shared resources".

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.

   **Solution :** Left as an exercise to the reader. (Really. Read the parallelism notes!)

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?

   **Solution :** Double. Look up the formula, plug, and chug. Then, ask yourself some more interesting questions, like what happens when $S$ depends on $n$!

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.

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

   **Solution :** $p \in \Omega(\lg n)$ because our performance guarantee (expected case, with a good scheduler) is $O(\frac{n \lg n}{p} + (\lg n)^3)$. With $p \in \Omega(\lg n)$, $O(\frac{n \lg n}{p} + (\lg n)^3) = O(\frac{n \lg n}{\lg n} + (\lg n)^3) = O(n + (\lg n)^3) = O(n)$, which is the sequential algorithm's performance.

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.

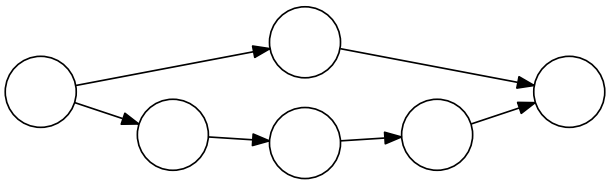
   **Solution :** (d) is irrelevant. (We do the same tuning on B-trees, which don't have the property mentioned.)

   (b) is extremely contextual but certainly untrue in general (and especially in the cases where we bother tuning B+-trees).

   (a) is weird and wrong.

   (c) is quite accurate. We're trying to minimize constants by accounting for the fact that, e.g., the disk access required to get to a node is *far* more expensive than any single operation (or modestly large sequence) that we might perform on that node.

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.



   **Solution :** Sure:

   The longest path in this graph is length 4. The highest in-degree is 2. $|V| = 6$. And, $6/2 < 4$.

## 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.

   **Solution :** (1) best describes work.

   (2) is inaccurate because the "real" sequential version may differ from the best parallel version (since we sometimes sacrifice single-processor performance (work) to improve span). It's useful because the performance of the best available sequential algorithm is often already known when we set to work on a parallel version... and even when it's not, it's an important benchmark! (Why bother parallelizing if it doesn't do any better than the sequential solution?!)

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.

**Solution :** (2) best describes span.

(1) is inaccurate because not all nodes necessarily take "1 unit of time" each; so, we cannot just add up the number of nodes. However, this estimate is approximately correct if all nodes take about the same amount of time and asymptotically correct (for families of DAGs whose structure depends on $n$, else we couldn't talk about asymptotics!) if the work at each node is bounded by a constant (which is a typical goal we have in mind for load management in good task-based parallel algorithm design). Plus, DAGs are a convenient, clear representation for analysis.

# 6 Analyze This [SOME marks]

Give a big-$\Theta$ 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 >= 0; j /= 2) {
    count++;
  }
  for (int k = n; k > count; k--) {
    count++;
  }
  count++;
}
int p = n*count;
for (int m = 0; m < p; m++) {
  count++;
}
```

**Solution :** This is a super-tricky, fun problem. Let's start from the end. Whatever `count` is coming out of the top loop, the bottom loop runs $n$ times that long. Since the "inner loop" inside the top loop is always an increment of `count`, it's the bottom loop that will set our asymptotic bound.

That means we need to know `count` after the top loop, which we can calculate asymptotically.

The top loop runs $n^2$ times. (See why? You can tell just by looking at the `for (..)` part.)

Within the loop, we're only concerned asymptotically with the statement (of the first loop, the second loop, or the final increment) that increases `count` the most. Since the loops run at least once, that means we can ignore the final increment.

How about the first loop? Since we divide `j` in half each time, it runs $\lg i$ times. That's:

$$\lg 1 + \lg 2 + \lg 3 + \cdots + \lg n^2 - 2 + \lg n^2 - 1 + \lg n^2$$

We've done a similar analysis several times. We can upper-bound by raising everything to $\lg n^2$ and get $n^2 \lg n^2 = 2n^2 \lg n \in O(n^2 \lg n)$. We can similarly lower-bound by dropping the bottom half of the terms and

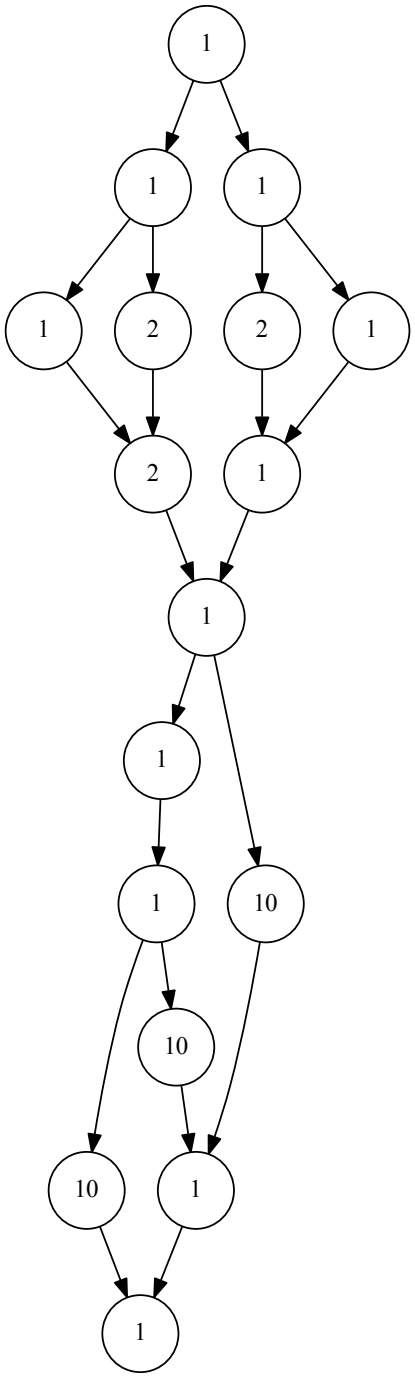
lowering the top half to roughly $\lg \frac{n^2}{2}$, which gives us a $\Omega(n^2 \lg n)$ bound. Thus, the first loop *in total* contributes $\Theta(n^2 \lg n)$ to `count`.

The second loop is strange. Clearly, however, it doesn't run at all once `count` reaches `n`. But, each time the loop *does* run, it increases `count`. So, no matter *what* else is happening, that loop cannot run more than `n` times. Thus, the second loop *in total* contributes $O(n)$ to `count`, which is swamped by the first loop.

So, by the time we reach the line that sets `p`, `count` $\in \Theta(n^2 \lg n)$, which means the whole program runs in $\Theta(n^3 \lg n)$ time.

## 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!




**Solution :** Sorry for writing it this way but... starting from the top: left, right, drop, drop, left, drop, right, drop, drop, which yields $1 + 1 + 2 + 2 + 1 + 1 + 1 + 10 + 1 + 1 = 21$.

In the top half (above the "choke point"), the structure is symmetrical; so, we just trace the "heaviest" route, which is the one with two 2s.

In the bottom half, I just exhaustively tried the three paths and summed them up, looking for the most expensive.

# 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.

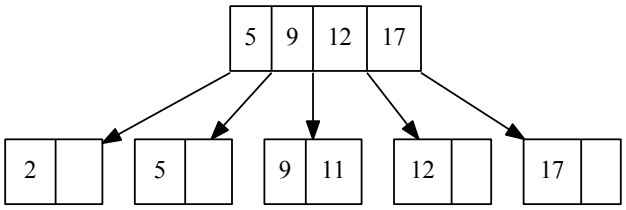
Parallel map using _____ from _____ to _____.
Parallel map using FUNCTION from SOURCE to TARGET.
Parallel pack filtering on FUNCTION from SOURCE to TARGET starting at index NUMBER.
Parallel prefix using FUNCTION from SOURCE to TARGET.
Parallel reduce using FUNCTION from SOURCE to TARGET.

**Solution :** Parallel map using `makePair` from INPUT to TEMP1.
Parallel prefix using `countLastURL` from TEMP1 to TEMP2.
(At this point, there's quite a bit of variation in the order we can do things. Here's one solution.)
Parallel pack filtering on `isBoundary` from TEMP2 to OUTPUT starting at index 0.
Parallel map using `getCount` from TEMP2 to TEMP3.
Parallel reduce using `max` from TEMP3 to FREQ.

# 9 Groundhog Dayta Structures [SOME marks]

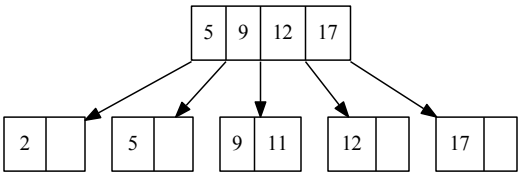
Each of the following problems starts with a picture of a data structure starting in a particular state. For each subproblem, assume the data structure starts in this state, and draw its state after the operation indicated. You may modify the given picture or draw a new one as you see fit, but make your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

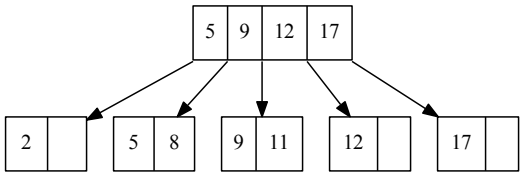
1. **Groundhog B+-Tree**: draw the result of applying the indicated operations to the following B+-tree.




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.

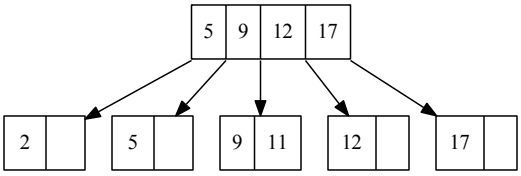
(a) INSERT 8

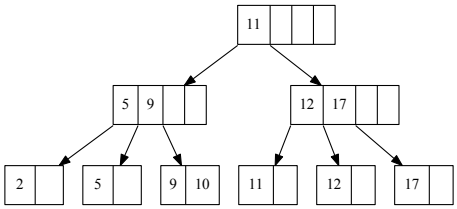



**Solution :**

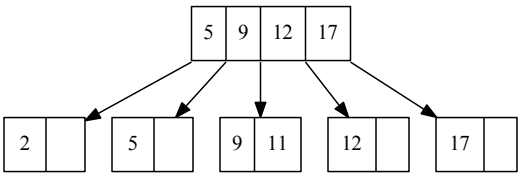
(b) INSERT 10

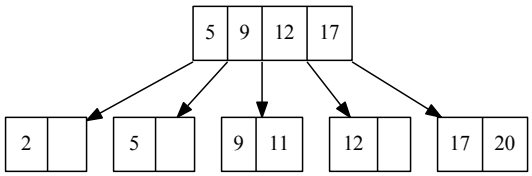





**Solution :**

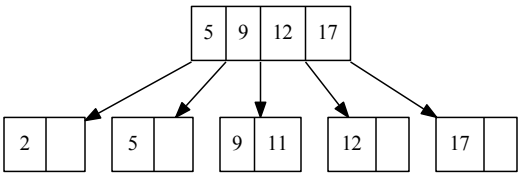
(c) INSERT 20

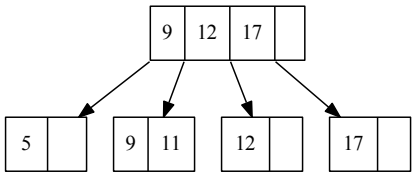



**Solution :**

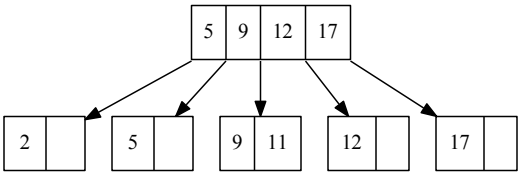
(d) DELETE 2

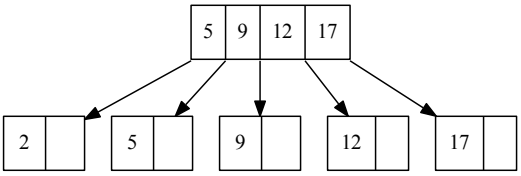



**Solution :**

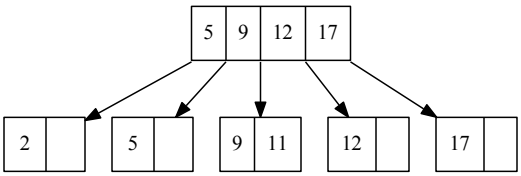
(e) DELETE 11

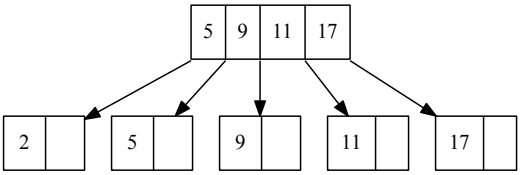



**Solution :**

(f) DELETE 12

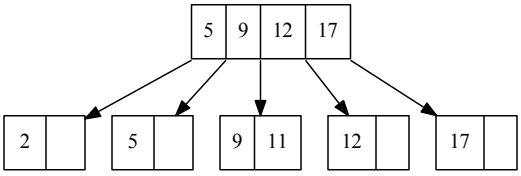


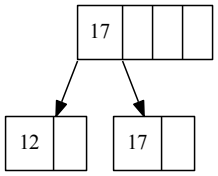




**Solution :**

Notice the *two* things that have changed on this delete/borrow!

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






**Solution :**

(The root can go down to 2 children without getting merged, since there's nothing to merge with!)

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* |

**Solution :**

| | |
|---|---|
| 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* |

**Solution :** Hashes to index 3, collides, probes at $3+1$ and slots right in.

| | |
|---|---|
| 0 | |
| 1 | 1 |
| 2 | 16 |
| 3 | 3 |
| 4 | 10 |
| 5 | 26 |
| 6 | *T* |

(c) INSERT 20

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

**Solution :** Hashes at index 6 and overwrites the tombstone.

| | |
|---|---|
| 0 | |
| 1 | 1 |
| 2 | 16 |
| 3 | 3 |
| 4 | |
| 5 | 26 |
| 6 | 20 |

(d) INSERT 30

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

**Solution :** Hashes at index 2, probes at index $2 + 1 = 3$, probes at $2 + 4 = 6$, replaces the tombstone.

| | |
|---|---|
| 0 | |
| 1 | 1 |
| 2 | 16 |
| 3 | 3 |
| 4 | |
| 5 | 26 |
| 6 | 30 |

(e) INSERT 50

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

**Solution :** Hashes at index 1, probes at index $1 + 1 = 2$, probes at index $1 + 4 = 5$, probes at index $1 + 9 \equiv 10 \equiv 3 (\mod 7)$, probes at index $1 + 16 \equiv 17 \equiv 3 (\mod 7)$, and we start repeating. From this

point, we'll repeat forever, but it turns out if we *ever* go $\frac{n}{2}$ probes in quadratic probing, the insert is going to fail.

So, we rehash to something like the following (although depending on your order, you may get something slightly different):

| | |
|---|---|
| 0 | |
| 1 | 1 |
| 2 | 16 |
| 3 | 3 |
| 4 | |
| 5 | |
| 6 | |
| 7 | 7 |
| 8 | 50 |
| 9 | |
| 10 | |
| 11 | |
| 12 | 26 |
| 13 | |

Note that we dump the tombstones since we're rehashing anyway. Also note that no matter what, this resize/rehash *cannot* fail because the load factor ends up less than one-half.

(f) DELETE 3

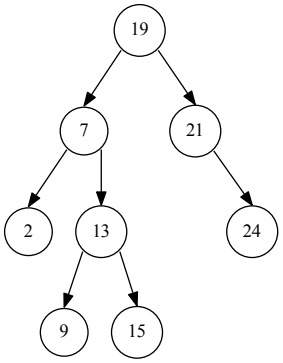
| | |
|---|---|
| 0 | |
| 1 | 1 |
| 2 | 16 |
| 3 | 3 |
| 4 | |
| 5 | 26 |
| 6 | *T* |

**Solution :**

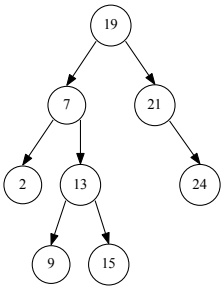
| | |
|---|---|
| 0 | |
| 1 | 1 |
| 2 | 16 |
| 3 | T |
| 4 | |
| 5 | 26 |
| 6 | *T* |

It's a tombstone, not an empty slot!

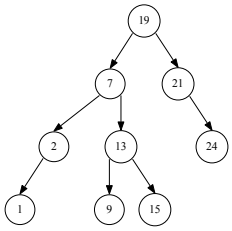
3. **Groundhog AVL Tree**: draw the result of applying the indicated operations to the following AVL tree.



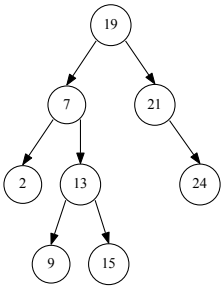

(a) INSERT 1



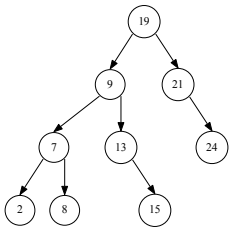

**Solution :** Just a normal insert:



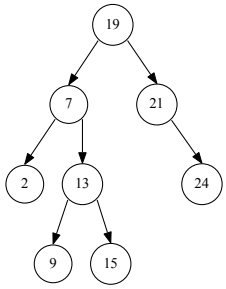

(b) INSERT 8



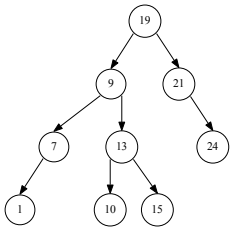

**Solution :** *Not* at node 8 but at node 9, the grandparent (7) is unbalanced. That's a zig-zag case:



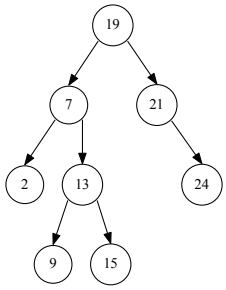

(c) INSERT 10



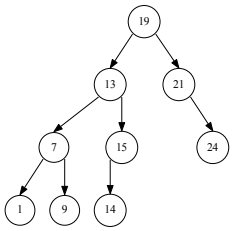

**Solution :** Again *not* at node 10 but at node 9, the grandparent (7) is unbalanced. That's a zig-zag case with the children falling a bit different:



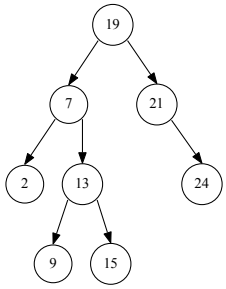

(d) INSERT 14




**Solution :** *Not* at node 14 but at node 15 this time, the grandparent (7) is unbalanced. That's a zig-zig case:

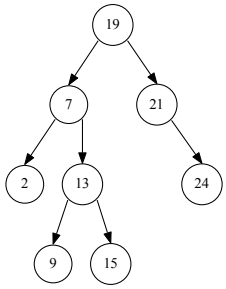



(e) INSERT 17



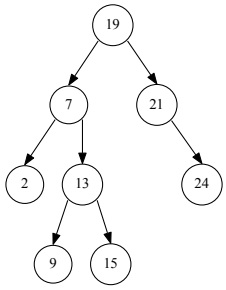

**Solution :** A zig-zig from 15 up to 7. Left as an exercise.

(f) INSERT 20



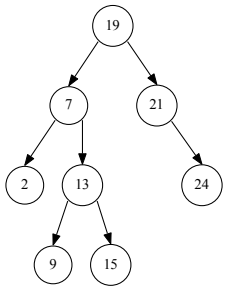

**Solution :** A normal insert. Left as an exercise.

(g) INSERT 23




**Solution :** A zig-zag from 23 up to 21. Left as an exercise.

(h) INSERT 26




**Solution :** A zig-zig from 26 up to 21. Left as an exercise.

## 10 PARAheapsLEL [SOME marks]

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

**Solution :** We'll provide a mix of pseudocode and C++ and assume that an `isLeaf` function, a `leftChild` function, a `rightChild` function, and a `percolateDown` function are all available. (Note: our code fails on an empty heap but could easily be modified not to.)

Switch to recursion, of course:

```
// call on the root of a non-empty heap to run.
// (be sure this is inside a #pragma omp parallel region)
buildHeap(heapArray, nodeIndex, heapSize):
  if !isLeaf(nodeIndex, heapSize):
#pragma omp task untied // nothing needs to be shared!
    buildHeap(heapArray, leftChild(nodeIndex), heapSize)
    buildHeap(heapArray, rightChild(nodeIndex), heapSize)
#pragma omp taskwait

  // This is a POST-ORDER traversal; so, we work from the leaves up.
  percolateDown(heapArray, nodeIndex, heapSize)
```

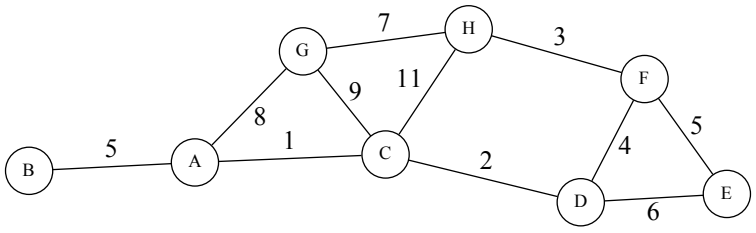
The work is left as an exercise.

The key piece of the span recurrence is a recursive call on a heap about half as large plus the $O(\lg n)$ work performed by the percolate down call:

$$\begin{aligned} T_\infty(n) &= T_\infty(n/2) + \lg n \\ &= \lg n + \lg n/2 + \lg n/4 + \cdots + \lg 1 \\ &= (\lg n) + (\lg n) - 1 + (\lg n) - 2 + \cdots + 2 + 1 + 0 \\ &\in \Theta((\lg n)^2) \end{aligned}$$

## 11 Computer Graph-ics [SOME marks]

Consider the following graph:




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

**Solution :** Assume blanks in the table indicate $\infty$. Since the graph is undirected, we provide only the upper-right triangle (but it would be fine to provide the lower-left or to provide both as long as they were symmetric about the diagonal of the matrix, i.e., mirror images of each other).

| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| A | | 5 | 1 | | | | 8 | |
| B | | | | | | | | |
| C | | | | 2 | | | 9 | 11 |
| D | | | | | 6 | 4 | | |
| E | | | | | | 5 | | |
| F | | | | | | | | 3 |
| G | | | | | | | | 7 |
| H | | | | | | | | |

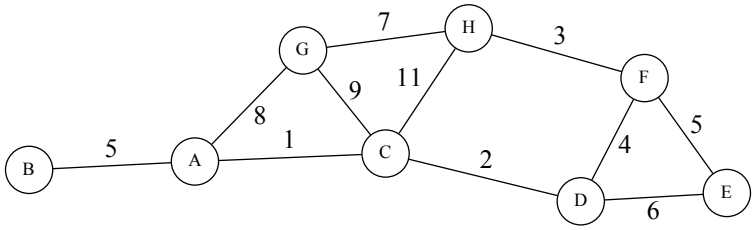
2. Write out an adjacency list representation of the graph.

**Solution :** We list *both* directions of every edge. That's probably the best idea if we need to be able to iterate over all neighbours of a node quickly. Do you see why? If we only care about saving storage space, we can list only the "upper right" as with the adjacency matrix above.

| | |
|---|---|
| A | (B, 5), (C, 1), (G, 8) |
| B | (A, 5) |
| C | (A, 1), (D, 2), (G, 9), (H, 11) |
| D | (C, 2), (E, 6), (F, 4) |
| E | (D, 6), (F, 5) |
| F | (D, 4), (E, 5), (H, 3) |
| G | (A, 8), (C, 9), (H, 7) |
| H | (C, 11),(F, 3), (G, 7) |

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:



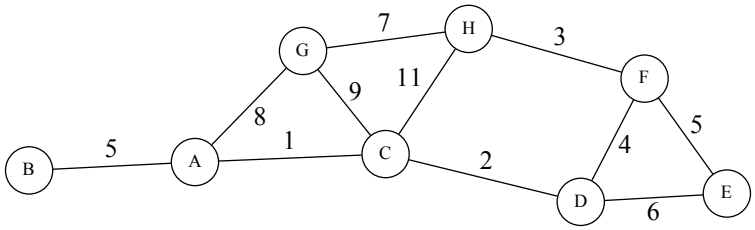

| *node* | *cost* | *order* |
|---|---|---|
| **A** | 0 | $1^{st}$ |
| **B** | | |
| **C** | | |
| **D** | | |
| **E** | | |
| **F** | | |
| **G** | | |
| **H** | | |

**Solution :**

| *node* | *cost* | *order* |
|---|---|---|
| **A** | 0 | $1^{st}$ |
| **B** | 5 | $4^{th}$ |
| **C** | 1 | $2^{nd}$ |
| **D** | 3 | $3^{rd}$ |
| **E** | 9 | $7^{th}$ |
| **F** | 7 | $5^{th}$ |
| **G** | 8 | $6^{th}$ |
| **H** | 10 | $8^{th}$ |

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** | | |

**Solution :** We made some changes as we created the solution, so please check if this is incorrect! Also, note that the two edges both with weight 5 mean a small amount of reordering (swapping the order of the two 5 edges) is possible. The MST does not change.

| *edge* | *in MST?* | *order* |
|---|---|---|
| **AB** | Yes | $5^{th}$ |
| **AC** | Yes | $1^{st}$ |
| **AG** | No | $9^{th}$ |
| **CD** | Yes | $2^{nd}$ |
| **CG** | No | $10^{th}$ |
| **CH** | No | $11^{th}$ |
| **DE** | No | $7^{th}$ |
| **DF** | Yes | $4^{th}$ |
| **EF** | Yes | $6^{th}$ |
| **FH** | Yes | $3^{rd}$ |
| **GH** | Yes | $8^{th}$ |

## 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.)

**Solution :** We propose a standard B+-tree and a slightly fancy hashtable with double hashing. Any reasonable hashtable with double hashing would do fine, but we think it would be fun to use whose size defaults to a power of 2 well above the maximum load size the game design company has measured/anticipates, and the double-hash mod base being one-fourth of the table size. (Why a power of 2? Mod by a power of 2 is very fast!) But, once we have the double-hash mod result, we multiply it by 2 and add 1 to get an odd number (so it's guaranteed to be mutually prime with the table size and we'll explore all cells of the table).

(Fun alternate hash table solution: use a normal double hash but insist that the client send both the quotient and the remainder of the key divided by the double hash's mod base. It's then relatively cheap to use the client's remainder and, if we want (which we should for security reasons!!) verify that the client did the right thing with a multiplication and an addition.

Advantages of the B+-tree over the hashtable: B+-trees are designed to work well managing enormous amounts of data. Perhaps this table is large enough for that to be very useful. (But it may fit in main memory on the server or the server might be unacceptably slow anyway.) Also, B+-trees scale gracefully as the maximum number of characters active at a time increases, while the hash table will suffer a very expensive resize/rehash at some point.

Advantages of the hashtable over the B+-tree: of course $O(1)$ expected time for operations! It's also relatively simple (even with the bells and whistles above). Finally, each insert into the hashtable changes just one local entry, whereas inserts into a B+ tree can result in changing many entries (although this is a case where not insisting on sorted nodes in the B+-tree might actually be advantageous).

Of course there are many others!

One last note: if you're curious to know more about "big data" data structures like B+-trees, look up "external memory data structures".

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

**Solution :** Well, maybe they could do deletes on the fly now (leaving them as tombstones) and batch removal of the tombstones when they were previously doing deletes. (Realistically, that's probably what they were already doing.)

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

**Solution :** Here's a couple of reasons:

They should distribute their server to avoid a "single point of failure", although this is somewhat outside our course's scope.

They can handle the (hopefully) common lookup case *very* quickly with little contention using a parallel solution, since multiple reads can happen concurrently. They *may* also be able to handle many inserts simultaneously, depending on their concurrency strategy.

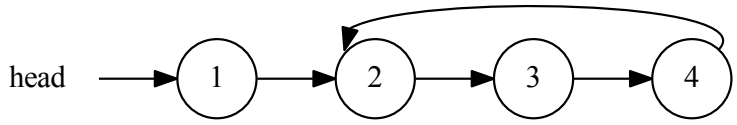
## 13 This Is The Problem That Never Ends... [SOME marks]

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`.




**Solution :**

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;
  }
}
```

**Solution :** A segmentation fault, quite likely. This is not tail recursive but will (try to) recurse infinitely. (Also, nothing prints.)

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;
}
```

**Solution :** An infinite loop, assuming we have basic optimazation on (else a seg fault again). This *is* tail recursive; so, we have a nice, tight infinite loop.

4. Return to part 1.

**Solution :** . . . it just goes on and on, my friend!

## 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.)

**Solution :**
```
int subSumHelper(int arr[], int lo, int hi) {
  if (hi - lo <= SEQUENTIAL_CUTOFF) return subSumSeqlHelper(arr, lo, hi);

  int mid = lo + (hi - lo) / 2;
  // the declaration of (at least) the shared variable
  // must go before we fork the task!
  int lSubSum, rSubSum;
```

```
#pragma omp task untied shared(lSumSum)
  int lSubSum = subSumHelper(arr, lo, mid);
  int rSubSum = subSumHelper(arr, mid+1, hi);
#pragma omp taskwait

  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) {
#pragma omp parallel
#pragma omp single
  return subSumHelper(arr, 0, n);
}
```

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.*

**Solution :** Here's the first loop:

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

What it's trying to do is find the largest subsequence sum with its endpoint "anchored" at the midpoint of the array (just before). So, let's have as an invariant that: `bestLMidSubSum` is the "best subsequence sum so far", i.e., the maximum subsequence sum for a subsequence of the form $\texttt{arr}[k \ldots mid - 1]$, where $i < k < mid$, and `lMidSubSum` is the sum of all the elements in $\texttt{arr}[i + 1 \ldots mid - 1]$.

Base case: when `i = mid - 1`, then the subsequence we're operating on is empty, and 0 (the value in both variables of interest) is trivially the best and full subsequence sum.

Inductive step: assume that the invariant holds at a particular value of `i` that's less than `mid` and at least `lo`. We must show that when `i` is decremented at the end of the loop, the invariant still holds.

`lMidSubSum` is updated to be `arr[i] + lMidSubSum`, which by our assumption is `arr[i] + arr[i+1] + ⋯ array[mid-1]`, which is correct for the next value of `i`, completing that portion of the proof.

`bestLMidSubSum` should either be the best subsequence sum so far up to this point (which it already is by assumption) or be updated to the new sum, the one that extends all the way from `i` to `mid - 1`. As we've just shown, that's the new value of `lMidSubSum`, and indeed the code ensures that if that value is larger than the value of `bestLMidSubSum`, we update `bestLMidSubSum`.

QED

So, the loop really does give us the best sum on the left that's "anchored at the middle". By similar reasoning, the second loop gives us the best sum on the right that's "anchored at the middle" (starting at `mid` rather than `mid-1`. If we add these together, that's the best sum that crosses the middle. We'll come back to that.

Now, let's prove that the whole function works.

Base Case: The maximum subsequence sum of an empty array is 0, which is what the code returns when `lo <= hi`.

Inductive Step: Assume the code works correctly on arrays that are "half as big" (floor and ceiling) as an array of length $n \geq 1$.

Then, the two recursive calls return the maximum subsequence sums of the left and right halves of the array. The max of those two is the maximum subsequence sum that does *not* include `mid` (or, therefore, "cross the middle" of the array), since I intentionally left that out of the recursive calls.

We already saw that the loops give us the maximum subsequence sum that *does* cross the middle. The max of these two is clearly the maximum subsequence sum in the array.

QED

(BTW, I note as I write the proof that we *know* that both the `mid` *must* be included if the "middle" sum unambiguously wins. Otherwise, the left recursion would "catch" it. So, we could use this insight to slightly improve the algorithm.)

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

   **Solution :** Work left as a standard recursion analysis exercise similar to the MergeSort analysis.

   Span is more interesting. For the recurrence for span, we get $T_\infty(n) = T_\infty(n/2) + n$. (As usual, letting $n$ stand in for a function that is $\Theta(n)$ and $T_\infty(n/2)$ stand in for something with a floor or ceiling. (And that latter just made me go back and change `mid` to `mid+1` in the recursive calls in the code. Can you see why? Hint: non-termination.)

   $T_\infty(n) = T_\infty(n/4) + n/2 + n = T_\infty(n/8) + n/4 + n/2 + n$ and so on. This sum is clearly lower-bounded by $n$ and converges to $2n$. So, $T_\infty(n) \in \Theta(n)$.

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 $n$.

**Solution :** Work will be unchanged. Analysis left to the reader.

The span *will* change. Analysis left to the reader, but start from the analysis of parallel quicksort if you need hints. You should see marked similarities (like the-samenesses).