1 Asymptotic Analysis

RAM (random access machine) model: each memory cell stores one word of data. Memory access and primitive operations are constant time.

Order Notation:

1. $f(n) \in O(g(n))$ if $\exists c, n_0 > 0$ such that $\forall n \geq n_0, f(n) \leq cg(n)$
2. $f(n) \in \Omega(g(n))$ if $\exists c, n_0 > 0$ such that $\forall n \geq n_0, f(n) \geq cg(n)$
3. $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
4. $f(n) \in o(g(n))$ if $\forall c > 0, \exists n_0 > 0$ such that $\forall n \geq n_0, f(n) < cg(n)$
5. $f(n) \in \omega(g(n))$ if $\forall c > 0, \exists n_0 > 0$ such that $\forall n \geq n_0, f(n) > cg(n)$

Important relationships:

1. $f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$
2. $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$
3. $f(n) \in o(g(n)) \iff g(n) \in \omega(f(n))$
4. $f(n) \in o(g(n)) \implies f(n) \in O(g(n))$ and $f(n) \notin \Omega(g(n))$
5. $f(n) \in \omega(g(n)) \implies f(n) \in \Omega(g(n))$ and $f(n) \notin O(g(n))$
6. If $f(n), g(n) > 0, \forall n \geq n_0$, then $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$ and $\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$. As well, $\max\{f(n), g(n)\} \in O(f(n) + g(n))$
7. If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n)$

Limits and Order Notation: If $f(n), g(n) > 0, \forall n \geq n_0$ and $L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, then,

$$f(n) \in \begin{cases} o(g(n)), & \text{if } L = 0 \\ \Theta(g(n)), & \text{if } 0 < L < \infty \\ \omega(g(n)), & \text{if } L = \infty \end{cases}$$

However, it is possible that $f(n)$ is in one of the above orders of $g(n)$ but L does not exist.

Other interesting relationships:

1. $n^x \in o(a^n), \forall x > 0, a > 1$
2. $(\log n)^x \in O(n^y), \forall x, y > 0$

Average case of algorithm A is on all inputs I of size n: $T_A^{\text{avg}}(n) = \frac{1}{|\{I\}|} \sum_{\{I\}} T_A(I)$. Worst case is the max of all $T_A(I)$.

Remember the recurrence relations and common sums and series, and $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$.

2 Priority Queues

2.1 Binary Heaps

Any binary tree with n nodes has height at least $\log(n+1) - 1 \in \Omega(\log n)$.

A max-heap has a structural property: all levels are completely filled, except possibly the last which is filled from the left, and an ordering property: for any node v, its key is larger than any other key in its subtree. Thus, the largest key is at the root. The height of a heap with n nodes is $\Theta(\log n)$.

Heaps are stored in arrays, where the root is at index 0 and the left child of $A[i]$ is at index $2i+1$ and the right child is at index $2i+2$ and the parent is at index $\lfloor \frac{i-1}{2} \rfloor$.

Operations on a max-heap:

1. Insertion: insert the new key as a leaf at the end of the array. Then, perform a fix-up. A fix-up means recursively swapping keys with the key of its parent until either the root is reached or the parent has a larger or equal key. This is $O(\log n)$.
2. DeleteMax: swap the root with the leaf at the last index. Then, perform fix-down on the new root. A fix-down means recursively swapping keys with the larger key of its two children until it is either a leaf or it is larger or equal to the the keys of both children. This is also $O(\log n)$.

A priority queue is a wrapper around a max or min-heap which supports insertion and deleteMax/deleteMin.

2.2 PQ-Sort, Heapsort

PQ-Sort: insert all n items into a priority queue then call deleteMax n times, this is $O(n \log n)$. It is possible to convert an array into a heap (heapify) then repeatedly call deleteMax n times, in place. Thus, this sort is $O(1)$ space.

Heapify: the idea is to convert an array A into a max-heap in-place. A already represents a max-heap, it is just out of order. Starting from the node at the largest index i which has at least one child, call fix-down, decrement i, call fix-down again, and do this until $i = 0$. This is $\Theta(n)$.

Proof of heapify run time:

3 Sorting and Randomized Algorithms

3.1 Quickselect

Selection problem: Given an array A of size n, find the item that would be at index k if A were sorted, the $(k+1)$th largest item. Can do this in $\Theta(n + k \log n)$ using a max-heap. Want $\Theta(n)$.

Quickselect uses two subroutines:

1. choose-pivot: chooses an index p as the pivot
2. partition: takes an array and pivot p, modifies A such that $A[i]$ will contain $A[p]$, the pivot, and $A[0...i-1] \leq A[i] \leq A[i+1...n-1]$. Puts the pivot in the right, sorted spot. Can be done in place in $\Theta(n)$ with Hoare's partition. Returns the space where the pivot is, i.

Quickselect first calls choose-pivot, then partition. If partition returns $i = k$, then it is done. If it returns $i > k$, then recursively calls itself on $A[0...i-1]$ and k. If $i < k$, then calls itself on $A[i+1...n-1]$ and $k - i - 1$.

Analysis: In the worst case, quickselect is $\Theta(n^2)$ if partition keeps placing the item at either end and the recursive call is on an array of size $n-1$. In the best case, only one call to partition is needed and this is $\Theta(n)$. In the average case, quickselect is $\Theta(n)$ and satisfies the recurrence $T(n) \leq cn + \frac{1}{n}\sum_{i=0}^{n-1} \max\{T(i), T(n-i-1)\}$, the i represents all possible indices returned by partition.

3.2 Randomized Algorithms

The goal of randomized algorithms is to shift run time dependency from input to randomness. This way, there is no more such thing as a bad input. Every input could be best or worst case.

Let $T(I, R)$ be the running time of a randomized algorithm with input I and random sequence R. Then, expected run time is $T^{(\exp)}(I) = E[T(I, R)] = \sum_R T(I, R) \cdot Pr(R)$. Worst case expected run time is the max of all $T^{(\exp)}(I)$ for all I of size n. Average is the sum of all $T^{(\exp)}(I)$ for all inputs I of size n divided by the number of inputs I of size n. For well-designed random algorithms, the expected, worst expected, and average expected are all the same.

Randomized quickselect: Choose a random pivot. There is a $\frac{1}{n}$ chance this pivot goes into index i, so the analysis is the same as for average case regular quickselect, $\Theta(n)$ expected. In comparison, if choosing the rightmost element as the pivot, since this is deterministic, the probability is either 0 or 1 that it goes in index i. Choosing a random pivot makes the run time independent of the input.

Expected vs average run time: A deterministic algorithm has only one run time for any specific instance I of size n whereas a random algorithm can have many different run times. The expected run time is the sum of all the different possible run times on a specific I and random sequence R multiplied by the respective probability of R occuring. The average run time is the sum of all the possible run times on an instance I of size n averaged over the number of instances of size n. Random algorithms have an expected average run time. Deterministic algorithms have just an average run time.

3.3 Quicksort

Quicksort calls choose-pivot, then partition. If partition returns i, then quicksort recursively calls itself on $A[0...i-1]$ and $A[i+1...n-1]$.

Analysis: In the worst case, partition returns i on either end, so the recursive calls are of size $n-1$. The recurrence is $T(n) + T(n-1) + \Theta(n)$, or $\Theta(n^2)$. In the best case, i is roughly in the middle with recurrence

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$, or $\Theta(n \log n)$. In the average case, since $\frac{1}{n}$ of permutations have partition return a certain i, the recurrence is $T(n) = cn + \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1))$. This resolves to $\Theta(n \log n)$. The average case can be proved using the average height of the recursion tree, which is $O(\log n)$.

By choosing a random pivot, quicksort will have expected run time $\Theta(n \log n)$. Auxiliary space is $\Theta(n)$ worst case for the recursion depth. In randomized quicksort, the input doesn't matter, all that matters is that partition places the randomly chosen pivot into the middle of the array. Thus, average expected, average best, and average worst are all the same $\Theta(n \log n)$. If the pivot is chosen using the median of median strategy, then the worst case is $\Theta(n \log n)$.

3.4 Lower Bound on Sorting

In the comparison model, which means two elements are compared at a time and data is swapped and moved around, all sorting is $\Omega(n \log n)$ in the worst case. This can be seen by the decision tree, which has $n!$ leaves, one for each permutation of the array, and the height of the tree, which is in $\Omega(\log(n!))$. The height represents the minimum number of comparisons needed to reach some sorted order in a leaf. Since $\log(n!) \in \Theta(n \log n)$, then the result follows.

3.5 Non-Comparison Based Sorting

Suppose the numbers are in base R and each have m digits.

Bucket sort: put the numbers into R buckets based on one digit, then combine together. This is stable and sorts one digit. Uses $\Theta(n)$ space and $\Theta(n + R)$ time.

Count sort: sorts based on one digit as well, uses a count array of size R to count the number of items of each number and an idx array to keep track of the index to put each number in the sorted array. Is stable, uses $\Theta(n)$ space and $\Theta(n + R)$ time.

MSD-Radix sort: sorts multi digit numbers, starting from the most significant (leftmost) digit. In place, has lots of recursion, and uses an auxiliary digit sorting algorithm. This digit sorting algorithm does not necessarily need to be stable but MSD-Radix sort itself will only be stable if the digit sorting algorithm is stable.

LSD-Radix sort: also sorts multi digit numbers, starting from the least significant digit. In place, no recursion, but needs a stable digit sorting algorithm, such as count sort.

Under certain input, count sort and radix sort are $o(n \log n)$. Overall, both MSD and LSD Radix sort use $\Theta(n + R)$ space and $\Theta(m(n + R))$ time.

4 AVL Trees

BST: has $\Theta(\log n)$ height in the average case. When deleting, if the node has fewer than two children, then remove it and move the child up. If the node has two children, the swap the key with the successor, then remove the original successor leaf. All binary trees have height $h \in \Omega(\log n)$. Thus, to show that a certain type has $h \in \Theta(\log n)$, it is only necessary to show $h \in O(\log n)$.

AVL: balance factor is defined as the height of the right subtree minus the height of the left subtree, where an empty tree has height -1 and a tree of one node has height 0. The balance factors of all nodes must either be $-1, 0, 1$. An AVL tree on n nodes has height in $\Theta(\log n)$. This is proved with a recurrence of $N(h)$ for the least number of nodes in an AVL tree of height h. Thus, search, insert, and delete are all $\Theta(\log n)$ in the worst case.

4.1 Insertion

1. Insert like in a regular BST, let z be the new leaf inserted
2. While z is not the root, move upwards through the parent pointer
3. If there's an imbalance, let y be the taller child of z, breaking ties arbitrarily, let x be the taller child of y, break ties to favor left-left/right-right. Then, call restructure on x

4.2 Rotations

There are four types of imbalances. Let x be the node restructure was called on, let y be its parent, let z be the parent of y. In all four situations, the middle key of x, y, z will become the new root.

1. right rotation: right rotation on z (left left case, y is left child of z, x is left child of y)
2. double right rotation: left rotation on y, right rotation on z (left right case)
3. double left rotation: right rotation on y, left rotation on z (right left case)
4. left rotation: left rotation on z (right right case)

4.3 Deletion

1. Remove like in a regular BST, let z be the parent of the deleted leaf
2. While z is not the root, move upwards through the parent pointer
3. If there's an imbalance, let y be the taller child of z, breaking ties arbitrarily, let x be the taller child of y, if there is a tie, must choose choose the left child of y if y is a left child of z, same for the right. Call restructure on x

Deleting a node from an AVL tree will cause the overall height to decrease by at most 1. Unlike in insert where at most one restructure is needed, more than one may be needed for deletion, so continue moving upwards until z is the root.

5 Skip Lists and Re-ordering

5.1 Skip Lists

A regular BST has average height $O(\log n)$. The goal is to shift this to expected height $O(\log n)$ using randomization.

A skip list contains $h+1$ levels of ordered linked lists $S_0, \ldots, S_h$. Each list starts with a negative infinity sentinel node and ends with a positive infinity sentinel. S_0 contains all the keys and values in non-decreasing order, S_h contains only the two sentinels, other levels contain some keys but no values. Each key belongs to a tower of nodes. For example, if a key goes up to S_3, its height is 3. Each node has links to the next node in its level and the node below it. Operations:

1. Search: returns a stack of nodes, the first node to be inserted in the stack is to the top-left negative infinity sentinel node. The rest of the nodes in the stack are the nodes reached when traversing. A search goes as far as it can on the current level until it reaches a key which is equal or greater than the target key, then the node gets pushed on the stack and the traversal moves down. Check whether the node after the top of the stack has key equal to the target key.
2. Insert: toss a coin until you get tails. If you have i heads, then your tower will be of height i. Increase the height of the skip list if needed such that $h > i$. Search for the key in the skip list and the top $i+1$ nodes of the stack will be the predecessor to the new node with the new key. Note that $i+1$ new nodes will be inserted into the skip list.
3. Delete: search for the key, if a node on the returned stack has the next node equal to the key, then remove it from the list. Decrease the height such that only the top level has only the two sentinel nodes.

Analysis: Expected $O(n)$ space, $O(\log n)$ height with high probability since a skip list with n items has height at most $3\log n$ with probability at least $1 - \frac{1}{n^2}$. In S_i, it is expected that there are $n \cdot (\frac{1}{2})^i$ nodes.

Search, insert, and delete are all expected $O(\log n)$. At each level, it's expected that there are less than or equal to 2 rightwards moves along the same level. This is because a node can only go rightwards into a top of a tower, and the next node is a top of the tower rather than just part of the tower with probability $\frac{1}{2}$. Thus, a node which is not the top of the tower is expected within the next 2 rightwards moves.

5.2 Re-ordering

Optimal static ordering: order A such that the most frequently searched for is at the front, the expected access cost is minimized. Sort by non-increasing access probability for each item. Requires knowing the access probabilities before hand.

Move to front: on a successful search, move it to the front. Transpose: on a successful search, swap the item with the one preceding it. If already at the front, do nothing. MTF works well in practise, transpose does not adapt well to changing access patterns. Implemented in arrays or lists.

6 Interpolation Search and Tries

6.1 Lower Bound on Searching

In the comparison based model, $\Omega(\log n)$ comparisons are required to search a dictionary of size n.

Proof: Searching can be visualized as a binary decision tree with n leaves. Thus, the height is equivalent to the number of comparisons that must be made to find the target key. A binary tree with n leaves has height in $\Omega(\log n)$.

6.2 Interpolation Search

Variation of binary search, where the relative position is guessed, rather than cutting out only half of the search space at each step. Works best if the keys are uniformly distributed. It can be shown that the array we recurse into has expected size $\sqrt{n}$, so the recurrence is $T^{(\text{avg})}(n) = T^{(\text{avg})}(\sqrt{n}) + \Theta(1)$. This resolves to $O(\log \log n)$ run time. Worst case is $\Theta(n)$. An example of the worst case is if the array is $\{0, 1, 2, 3, 4, 5, 5000\}$ and we are searching for $k = 5$.

6.3 Tries

Assume the strings in the dictionary are prefix-free. This can be guaranteed by appending a \$ terminator to the end of each word. Keys are stored only in the leaf nodes and edges are labelled with 0, 1, or \$. $\Theta(|x|)$ for search, insert, and delete, where x is the length of the binary string/key. Both insert and delete use search as a subroutine first and delete will recursively delete upwards all nodes until it reaches a node with two or more children.

Variations of tries:

1. Don't need to store keys in the leaf. The key is implicitly created along the path to the leaf
2. Stop adding nodes to the trie as soon as the key is unique
3. Use flags in each node to indicate if a key ends at that node, allows prefixes and removes the need for the \$ terminator

Compressed trie: Each node stores an index for the current comparison index. If there are n keys, there are at most $n - 1$ internal nodes. When a leaf is reached, the key it holds must be compared with the target key, since it's possible that indices were skipped along the path to this leaf. Deletion involves searching, removing the node, then compressing upwards. Insertion involves searching and uncompressing along the way, inserting the key, then compressing upwards. All operations take $O(|x|)$ time.

Multiway trie: Same as a regular trie but there may be up to $|\Sigma|+1$ children for each node, where the $+1$ represents the \$ terminator child. Run times are $O(|x| \cdot f)$ where f is the time to find the appropriate child. May be $O(1)$ if the children are stored in an array/hash table, may be $O(|\Sigma|)$ if stored in a list. There's a space-time tradeoff here.

7 Hashing

Direct addressing: Use a hash table of size M, put the key value pair with key k in index $k-1$. Requires $0 \leq k < M$, waste of space.

Hashing: map keys to small range of integers then use direct addressing. Keys come from some universe U, the hash function $h : U \to \{0, 1, \ldots, M-1\}$, hash table T is an array of size M, and key k is stored in $T[h(k)]$.

Load factor: $\alpha = \frac{n}{M}$. Rehash when load factor is too large or small so that space is always $\Theta(n)$.

7.1 Collision Resolution

1. Chaining: Table entry is an unsorted linked list, a bucket. When a collision occurs, add the key to the front of the list. Assuming uniform hashing, average bucket size is α. Searching is average case $\Theta(1+\alpha)$, worst case $\Theta(n)$. Insert is worst case $\Theta(1)$, delete is the same as search, space is $\Theta(M+n) = \Theta(\frac{n}{\alpha}+n)$. If $\alpha \in \Theta(1)$, then average costs are $O(1)$ and space is $\Theta(n)$. Keep α in $\Theta(1)$ by rehashing when $n < c_1 M$ or $n > c_2 M$ for some $0 < c_1 < c_2$ constants. Rehashing costs $\Theta(M+n)$.
2. Open addressing: one key can go in more than one space in T

7.2 Open Addressing

The first two types result in a probe sequence and use lazy deletion:

1. Linear probing: $h(k,i) = (h(k)+i) \mod M$, for some hash function h and i starts at 0 and increments by 1 in each collision. If i reaches M, then the table is full and cannot insert. For search, keep going past slots which contain different keys or were lazy deleted until either the target key is found or an empty space is reached. Has good memory locality which reduces block transfers but may cause clustering issues.
2. Double hashing: Uses two independent hash functions h_1, h_2, $h_2(k) \neq 0$ and $h_2(k)$ is relative prime with M for all keys, M is prime. Then, $h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod M$. i starts off at 0 and increments by 1 in each collision, just like in linear probing. Note that double hashing is a generalization of linear probing where $h_2(k) = 1$ for all k. Worst case is $\Theta(n)$. If $h_2(k)$ is chosen such that $h_2(k)$ is coprime with M for all k, then a probe sequence of k will reach every space in the table.
3. Cuckoo hashing: Uses two independent hash functions h_0, h_1 and two tables T_0, T_1. Key k can only be in $T_0[h_0(k)]$ or $T_1[h_1(k)]$. Search and delete are constant time worst case, insert initially puts the key into $T_0[h_0(k)]$. If this spot is occupied by key k', kick it out and put k' into $T_1[h_1(k')]$. Repeat until either some spot is empty or the insertion is aborted after $2M$ failed attempts and rehashing with a larger M and new functions is done. Insert is expected to be constant time if α is small enough. M is the size of one table so there are a total of $2M$ spaces and $\alpha = \frac{n}{2M}$ where n is the total number of keys. Not necessary to do lazy deletion. Requires $\alpha < \frac{1}{2}$ to get $O(1)$ average insert.

There are two basic types of hash functions. One is modular, $h(k) = k \mod M$ and the other is multiplicative, $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$, where $0 < A < 1$ is a decimal. Note that $(kA - \lfloor kA \rfloor) \in [0, 1)$ computes the fractional part of kA.

7.3 Analysis, Independence, Randomization

Independent hash functions: Two functions h_1, h_2 are independent if $P(h_1(k) = i)$ and $P(h_2(k) = j)$ are independent, neither function relies on the other. Choosing two modular functions may be bad. Eg, if $h_1(k) = k \mod 8, h_2(k) = k \mod 4$, then these are not independent since if $h_1(k) = 0$, then it must be that $h_2(k) = 0$ for that k. Any time the mods are not coprime, the functions are not independent. To improve on this, h_2 in double hashing is typically multiplicative. A good indication that h_1, h_2 are independent is that if $h_1(k_1) = h_1(k_2)$ for two keys k_1, k_2, then $h_2(k_1) \neq h_2(k_2)$.

Moreover, if there is a key k such that $\gcd(h_2(k), M) = d$, then a probe sequence for k will only reach $\frac{1}{d}$th of the table. This was proven in tutorial 7.

Summary of run times: All open addressing schemes have $\alpha < 1$ and cuckoo hashing has $\alpha < \frac{1}{2}$. All operations: search, insert, delete is $O(1)$ average if the hash function is uniform and α is sufficiently small, but the worst-case run time is usually $\Theta(n)$.

Randomized hash function: When initializing or re-hashing, choose a prime $p > M$ and random numbers $a, b \in \{0, \ldots, p-1\}, a \neq 0$ then use $h(k) = ((ak + b) \mod p) \mod M$. Can prove that for any fixed $x \neq y$, the probability of collision is at most $\frac{1}{M}$ so the expected run time is $O(1)$. Thus, average case performance has been shifted to expected performance.

Chaining guarantees that insertion will work without needing to rehash, linear probing and double hashing will complete insertion unless the table is full. Cuckoo hashing may fail in insertion even if the tables aren't full and the hash functions are chosen well.

8 Range Search

Motivation: Find all points in $d-$dimensional space which fit in a certain range for each of the d dimensions. Can use a partition tree, which is a tree of n leaves where each leaf is a point and the internal nodes are a region (for example, kd-trees, quadtrees) or use a multi-dimensional range-tree. Note that since a range search returns s points, $0 \leq s \leq n$, then range search is never in $o(s)$.

8.1 Quadtree

Suppose there are n points in some square R. The root of the range tree corresponds to the region R. If R has 0 or 1 points, then its a leaf which stores the point or is an empty node. Or else, partition R into four equal quadrants: $R_{NE}, R_{NW}, R_{SW}, R_{SE}$. The root has four subtrees for each of these quadrants, ordered in the same way from left to right. Recursively build the tree in this manner. If a point is on a boundary between quadrants, it belongs to the upper-right quadrant. One alternative to forcing every internal node to have four subtrees, where some may be empty, is to omit the empty subtrees but label the edges. Operations:

1. Searching for a point is similar to BST search.
2. Inserting a point first performs a search, then splits if there are two points in the region.
3. Deleting first performs a search, then if the parent has only one other point, the parent is deleted and the other point is promoted. Repeat recursively upwards for all ancestors which now only have one point.
4. Range search: Assume each node in the tree stores the square it is associated with and a search rectangle is passed in.

Analysis: Let $\beta(S) = \frac{\text{sidelength of } R}{d_{\min}}$ be the spread factor for a set of points S. Then, the height $h \in \Theta(\log \beta(S))$. Worst case for building the initial tree is $\Theta(nh)$, worst case for range search is $\Theta(nh)$, even if no points are returned. Worst case comes when points are in pairs, where each pair has points that are very close together but the pairs are far apart from one another. Then, there would be $\Theta(nh)$ nodes in the tree. However, it is typically much faster. Overall, quadtrees are easy to compute and handle but space is potentially wasteful unless the points are well-distributed. One variation that could be used is to allow more than one point in a leaf, this results in fewer splits.

For any n, it is possible to place them on a plane such that there is a rectangle A such that a 2D range query on A returns nothing but forces the search to visit all nodes in the tree.

8.2 kd-tree

The idea is to split regions, alternating between vertical and horizontal splits, such that half of the points are in each half. Each internal node keeps track of its split line and each leaf is a point. For 2D data, the root node splits on x and might contain something like $[x < p_5.x?]$. Operations:

1. Construction: Use quick-select to find the x or y coordinate which is the $(\lfloor \frac{n}{2} \rfloor + 1)$th largest. Since this must be done at each level, the overall expected time for construction is $\Theta(h \cdot n)$, can be reduced to $\Theta(n \log n + h \cdot n)$ by pre-sorting. Assuming the split at each node roughly partitions the nodes into two halves, $h(n) = \max(h(\lfloor \frac{n}{2} \rfloor) + h(\lceil \frac{n}{2} \rceil)) + 1$, so $h \in \Theta(\log n)$. Therefore, construction is $\Theta(n \log n)$ and the tree takes $O(n)$ space. If all the points are in a line, for example, all on $y = c$, then $h(n) \approx h(\frac{n}{2}) + 2, h(n) \in \Theta(\log n)$. If the points are in an L shape, then ?????
2. Searching for a point is the same as BST search.

3. Inserting or deleting a point first involves searching for it then appropriately creating a new leaf or removing internal nodes with only one child. After certain inserts or deletes, the height might no longer be $O(\log n)$, then the entire tree is re-built.

4. Range search is very similar to Quadtree range search. Each internal node has an associated R region, there's a query rectangle A and if $R \not\subseteq A$ and $R \cap A$ is not empty, then both sides of the split R are searched.

Range search analysis: Complexity is $O(s + Q(n))$, where s is the number of points reported and $Q(n)$ is the number of nodes in the tree which are visited such that its corresponding region R is neither $R \subseteq A$ nor $R \cap A \neq \emptyset$. It can be shown that $Q(n) \leq 2Q(\frac{n}{4}) + O(1), Q(n) \in O(\sqrt{n})$, so overall, range search is $O(s + \sqrt{n})$.

$d-$dimensional kd-tree: The root partitions based on the first coordinate, its subtrees partition on the second coordinate, the $d-1$ level partitions based on the last coordinate, then the d level partitions based on the first coordinate again. Storage is $O(n)$, construction is $O(n \log n)$, and range search is $O(s + n^{1-\frac{1}{d}})$, assuming $o(n)$ points share coordinates and d is a constant.

8.3 Range tree

Range trees use more space than Quadtrees or kd-trees, but have a much faster range search. It is made of a BST T that sorts by x-coordinate and each node v has an auxiliary tree $T(v)$ which is a BST that sorts all the points in the subtree rooted at v by y-coordinate.

1D range search on regular BST: Suppose we want all keys between k_1 and k_2. Let P_1 be the path that comes from searching for k_1 and let P_2 be the path for k_2. Then, report all nodes that are to the right of P_1 and left of P_2 and check each node on a path, the boundary nodes, individually to see if its between k_1 and k_2. Assuming the BST is somewhat balanced, there are $O(\log n)$ boundary nodes. An allocation node is such that it's not in P_1 or P_2 but the parent is in P_1 or P_2 and if the parent is P_1, it must be the right child, if the parent is P_2, it must be the left child. All nodes in subtrees rooted at an allocation node are reported. Therefore, the overall run time is $O(\log n + s)$ where s is the number of points reported.

Range tree structure: Assume the primary tree that sorts by x is balanced and has height $O(\log n)$. Also, the root of each auxiliary tree may not be the same as the root of its corresponding subtree in the primary tree. The primary tree uses $O(n)$ space. Each of the n nodes has $O(\log n)$ ancestors in the primary tree, thus, it is present in $O(\log n)$ auxiliary trees. So the total space is $O(n \log n)$.

Operations:

1. Search: same as in a BST.

2. Insert: first, insert the point by x coordinate in the primary tree. Then, walk back up to the root and insert into the auxiliary tree by y of each node it reaches in this walk.

3. Delete: same as insertion

4. Range search: suppose we want to find points in $[x_1, x_2] \times [y_1, y_2]$. First, search for x_1 and x_2 in the primary tree and keep track of the path, these are the boundary nodes. For each allocation node, search for $[y_1, y_2]$ in the auxiliary tree, since all these nodes are guaranteed to have x coordinate in the interval $[x_1, x_2]$. For each boundary node, individually check if its in the range.

We want the BSTs to be balanced but insertion and deletion will be slow with AVL trees. Thus, allow certain imbalances then rebuild the entire structure when needed.

Analysis of range search: $O(\log n)$ to find the boundary and allocation nodes in the primary tree and there are $O(\log n)$ allocation nodes. For each allocation node v, takes $O(\log n + s_v)$ where s_v is the number of points reported, using range search on the auxiliary BST. Every point in an auxiliary tree of an allocation node is only in one such auxiliary tree. Thus, the sum of all the points reported for each allocation node is

less than or equal to the total number of points reported. So overall, this is $O(s + \log^2 n)$ where s is the total number of points reported. Also, $O(\log n)$ to go through each boundary node but this does not change the complexity.

Higher dimension d range tree: In $d-$dimensional space, the space of a range tree is $O(n(\log n)^{d-1})$, construction is $O(n(\log n)^{d-1})$, and range query is $O(s + (\log n)^d)$. In comparison to kd-trees, range trees take more space and are slower to construct but have faster range query.

8.4 Summary

Quadtrees are simple, good only if the points are well-distributed, and wastes space for higher dimensions. kd-trees are linear in space, have query time $O(s + \sqrt{n})$, insert and delete can destroy balance, and do not work as well for many repeated coordinates. Range trees have the fastest range search $O(s + \log^2 n)$, wastes space, and have complicated insert and delete.

Range searching can solve many problems relating to searching for multi-dimensional data.

9 String Matching

Want to search for pattern P of size m in a text T of size n by returning the index of first occurrence or fail, if P is not in T.

A guess is an index $i, 0 \leq i \leq n - m$, such that P may start at $T[i]$. A check is a position $j, 0 \leq j < m$ where $T[i + j]$ is compared with $P[j]$. A correct guess requires m checks, an incorrect guess may require fewer.

Brute force for string matching is $\Theta(nm)$. Can improve on this by either doing preprocessing on the pattern P, like is done in Rabin-Karp, Boyer-Moore, KMP, or do preprocessing on the text T, like is done with suffix trees.

9.1 Rabin-Karp

Choose a random, large prime p. Calculate the modular hash of the pattern P with p. Then, calculate rolling modular hashes of substrings of T of length m with p. If it matches with the pattern hash, compare character by character. If not, move on. Subsequent rolling hashes can be calculated in $O(1)$ given the previous one. Expected running time is $O(m + n)$, might be $\Theta(mn)$ if we keep having hashes which match but the pattern does not match but this is rare. Good when searching for multiple patterns in the text, since each pattern only needs to be hashed once.

9.2 Boyer-Moore

Uses a last occurrence function/array of P which maps characters to the index of its last occurrence, put -1 for characters which do not appear in P. Compare with P backwards, start from the end of P. If there is a mismatch, can skip large chunks of the text which we know will not match the pattern. For example, if $T[i + j]$ is a mismatch with $P[j]$ but $T[i + j]$ is not anywhere in P, i will increment to $i + j + 1$.

On typical English text, BM is the fastest and probes approximately 25% of letters in T. Worst case with this bad character heuristic is $\Theta(mn + |\Sigma|)$ but typically faster. The $\Theta(|\Sigma|)$ comes from creating the last occurrence array. An example of a worst case is $P = abbb, T = bbbbbbbbbb$.

9.3 Knuth-Morris-Pratt (KMP)

Uses a failure function/array $F[0...m - 1]$ where if $P[j] \neq T[i]$, then set j to $F[j - 1]$ if $j > 0$, or otherwise increment i. The idea is that when we fail to match at $P[j]$, shift j such that the prefix matches with the suffix of what we were just comparing, $P[1...j]$, so we don't have to compare these substrings again. $F[j]$ should store the length of the longest prefix of P that is a suffix of $P[1...j]$. Note that this is $P[1...j]$ and not $P[0...j]$ because $P[0...j]$ is a prefix of the entire $P[0...j]$ and in case of failure, we must move forwards by at least 1. It's easy to compute this using a table where the columns are $j, P[1...j]$, the longest prefix which is a suffix of $P[1...j]$, and $F[j]$. Denote empty string as $\wedge$. The failure array can be computed in $\Theta(m)$ by using a variation of KMP. It is $\Theta(m)$ because in the algorithm $2i - j$ increases in each iteration and $2i - j \geq 0$ at the start and $2i - j \leq 2m$ at the end. By a similar analysis, the overall algorithm is $\Theta(n + m)$, even in the worst case.

9.4 Suffix Trees

Suffix trees will preprocess text T rather than pattern P. This is good for searching for many different P in one single T. The idea is that P is a substring of T iff it is a prefix of some suffix of T. So, store all suffixes

of T with a \$ terminator in a compressed trie, then search for P in this trie. If T has length n, there are $n+1$ suffixes, including the empty string, so there are $n+1$ leaves in this trie, the extra $+1$ is for the empty \$ string. The suffix tree can be built in $\Theta(n^2)$ by inserting each suffix into the trie. It is possible to build in $\Theta(n)$. Since this is a compressed trie, we need to compare P with one of the leaves of the node the search ends at. The search stops at either a leaf, or if the index in the node is greater than or equal to m.

9.5 Summary

	Brute-Force	**KR**	**BM**	**DFA**	**KMP**	**Suffix trees**
Preproc.	—	$O(m)$	$O(m+\|\Sigma\|)$	$O(m\|\Sigma\|)$	$O(m)$	$O(n^2)$ ($\rightarrow O(n)$)
Search time	$O(nm)$	$O(n+m)$ (expected)	$O(n)$ (often better)	$O(n)$	$O(n)$	$O(m)$
Extra space	—	$O(1)$	$O(m+\|\Sigma\|)$	$O(m\|\Sigma\|)$	$O(m)$	$O(n)$

Typo in the above table: space for BM is $O(|\Sigma|)$.

10 Compression

10.1 Basics

Source text S: original data, from source alphabet Σ_S.

Coded text C: encoded data, from coded alphabet Σ_C, usually $\{0, 1\}$.

Encoding schemes which try to minimize the size of C perform data compression. The compression ratio used to measure data compression is $\frac{|C| \cdot \log|\Sigma_C|}{|S| \cdot \log|\Sigma_S|}$

Logical compression uses the meaning of the data whereas physical compression only knows the physical bits of the data. Lossy compression achieves better compression ratios but the decoding is approximate and S is not always recoverable whereas lossless compression decodes S exactly. This course focuses on lossless physical compression.

Encoding $E : \Sigma_S \to \Sigma_C^*$. For each $c \in \Sigma_S, E(c)$ is the codeword of c. Fixed length code, variable length code refers to whether $E(c)$ is the same length for all $c \in \Sigma_S$.

Not all encodings E are prefix-free. A prefix-free E is such that $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$. Any prefix-free code is uniquely decodable.

10.2 Huffman

The goal is to minimize the encoding length by assigning more frequent characters shorter codes.

Huffman encoding algorithm:

1. Count frequency of each character $c \in \Sigma_S$ in S.
2. For each c, construct a trie containing only c.
3. Assign a weight to each trie, where the weight is the sum of frequencies of all characters in the trie.
4. Merge two tries with the smallest weight using a new root, new weight is the sum. Assign the left trie with 0, right trie with 1. Thus, $\Sigma_C = \{0, 1\}$.
5. Repeat until there is only one trie left.

Use a min-ordered heap so that removing the two tries with the minimum weight is fast and inserting the new trie into the collection is easy. The constructed trie is not unique so the trie must be passed along with C in order to decode. The trie is optimal in the sense that C is shortest.

Encoding run time is $O(|\Sigma_S|\log|\Sigma_S|+|C|)$, the $|\Sigma_S|\log|\Sigma_S|$ comes from building the trie from the min-ordered heap of tries, the $|C|$ comes from encoding from the trie. Encoding from a trie, where a hashmap is first built that maps characters to leaves in the trie, is $O(|T|+|C|) = O(|\Sigma_S|+|C|)$, where $|T|$ is the size of the trie. So overall, this is $O(|\Sigma_S|\log|\Sigma_S|+|\Sigma_S|+|C|) = O(|\Sigma_S|\log|\Sigma_S|+|C|)$.

Decoding run time is $O(|C|)$. Since encoding and decoding have different run times, Huffman is asymmetric. The dictionary is static; same throughout the whole process of encoding and decoding.

10.3 Run-Length Encoding

RLE is a multi-character encoding since multiple source-text characters receive one code-word. Source and coded alphabet are both $\{0, 1\}$. The goal is to make use of long runs of 0 or 1.

RLE encoding algorithm: $O(|S|+|C|)$

1. Output first bit of S (this step is easy to forget).
2. Count first run of 0 or 1's. If this run has length k, output $\lfloor \log k \rfloor$ 0's then output k in binary.
3. Repeat until the end of S is reached.

The binary representation of k must start with a 1. Moreover, $\lfloor \log k \rfloor + 1$ bits are required to represents k in binary. If j 0's are read, this indicates the next $j + 1$ bits are for the binary representation of k, where $j = \lfloor \log k \rfloor$. As well, $\lfloor \log k \rfloor$ and the binary representation of k can be calculated simultaneously while building C, there is no extra cost associated with it.

RLE decoding algorithm: $O(|S|+|C|)$

1. Remove first bit, this bit indicates whether S starts with a 0 or 1.
2. Perform decoding in reverse; count the number of 0's, get the binary representation of k, append k 0's or 1's.

If S were all 0's of length n, then C would be $1 + \lfloor \log n \rfloor + \lfloor \log n \rfloor + 1 = 2\lfloor \log n \rfloor + 2 \in o(n)$ bits.

If the run length k is 2 or 4, the code is actually longer (expansion). There is no compression until the run length is at least 6. A run of length k not at the start of S is encoded into $2\lfloor \log k \rfloor + 1$ bits.

10.4 Lempel-Ziv-Welch

The goal is to optimize for repeated substrings without knowing what they are beforehand. LZW works with a stream and does not need to go through the entire S first like Huffman does. LZW is adaptive encoding because the dictionary changes during both encoding and decoding. D_0 is the original dictionary, usually ASCII. For $i \geq 0, D_0$ is used to determine the ith character output. Then, both the encoder and decoder update D_i to D_{i+1} by adding a single multi-character substring pattern, starting from number 128. Every number in C refers to a single character in Σ_S or a substring. Numbers in C are usually converted to a bit-string with fixed-width encoding using 12 bits. Thus, 2^{12} patterns are possibly recognizable. LZW is a fixed-length encoding, whereas Huffman and RLE are variable-length. Especially good for English text.

Encoding algorithm:

1. Start with D_0 of just ASCII letters, represented as a trie.
2. Parse trie to find longest prefix x already in D_i, output the number associated with the path for x in the trie. Update the trie with the letter following x along the same path, this adds a new leaf which is a number. Restart with this letter.

Special case when decoding: If a code (number) is not in D yet, then it encodes the previous string + the first character of the previous string.

10.5 bzip2

bzip2 uses text transform. If S has repeated longer substrings, then C after BWT will have long runs of characters. This makes C a good candidate to then put through MTF. Since this text has long runs of characters, MTF will give a code that has long runs of zeros. This is then good for RLE compression. Lastly, use Huffman encoding for additional compression.

10.6 MTF heuristic

MTF takes advantage of data locality: the current character will probably be seen again soon. Is an adaptive text-transform algorithm. If $|\Sigma_S| = m$, then $\Sigma_C = \{0, 1, \ldots, m-1\}$. Encoding and decoding work exactly the same way and make use of an array L containing the characters in Σ_S, in some pre-agreed, fixed order before MTF. For example, L might be of size 26, where $L[i]$ is the ith letter of the English alphabet. The output C is such that $C[i] = j$, where $S[i]$ was at index j in L.

A run in S encodes to some number then a run of 0's in C. A run in C could mean a repeated substring in source S. For example, 1111 means two letters are alternating back and forth.

10.7 Burrows-Wheeler Transform

C has the same letters as S, just in a different order. The idea is that although C isn't compressed, it is easily compressible with MTF. S ends with special \$ character and encoding requires all of S at once, does not work with streams. Decoding is more efficient than encoding.

BWT encoding algorithm: $O(n^2)$ using MSD radix sort, often better

1. Write all cyclic shifts of S then sort the cyclic shifts, where \$ comes first.
2. The last character from each row of the sorted shifts is C.

BWT decoding algorithm: $O(n)$

1. Create an array A of size n which stores pairs, where $A[i]$ is $(C[i], i)$.
2. Stable sort A by the first entry, the character.
3. Set j to be such that $C[j]$ is \$, this is a linear search through C.
4. Set j to be the second entry of $A[j]$ and append $C[j]$ to S.
5. Repeat until $C[j]$ is \$, S now is the decoded text.

Both encoding and decoding use $O(n)$ space.

If S has repeated longer substrings, then C after BWT will have long runs of characters.

Note that the order in which you write the cyclic shifts does not matter. There is only one unique way to sort them, and one unique C anyway. Moreover, in the decoding, A must be stable sorted because the ith letter c in the sorted array should correspond to the ith occurrence of c in the code C. This is what the "disambiguate by row index" part is doing.

11 External Memory

External Memory Model: Unbounded external memory (disk), internal memory (RAM, cache) of size M, data is transferred between external and internal memory in slow block transfers, data is transferred between internal memory and CPU very quickly. We want to minimize block transfers.

11.1 External Sorting

Suppose we want to sort an array A of size n where n is very large and A is stored in blocks in disk. Heapsort has bad memory locality, merge sort will work better. This sorting uses $d-$way merging, which is a method of merging d sorted arrays using a priority queue. Let B be the block size, let M be the size of internal memory.

1. Create $\frac{n}{M}$ sorted runs of length M, results in $\Theta(\frac{n}{B})$ block transfers.
2. Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using $d-$way merge. The -1 is there to give space in internal memory for sorted output to write back to disk. The sorting is done in internal memory by partitioning into spaces of size B, where each space is reserved for some elements of one sorted run. These elements are eventually removed as they are sorted, new elements in the run is added when the space is empty.
3. Repeat step 2 until all original $\frac{n}{M}$ runs are sorted. Since groups of d runs were sorted into one longer run, after this round, there are a factor of d fewer runs to sort. $\Theta(\frac{n}{B})$ block transfers are needed in each round of merging. Therefore, there are overall $\log_d(\frac{n}{M})$ rounds of merging to create one sorted array, so the number of block transfers is $O(\log_d(n) \cdot \frac{n}{B})$. As well, scanning n elements requires $\Omega(\frac{n}{B})$ block transfers. $d-$way merge sort with $d \approx \frac{M}{B}$ is optimal, up to constant factors.

11.2 2-4 Tree

External dictionaries: Tree-based dictionaries typically have poor memory locality. In an AVL tree, $\Theta(\log n)$ blocks are loaded in the worst case. B-trees improves on this.

2-4 tree: Special case of B-trees. Is a balanced search tree where every node is either a 1-node (1 key, 2 possibly empty subtrees), a 2-node (2 keys, 3 possibly empty subtrees), or a 3-node (3 keys, 4 possibly empty subtrees) and has a pointer to its parent. All empty subtrees are at the same level and every node has one fewer keys than the number of subtrees. Operations:

1. Searching is similar to BST search.
2. Insert first searches to find the node where the key should be inserted, then recursively splits upwards while the node is overflowing (has 4 keys). Inserting may cause the overall height to increase by 1, if the overflow continues up to the root and a new root needs to be created. If the node has p nodes which causes the overflow, then its always the $\lfloor \frac{p}{2} \rfloor + 1$ largest which gets removed and moved up.
3. Delete first searches to find the key k. Then finds the predecessor or successor leaf k' of k, swaps k' with k, deletes the old k', and recursively deletes upwards while the current node has an underflow (has 0 keys). Either does a transfer/rotate if the node has a sibling with 2 or more keys and can give one away or does a merge if there is no sibling or all siblings have only 1 key. Always break ties to the right. May decrease overall height by 1, if the underflow continues up to the root and an empty root is deleted.

11.3 a-b Tree

a-b tree: The 2-4 tree is an a-b tree where $a = 2, b = 4$. Satisfies the same properties as an a-b tree, where every node has at most b subtrees and least a subtrees except for the root which has at least 2 subtrees. Also,

every node has one fewer key than number of subtrees. If $a \leq \lceil \frac{b}{2} \rceil$, then the operations are the same as for the 2-4 tree, except with different thresholds for over and underflow.

Analysis of a-b trees: The least number of nodes in an a-b tree at level 0 is 1, the least number of nodes at level 1 is 2, the least number at level h is $2a^{h-1}$. Since each node has at least $a-1$ keys, except the root which has at least 1 key (since the root has at least 2 subtrees and 2 - 1 = 1), then the least number at level h is $2a^{h-1}(a-1)$. Thus, if there are n keys, $n \geq 1 + 2(a-1)\sum_{i=0}^{h-1} a^i = 2a^h - 1$, so $\log_a(n) \geq h$, so $h \in O(\log_a(n))$.

The most number of nodes at level 0 is 1 and at level h is b^h. Since each node has at most $b-1$ keys, the most number of keys at level 0 is $b-1$ and at level h is $b^h(b-1)$. Thus, in a B-tree with n keys and height h, $n \leq \sum_{i=0}^{h} b^h(b-1) = (b-1)\sum_{i=0}^{h} b^h = (b-1)\frac{b^{h+1}+1}{b-1} = b^{h+1} + 1$. Therefore, $h \in \Omega(\log_b(n))$. From above, $h \in O(\log_a(n))$.

11.4 B-Tree

A B-tree of order m is an $\lceil \frac{m}{2} \rceil$-$m$ tree, a 2-4 tree is a B-tree of order 4. Suppose each node of a B-tree stores its keys and subtree pointers in a dictionary with $O(\log m)$ search, insert, and delete. For example, each node of a B-tree might be an AVL tree. Then, search, insert, and delete of the B-tree takes $\Theta(h)$ node operations. From the analysis above, $h \in O(\log_{m/2}(n)) = O(\log_m(n))$, and at each node, $O(\log m)$ operations are done, so overall, this is $O(\frac{\log n}{\log m} \cdot \log m) = O(\log n)$. Moreover, if m is chosen such that any m-node in the B-tree fits into a single block, typically $m \in \Theta(B)$, then each operation only needs $\Theta(h) = \Theta(\log_m(n)) = \Theta(\log_B(n))$ block transfers. This is much better than an AVL tree which needs $\Theta(\log n)$ transfers.

Variations of B-trees:

1. Preemptive splitting/merging: while searching for insert or delete, split and merge nodes which are close to under/overflow. This allows us to insert or delete just at the leaves, without recursively going back up.
2. B^+-trees: all keys are in leaves, which link together in a list, interior nodes just store keys to guide the search.
3. Cache-oblivious tree: a binary tree T hides another binary tree T' of size $\Theta(\sqrt{n})$, allows us to achieve $O(\log_B(n))$ block transfers without knowing B.

12 Other

Theorem: If $f(n) \in o(g(n))$ and $g(n) \in o(h(n))$, then $f(n) \in o(h(n))$.

Proof: Choose a $c > 0$. Then, $\exists n_1, n_2$ such that $f(n) < cg(n)$ for all $n \geq n_1$ and $g(n) < ch(n)$ for all $n \geq n_2$. Taking $n_3 = \max\{n_1, n_2\}$, then $f(n) < cg(n) < c^2 h(n)$. Since c is arbitrary, then c^2 is as well. Thus, $f(n) \in o(h(n))$.

Theorem: If $f(n) \in o(g(n))$ and $g(n) > 0, \forall n$, then n_0 must depend on c.

Proof: Suppose for contradiction there was a constant n_0. Since $f(n) \in o(g(n))$, then $f(n_0 + 1) < cg(n_0 + 1)$ for any c. Choose $c = \frac{f(n_0+1)}{g(n_0+1)}$. But, $cg(n_0 + 1) = f(n_0 + 1)$ so this is a contradiction.

Theorem: If $f(n) \in o(g(n))$, then $nf(n) \in o(ng(n))$.

Theorem: If $f(n) \in O(g(n)), g(n) > 1$, then $\log f(n) \in O(\log g(n))$.

Proof: Since $f(n) \in O(g(n))$, then $\exists c, n_0 > 0$ such that $f(n) \leq cg(n), \forall n \geq n_0$. Then,

$$\log f(n) \leq \log cg(n) = \log c + \log g(n)$$

INCOMPLETE!!!!!!!!!

If $g(n)$ doesn't have to be greater than 1, then $f(n) = 2, g(n) = 1$ is a counter proof to this theorem.

Theorem: Although $\lfloor \log n \rfloor \in \Theta(\log n), n^{\lfloor \log n \rfloor}$ is not necessarily in $\Theta(n^{\log n})$.

Theorem: For any $1 < a < b, a^n \in o(b^n)$.

Proof: Let $1 < a < b$. Then, for all $n > 0$, $a^n < a^{n+1} = \frac{a^{n+1}}{b^n} b^n$ and in order to satisfy the inequality $\frac{a^{n+1}}{b^n} b^n \leq cb^n$ for any c,

$$a\frac{a^n}{b^n} \leq c \rightarrow n \log\Big(\frac{a}{b}\Big) \leq \log\Big(\frac{c}{a}\Big) \rightarrow n \geq \frac{\log\big(\frac{c}{a}\big)}{\log\big(\frac{a}{b}\big)}$$

Therefore, taking n_0 as the max of this value and 1, $a^n \in o(b^n)$.

Other: Suppose we are trying to prove $f(n) \in o(g(n))$ and we have that $f(n) < cg(n)$ so long as $n_0 \geq \frac{1}{c}$. Then, we have to choose $\frac{1}{c} + c$, to ensure rounding up.

13 Other Problems

Simplify the following:

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 &= \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) \\
&= \sum_{i=1}^{n}\left[\sum_{j=i}^{n} j - \sum_{j=i}^{n}(i-1)\right] \\
&= \sum_{i=1}^{n}\left[\sum_{j=i}^{n} j - (n-i+1)(i-1)\right] \\
&= \sum_{i=1}^{n}\left[\sum_{j=1}^{n} j - \sum_{j=1}^{i-1} j - (n-i+1)(i-1)\right]
\end{aligned}$$