B01334462 () | on time | CS 157 – Fall 2017
B01507364 () | **Homework 4** | October 13, 2017

Problem 1: Universal Hashing

1. (a) If we have a set of parameters P, we want each parameter $p \in P$ to produce a hash h_p such that the probability of two elements in the domain a, b where $a \neq b$, hashing to the same value is $\frac{1}{m}$ where m is the total number of elements in the image of the hash.

   For a particular hash family $H = \{h_p : p \in [m]\}$ where m is a fixed prime and

   $$h_p(x) = px \text{ mod } m$$

   let's prove that for any hash $h_p \in H$ that we choose randomly the probability $(h_p(a) = h_p(b)) = \frac{1}{m}$.

   Like we showed in class, we can treat modular arithmetic like regular arithmetic in $\mathbb{R}$. Thus, let's solve for $p_1(x_1) = p_1(y_1)$. Combining terms we get $p_1(x_1 - y_1) = 0$. Since we're modding by m, which is prime, and $x_1 \neq y_1$ we're assured $(x_1 - y_1)$ has a unique inverse, i.e.:

   $$p_1 = \frac{0}{x_1 - y_1}$$

   Thus, for the hashes of x_1 and y_1 to be equivalent p_1 can only be a single value out of m, i.e. 0, meaning the probability $(h_1(x_1) = h_1(y_1))$ is $\frac{1}{m}$. Since we chose h_1 randomly, this holds for any hash in the family making H a universal hash family.

   The number of parameters in $P = [m]$, making the number of bits needed to compute a random element $\lceil \log_2 |P| \rceil = \lceil \log_2 [m] \rceil = \lceil \log_2(m) \rceil)$.

   (b) For a particular hash family $H = \{h_{p1,p2} : p_1, p_2 \in [m]\}$ where m is a fixed prime and

   $$h_{p1,p2}(x_1, x_2) = (p_1 x_1 + p_2 x_2) \text{ mod } m$$

   let's prove that any hash $h_{p1,p2}$ chosen randomly from H is universal such that the probability $((p_1 x_1 + p_2 x_2) = (p_1 y_1 + p_2 y_2)) = \frac{1}{m}$.

   Similar to above, let's group like terms $p_1(x_1 - y_1) + p_2(x_2 - y_2) = 0$. Renaming $p_2(x_2 - y_2) = q$ we get:

   $$p_1 = -q/(x_1 - y_1)$$

   Again, since we're modding by m, which is prime, and $x_1 \neq y_1$ we're assured $(x_1 - y_1)$ has a unique inverse. Thus, there's a single value for p_1 to make

$((p_1x_1 + p_2x_2) = (p_1y_1 + p_2y_2))$. Since p_1 can take on m values, there's a $\frac{1}{m}$ chance two values in the domain, not equal to each other, hash to the same value, making this hash family universal.

The number of parameters in $P = [m] \times [m]$ making the number of bits needed to compute a random element $\lceil \log_2 |P| \rceil = \lceil \log_2([m]^2) \rceil = \lceil \log_2(m^2) \rceil = \lceil 2 * \log_2(m) \rceil$.

(c) This hash family, where H is the same as in *part b*, but with $\{m = 2^k : k \in \mathbb{N}_0\}$ is not universal. In order to prove this, let's use a similar derivation as in *part b*, but show that given a particular (x_1, y_1) multiple values p_1 map to the same value.

Let's say, for example, that q from part 2 equals 0 and $m = 8$. In this case we have:

$$p_1(x_1 - y_1) = 0$$

Since m is not prime, there isn't guaranteed to be a unique inverse. If $x_1 = 4$ and $y_1 = 2$ we get:

$$p_1(2) = 0$$

In which case both $p_1 = 0$ and $p_1 = 4$ satisfy the above equation. In this case, the probability $((p_1x_1 + p_2x_2) = (p_1y_1 + p_2y_2)) = \frac{2}{m}$, which means this isn't universal.

Since the number of parameters in $P = [m] \times [m]$, like above, the number of bits is the same: $\lceil 2 * \log_2(m) \rceil$.

(d) Likewise, this family is not universal. We just proved above in *part c* that a function:

$$h_{p1,p2}(x_1, x_2) = (p_1x_1 + p_2x_2) \bmod m \text{ where } \{m = 2^k : k \in \mathbb{N}_0\} \tag{1}$$

is not universal. Thus, a family H made up of all functions mapping $((x_1, x_2) \in [m] \times [m]) \rightarrow [m]$ would include the above function. When we're checking if a family of hash functions is universal, we check to see if a random function chosen from the family is universal. Since (1) isn't universal, and there are many others like it $\in H$, H is not universal.

Since H is the family of hash functions mapping $[m] \times [m] \rightarrow [m]$, there's no stipulation about either the number of parameters h is parameterized by or the domain of p. For example:

$$h_{p_1,p_2,p_3,...p_n}(x_1, x_2) = p_1x_1 + p_2x_2 + p_3 + ...p_n : \{p_i \in \mathbb{Z}\} \tag{2}$$

Is a valid function $\in H$. Therefore, it seems the number of bits to needed to compute a random element could be unbounded.

2. If we've already found 2 pairs of inputs, $\{(x_1, x_2), (x_1', x_2')\}$, that hash to the same value we can try to solve for p_1 or p_2. Setting up an equivalence like above:

$$p_1(x_1) + p_2(x_2) = p_1(x_1') + p_2(x_2') \tag{1}$$

We can easily combine terms to get $p_1(x_1 - x_1') + p_2(x_2 - x_2') = 0$. Since we know each pair of values, let's substitute constants c_1 and c_2 above to get:

$$p_1(c_1) + p_2(c_2) = 0 \tag{2}$$

WLOG, let's now solve for either p_1 or p_2, eliminating one of the hash's parameters:

$$p_1 = \frac{-p_2(c_2)}{c_1} \tag{3}$$

To find another pair of inputs, (y_1, y_2), that hash to the same value, let's solve equation (1), but substitute in the new value for p_1 we've calculated in (3):

$$\frac{-p_2(c_2)}{c_1}(y_1 - x_1) + p_2(y_2 - x_2) = 0 \tag{4}$$

Multiplying through by c_1 and combining like terms we can simplify (4) into:

$$p_2((c_2x_1 - c_1x_2) + (c_1y_2 - (c_2y_1)) = 0 \tag{5}$$

We know the values of x_1, x_2, c_1, and c_2, thus we can simplify further grouping them into a constant c_3:

$$p_2(c_1y_2 - c_2y_1 + c3) = 0 \tag{6}$$

In order to find future pairs, (y_1, y_2), that hash to the same values as (x_1, x_2) and (x_1', x_2'), simply pick values y_1 and y_2 that make $(c_1y_2 - c_2y_1 + c3)$ equal 0.

As a quick example, if $m = 13$ and $(p_1, p_2) = (3, 7)$, then pairs $(2, 1)$ and $(4, 2)$ would both hash to the same value, i.e. 0. Even if we don't know p_1 and p_2, using the above derivation we get an equation of the form (6):

$$p_2(2y_2 - y_1 + 0) = 0$$

Finding the values of (y_1, y_2) that make $(2y_2 - y_1 + 0) = 0$ quickly yields suitable pairs: $(6, 3)$, $(8, 4)$, $(10, 5)$, etc.

Problem 2: Constant Initialization

1. Our goal in this problem is to create a data structure, much like an array, where getting the value from an index and setting the value at an index are constant time operations with the additional property that initialization of the data structure is also O(1).

   Data Structure: We will satisfy the above invariants by using additional memory, specifically, every time n blocks of memory are requested we will allocate 3 times that, $3n$, broken up into 3 arrays A, B, C. We will also maintain a *counter*, which tracks how many elements have been SET in our data structure, initialized to 0. The only value we'll initialize in our arrays will be $C[0] = 0$. Thus, with g_i indicating a garbage value at index i, our data structure initially looks like this:

   $$counter = 0$$

   $$A = \begin{array}{|c|c|c|c|c|c|}\hline g_0 & g_1 & g_2 & g_3 & \dots & g_n \\ \hline\end{array}$$

   $$B = \begin{array}{|c|c|c|c|c|c|}\hline g_0 & g_1 & g_2 & g_3 & \dots & g_n \\ \hline\end{array}$$

   $$C = \begin{array}{|c|c|c|c|c|c|}\hline 0 & g_1 & g_2 & g_3 & \dots & g_n \\ \hline\end{array}$$

   Since the total amount of memory used, including the counter, is $(3n + 1)$, asymptotically our memory usage is still $O(n)$. Furthermore, we only ever initialize 2 values, thus initialization is $O(1)$. Using the above data structure, our algorithm is as such:

   Algorithm: Whenever we try to set an index i to a value v, we will use array A to store that value. Since we are not initializing any of the values in our arrays except $C[0]$, we need to keep track of what indices have been modified. In order to do this, array B, at index i, will maintain the index into array C, which holds where v was inserted into A. **The sub-array in C from [0, *counter*) thus maintains a record of all indices in A that we've modified and are, therefore, valid parameters to GET.** Our variable *counter* maintains the size of this record in C so that, regardless what garbage values are initially in C, we only ever look in the portion we've modified. Thus, to perform each operation:

   (a) **SET**: Whenever we SET a value v at index i, we'll first index into B at i. If $B[i] \geq counter$ or $C[B[i]] \neq i$, we know the index we're trying to SET isn't in C and therefore hasn't been modified yet.

In this case, we will: **(1)** $A[i] = v$ **(2)** $B[i] = counter$ **(3)** $C[counter] = i$ **(4)** $counter = counter + 1$.

Thus we'll place the element into A, the index of A into C, and the *counter* of when we placed the value into A into B. What's key about *counter* is it keeps track of the sub-array of C that holds values *we know* we've modified and are thus valid to GET.

If $B[i] < counter$ and $C[B[i]] = i$, i has been modified already. In this case we don't need to increment the size of the sub-array in C holding modified indices, we can simply update $A[i] = v$.

(b) **GET**: When we try to GET a value at a particular index i, we'll first check the value of $B[i]$, which is an index into C. If this value $\geq counter$, then it references a portion of C we've yet to modify, therefore we'll return *empty*. Otherwise, we'll check that $C[B[i]] = i$. If it does, we know that index i has indeed been SET by us so we'll return the value at $A[i]$. If it doesn't we'll return *empty*.

Proof: Let's now prove that this data structure behaves the same as an array, where GET and SET operations correctly update the array in constant time.

When we SET a value v at index i, we do a maximum of 2 array indexations, 3 array assignment operations, 2 comparisons, and a single increment to *counter*. Since each of these operations is constant and we're doing a set number of them not dependent on the size of the array, SET is a constant time operation. Furthermore, we're always updating index i in array A to hold the value we're inserting. Thus we know SET correctly places elements into our array.

When we GET an element at index i, we do a maximum of 3 array indexations and 2 comparisons. Likewise, since each of these operations is constant and we're doing a set number of them not dependent on the size of the array, GET is a constant time operation. We are sure GET returns only values we have actually SET because on each GET we check the sub-array in C from $[0, counter)$. This is the portion of C only we've updated, so we are sure that if we find an index in this region it's been SET by us.

Let us say a garbage value from $B[i]$ indexes into this sub-array in C where we're storing all the indices we've SET. Even if this were to happen, we know i won't be in this sub-array since we start with $C[0] = 0$ and $counter = 0$ and only increment the size of the sub-array and include a new i when we SET a new value. Thus, we're guaranteed that GET correctly returns only those values that've been SET and *empty* otherwise.

Problem 3: Binary Search Trees

1. **AVL Trees** For insertion, AVL trees require either 0, 1, or 2 single rotations (you could call 2 single rotations a double rotation). If the insertion doesn't change the depth differential between sub-trees > 1, then no rotations are needed. If the depth differential between two sub-trees is increased to 2, then depending on the position of the sub-tree with the greater depth it could take either 1 or 2 single rotations (1 when the shallower sub-tree is to the left of a parent node, 2 when the shallower sub-tree is to the right).

   Deletions are a bit more complicated. When we delete a node out of an AVL tree we again might need to use a rotation if the height differentials of two sub-trees is 2. This re-balancing could affect the height of the tree containing these sub-trees, however, and therefore the number of rotations necessary for a deletion could be proportional to the depth of the tree i.e. $log(n)$ where n is the number of nodes in the tree.

   Red-Black Trees According to the Wikipedia page under Operations, Red-Black trees require at most 2 rotations for insertion and 3 rotations for deletion.

2. In this problem we will show that the longest path from the root of an arbitrary red-black tree to a leaf contains at most twice as many nodes as the shortest path from the root to a leaf.

   Key to solving this is the invariant of red-black trees that **every path from a given node to any of its descendant leaf nodes contains the same number of black nodes**. So, given a tree T, let the shortest path from root to leaf contain a total of B black nodes. Longer paths can be constructed, but in order to keep B constant we can only insert red nodes.

   Also important is the fact that **if a node is red then both its children must be black**. Thus, we can't simply insert an arbitrary number of red nodes; each red node must be interleaved between black nodes i.e. the black and red nodes must alternate. Thus, the longest possible path would be the one with alternating red black nodes. This path would have a length $2B$, while the shortest path, the one with only black nodes, would have length of B.

   Therefore, the longest path from the root of an arbitrary red-black tree to a leaf contains at most twice as many nodes as the shortest path from the root to a leaf.

3. There will be an array of size three that is associated with each of the nodes in the tree. The first entry of the array is minimum distance between any two nodes in the

node's subtree. The second entry will be the minimum value of a node in the node's subtree. The third entry will be the maximum value of a node in the node's subtree. Given a tree T, we only need to look at the first entry of the array associated with the root to find the minimum distance between any two elements of the tree. This lookup is constant time because we will always be looking at the root's array regardless of how large the tree T grows.

We will now explain how we plan on updating the information given the four subroutines Add-Leaf, Remove-Node-With-At-Most-One-Child, Edit-Internal-Node-Value, and Rotate.

Add-Leaf: When we are adding a leaf, we will give the new node an associated array of $[\infty, node.val, node.val]$, where $node.val$ is the value of the new leaf node. We then go to the next parent and update the minimum distance between any two elements in the subtree. There will be four computations to find the minimum difference. The minimum difference of the current is either the minimum difference of one of the child subtrees, the current node value subtracted by the maximum of the left subtree or the minimum of the right subtree subtracted by the current node value. Last, we either update the minimum (if we are coming from the left subtree) or the maximum (if we are coming from the right subtree) if necessary. These updates will look like this for each parent in the new leaf's path to the root (where RC and LC refer to the right and left child subtree arrays, respectively):

$$Array = [\min(LC[0], RC[0], |node.val - LC[2]|, |node.val - RC[1]|), LC[1], RC[2]] \quad (1)$$

If we continue these updates through parents until we reach the root, we will be performing these constant operation on at most $logn$ nodes since this is a balanced tree.

Remove-Node-With-At-Most-One-Child: When we are removing a node with at most one child, we must similarly traverse the tree through successive parents updating the array.

If this node had one child, then this child be a leaf since this is a balanced tree. Therefore, we can update this the same way that we would update the tree when adding a leaf with the array $[\infty, node.val, node.val]$. If this node does not have any children, then the we need to update the parent with this loss of information (using equation 1) and continue to move through the parents to the root, updating the nodes' arrays.

Edit-Internal-Node: When editing an internal node, we do not need to update and of the node's subtrees since the arrays of the subtrees do not depend on the parents, only the children. Starting with the new node value, we can calculate the new node's array using the equation 1. From the new node, we can follow the parents to the root while updating each parent.

Rotate: When we must do some form of rotation, we can continue to preserve these data storing arrays. In a rotation, there are two nodes with at most three subtrees. After the rotation, we must recalculate the arrays of the two nodes using the equation 1. Once we recalculate these two nodes, we are finished updating for that rotation. We do not need to update any parents because the rotation will not be adding any new nodes and to any of the subtrees, we are just rearranging them. Since the parent of one of the nodes is the grandparent of another, we know that we do not need to adjust any other arrays. For a single rotation, this subroutine takes constant time.

The total time that this algorithm takes is $O(logn)$ when inserting or deleting. Adding or removing *any* node takes $O(logn)$ from any necessary rotations. Since each rotation takes $O(1)$ and adding a leaf or removing a node with at most one child is also $O(logn)$, we can conclude that inserting or deleting any node will take $O(logn)$ time.

4. (a) We can use a self-balancing binary search tree to store the positions of people on an infinite line. Given that someone at an arbitrary position will throw a knife, we can calculate who will be hit (if anyone) by traversing the tree from the knife thrower's position node to the root of the tree by recursively visiting parent nodes.

   If the parent node has a position that is less than the that of the child which we just visited, then we continue to the parent. If we hit the root and the root is less than the previous node, then the knife will not hit anyone and the thrower is positioned at the front of the line.

   If there is a parent node that we pass through that has a position greater than the previously visited child, then we stop the algorithm and call the SWAT team (or the A-Team for those Mr. T fans) and protect the person that is associated with that parent node.

   Since this is a self-balancing tree, we know that the depth will be $\leq log_2 N$. Since we are only traversing along the depth, this runtime will be on the order of $O(logn)$.

   (b) Now that we have an algorithm to find the next person in line, we can store the height h of a person and the maximum height that is in each of its subtrees. When we traverse the tree, we will still only consider parents that have a position greater than its previously visited child.

   If the height of the parent is less than the height of the knife j, then we look at the maximum height of the right subtree. If the maximum height is less than j then we will move to the next parent. If we evaluate the root and still do not have a possible target, then the knife goes soaring over people's heads and everyone is safe.

   If the height of the parent that has a position greater than the previously visited child's position is greater than j, this person will be hit (call the SWAT team).

   If the height of this parent is less than or equal to j, but there is a person in its right subtree that has a height greater than j, then we visit the right child node. We will now descend the tree from this point. If the left subtree of this node has a maximum height that is greater than j go down that subtree and recursively look at the left child node.

If the left subtree does not have a person that will get hit, but current node would, we need to save the person associated with the current node.

If the left subtree does not have a person that will get hit and the current node does not current node, then the right subtree must have the unlucky victim. In this case, we will call this descent recursively on the right child node.

Like the previous part of the problem, we know that we ascending the tree will take at most $log_2 N$ time. The descent will also take at most $log_2 N$ time since we will only need to take one path down. Since the total time is at most $2log_2 N$, we can say that the runtime is $O(logn)$ as well.

Problem 4: Allocation

1. (a) This scheme is broken up into $log_2 N + 1$ regions all of size N. Therefore, this will require $N(log_2 N + 1)$ memory. This is on the order of $O(NlogN)$ memory.

   (b) This scheme would fail if there are no available chunks to be allocated. The most embarrassing situation would be continuous calls to allocate memory of size $2^{k-1} + 1$ for each value of k where $k > 0$. When $k = 0$, we will fill the region of N with N calls of $allocate(1)$. This sequence will cause our scheme to use a chunk of size 2^k to accommodate this allocation request. This would be the sequence with least total memory used because we are requesting the minimum amount of memory that would satisfy an allocation of chunk size 2^k

   (c) The efficiency of this scheme converges to 1/2 of total memory since at sufficiently large values of k, $\frac{2^{k-1}+1}{2^k} = \frac{1}{2}$.

2. To implement the allocate and free functions in constant time given the aboce scheme, we will use $log_2 N$ stacks for each region of memory with the same chunk size. In each stack, we will store the address of the first bit in each unallocated chunk of memory. When we need to allocate a chunk with an arbitrary size 2^i, we look to the stack with chunks of size 2^i and pop off the first top entry and allocate the desired space at memory address in the stack entry. If the stack is empty, then we know that there are no unallocated chunks of size 2^i and the algorithm will fail. When we are calling the free function on an address, we will take the address of the beginning of the chunk and put that on the top of the stack since it is now unallocated space.

   We use stacks here because stacks are able to quickly add and remove entries. Since it does not matter which chunk we use to allocate only that it is the correct size, we can use stack to return an address in constant time.

3. We consider the following alternate scheme to allocate and free memory. Instead of separating the memory into regions of size N, we leave the entire block of memory together. We will store the memory locations of unallocated and allocated chunks in a linked list that keeps track of where the memory chunk begins and how large the chunk is.

   When we want to allocate a region of memory with an arbitrary size i, we will loop through the linked list and find the chunk of unallocated memory that is closest and greater than or equal to the requested size. We call this scheme best fit since we look for the unallocated chunk that best fits the requested size.

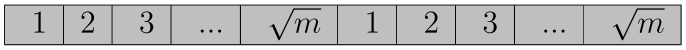
   Whenever the chunk is freed, the entry in the linked list is marked as unallocated. If there are regions directly next to this unallocated region that are also unallocated, we will combine these regions together to make a large chunk of unallocated memory.

4. (a) Consider an algorithm that makes a sequence of m allocation calls of size 1, frees every other address space, and then makes an allocation call of size 2.

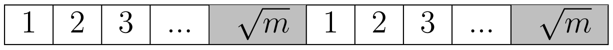
Consider a scheme **1** that leaves the memory in one continuous region. Regardless of the scheme we use to decide where to put a new allocation, if we fill up memory and then call $free$ on every other memory address, we will be left with $m/2$ fragmented chunks of size 1. A subsequent $allocate(2)$ call will cause this scheme to fail.

Now, consider a scheme **2** that breaks up memory into separate regions with varying sizes of chunks. Each of these regions must be of size $< m$ with some regions > 1, otherwise this scheme would fail in a way analogous to the scheme of having one continuous region of memory. Only one of these regions can support a call of $allocate(1)$. Since we've separated the memory into regions, once this region is filled, a subsequent $allocate(1)$ will cause the scheme to fail while there is still memory available in other regions.

(b) Similarly, we can fill the entire memory with $allocate(1)$ calls, but for this sequence, for each block of size $\sqrt{m}$ we will free all memory blocks from 1 up to $\sqrt{m} - 1$. If we group the memory in chunks, allocated memory will go from looking like this:



To looking like this:



This will leave a total of $\sqrt{m}$ single blocks of memory allocated, with $\sqrt{m}-1$ free blocks between them. Thus, a single call to $allocate(\sqrt{m})$ will fail since there's no contiguous region large enough to hold it. Our argument for scheme 2 from *part a* is still valid in this case.

(c) The algorithm that causes any scheme of a memory allocator to fail using at most $\frac{m}{logm}$ memory, where m is the total size of the memory, is as follows. First, we will consider a reparameterization for $\frac{m}{logm}$ where we will use the parameter $s \approx \frac{m}{logm}$. This means that we want to cause a system of $O(slogs)$ (rather than $O(m)$) to fail while only having s memory allocated.

Algorithm: Now, we will allocate $\frac{s}{2}$ memory by calling $allocate(1)$ $\frac{s}{2}$ times. Here, we can initialize a variable L that we can use to keep a count of how many allocated chunks we have at any given point. After we allocate $\frac{s}{2}$ chunks, $L = \frac{s}{2}$. We can now divide each our memory into regions of size $2^0 = 1$. These regions will be necessary in our freeing step of the algorithm (labeled ii.) below. Our next step is to loop through $allocate$ and $free$ calls as follows from $i = 1$ to $i = 4c \cdot logs - 2$ (where $c < 1/8$):

i. We will allocate $\frac{s}{2}$ memory by calling $allocate(2^i)$ $\frac{s}{2}\frac{1}{s^i}$ times. This will allocate $\frac{s}{2}\frac{1}{s^i}$ chunks of size 2^i. We will increase L by $\frac{s}{2}\frac{1}{s^i}$.

ii. We will then free half of the allocated chunks ($\frac{L}{2}$ chunks). To decide which chunks to free, we must first merge every other subdivided region so that now we have $\frac{s}{2}\frac{1}{s^i}$ regions of size 2^i. We will selectively chose to free the larger block of memory that has a **center** in a given region if there are more than 1 blocks with a center in a given region. The center is defined by the address of the middle of the chunk. We will prove that there can only be at most 2 centers in any given region before freeing.
If we have freed less than $\frac{L}{2}$ chunks and every region contains only one center, then we will remove the largest chunks of memory until we have freed $\frac{L}{2}$ chunks.
Once we finish freeing the $\frac{L}{2}$ chunks, we can set $L = \frac{L}{2}$

Proof: First, we must prove that this loop will cause the memory allocator to fail before it is terminated and it will not request more than s memory. Our algorithm will fail when $2^i \cdot L = m$. Since m is on the order of $O(slogs)$, we can also say $2^i \cdot L = c \cdot slogs$ where c is some constant. We can think of 2^i as the size of the regions and L as the number of chunks. If we have *at most* 1 chunk center per region, L also represents the number of regions occupied after each iteration of the algorithm. Therefore, $2^i \cdot L =$ represents the total number of bits that the algorithm has "clogged" with fragmentation. By "clogged" we mean that it is unavailable to the next allocation call of size 2^{i+1}. When the total number of clogged bits is equal to the total memory, the memory allocator will not be able to complete another call of $allocate(2^{i+1})$.

However, we cannot request more than s memory from the allocator. We need to set a bound on our iterator i. To set this bound, we need a better understanding of how to calculate L. For every iteration of i, we are allocating $\frac{s}{2}\frac{1}{s^i}$ more chunks than the previous iteration and then halving this new number of chunks. More generally, we can say:

$$L_i = \frac{L_{i-1} + \frac{s}{2}\frac{1}{2^i}}{2}$$

Now, if we look at the total number of clogged bits,

$$2^i \cdot L_i = 2^i \cdot \frac{L_{i-1} + \frac{s}{2}\frac{1}{2^i}}{2} = 2^{i-1} \cdot L_{i-1} + \frac{s}{4}$$

More generally, this is equivalent to

$$2^i \cdot L_i = i \cdot \frac{s}{4} + L_0 = (i+2) \cdot \frac{s}{4} \text{ , where } L_0 = \frac{s}{2}$$

Returning to our conclusion that the algorithm will fail at $2^i \cdot L_i = c \cdot slogs$, we can solve for i to determine the upper bound of our iterator i.

$$2^i \cdot L_i = (i+2) \cdot \frac{s}{4} = c \cdot slogs \rightarrow i = 4c \cdot logs - 2$$

However, our algorithm cannot request more than s memory at any given time. At the end of any given iteration we can have up to $s/2$ memory already allocated. This means that we cannot request allocations of over size $s/2$ or we need $i \leq \frac{logs}{2}$. If we include this bound to the above bound for i, we can solve for c and give a range of possible values of the constant c.

$$i = 4c \cdot logs - 2 \leq \frac{logs}{2} \rightarrow c < \frac{1}{8} \leq \frac{1}{8} + \frac{1}{2logs} \text{ , for } s > 1$$

Now, we have a bound of i from 1 to $4c \cdot logs - 2$ where $c < \frac{1}{8}$.

Next, we must prove that if there exists a center in a region of size 2^i, we cannot allocate a chunk of size 2^{i+1} with a center in that region. If we have a chunk of size 2^{i+1}, then there needs to be 2^i memory on either side of the center. If one of these chunks is centered in a region of size 2^i, the chunk would need the entire region to allocate each of its 2^i memory halves.

We have proved that our algorithm will reach a value that will cause the memory allocator to fail before we reach an iteration that requests more than s memory, but this is dependent that after each iteration, we only have *at most* 1 center in each region of size 2^i. We will prove that at the end of each iteration, we will have at most 1 center in each region by induction.

As a base case, we look at the $i = 0$ iteration. This iteration finishes with $\frac{s}{2}$ chunks of size 1 in regions of size 1. Since the regions are of size 1, then we cannot have more than 1 center in these regions.

Consider an inductive hypothesis that after the i-th iteration, there is at most 1 center in each region of size 2^i. Now consider the next iteration through the algorithm. We must allocate chunks of size 2^{i+1}. As we proved above, we cannot allocate a chunk of size 2^{i+1} centered in a region of size 2^i that already has a chunk of memory centered in it. After we have allocated our chunks, we have at most 1 chunk in each of the 2^i-sized regions. When we merge every other region together, we are left with at most 2 centers per region of size 2^{i+1}. As our algorithm dictates, we will remove the larger chunk from each of these regions. Thus, after this iteration, we continue to have at most 1 center in each region.

Since we have proved that there is a point that our algorithm will cause the allocator to fail and we proved that our algorithm will hit this point before we request more than s memory, we know that this algorithm will always cause a memory allocator to fail regardless of the scheme.