Data Structures and Algorithms

Raymond Bian

November 27, 2023

Contents

Lecture 1: Arrays, ArrayList

Lecture 2: LinkedList

Lecture 3: DLL, CSLL

Lecture 4: Recursion, Stacks

Lecture 5: Queues

Lecture 6: Deques

Lecture 7: Intro to Trees

Lecture 8: Tree Traversals

Lecture 9: Level Order, BST Operations

Lecture 10: BST Remove

Lecture 11: Heaps

Lecture 12: Build Heap

Lecture 13: Hashmaps; External Chaining

Lecture 14: Linear Probing

Lecture 15: Quadratic Probing

Lecture 16: Skip Lists

Lecture 17: AVL

1 AVL Tree

What if we want guaranteed $O(\log(n))$ operations on everything?

Definition 1. An AVL is a BST that is *always* balanced. It stores a balance factor that is the difference between the left node's and right node's height.

If the balance factor has magnitude greater than or equals to 2, then it is learning too far left (positive) or too far right (negative). We can maintain the tree's balance with rotations.

Note. Getting the height of a BST is $O(n)$. However, we can store the height of a node in the node itself. This makes getting the height of a node $O(1)$.

1.1 Operations

1.1.1 Update

When we add or remove data, the heights and balance factors of each ancestor will change. Therefore, we must update these values.

Algorithm 1: Update

```
Input: curr, the node to update
curr.height = max(kids.height) + 1;
curr.bf = curr.left.height - curr.right.height; // Children must be updated first!
```

1.1.2 Add

How do we add data?

1. We add the data to the leaf position, as you would in a BST.
2. If the tree is no longer balanced, we then rotate the tree to balance it.

Note that when a node is added, at most $\log(n)$ nodes (its ancestors) have a new balance factor.

Algorithm 2: Add

```
if $curr = null$ then
    add new node;
else if $curr.data < data$ then
    recurse left;
else if $curr.data > data$ then
    recurse right;
else if $curr.data = data$ then
    // Duplicate, do nothing
end
foreach $node : curr \to root$ do
    update(node);
    if node.bf is bad then
        rotate
    end
end
```

Note. You should update after recursion so the height values for the children are correct.

1.2 Rotations

There are four types of rotations: left, right, left-right, right-left.

1.2.1 Left Rotation

This type of rotation is used when:

- The node is leaning right (balance factor is -2)
- The right child is also leaning right (balance factor is -1)

Algorithm 3: Left Rotation

```
Input: A, the root of the tree
Output: the new root of the tree
B = A.right;
A.right = B.left;
B.left = A;
update(A);
update(B);
return B
```

Note. This is an O(1) operation.

1.2.2 Right Rotation

This type of rotation is used when:

- The node is leaning left (balance factor is 2)
- The left child is also leaning left (balance factor is 1)

Algorithm 4: Right Rotation

```
Input: A, the root of the tree
Output: the new root of the tree
B = A.left;
A.left = B.right;
B.right = A;
update(A);
update(B);
return B
```

Lecture 18: AVL Continued

1.2.3 Right Left Rotation

This type of rotation is used when:

- The node is leaning right (balance factor is -2)
- The right child is leaning left (balance factor is 1)

We rotate the right child to the right. Then, we rotate the root to the left. Even though this is a combination of two rotations, it is still considered a single operation.

1.2.4 Left Right Rotation

This type of rotation is used when:

- The node is leaning left (balance factor is 2)
- The left child is leaning right (balance factor is -1)

This is just the mirror of the previous operation: we rotate the left child to the left, and the root to the right.

1.3 Runtime

How long do operations on an AVL tree take?

- When you add data into an AVL tree, you do at most one rotation, so the runtime is $O(\log(n))$.
- When you remove from an AVL tree, you do at most $\log(n)$ rotations. Even still, the runtime is still $O(\log(n))$.
- Remember that each rotation is $O(1)$.

Note. An AVL can be at most $1.44\log(n) = \log(n)$ tall. This is derived from $\frac{1}{\log_2(\phi)}$.

Exercise 1

What does the following AVL tree look like after these operations?

- add(56, 75, 61, 88, 93, 77)
- remove(61)

Lecture 19: 2-4 Trees

2 2-4 Trees

Another $O(\log n)$ data structure.

Definition 2. A **perfect** binary search tree is a full binary search tree where all leaves are at the same depth.

We maintain the perfect shape for any operation on a 2-4 tree. However, this means that they can only have 1, 3, 7, 15, 31 nodes, etc. Our solution is to add multiple data items to each node.

Property. Every node has 1-3 data items, and 2-4 children. Every internal node always has one more child than the number of data items.

Note. Note that 2-4 trees are not binary trees.

Property. Let d_1, d_2, d_3 be the data in the node, and t_1, t_2, t_3, t_4 be the data of the children.

- The data in each node must be sorted ($d_1 < d_2 < d_3$).
- The data of each children must be sorted between the node's data: $t_1 < d_1$, $d_1 < t_2 < d_2$, $d_2 < t_3 < d_3$, $d_3 < t_4$.

2.1 Operations

2.1.1 Contains

Very similar to a binary search tree.

Algorithm 5: Contains

```
if $data < d_1$ then
    explore $t_1$;
else if $data < d_2$ then
    explore $t_2$;
else if $data < d_3$ then
    explore $t_3$;
else
    explore $t_4$;
end
```

2.1.2 Add

A little bit harder. We must maintain the property that all leaves have the same height. We use the same logic as the contains algorithm, and add it to the node.

If there are already three data items in the leaf (overflow), we solve it with a process called promotion. We push one data item into the parent (either d_2, d_3), and split the rest of the data into two leaves around the promoted data.

If the parent is now too full, repeat the process again. If the root is too full, then the promoted data becomes the new root, and the root splits into two children.

2.1.3 Remove

There are several cases to consider:

Case 1: Removing from a leaf with 2 or 3 data items. This case is very easy, as we just remove the data from the leaf.

Case 2: Removing from an internal node. This case is simple as well, as we maintain the structure of the tree, replacing the data with its predecessor or successor. Because the predecessor or successor is always in a leaf, we can use Case 1, 3 or 4 to remove the predecessor or successor.

Lecture 20: 2-4 Trees Continued

Remove operation continued:

Case 3: We want to remove from a leaf with only one data item, but the siblings have spare data. This operation is called a **transfer** or **rotation**. We take the spare data directly to the left or right, promote it into the parent, and pull the parent into the removed leaf. Note that if you transfer an internal node, you must move the subtrees as well!

Case 4: We want to remove from a leaf with only one data item, and there is no spare data in the left or right siblings. Then, we pull down from the parent (left or right) and **fuse** two nodes together.

However, you might empty the parent, which requires you pull from the grandparent. However, the grandparent might be empty after that operation as well! Therefore, we stop when the parent has 2 or 3 data items, we use a transfer, or the root becomes empty.

Still confused: here is a handly flowchart:

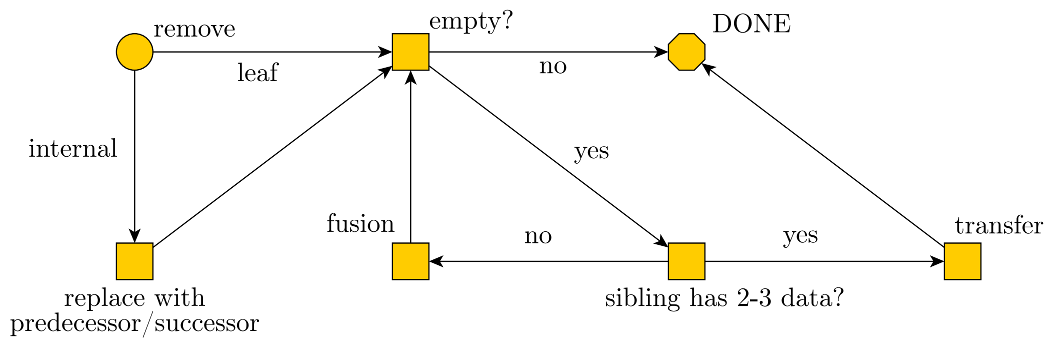

Figure 1: 2-4 Tree Remove Flowchart

2.2 Runtime

The runtime of a 2-4 tree is as follows:

- Adding a node may require up to $\log n$ promotions, which is $O(\log n)$.
- Removing a node may require up to $\log n$ fusions, which is $O(\log n)$.

Note. Note that even though an AVL and a 2-4 tree have the same runtime complexity, the depth of a 2-4 tree is less than or equal to that of an AVL tree. Therefore, it is used when going down the tree is costly - for example, when the data is stored on disk. For example, databases use a generation called a B-tree (where there are many items per node).

Lecture 21: Sorting

3 Sorting

What is an algorithm?

- A sequential list of instructions to accomplish some goal.

For each sorting algorithm, we will look at:

- Time complexity (best, average, worst)
- Stability: equal valued items maintain their relative order. This is important because it allows us to sort by multiple criteria.
- Adaptivity: an algorithm is faster if data is (partially) sorted.
- In place: uses O(1) extra memory to sort the data.

There are also two types of sorting algorithms:

- Iterative: uses loops to iterate over the data. It is easier to implement and is usually in-place, but only sorts one item at a time.
- Divide and Conquer: (usually) uses recursion that splits the data into smaller pieces, sorts them, and then merges them. It is usually faster, but requires more memory.
- Non-Comparitive: uses special properties of the data to sort it. It is usually faster, but is only applicable to certain data.

3.1 Bubble Sort

For bubble sort, we iterate through the array from beginning to end, comparing pairs of adjacent items, which are swapped if they are out of order. We repeat this process, stopping at an earlier ending value until no swaps are made.

Algorithm 6: Bubble Sort

```
Input: A, the data to be sorted
for i from $0 \to n-1$ do
    for j from $0 \to n-1-i$ do
        if A[j] > A[j+1] then
            swap(A[j], A[j+1]);
        end
    end
end
```

3.1.1 Optimizations

1. If we make no changes, we know that the array is sorted. This means that we can stop the algorithm early.
2. Track the last index where a swap was made, and only iterate up to that index. We can do this because we know that all elements after that index are sorted.

Algorithm 7: Optimized Bubble Sort

```
Input: A, the data to be sorted
end = n - 2;
while end > -1 do
    lastSwap = -1;
    for i : 0 → end do
        if A[i] > A[i+1] then
            swap(A[i], A[i+1]);
            lastSwap = i;
        end
    end
    end = lastSwap - 1;
end
```

3.1.2 Runtime

The best case for bubble sort occurs when the data is already in sorted order. We still have to check that the data is sorted, so the best case is $O(n)$.

The worst case for bubble sort occurs when the data is in reverse sorted order. We then do $n + n - 1 + n - 2 + \ldots + 1 = \frac{n}{2}(n+1) = \frac{n^2+n}{2}$ operations, which is $O(n^2)$.

The average case is, unfortunately, very close to the worst case. Therefore, it is $O(n^2)$. Why is this?

If we randomly order the data, then on average half of the pairs are out of order. Every swap in our algorithm fixes one pair. There are on average $\frac{n(n-1)}{4}$ pairs that are out of order, therefore our runtime is $O(n^2)$ on average.

3.1.3 Conclusion

Bubble sort is stable, adaptive, and in-place.

3.2 Insertion Sort

We iterate over the elements, swapping the current element backwards until they are at the proper position in the first part of the array.

3.2.1 Runtime

The best case for insertion sort occurs when the data is already in sorted order. We iterate over all data, and we do one comparison for each item, which results in a runtime of $O(n)$.

The worst case for insertion sort is when the data is in reverse sorted order. Similar to bubble sort, the worst case is $O(n^2)$.

The average case, also like bubble sort, is $O(n^2)$.

3.2.2 Conclusion

Insertion sort is stable, adaptive, and in-place.

3.3 Selection Sort

We find the largest item in our data, and swap it to the end of the array. We then repeat this process, swapping the next largest item to the second to last position, and so on until the data is sorted.

3.3.1 Runtime

All cases for selection sort is $O(n^2)$. We don't do anything depending on the data, so our runtime is quadratic even if our data is sorted!

Lecture 22: Sorting Continued

Continuing on with selection sort:

Algorithm 8: Selection Sort

```
  Input: A, the array to sort
1 for $i = A.length - 1$ to 2 do
2     Swap max(A[1..i]) with A[i];
3 end
```

3.3.2 Conclusion

Selection sort is in-place. However, it is not adaptive and not stable. Even still, selection sort uses the fewest swaps, which makes it applicable for applications where writing to memory is costly.

3.4 Cocktail Shaker Sort

Bubble sort is (kind of) adaptive. The Cocktail Shaker sort builds upon this, becoming adaptive for arrays like [2, 3, 4, 5, 6, 7, 8, 1].

In one iteration of the Cocktail Shaker sort, we run bubble sort from left-to-right, then once again from right-to-left. Note that as a result, both sides of the array will be sorted.

3.4.1 Optimizations

We can track the last swap in both directions. Then, we can stop the algorithm once we reach the last swap in both directions.

3.4.2 Runtime

The best case for this sort (sorted data) is $O(n)$.

The average case for this sort is $O(n^2)$.

The worst case for this sort (reverse sorted data) is $O(n^2)$.

3.4.3 Conclusion

Just like Bubble sort, it is in-place, stable, and adaptive.

Note. Why do we study bad sorting algorithms? They are stable, adaptive, and in-place. Fast sorts often sacrifice one of these properties, and are often harder to implement. Complex sorts use these sorts as building blocks, and they aren't even the worst (Bogosort, Stalinsort, etc)!

Note. Theoretical computer scientists have found that the best possible worst case sorting algorithm sorts in $O(n \log(n))$.

3.5 Heap Sort

We can achieve an $O(n \log(n))$ sort by adding everything to a heap in $O(n)$, and removing data one at a time in $O(log(n))$.

3.5.1 Conclusion

Heap sort is not in-place: while you can define and apply the heap logic on any backing array, more often than not you will be inserting your data into an external heap, which is $O(n)$ extra memory.

Because heap sort is $O(n\log(n))$ for every case, it is not adaptive.

Note. What other data structures can we sort with? Are they good? Well, we can use an AVL/BST, add all the data in $O(n\log(n))/O(n^2)$, and get the inorder traversal in $O(n)$. A similar implementation can be created with skiplists.

Lecture 23: Merge Sort

3.6 Merge Sort

Merge sort is an example of a divide-and-conquer sort. It breaks a large problem into smaller problems, solves them, and combines the solutions into one larger solution.

It takes an array of data, splits them into two, sorts them (using merge sort yet again), and merges the two sorted lists back together.

Algorithm 9: Merge Sort

```
if array.length == 1 then
    return;
end
left = arr[0:arr.length/2];
right = arr[arr.length/2:arr.length];
mergeSort(left);
merseSort(right);
merge(left, right, arr);
```

How do we merge the two sorted sublists? We track i and j, which both start at 0. We compare the two elements at left[i] and right[j], and add the smaller one to the array. We then increment the index of the smaller element.

Note. If there are two equal values, we take the element from the left sublist to maintain stability.

3.6.1 Runtime

The operation list of merge sort is $2\log(n)$ tall, and there are n operations per level. Therefore, the runtime is (always) $O(n\log(n))$.

Note. Note that one such "worst case" (most comparisons) of merge sort for an array of size 8 is $[5, 1, 7, 3, 2, 8, 6, 4]$. This is because we have to compare every element in the first half with every element in the second half, and so on for each merge. Even still, this is $O(n\log(n))$.

3.6.2 Conclusion

Merge sort is stable, but not adaptive (same runtime for all case). We also need to create a second array to split and merge the arrays, so it is not in-place.

3.7 Quicksort

Quicksort is also a divide-and-conquer algorithm. However, unlike merge sort, it is in place. It also can be done non-recursively, which is more efficient.

The overall plan of Quicksort is as follows:

1. Pick one data as the pivot.
2. Partition the data into two sublists: one with elements less than the pivot, and one with elements greater than the pivot.
3. We repeat this plan in each of the sublists, until the data is sorted.

Lecture 24: QuickSort

How do we partition the list? After picking the pivot,

1. We swap the pivot with arr[0],
2. Then we maintain two pointers i and j and walk them towards the center until both see two elements that are in the wrong spot.
3. We swap arr[i] and arr[j]
4. We then stop once the two pointers cross each other, and swap the pivot with j.

3.7.1 Runtime

The average runtime is $O(n \log n)$, and the worst case is $O(n^2)$. Note that the best pivot is the median. Below are some sample pivots:

- arr[0] as the pivot runs in $O(n^2)$ if the data is already sorted.
- Median as the pivot runs in $O(n^2)$ when we have a malicious user.
- Random pivot runs in $O(n \log n)$ on average. But you could pick the worst pivot every time, which would have worst case $O(n^2)$.

Remark. In this class, QuickSort refers to QuickSort with a random pivot.

Note. There is an algorithm called Median-of-medians (grad algo), which gives you a guaranteed "good enough" pivot and runs fast enough such that the worst case is $O(n \log(n))$. However, due to constant time factors, it is not used in practice.

3.7.2 Conclusion

QuickSort is in-place, but not adaptive or stable. Note that QuickSort with recursion is technically not in-place due to the $\log(n)$ to n memory complexity of the call stack.

3.7.3 QuickSelect

Given an array of n elements, find the k-th smallest element in the array. Note that if $k = 1$ or $k = n$, we can find the smallest or largest element in $O(n)$ timewith linear search. Also, if the array was already sorted, we can find the k-th smallest element in $O(1)$ time.

QuickSelect is important because it beats the obvious plan of first sorting the array and getting the k-th element, instead running in $O(n)$ time on average. Instead of recursing to both sides as in QuickSort, we recurse only to the side with the k-th smallest element.

1. Same as steps of QuickSort
2. If $j = k + 1$, we return the pivot. If $j > k - 1$, we find the k-th smallest element in the left subarray. If $j < k + 1$, we find the $k - j - 1$-th smallest element in the right subarray.

Note that because each time we pick an array, we do half as much work, such that we have a runtime of

$$n + \frac{n}{2} + \frac{n}{4} + \ldots = 2n = O(n).$$

on average. In the worst case, we have a runtime of $O(n^2)$.

Lecture 25: RadixSort

3.8 RadixSort

RadixSort has an average case runtime of $O(n)$! But the best case we can do with sorting is $n \log(n)$, so what's the catch?

Note. The bound of $n \log(n)$ only applies to comparison-based sorting algorithms.

Because RadixSort cannot do comparisons, it can only sort string like things, including integers. Suppose you have a ton of integers. The idea of RadixSort is to sort by ones digit, then tens digit, then hundreds digit, etc. This order guarantees stability of the sort.

However, we don't use the other sorts to sort by digits. Instead, we create buckets/queues and place items in their proper buckets.

Algorithm 10: RadixSort

```
for $i = 0$ to max number of digits do
    for each item do
        enqueue item into bucket corresponding to digit i
    end
    for each bucket do
        dequeue all items into original list
    end
end
```

Note. Note for negative digits, you can use 19 buckets to store all possible digits.

How are we going to manipulate the digits of a number? We could convert this string to a char array, but that would be expensive. Instead, we can use the mod operator and integer division to get the digits.

Property. For digit number i, we can divide the number by 10^{i-1} and then mod by 10. This will give us the digit we want.

What base should we choose for RadixSort? Note that if we choose a super large base, such as 2000, then we need to use 2000 buckets! The optimal base is usually 2^8, but varies depending on use case.

3.8.1 Runtime

We have to make n operations for each digit, and we have to do this for each digit. So the runtime is $O(nk)$, where k is the number of digits in the largest number.

3.8.2 Conclusion

RadixSort is stable, but it is not adaptive and in-place.

Lecture 26: Pattern Matching

4 Pattern Matching

What is our motivation for this problem? Imagine we want to find a pattern in a text. What is the most efficient way of doing so?

Example. Google searches the entire internet for a pattern: your name. We can also search very long DNA for specific codons (the pattern). $\diamond$

Notation. Let the length of the text be n. Let the length of the pattern be m.

Definition 3. Let the **alphabet** be the set of valid characters for the text.

There are two types of pattern matching, which differ on what to do when we find the pattern. If we stop, that means we are searching for a single occurence. If we continue, that means we are searching for all occurences.

4.1 Brute Force

Brute force is not just for pattern matching. Instead, we try every possible solution. For pattern matching specifically, we have $O(n)$ $(O(n-m))$ choices for the start of the pattern. We check each starting spot for the pattern.

Algorithm 11: Brute Force Pattern Matching

```
i = 0;
while $i < len(text) - len(pattern)$ do
    j = 0;
    while $j < len(pattern)$ and $text[i+j] = pattern[j]$ do
        j++;
    end
    if $j = len(pattern)$ then
        return True;
    end
    j = 0;
    i++;
end
```

4.1.1 Runtime

The best case for our brute force pattern matching is $O(n)$. The worst case is $O(nm)$, because we need to check every $n - m$ starting positions at most $m - 1$ times. For English, the average case is $O(n)$.

Note that we don't usually care about the average case because english text is not random. Even though brute force is fine, is there a better way to search for patterns?

Note. Java String.contains() uses brute force.

4.2 Boyer-Moore

The idea for this algorithm is to use the wrong character to shift the pattern by a certain amount with a lookup table. Our lookup table stores the last appearance of each character in the pattern. If a character has not appeared, this index is -1.

Algorithm 12: Boyer-Moore Pattern Matching

```
Input: text, pattern
Output: True if pattern in text, False otherwise
for $i < len(pattern)$ do
    lookup[pattern[i]] = i;
end
i = 0;
while $i < len(text) - len(pattern)$ do
    j = len(pattern) - 1;
    while $j >= 0$ and $text[i+j] = pattern[j]$ do
        j -= 1;
    end
    if $j = -1$ then
        return True;
    end
    i += max(1, j - lookup[text[i+j]]);
end
return False;
```

Lecture 27: Pattern Matching Continued

4.2.1 Runtime

The worst case runtime is $O(mn)$. The best case for finding a single occurence is $O(m)$. The best case for failing to find an occurence or finding all occurences is $O(\frac{n}{m})$. However, be careful that building the lookup table takes $O(m)$ time, so technically our best case runtime for these two cases is $O(\frac{n}{m} + m)$.

Note. We should use Boyer-Moore whenever the text has characters not in the pattern. This is more likely as the alphabet grows, so it is better for larger alphabets.

Traditional Boyer-Moore includes a good suffix rule, which improves big O runtime, but doesn't really speed up runtime in real life scenarios. It is also quite similar to KMP, but we will not cover it.

4.3 Knuth-Morris-Pratt

What is the longest word that you can think of which has no repeated letters? Demographic.

Note that when we try to find pattern "demographic" in a text, we can shift, no matter what, the d all the way to under the c. However, if there are repeated patterns, we can shift the word to align with another pattern to save work. That is the core idea of KMP.

Lecture 28: KMP Continued

Depending on which letter we failed on when checking the pattern, we can shift by a predetermined amount. Note that we keep track of $j - \text{shift} - 1$ (j is index of pattern) and insert this value into the failure table into $j - 1$, as this is the number of characters in our pattern that are already matched for the next iteration of the algorithm.

How do we get the failure table? We need to know how much of the beginning matches the end, i.e. is there a matching prefix/suffix? More formally, the failure table at j is the length of the longest string which is both a prefix of the pattern and a proper suffix of $p[0 \dots 1]$.

Example. Let the pattern be "axabaxax". The failure table is as follows:

Prefix	**Longest Prefix/Suffix**
a	0
ax	0
axa	1
axab	0
axaba	1
axabax	2
axabaxa	3
axabaxax	2

◇

Example. Let the pattern be "eminem". The failure table is as follows:

Prefix	**Longest Prefix/Suffix**
e	0
em	0
emi	1
emin	0
emine	1
eminem	2

◇

Example. Let the pattern be “ababaab”. The failure table is as follows:

Prefix	**Longest Prefix/Suffix**
a	0
ab	0
aba	1
abab	2
ababa	3
ababaa	1
ababaab	2

◇

Example. Let the pattern be “salsas”. The failure table is as follows:

Prefix	**Longest Prefix/Suffix**
s	0
sa	0
sal	0
sals	1
salsa	2
salsas	1

◇

How do we run KMP?

1. We iterate L to R through the pattern
2. If we fail at an index j (in the pattern), suppose $FT[j-1] = k$.
3. If $k > 0$, we can save some matched characters, so we shift until k characters of the patterns still overlap with previous matched characters (i.e. shift by $j-k+1$). Note that we don’t need to recompare these still overlapping characters!
4. If $k = 0$, then there is no useful repetition. Therefore, we should align the pattern with the mismatched character.
5. If $j = 0$, then we cannot access the failure table. Therefore, we should just shift by one.
6. If we have a complete match, use the last entry in the failure table (i.e. $FT[m-1]$ where m is the length of the pattern).

Algorithm 13: Failure Table Generation

```
Input: Our pattern
i = 0 ;                                                    // prefix
j = 1 ;                              // suffix, also current ft entry index
ft = int[len(pattern)];
ft[0] = 0;
while j < len(pattern) do
    if pattern[i] == pattern[j] then
        ft[j] = i + 1;
        i++; j++;
    else
        if i == 0 then
            ft[j] = 0;
            j++;
        else
            i = ft[i-1] ;      // progress before, try smaller prefix suffix pair
        end
    end
end
```

Lecture 29: KMP Continued Continued

Here is the pseudocode for KMP:

Algorithm 14: KMP

```
Input: Pattern $P$ of length $m$, text $T$ of length $n$
Output: All indices $i$ such that $T[i..i + m - 1] = P$
ft = makeFT($P$);
i = 0;
j = 0;
while $i \leq n - m$ do
    if T[i] == P[j] then
        if j == m - 1 then
            j = ft[j];
            i++;
        else
            i++;
            j++;
        end
    else
        if j == 0 then
            i++;
        else
            j = ft[j] - 1;
        end
    end
end
```

4.3.1 Runtime

The worst case time is $O(m+n)$. Note that the trivial best case is when you are trying to find one occurrence and it is at the start of the case, which is $O(m)$.

Note. Note that KMP can get stuck sometimes.

Example. Let the text be aaab..., and the pattern be aaaa. Then, we would check the pattern 4 times here, which is less efficient that Boyer-Moore. In other words, KMP does not recognize that b is not in the pattern. ◇

Note. Boyer Moore has a better best case with $O(\frac{n}{m}+m)$, but a worse worst case with $O(mn)$. Of the two, KMP is preferred for smaller alphabets, and Boyer-Moore is preferred for large alphabets.

Lecture 30: Robin Karp

4.4 Robin Karp

Instead of comparing character by character, we can use a rolling hash function to compare substrings. For each alignment, compare $hash(pattern)$ to $hash(text[i \dots i+m-1])$. If the hashes are different, we know the two strings are different. However, if they are the same, then you have to check character by character. Can we optimize finding the hash of the substring $O(1)$?

4.4.1 Rolling Hash

The first hash we calculate is $O(m)$, but updating the hash is $O(1)$. Each hash is calculated from the previous hash in $O(1)$ time. A potential hash is to add all the algorithms together. Then, we can subtract the first one and add the next one to roll the hash. However, this leads to many collisions.

The actual hash uses a base B, which is prime. Then, we treat the string as a base B number. Then, abcd would be represented as

$$a \times B^3 + b \times B^2 + c \times B + d.$$

Note that we can roll our hash by subtracting $a \times B^3$, multiplying what remains by B, and then adding the new character e.

4.4.2 Runtime

The best case runtime is when we find one occurence at the beginning, which is $O(m)$. For all occurences, the hash never matches, so we have a runtime of $O(n+m)$. In the worst case, the hash always matches the text, so we have to compare character by character, which is $O(nm)$.

4.5 Galil Rule

The Galil rule is a combination of KMP and Boyer Moore. Essentially, when we have a full match in Boyer Moore, we can shift by the period $(m - ft[m-1])$ as we do in KMP. This optimizes the worst case in which the text consists of many patterns.

Lecture 31: Graphs

5 Graphs

For motivation, think about landlocked and doubly landelocked countries. How would we find the number of doubly landlocked countries? We could use a graph where countries are nodes, and edges are placed between two things that are touching.

Definition 4. A **vertex** is a node in a graph.

Definition 5. An **edge** connects two vertices in a graph.

Note. The way you draw the graph doesn't matter!

Definition 6. Two vertices are **adjacent** if they share an edge.

Definition 7. A vertex is **incident** to an edge if it is an endpoint of that edge.

Definition 8. The **degree** of a vertex is the number of vertices it is adjacent to.

Definition 9. A vertex is **isolated** if its degree is 0.

Definition 10. A **simple** graph is a graph without self-loops and multiple edges.

Definition 11. The number of vertices in G, denoted the **order** of G, is $|V|$.

Definition 12. The number of edges in G, denoted the **size** of G, is $|E|$.

Definition 13. A **directed** graph is a graph where edges have a direction.

Example. Currency exchange rates can be modeled with a directed graph. ◇

Example. Flights can be modeled with a directed graph. ◇

Example. Pathfinding between locations in a map. ◇

How do we traverse a graph?

Definition 14. A **walk** is a traversal across a graph through a series of edges.

Definition 15. A **path** is a walk in which no vertex or edge is repeated.

Definition 16. A **trail** is a walk in which no edge is repeated.

Definition 17. A **cycle** is a path in which the first and last vertices are the same.

Definition 18. A **circuit** is a trail in which the first and last vertices are the same.

Definition 19. Two vertices are **connected** if there is a path from one to another.

Definition 20. A graph is **connected** if every pair of vertices is connected.

Definition 21. A **tree** is an acyclic connected graph.

Note. All trees have $|V| - 1$ edges.

Definition 22. A **clique** is a graph with the maximum number of edges.

Note. All cliques have $\frac{n(n-1)}{2} \approx O(|V|^2)$ edges.

Definition 23. A **sparse** graph has $\approx |V|$ edges (tree).

Definition 24. A **dense** graph has $\approx |V|^2$ edges (clique).

Lecture 32: DFS

5.1 Graph Representations

How do we store a graph in a computer? There are to main ideas: adjacency matrices and adjacency lists. The rows are the "out" vertices and the columns are the "in" vertices, and the matrix itself can store the label or weight or some other value of the edge exists.

Note. An undirected graph will have a symmetric adjacency matrix.

The major appeal of using a matrix is having O(1) operations to remove, add, or set edges. However, there is a lot of space used for nothing - we will always use $O(|V|^2)$ memory. Also, adding or removing new vertices to the graph is expensive, also time $O(|V|^2)$ as we need to copy our data to a new array. Also, getting the list of adjacent vertices is time $O(|V|)$.

An adjacency list is a map from vertices to a list of edges incident to the vertex. With this representation, we can add/remove a vertex in time $O(1)$. The worst case for removing an edge is $O(\deg(V)) \approx O(|V|)$. And, we can get edges incident to this vertex in time $O(1)$, and the vertices at the end of the edges in time $O(\deg(V)) \approx O(|V|)$

Note. An adjacency matrix should be used for dense graphs, and an adjacency list should be used for sparse graphs.

We can also use an edge list, which is a lot worse. This is because we need to look through every single edge in our list to find specific information, which is $O(|E|)$.

Starting at a vertex in the graph, which vertices are reachable?

The naive solution is to wander through the grarph randomly, and note when a new vertex is seen. This is very slow because it repeats vertices. We can optimize this by marking nodes as we visit them. However, we can get trapped and miss some nodes.

The actual plan is, as we traverse through the graph, we will store the vertices seen but not traversed into a to-do list. When we have nothing to do, we will check the to-do list to jump to the next vertex. If we insert new items into the to-do list at the front, we have DFS. If we insert new items to the back of the to-do list, we have BFS.

5.2 Depth-First Search

Algorithm 15: DFS

```
Input: Graph $G = (V, E)$, start vertex $v$
for $w$ adj to $v$ do
    if $w$ is not marked then
        mark $w$ as seen;
        DFS($G, w$);
    end
end
```

Note. In this class, the tiebreaker for which node to visit first is alphabetical.

DFS can be used for finding connected components, cycles, spanning trees, as well as finding "a" path between vertices.

5.2.1 Runtime

DFS has a time complexity of $O(|V| + |E|)$.

Lecture 33: DFS, BFS, Dijkstra's

TODO

Lecture 34: Dijkstra's Algorithm

5.3 Dijkstra's Algorithm

To find a path from A to B with weighted edges, we can used Dijkstra's.

Example. Note that Dijkstra's does not support negative edge weights. ◇

Given that we know the best path to a node, we can go from that node to the next to find a best path to the neighboring node. We use a min heap based on shortest total length, popping the first one in the heap and processing it.

Algorithm 16: Dijkstra's Algorithm

```
pq = new PriorityQueue();
pq.enqueue(start, 0);
while pq is not empty or visited not full do
    v, dist = pq.poll();
    if v not in visited then
        visited.add(v);
        for w adj to v do
            if !w.visited then
                w.dist = min(w.dist, v.dist + e.weight); pq.add(w);
            end
        end
    end
end
```

5.3.1 Runtime

There can be at most $|E|$ items in the heap, and each takes $\log(|E|)$ time to remove, which means that the runtime is $O(|E|\log(|E|))$.

Note. Some special priority queues can have a runtime of $O(|E| + |V|\log(|V|))$, which uses a decrease priority function. With a Fibonacci Heap, you can achieve a runtime of $O(|E| + |V|\log(|V|))$.

5.3.2 Negative Edge Weights

What happens if we have negative edge weights?

Definition 25. A **negative cycle** is a cycle in which the sum of edge weights is negative.

The shortest "path" problem is not defined on negative cycles, so we don't really need to worry about this case. However, even without a negative cycle, a path might look bad at first, but might be good later on. This is why we need to be careful with negative edge weights.

Note. We can use Bellman-Ford, Floyd Warshall, or Johnson's Algorithm to find shortest paths in such graphs.

Lecture 35: Minimum Spanning Trees

5.4 Prim's Algorithm

A Minimum Spanning Tree is a spanning tree of a graph with the smallest possible weight.

Note. A tree always has $|V| - 1$ edges.

How do we find this MST? One plan is to pick a start vertex and explore the graph the usual way, growing a single conneced tree throughout our algorithm. We add the lightest edge that:

- Does not form any cycles
- Is connected to the rest of the tree

Note that we can accomplish the first point with a visited set, and the second point by only adding adjacent vertices. We can get the lightest edge with a priority queue.

Algorithm 17: Prim's Algorithm

```
visited = Set<vertex>;
PQ = PriorityQueue<edge>;
MST = Set<edge>;
for e incident to s do
    PQ.insert(e);
end
visited.add(s);
while !PQ.isEmpty() do
    e = PQ.dequeue();
    u,v = e.endpoints();
    if v is not in visited then
        MST.add(e);
        visited.add(v);
        for f incident to v do
            if f.getOtherEndpoint(v) is not in visited then
                PQ.insert(f);
            end
        end
    end
end
```

5.4.1 Runtime

The priority queue could contain every edge, and could be $O(|E|)$. Removing is $O(|E|)$. Therefore, the total runtime is $O(|E| \log |E|)$.

5.5 Kruskal's Algorithm

The other option to find a MST is using Kruskal's Algorithm.

Definition 26. A **greedy** algorithm is an approach that makes the locally optimal choice at each step.

To greedily build an MST, we want the minimum edge sum. Therefore, we add all the minimum edges that do not create cycles.

Algorithm 18: Kruskal's Algorithm

```
PQ = PriorityQueue<edge>(G.edges());
MST = new Set<edge>;
while PQ is not empty and MST is not full do
    e = PQ.dequeue();
    if e does not create a cycle in MST then
        MST.add(e);
    end
end
```

To check that an edge does not create a cycle, we can use the union-find data structure.

5.5.1 Union-Find

Find(v) returns the label of vertex v. All connected vertices have the same label.

Note. For our case, adding an edge between x and y would create a cycle if the label of x is the same as the label of y.

Union(x, y) updates the labels of everything connected to x or y to be the same.

With the best possible optimizations, we can get a runtime of $O(A^{-1}(G))$ for these operations. However, note that A^{-1}(number of atoms in the universe) = 5. Therefore, this data structure does not affect the runtime of Kruskal's in a significant way.