CS 162 Notes (Winter 2017)

Justin Lubin

Contents

1 Asymptotic Analysis **7**
- 1.1 Big-O Notation 7
  - 1.1.1 Definition 7
  - 1.1.2 Assumption: Constant Operations (Approximations) 7
  - 1.1.3 Non-Constant Operations 7
- 1.2 Reducing Big-O Expressions 8
  - 1.2.1 What to Eliminate 8
  - 1.2.2 Examples 8
- 1.3 Linear Search 8
  - 1.3.1 Code 8
  - 1.3.2 Analysis 8
- 1.4 Binary Search 9
  - 1.4.1 Code 9
  - 1.4.2 Analysis 9
- 1.5 Methods of Asymptotic Analysis 9
  - 1.5.1 Expansion Method 9
  - 1.5.2 Substitution Method 10
- 1.6 The Towers of Hanoi 11
  - 1.6.1 Gameplay 11
  - 1.6.2 Algorithmic Solution 11
- 1.7 Mergesort 12
  - 1.7.1 Description 12
  - 1.7.2 Example 12
  - 1.7.3 Analysis 12

2 Algorithm Correctness **15**
- 2.1 Key Parts of an Algorithm 15
- 2.2 Proving Correctness 15
  - 2.2.1 Basic Example 15
  - 2.2.2 Loop Invariants 16
  - 2.2.3 Strong Induction 16
  - 2.2.4 Mergesort 16

3 Data Structures I **19**
- 3.1 Dictionaries 19
  - 3.1.1 Description 19

- 3.1.2 Implementations . . . 19
- 3.1.3 Better Implementations . . . 20
- 3.2 Trees . . . 20
  - 3.2.1 Terminology . . . 20
- 3.3 Binary Trees . . . 20
  - 3.3.1 Description . . . 20
  - 3.3.2 Binary Tree Traversal . . . 21
- 3.4 AVL Trees . . . 22
  - 3.4.1 Motivation . . . 22
  - 3.4.2 Example . . . 23
  - 3.4.3 Maintaining the AVL Condition . . . 23
- 3.5 Hashtables . . . 25
  - 3.5.1 Description . . . 25
  - 3.5.2 Example Hash Function . . . 25
  - 3.5.3 Collisions . . . 25
  - 3.5.4 Load Factor . . . 26
  - 3.5.5 Linear Probing . . . 26
  - 3.5.6 Quadratic Probing . . . 26
  - 3.5.7 Double Hashing . . . 27
  - 3.5.8 Universal Hash Functions . . . 27
  - 3.5.9 Perfect Hash Functions . . . 28
- 3.6 Union-Find Data Structure . . . 28
  - 3.6.1 Mathematical Definitions . . . 28
  - 3.6.2 The Union-Find Algorithm . . . 28
  - 3.6.3 Union-Find Operations . . . 29
  - 3.6.4 A Cute Application of Union-Find . . . 29
  - 3.6.5 Up-Tree . . . 30
- 3.7 Priority Queue . . . 31
  - 3.7.1 Binary Min-Heap . . . 31
  - 3.7.2 Array Representation of Binary Trees . . . 32
  - 3.7.3 Fast Way to Build Heap . . . 33

4 Graphs 35
- 4.1 Introduction to Graphs . . . 35
  - 4.1.1 Undirected Graphs . . . 35
  - 4.1.2 Directed Graphs (Digraph) . . . 36
  - 4.1.3 Weighted Graphs . . . 36
  - 4.1.4 Connectivity and Completeness . . . 36
  - 4.1.5 Trees . . . 36
  - 4.1.6 Directed Acyclic Graph (DAG) . . . 36
  - 4.1.7 Representations of a Graph . . . 37
- 4.2 Topological Sort . . . 37
  - 4.2.1 Definition . . . 37
  - 4.2.2 Examples . . . 37
  - 4.2.3 Implementation . . . 37
  - 4.2.4 Better Implementation . . . 38
- 4.3 Graph Traversal . . . 39
  - 4.3.1 Introduction . . . 39

4.3.2 Depth-First Search 39
4.3.3 Breadth-First Search 40
4.3.4 Comparison of Depth-First and Breadth-First Searches 40
4.3.5 Iterative Depth-First Search 41
4.3.6 Saving the Shortest Path 41
4.4 Minimum Spanning Tree 41
4.4.1 Definition 41
4.4.2 Algorithm Overviews 41
4.4.3 Kruskal's Algorithm 41
4.4.4 Dijsktra's Algorithm 42
4.4.5 Correctness of Dijsktra's Algorithm 43
4.4.6 Asymptotic Analysis of Dijsktra's Algorithm 43
4.4.7 Dijsktra's Algorithm with a Priority Queue 43
4.4.8 Density vs Sparsity 44

5 Sorting **45**
5.1 Definition 45
5.2 Specification 45
5.3 Variations 45
5.4 Categories of Sorting Algorithms 45
5.5 Insertion Sort 46
5.6 Selection Sort 46
5.7 Heapsort 46
5.8 Divide and Conquer 47
5.9 Mergesort 47
5.10 Quicksort 47

Chapter 1

Asymptotic Analysis

1.1 Big-O Notation

1.1.1 Definition

We have that a function $f(n) \in O(g(n))$ if and only if there exists constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

We have that a function $f(n) \in \Omega(g(n))$ if and only if there exists constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \leq n_0$.

We have that a function $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f \in \Omega(g(n))$.

This notation can be used to analyze the best-case, average-case, and worse-case efficiency of an algorithm, but this class typically concerns the worst-case efficiency of an algorithm.

Note that efficiency can be a measure of time, space, or even power complexity.

1.1.2 Assumption: Constant Operations (Approximations)

- Arithmetic (fixed width)
- Assignment
- Access any array element

1.1.3 Non-Constant Operations

Control Flow	Time
Consecutive statements	Sum of time of each statement
Conditional	Time of test + time of the slower branch
Loop	Number of iterations * time of body
Function call	Time of function body

Control Flow	Time
Recursion	Solve recurrence relation

1.2 Reducing Big-O Expressions

1.2.1 What to Eliminate

- Eliminate low-order terms
- Eliminate coefficients

1.2.2 Examples

- $4n + 5 \in O(n)$
- $\frac{1}{2}n \log n + 2n + 7 \in O(n \log n)$
- $n^3 + 2^n + 3n \in O(2^n)$
- $n \log(10n^2) + 2n \log n \in O(n \log n)$
  - Note that $n \log(10n^2) = 2n \log(10n)$

1.3 Linear Search

1.3.1 Code

```
int find(int[] arr, int arr_length, int k) {
    for (int i = 0; i < arr_length; ++i) {
        if (arr[i] == k) {
            return 1;
        }
    }
    return 0;
}
```

1.3.2 Analysis

- Best case: approximately six steps
  - $O(1)$
- Worst case: $6 * \text{arr\_length}$ steps
  - $n = \text{arr\_length}$, so $O(n)$

1.4 Binary Search

1.4.1 Code

```
int find(int[] arr, int k, int lo, int hi) {
    return help(arr, k, 0, arr_length);
}

int find(int[] arr, int arr_length, int k) {
    int mid = (hi + lo) / 2;
    if (lo == hi) {
        return 0;
    }
    if (arr[mid] == k) {
        return 1;
    }
    if (arr[mid] < k) {
        return help(arr, k, mid + 1, k);
    } else {
        return help(arr, k, lo, mid);
    }
}
```

1.4.2 Analysis

Let $T(n)$ be the efficiency of `find`. Then, because each split takes approximately ten operations, we have that:

$$\begin{aligned} T(n) &= 10 + T\left(\frac{n}{2}\right) \\ &= 10 + \left(10 + T\left(\frac{n}{4}\right)\right) \\ &= 10 + \left(10 + \left(10 + T\left(\frac{n}{8}\right)\right)\right) \\ &= 10k + T\left(\frac{n}{2^k}\right). \end{aligned}$$

To solve this, there are a couple methods.

1.5 Methods of Asymptotic Analysis

1.5.1 Expansion Method

We know $T(1)$, so we should try to express this formula in terms of $T(1)$. To do so, let $\frac{n}{2^k} = 1$, so then $k = \log n$. Then

$$\begin{aligned}
T(n) &= 10\log n + T\left(\frac{n}{2^{\log n}}\right) \\
&= 10\log n + T(1) \\
&= 10\log n + 10 \\
&\in O(\log n).
\end{aligned}$$

However, this method actually gives you a big-theta approximation for T; in other words, not only is $T \in O(\log n)$, we also have that $T \in \Theta(\log n)$.

1.5.2 Substitution Method

IMPORTANT NOTE: This method is *not* recommended because it actually requires an inductive proof (not covered).

Guess $O(?)$, then check. For example (in this case), guess $\log n$ because we have something like $\frac{n}{2^n}$ in the formula. Then:

$$\begin{aligned}
T(n) &= 10 + T(n/2) \\
&= 10 + \log(n/2)
\end{aligned}$$

Because we have guessed that $T \in O(\log n)$, we have that $T(n) \leq c\log n$ for all $n \geq n_0$, for some constants c, n_0.

$$\begin{aligned}
T(n) &\leq c\log n \\
10 + \log(n/2) &\leq c\log n \\
10 + \log(n) - \log(2) &\leq c\log n \\
\frac{10 + \log(n) - \log(2)}{\log n} &\leq c \\
\frac{10}{\log n} + 1 - \frac{\log(2)}{\log n} &\leq c \\
\frac{10}{\log n} + 1 - \frac{1}{\log n} &\leq c
\end{aligned}$$

Now take $n_0 = 2$, so $n \geq 2$ and thus $\log n \geq 1$ (log is base two). We then have:

$$\begin{aligned}
c &\geq \frac{10}{1} + 1 - \frac{1}{1} \\
c &\geq 10
\end{aligned}$$

Therefore, $T \in O(\log n)$ because with $c = 10$ and $n_0 = 2$, we have that $T(n) \leq c\log n$ for all $n \geq n_0$.

Note that the substituion method is more general than the **Expansion Method**, but it does *not* give you a big-theta approximation (unlike the **Expansion Method**).

1.6 The Towers of Hanoi

1.6.1 Gameplay

The goal of the Towers of Hanoi is to move all disks to goal peg, with the following rules:

- You can only move one disk at a time
- You can only move the top-most disk in a pile
- You cannot put a larger disk on top of a smaller one

1.6.2 Algorithmic Solution

```
if n = 1:                                 => T(1)
    move to goal (base case)                   = 1
else:                                     => T(n)
    move top n-1 disks to temporary peg        = T(n - 1)
    move bottom disk to goal                   + T(1)
    move the n-1 disks to goal                 + T(n - 1)
```

Thus, $T(n) = T(n-1) + T(1) + T(n-1)$, with $T(1) = 1$. So:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 4T(n-2) + 3 \\ &= 4(2T(n-3) + 1) + 3 \\ &= 8T(n-3) + 7. \end{aligned}$$

By inspection, we have:

$$T(n) = 2^{n-1}T(1) + (2^{n-1} - 1),$$

but $T(1) = 1$, so we have:

$$\begin{aligned} T(n) &= 2^{n-1} + 2^{n-1} - 1 \\ &= 2^n - 1 \\ &\in \Theta(2^n). \end{aligned}$$

1.7 Mergesort

1.7.1 Description

To mergesort a list, split the list into two and sort the sublists. To merge them back together, interleave the elements. Interleaving / merging is $O(n)$ and there are $O(\log n)$ splits, so mergesort is $O(n \log n)$.

Mergesort is based on the trick that it is really easy to interleave two sorted lists.

1.7.2 Example

```
8 2 9 4 5 3 1 6
=> 8 2 9 4
   => 8 2
      => 8
      => 2
      merge: 2 8
   => 9 4
      => 9
      => 4
      merge: 4 9
   merge: 2 4 8 9
=>  5316
   => 5 3
      => 5
      => 3
      merge: 3 5
   => 1 6
      => 1
      => 6
      merge: 1 6
   merge: 1 3 5 6
merge: 1 2 3 4 5 6 8 9
```

1.7.3 Analysis

We have that $T(1) = 1$ and $T(n) = 2T(n/2) + n$ (split into two sublists, then mergesort them, then merge / interleave them). We then have that:

$$\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(T(n/4) + n/2) + n \\
&= 2T(n/4) + 2n \\
&= 4(2T(n/8) + n/8) + 2n \\
&= 8T(n/8) + 3n \\
&= 2^k T(n/2^k) + kn
\end{aligned}$$

Set $n/2^k = 1$ (because we are using the **Expansion Method**), so $k = \log n$. Then:

$$\begin{aligned}
T(n) &= 2^{\log n} T(n/2^{\log n}) + (\log n)n \\
&= nT(1) + n \log n \\
&= n + n \log n \\
&\in \Theta(n \log n)
\end{aligned}$$

Chapter 2

Algorithm Correctness

2.1 Key Parts of an Algorithm

There are a couple key things that every algorithm needs:

- Inputs
- Outputs
- Preconditions (restrictions on input)
- Postconditions (restrictions on output)
- Step-by-step process specification (either in natural language or pseudocode)

Therefore, we can define a "correct" algorithm to be one that, given any input data that satisfies the precondition, produces output data that satisfies the postcondition *and* terminates (stops) in finite time.

2.2 Proving Correctness

2.2.1 Basic Example

Consider the following pseudocode to swap two variables:

```
swap(x, y)
    aux := x
    x := y
    y := x
```

The following is a proof of correctness of `swap`:

1. Precondition: `x = a` and `y = b`.
2. Postcondition: `x = b` and `y = a`.
3. `aux := x` implies `aux := a`.
4. `x := y` implies `x := b`.
5. `y := aux` implies `y := a`.

6. Thus, `x := b` and `y := a`, so the postcondition is satisfied.

2.2.2 Loop Invariants

A *loop invariant* is a logical predicate that, if true before any single iteration of a loop, then is also true after the iteration of the loop. It is called an “invariant” because it is always true.

When using induction to prove the correctness of an algorithm, the loop invariant is the inductive hypothesis.

Consider the following psuedocode to sum the first `N` elements of a list `a`:

```
sum_list(a, N)
    s := 0
    k := 0
    while (k < N):
        s := s + a[k]
        k := k + 1
```

To prove this algorithm is correct, we must show:

1. The loop invariant (that at step `k`, `s =` sum of first `k` numbers in `a`) holds true; and
2. The algorithm terminates.

We want to show that, at step `k`, `s =` sum of first `k` numbers in `a`. Hence, we will induct on `k`.

1. At `k = 0`, we have that `s = 0`, so the algorithm is correct for `k = 0`.
2. We will assume that the algorithm holds for some arbitrary `k`.
3. We will now prove that the algorithm holds for `k + 1`. In the `(k + 1)`-th iteration of the loop, we set `s = s + a[k + 1]`, so `s` is the sum of the first `k` numbers (because, by the induction hypothesis, before the iteration of the loop we have that `s` is the sum of the first `k` numbers) plus `a[k + 1]`; i.e., it is the sum of the first `k + 1` numbers.

Hence, the loop invariant holds for all `k`. However, we must also prove that the algorithm terminates. For each iteration, we have that `k := k + 1`, so after `N` iterations, `k = N`, and hence the loop will terminate.

2.2.3 Strong Induction

Strong induction is just like regular induction, but instead of assuming the inductive hypothesis for some `k`, we assume it for all values less than or equal to `k`. We will use strong induction in the proof of correctness of mergesort.

2.2.4 Mergesort

Consider the following pseudocode of mergesort:

```
mergesort(A, l, r):
    if l < r:
        m = floor((l + r) / 2)
        mergesort(A, l, m)
        mergesort(A, m + 1, r)
        merge(A, l, m, r)
```

Note that the precondition of `merge` is that `A[l ... m]` and `A[m + 1 ... r]` are sorted, and the postcondition is that `A` is sorted.

The precondition for this algorithm is that `A` has at least `1` element between the indices `l` and `r` (otherwise, this code doesn't have anything to do). The postcondition is that the elements between `l` and `r` are sorted.

To prove the correctness of this algorithm, we must show that the postcondition holds. To do so, we will prove that `mergesort` can sort `n` elements (and so it will therefore be able to sort `n = r - l + 1` elements).

The base case of `n = 1` is true because `A` is simply a one-element list (which is always sorted).

We will let the inductive hypothesis be that mergesort correctly sorts `n = 1...k` elements (i.e., everything less than or equal to `k`) This is strong induction.

To show the inductive step (`n = k + 1`) true, consider the two recursive calls in the function.

1. For the first recursive call, we have that `mergesort` is sorting `m - l + 1 = (k + 1) / 2 <= k` elements, so by the inductive hypothesis, `mergesort` holds. Hence, `A[l ... m]` is sorted.
2. For the second recursive call, we have that `mergesort` is sorting `r - (m + 1) + 1 = r - m = (k + 1) / 2 <= k` elements, so by the inductive hypothesis, `mergesort` holds. Hence, `A[m + 1 ... r]` is sorted.

We therefore have that the precondition of `merge` is upheld, so the postcondition of `merge` must also be upheld. Therefore, we have that `A` is sorted, and so the postcondition of `mergesort` is upheld.

Lastly, we must show that `mergesort` terminates. Note that for each recursive call, the length of the subarray between `p` and `q` decreases, so eventually it must reach a point where there is only one element in the array, in which case we reach termination.

Chapter 3

Data Structures I

3.1 Dictionaries

3.1.1 Description

A *dictionary* is a set of `(key, value)` pairs. The keys must be comparable. A dictionary has the following operations:

- `insert(key, value)`
- `find(key)`
- `delete(key)`

3.1.2 Implementations

Here are some possible underlying data structures for the dictionary:

	`insert`	`find`	`delete`
Unsorted linked list	`O(1)`	`O(n)`	`O(n)`
Unsorted array	`O(1)`	`O(n)`	`O(n)`
Sorted linked list	`O(n)`	`O(n)`	`O(n)`
Sorted array	`O(n)`	`O(log n)`	`O(n)`
Balanced tree	`O(log n)`	`O(log n)`	`O(log n)`
“Magic array”	`O(1)`	`O(1)`	`O(1)`

A “magic” array is a hashtable!

3.1.2.1 Side Note: Lazy Deletion

Rather than actually deleting and shifting the contents of an array, you can just “mark” it as deleted. This is simpler, faster, and you can do removal in batches, but takes up extra space and may complicate the implementation.

3.1.3 Better Implementations

The following are some better implementations for a dictionary:

1. Binary trees
2. AVL trees (guarenteed to be balanced)
3. B-trees (guarenteed to be balanced)
4. Hash tables

We will implement a dictionary as a binary tree where the data of each node is a `(key, value)`, and the left child has `key` that is less than (or equal to) that of the parent node, and the right child has a `key` that is greater than that of the parent node. Note that such an ordering must be defined because we require that the keys are comparable.

3.2 Trees

3.2.1 Terminology

Here is some terminology for trees:

- Root: top of tree
- Leaves: at bottom of tree
- Children: the nodes directly below (and attached to) another node
- Parent: the node to which children are attached
- Siblings: two nodes are siblings if they are children to the same parent
- Descendants: subtree with a particular node at root; includes children, children of children, etc.
- Depth (of a node): distance to root
- Height (of a tree): distance from root to furthest leaf
- Branching factor: how many children each node can have (for a binary tree, this is 2)
- A *balanced* tree has height $\in O(\log n)$, where n is the number of nodes.

3.3 Binary Trees

3.3.1 Description

Here are some facts that are true of any binary tree with height h:

- The maximum number of leaves is 2^h

- The maximum number of nodes is $2^h + (2^h - 1) = 2^{h+1} - 1$
- The minimum number of leaves is 1
- The maximum number of leaves is $h + 1$

Here is a function to determine the height of a tree:

```
int tree_height(Node *root) {
    if (root == NULL) {
        return -1;
    } else {
        return 1 + max(tree_height(root->left, root->right));
    }
}
```

3.3.2 Binary Tree Traversal

Suppose that we have a binary tree defined as follows:

```
+
  *
    2
    4
  5
```

Then we can process t either in-order, pre-order, or post-order.

- In-order: `2 * 4 + 5`
- Pre-order: `+ * 2 4 5`
- Post-order: `2 4 * 5 +`

Here is a slightly more abstract example of ordering. Consider the following tree:

```
A
  B
    D
    E
  C
    F
    G
```

Then:

- In-order: `DBEAFCG`
- Pre-order: `ABDECFG`
- Post-order: `DEBFGCA`

Note that, given an order trace, one cannot determine the structure of the original tree (but given a pre/post-order trace *and* an in-order trace, one can do so).

The following code performs an in-order traversal of a tree:

```
void in_order_traversal(Node *t) {
    if (t != NULL) {
        // for pre-order, process here instead
        in_order_traversal(t->left);
        process(t->data);
        in_order_traversal(t->right);
        // for post-order, process here instead
    }
}
```

3.4 AVL Trees

3.4.1 Motivation

For a binary treee, we have that:

- Each node has less than or equal to two children
- Operations are simple
- Order property
    - All keys in left subtree are smaller
    - All keys in right subtree are larger
    - This gives us $O(\log n)$ find — **but only if balanced**

This is what an unbalanced tree looks like:

```
A
  B
    C
      D
        E
```

It basically is a linked list, so `find`, `insert`, and `delete` are all $\Theta(n)$. If you wanted to build this tree (with n items), it would be $\Theta(n^2)$. This is bad, so we want to keep the tree *balanced*.

We must keep balance not just at build time, but also every time we insert or delete. To do this, we will need to define what constitutes a “balanced” tree. However, the conditions cannot be too weak (useless) or too strong (impossible). The sweet spot is the **AVL condition**.

Here are some possible conditions, and why they are not really that good.

1. Left + right subtrees of the root have the same number of nodes.
    - Too weak because the left and right subtrees can just themselves be really unbalanced.
2. Left + right subtrees of the root have the same height.
    - Too weak because he left and right subtrees might just both be linear (i.e. look like linked lists).
3. Left + right subtrees of **every node** have equal number of nodes.
    - This ensures that the tree will always be perfect, but is too strong because this is extremely difficult/expensive to maintain.
4. Left + right subtrees of **every node** have equal height.
    - Same thing as previous condition; too strong.

So, what is the solution? The AVL condition:

- Left + right subtrees of every node differ by at most 1.

3.4.2 Example

The following is a valid AVL tree:

```
6 (3)
  4 (1)
    1 (0)
    _
  8 (2)
    7 (0)
    11 (1)
      10 (0)
      12 (0)
```

The numbers in parentheses are the height of that particular node. The height is defined as the maximum distance to the bottom. It can recurisely be defined as `max(height(child1), height(child2)) + 1`; i.e., one more than the maximum height of the children.

3.4.3 Maintaining the AVL Condition

There are four cases that we have to handle when updating the AVL tree to make sure that it is valid. Fixing invalid AVL trees is done via a technique known as *rotation*.

Note that, in the following examples, `[x]` denotes an arbitrary (sub)tree.

3.4.3.1 Left-Left

```
a (h+3)
  b (h+2)
    [x] (h+1; originally h)
      *added element*
    [y] (h)
  [z] (h)
```

becomes

```
b (h+2)
  [x] (h+1)
    *added element*
  a (h+1)
    [y] (h)
    [z] (h)
```

3.4.3.2 Right-Right

```
a (h+3)
  [x] (h)
  b (h+2)
    [y] (h)
    [z] (h+1; originally h)
      *added element*
```

becomes

```
b (h+2)
  a (h+1)
    [x] (h)
    [y] (h)
  [z] (h+1)
    *added element*
```

3.4.3.3 Right-Left

This utilizes a **double rotation**, but it is helpful to just think of moving the problematic node to its grandparent position, then just putting everything else back in how it fits (which will be unique).

```
a (h+3)
  [x] (h)
  b (h+2)
    c (h+1)
      [u] (h; originally h-1)
        *added element*
      [v] (h-1)
    [z] (h)
```

becomes

```
c (h+3)
  a (h+2)
    [x] (h+1)
    [u] (h)
      *added element*
  b (h+1)
    [v] (h-1)
    [z] (h)
```

3.5 Hashtables

3.5.1 Description

A hashtable maps keys to indices in an array via a hashing function. Keys can be anything, so long as the hash function maps it to an integer.

For insert, find, and delete, hashtables are really fast. However, the hashtable is just a random mapping between keys and indices, so there is no ordering. Hence, things like `findmin`, `findmax`, `predecessor`, and `successor` would all be $O(n)$. AVL trees would be a better choice if you need these functions.

Hashtables are useful if the key-space is very large, but the actual number of keys that you use is small. Note that the array backing a hashtable is almost always much smaller than the key-space.

Here are some examples of when it would be good to use a hashtable:

- Compiler: all possible identifiers for a variable vs. those used in a program. This is a large key-space with a small number of keys actually used, so a hashtable would be useful here.
- Database: all possible student names vs. the students in a class.
- AI: all possible chess configurations vs. ones seen in a single game

3.5.2 Example Hash Function

Here is a simple hash function:

$$h(k) = k \ \% \ n,$$

where k is the key and n is the table size.

If you have a bunch of numbers that are multiples of each other as your key, then you would want the table size to be prime (or just coprime to the multiple in question) in order to minimize collisions.

3.5.3 Collisions

A *collision* occurs when two keys map to the same index. One way to resolve this is to utilize *chaining*. Chaining is when you store a linked list at each index of the array (instead of just the key). Then, once you hash a key, you can look in / append to the linked list for it. If you have a really bad hashing function and you chain, then your hashtable just becomes a linked list.

3.5.4 Load Factor

When chaining, we can define λ to be the *load factor* of a hashtable, which corresponds to the average number of elements in each *bucket* (index in the array). We have that $\lambda = k/n$, where k is the number of elements and n is the tablesize.

One application of this is that we can be more precise in our analysis of `find`. We have that `find` is in $O(\lambda)$ in the unsuccessful case, and $O(\lambda/2)$ in the successful case (so it is all really $O(\lambda)$).

3.5.5 Linear Probing

Instead of having a linked list at each index (which takes more memory and is dynamically allocated), we can use a technique called *linear probing.* If there is a collision, then just store the key in the next available spot in the array. The hash function then becomes

$$h'(k) = h(k) + i,$$

where i is the number of times that you have tried to place the key.

However, you must be careful when finding and deleting. When you are looking for an element, you have to hash it, then do linear search until you either find the key or you find an empty spot. However, when deleting, you must use lazy deltion, otherwise you would mess up the linear probing search.

This is not usually used in practice. Quadratic probing is preferred.

3.5.6 Quadratic Probing

Quadratic probing is just linear probing, but instead of storing the colliding key in the next index, you store it according to:

$$h'(k) = h(k) + i^2,$$

where i is the number of times that you have tried to place the key. This ensures that the indices are more spread out, so clusters don't arise (the goal of a hashtable is to have a good spread of indices). It does so by changing the "stride" of the probing, so that if a key is placed in what was a long series of probes for some other key, the hash function doesn't have to go through the entire series (it goes through a different one for each key).

Quadratic probing avoids the dynamic allocation of memory present in the chaining technique, while at the same time improving upon the naive linear probing method.

There are, however, some issues with quadatic probing. If you are unlucky, you may get a *cycle*, where the hashing function keeps jumping on the same (full) indices over and over again, never finding an open slot (even if one exists). To combat this, you can make the table size prime and take $\lambda < 1/2$. There is a proof that if these conditions are upheld, then quadratic probing will find an empty slot in $n/2$ probes, where n is the size of the table (this is not a very good bound, but at least it's something).

3.5.7 Double Hashing

Double hashing is like linear probing, but with an extra hash function that adjusts the stride of the probing. We let

$$h'(k) = h(k) + i * g(k),$$

where h and k are hashing functions, and i is the number of times that you have tried to place the key. The one caveat is that it must be the case that $g(k) \neq 0$ for any key k, otherwise the probing would never advance.

One good implementation of double hashing defines h and g as:

$$\begin{aligned} h(k) &= k \ \% \ p \\ g(k) &= q - (k \ \% \ q), \end{aligned}$$

where p, q are prime and $2 < q < p$. These two functions result in no cycles for double hashing, which is really nice (proof not given).

3.5.8 Universal Hash Functions

Define a hashing function by:

$$h(k) = ((ak + b) \ \% \ p) \ \% \ n,$$

where n is the table size, p is a prime such that $p > m$, and a and b are random integers (fixed for a particular function, i.e. h is referentially transparent).

Because h has a couple parameters, we can easily change it if we want to, but then we will have to rehash the table. This is a very expensive process, but can be useful; for example, if we detected that the number of collisions is greater than some threshold, we can dynamically change the hash function, double the table size (for good measure), and then rehash the table. We combine the operations of changing the hash function and the table size, because we have to rehash each entry after doing one of these operations, so it makes sense to combine it all in one so that rehashing only has to occur once.

3.5.9 Perfect Hash Functions

Perfect hashing must be done statically (no one knows how to do it dynamically). To hash n keys perfectly, randomly generate some universal hash functions with table size n^2 until there are no collisions (with n keys and a table size of n^2, the probability of a collision is less than 1/2, so it is more likely than not that a random universal hash function will work after a couple tries).

Unfortunately, n^2 is not that good. To improve this, we can make take a table with $O(n)$ size, use a universal hash function, and for each collision, generate another hash table that is hashed perfectly. Each of these smaller hash tables will only have m keys (where m is the number of collisions), so hashing perfectly will only take m^2 slots. Note that m will (hopefully) be small, so this is better than n^2 slots.

3.6 Union-Find Data Structure

3.6.1 Mathematical Definitions

A *set* is a collection of elements with no repeated elements. Two sets are said to be disjoint if they have no elements in common. For example, $\{1, 2, 3\}$ and $\{4, 5, 6\}$ are disjoint, as are $\{a, b, c\}$ and $\{t, u, x\}$.

A *partition* of a set S is a set of sets $\{S_1, S_2, \ldots, S_n\}$ with each $S_i \subseteq S$ such that every element of S is in **exactly one** S_i.

The *Cartesian product* $A \times B$ of two sets A, B is the set of all pairs where the first element in the pair is from A and the second element is from B. In particular, for any set S, we have that $S \times S$ is the set of all pairs of elements in S.

A *binary relation* R on some set S is any subset of $S \times S$. For example, let S denote the people in a room. Some binary relations could be the people sitting next to each other, where the first element of the pair is sitting to the right of the second element.

An *equivalence relation* is a binary relation which is *reflexive*, *symmetric*, and *transitive*.

- Reflexive: x relates to x for all $x \in S$.
- Symmetric: if a relates to b, then b relates to a for all $a, b \in S$.
- Transitive: if a relates to b and b relates to c, then a relates to c for all $a, b, c \in S$.

Equivalence relations create a partition of a set. Furthermore, every partition of a set gives you an equivalence relation (let the relation be “Are these two elements in the same partition?”).

3.6.2 The Union-Find Algorithm

The union-find algorithm keeps track of a set of elements partitioned into a number of disjoint subsets. It first partitions the set into a bunch of one-element sets, then unions all the sets that are related to each other to form more useful sets.

Here are some uses of this:

- Road connectivity
- Connected components of social network graph
- Connected components of an image
- Type inference in programming languages (yasssssssss)

In general, connected components are the most useful application of union-find.

Union-find is a much more specialized data structure than, for example, AVL trees, but it is still useful nonetheless.

3.6.3 Union-Find Operations

Given an unchanging set S:

- `create()`
  - Generate initial partition of a set where each element gets their own subset (typically)
  - Give each subset a "name", usually by choosing a "representative element"
- `find(e)`
  - Take element e of S and returns the "name" of the subset containing e.
- `union`
  - Take two subsets and (permanently) make a larger subset, then choose a new name for this new subset
  - From this, we get a different partition of the set S with one fewer set
  - This affects future `find` operations

3.6.4 A Cute Application of Union-Find

We can create a maze-building application with union find! A naive way to make a maze would be to simply delete random edges. However, this is bad because it can create cycles in the maze, which is undesirable.

A more sophisticated algorithm would be to randomly remove edges, but only if doing so does not introduce a cycle. We can do this with union-find! To implement this using union-find, give each cell in the maze a number. Define an "edge" between adjacent cells x and y to be the pair (x, y).

we must now partition the cells into disjoint sets, such that if a cell x is in a subset with a cell y, then x is connected to y. (Initially, each cell is in its own subset.) If removing an edge would connect two subsets, then remove the edge and union the subsets. Otherwise, leave the edge (because this would actually introduce a cycle; i.e. connect a cell to another cell in its own subset).

More precisely:

- Let P be a partition of the cells
- Let E be the set of edges not yet processed
- Let M be the set of edges that are kept in the maze

```
while P has more than one subset:
    pick a random edge (x, y) to remove from E
    u = find(x)
    v = find(y)
    if u == v:
        add (x, y) to M // mark (x, y) as "don't remove"
    else:
        union(u, v)
    remove (x, y) from E
add remaining edges of E to M
```

This is a unique way of using union-find, because normally when you use union-find, you simply want to grow the connected components.

3.6.5 Up-Tree

An up-tree is a tree that only keeps track of what is above it. We start with a “forest” of trees, where each tree is just one node. When we want to union two elements, we choose one of the elements to be the parent of the other, and connected the child going to the parent (but not the other way around). With this schema, we have that the root of the up-tree is the representative element of the set, so to find what set a node is in, simply follow the tree up that the node is in.

The simplest implementation of an up-tree is just an array. This will work if each of the nodes are contiguous numbers (otherwise, there will need to be some sort of translation process, perhaps using a hashtable or a dictionary).

So, if the set elements are contiguous numbers $[1...n]$, we will use an array A of length n, initially filled with zeros, where a zero in index i means that node i has no parent. This makes sense because initially no node has a parent. If you want to make node p a parent of node c, simply set $A[c] = p$.

Then `find` will look like this:

```
find(x):
    lookup x in the array
    follow parent until 0 // until the root
    return root // root is the representative element of the set
```

Similar to AVL trees, we want to make sure that the up-tree actually looks like a tree, and not a linked list. Therefore, when implementing `union` we must be careful to make sure the tree stays somewhat balanced.

One interesting thing to note is that the *in-degree* of an up-tree is not limited at all. That is to say, there can be arbitrarily many nodes pointing to a parent node, and in fact this is more efficient because there are less parents to traverse. This is in contrast to normal downward trees, where more children equates to more branching and less efficiency.

Here is what `union` will look like:

```
union(u, v):
    set A[u] = v or A[v] = u // you have a choice
```

So what should our choice be? There are two key optimizations that we can implement to ensure that A stays tree-like.

1. Union by size; connect smaller tree to larger tree (keeps traversal lengths down for the larger tree). This ensures that the tree is approximately balanced, so `find` $\in O(\log n)$.
2. Path compression; connect a node directly to the root during each `find`. This actually makes union-find $O(\log^* n)$ (iterated logarithm). This is basically constant because it grows *extremely* slowly!

To implementing union by size with an array, we can keep track of the "weight" (size) of each root in the array. This can be done in an additional `weight` array, or you can simply keep the value `-weight` instead of zero in the root; the negative number signals that it is not a pointer to a parent, and the magnitude is the weight. With this, we have that `union` $\in O(1)$.

Now we shall consider the Big-O complexity of `find`. First note that, if we union by size, an up-tree with size h will have at least 2^h nodes. Here is a proof by induction:

- **Base case ($h = 0$).** Because $h = 0$, the tree must have only one node. We have that $2^h = 2^0 = 1$, and so the tree has at least 2^h nodes.
- **Inductive hypothesis**. Assume that an up-tree with size h has at least 2^h nodes for some $h \in \mathbb{N}$.
- **Inductive case**. We will consider a tree T with height $h + 1$. We know that T must be the union of two smaller subtrees (T_1 and T_2), each of height h. By the inductive hypothesis, both T_1 and T_2 have at least 2^h nodes. Note that the number of nodes in T is equal to the sum of the number of nodes in T_1 and T_2, so the number of nodes in T is at least $2^h + 2^h = 2^{h+1}$. Therefore, an up-tree with size $h + 1$ has at least 2^{h+1} nodes.

Thus, by the principle of induction, the proposition holds for all $h \in \mathbb{N}$. With this, we know that `find` $\in O(\log n)$, because we know that the tree will never grow by more than a factor of two.

3.7 Priority Queue

A *priority queue* holds comparable data (data with an ordering) and keeps track of the minimum element. The minimum element is said to have the highest *priority*; thus, the priority queue keeps track of the element with the highest priority.

3.7.1 Binary Min-Heap

The most common implementation of this data structure is a *binary min-heap*, also known as a *binary heap*. A min-heap is a tree consisting only of nodes whose children are larger than it (a max-heap would be a tree consisting only of nodes whose children are smaller than it). Thus, in a min-heap tree, the root of the tree is the smallest element.

A binary heap has three operations: `insert`, `delete_min`, and `is_empty`.

For `delete_min`, do the following steps:

1. Remove the root node
2. Move the rightmost node in the last row to the root of the tree
3. "Percolate down"

To percolate, compare the new root with each of its children, and swap it with the one with higher priority. Propogate this downward until the node that is being percolated reaches the bottom, or has higher priority than both its children.

Thus, `delete` $\in O(\log n)$ because it will have to percolate through at most $\log n$ nodes in the tree (because a balanced tree has height $\log n$).

For `insert`, do the following steps:

1. Put the node in the bottom right of the three
2. "Percolate up" (do the swapping thing described in the previous paragraph but upward)

Thus, `insert` $\in O(\log n)$ because it will have to percolate through at most $\log n$ nodes in the tree (because a balanced tree has height $\log n$).

To implement a binary-min heap, an array works nicely because we are dealing with perfect binary trees.

3.7.2 Array Representation of Binary Trees

Say you have the following binary tree:

```
A
  B
    D
      H
      I
    E
      J
      K
  C
    F
      L
    G
```

This is a perfect binary tree. You can represent it in an array as follows:

```
0 1 2 3 4 5 6 7 8 9 10 11 12
  A B C D E F G H I J  K  L
```

Then, from node i...

- To get the left child, go to index $i * 2 = i << 1$
- To get the right child, go to index $i * 2 + 1 = i << 1 + 1$
- To get the parent, go to index $i/2 = i >> 2$

This is really fast because multiplication and division by two can be computed with left and right bit shifts, which are really fast.

3.7.3 Fast Way to Build Heap

If you wanted to insert n items, it would normally be $O(n \log n)$, but *Floyd's Method* makes it $O(n)$. In Floyd's Method, you put the leaves in the binary tree array in any order, then percolate row-by-row.

Chapter 4

Graphs

4.1 Introduction to Graphs

A *graph* is an abstraction that represents relationships between objects.

4.1.1 Undirected Graphs

An undirected graph G consists of a set of *vertices* (also known as *nodes*) and a set of *edges*, denoted $G = (V, E)$. Here are some details about the components of an undirected graph:

- V: set of vertices $\{v_1, \cdots, v_n\}$ with $|V| = n$
- E: set of edges $\{e_1, \cdots, e_m\}$ with $|E| = m$
    - For each $i \in [m]$, we have that e_i is an *unordered* pair of vertices (endpoints). We can denote this as $e_i = (v_j, v_k)$.
- We have that $0 \leq m \leq \binom{n}{2}$, so $m \in O(n^2)$

Here is some terminology for an undirected graph:

- The *degree* of a vertex is the number of edges that are connected to it.
- A *path* is a sequence of vertices where each vertex is connected with the next by an edge.
- A *simple path* is a path with no vertices.
- The *path length* of a path is the number of **edges** (not vertices) in a path.
- A *cycle* is a path with the same starting and ending vertices.
- The *cycle length* of a cycle is the number of edges in a cycle.
- A *self-loop* is when a vertex is connected to itself via an edge.
- Two edges are said to be *parallel* if they both form the same connection between two vertices (so one is essentially redundant)
- A graph is said to be *dense* if $m \in \Omega(n^2)$ (thus implying that $m \in \Theta(n^2)$)
- A graph is *sparse* if $m \in O(n)$

4.1.2 Directed Graphs (Digraph)

Most of the terminology is the same between an undirected and a directed graph, but now edges are **ordered** pairs of vertices; i.e., there is a starting point and an ending point.

We also now have that $0 \leq m \leq n^2$, so $m \in O(n^2)$ (same as before).

Additionally, instead of each vertex having a degree, they have an *in-degree* (the number of in-bound edges) and an *out-degree* (the number of out-bound edges).

4.1.3 Weighted Graphs

- A *weighted graph* is a graph where each edge also has a *cost* (or *weight*).
- The *path cost* of a path is the sum of the costs of the edges in the path.

4.1.4 Connectivity and Completeness

4.1.4.1 Undirected Graphs

- An undirected graph G is said to be *connected* if, for all pairs of vertices (u, v) there exists a **path** from u to v.
- An undirected graph G is said to be *complete* if, for all pairs of vertices (u, v) there exists a **edge** from u to v.

4.1.4.2 Directed Graphs

- A directed graph G is said to be *strongly connected* if, for all pairs of vertices (u, v) there exists a **path** from u to v.
- A directed graph G is said to be *weakly connected* if, for all pairs of vertices (u, v) there exists a **path** from u to v **if we ignore directionality**.
- A directed graph G is said to be *complete* (or *fully connected*) if, for all pairs of vertices (u, v) there exists a **path** from u to v.

4.1.5 Trees

- A *tree* is an undirected, acyclic, connected graph.
- A *forest* is the union of trees. It is a tree that does not have to be connected.
- A *rooted tree* is a tree with a specified root node. It does not have to be undirected.

4.1.6 Directed Acyclic Graph (DAG)

A *directed ayclic graph* is a digraph with no directed cycles. It is similar to a tree, but may have directed cycles (so long as they are not also undirected cycles).

4.1.7 Representations of a Graph

Here are a few common representations of a graph:

- **Edge set/list**: a set/list of edges (as suggested by the mathematical definition of a graph)
- **Adjacency matrix**: a 2D array of booleans indicating whether or not two vertices are connected
  - Note that undirected graphs will be symmetric
- **Adjacency list**: a 1D array where each cell represents a vertex and points to a linked list of the vertices that it is connected to

4.2 Topological Sort

4.2.1 Definition

A *topological sort* works on a DAG $G = (V, E)$; it outputs vertices in order such that no vertex appears before another that has an edge to it.

Why must a topological sort only work on DAGs? If the graph is not cyclic, then it is impossible to sort, and if its not directed, then there is no ordering.

Is there always one unique topoligcal sort for a DAG? No, consider the following tree:

```
A
  B
  C
```

Both `ABC` and `ACB` are valid topological sorts.

More formally, a DAG gives you a partial ordering, and a topological sort gives you a total order that is consistent with the partial order of a DAG.

4.2.2 Examples

Here are some examples when you might want to use a topological sort on a DAG:

- Compiling code
- Course order to graduate
- Makefile (determining order to compile files)

In general, you may want to use a topological sort whenever you have a dependency graph of some sort.

4.2.3 Implementation

Here is a naive implementation of a topological sort:

```
NaiveTopologicalSort(DAG):
    // you could do the following by adding an in-degree field to the vertex
    // (more common), or you could use another structure (like an array)
    label each vertex in DAG with its in-degree

    while there are vertices in DAG to output:
        // because the following choice is arbitrary, that is why topological
        // sort is not unique
        choose a vertex v with in-degree = 0 from DAG

        output v

        // "conceptually" remove v from DAG:
        for each vertex u adjacent to v:
            decrement in-degree of u
```

Let's take a look at the runtime of this topological sort:

When analyzing graph algorithms on a graph $G = (V, E)$, it is common to write the Big-O notation in terms of $|V|$ and $|E|$.

- The initialization portion of topological sort takes $O(|V| + |E|)$ time. The part where $|V|$ is added is more of an edge case; it accounts for the case in which the number of vertices is greater than the number of edges (like if there are a lot of singletons)
- Finding a new vertex with `in-degree = 0` takes $O(|V|)$ times (look through all vertices to find one)
- Total number of times a new vertex need to be found is $O(|V|)$, so total finding process is $O(|V|^2)$
- Thus, total running time is $O(|V| + |E| + |V|^2) = O(|V^2| + |E|)$. However, for a DAG, it is always true that $|E| < |V|^2$, so we have that this topological sort is in $O(|V^2|)$.

4.2.4 Better Implementation

This is a better implementation of a topological sort using a queue:

```
BetterTopologicalSort(DAG):
    label each vertex with an in-degree
    enqueue all the zero-degree nodes
    while the queue is not empty:
        v = dequeue()
        output v
        remove(v)
        for each u adjacent to v:
            decrement in-degree of u
            if new degree of u = 0:
                enqueue u
```

Let's take a look at the runtime of this new topological sort:

- Initialization: $O(|V| + |E|)$. This is the same as before.

- Sum of all enqueues and dequeues: $O(|V|)$. Every vertex gets enqueued exactly once and dequeued exactly once.
- Sum of all decrements: $O(|E|)$. Each in-degree corresponds to an edge, so we must decrement exactly as many times as there are edges.
- Total: $O(|V| + |E|)$. Note that now $|V|$ may not dominate $|E|$, so we need to include both.

4.3 Graph Traversal

4.3.1 Introduction

Given a start node v, a *graph traversal* will find all reachable nodes from v; i.e., the nodes for which there exists a path from v. Note that graph traversals work on all graphs, not just DAGs.

To do so, we will just keep following nodes. We will also “mark” visited nodes so that we process each node only once, and (more importantly) so that the algorithm will actually know when to terminate.

Let’s take a look at the pseudocode for a general traversal:

```
TraverseGraph(startNode):
    set pending = empty set
    pending.add(startNode)
    while pending is not empty:
        next = pending.remove()
        for each node u adjacent to next:
            if u is not marked:
                mark u
                pending.add(u)
```

And also the running time! We will assume that adding and removing to a set are $O(1)$. Then, the entire traversal is $O(|E|)$ because you have to check (at most) *all* the edges.

Now let’s look at some more specific traversals.

4.3.2 Depth-First Search

A depth-first search goes as deep as possible, then backtracks. Here’s the pseudocode:

```
DFS(startNode):
    mark and process startNode
    for each node u adjacent to startNode:
        if u is not marked:
            DFS(u)
```

Let’s say you have a tree like this:

```
A
  B
    D
    E
  C
    F
      G
      H
```

Then a depth-first seach might go in the order `ABDECFGH` (this would be called pre-order).

Here is another implementation of depth-first search that uses an explicit stack rather than recursion:

```
DFS2(startNode):
    stack_push(startNode)
    mark startNode as visited
    while stack_not_empty():
        next = stack_pop()
        for each node u adjacent to next:
            if u is not marked:
                mark u
                stack_push(u)
```

However, this implementation is more of a "last-in first-out" implementation, and so it goes right-to-left (post-order). Thus, it would process the above tree as `ACFHGBED`.

4.3.3 Breadth-First Search

A breadth-first search advances in a "frontier"; i.e., level by level of the DAG. Here's the pseudocode (it requires a queue):

```
BFS(startNode):
    enqueue(startNode)
    mark startNode as visited
    while queue_not_empty():
        next = dequeue()
        for each node u adjacent to next:
            if u is not marked:
                mark u
                enqueue(u)
```

Basically, it is exactly the same the second implementation of depth-first search, but every time there is a stack operation it's a queue operation.

This breadth-first search would process the above tree as `ABCDEFGH`.

4.3.4 Comparison of Depth-First and Breadth-First Searches

- Depth-first search uses less memory

  - If p is the length of the longest path and d is the highest out-degree, then the size of the DFS stack is less than $d * p$ elements.
  - On the other hand, the breadth-first search can take order $O(|V|)$ space for the queue.

4.3.5 Iterative Depth-First Search

Pick some number k; then, do a depth-first search that is k levels deep. If this fails, try again with a larger k. THis is a compromise between DFS and BFS, it takes the order $O(d * p)$ space of DFS, but also kind of acts like a "frontier" as in BFS. If you don't find the element you are looking for in the first iteration of the algorithm, you have to re-traverse everything that you already did but with a bigger k, so it can be inefficient.

4.3.6 Saving the Shortest Path

So far, these searches have answered the question "is there a path?", but sometimes you need to actually store the path to the node. To do so, you can keep a backward pointer in each of the nodes that you set as you traverse them, then when the search finds a particular node, you have a path back to the root.

Alternatively, if you only want the path length, you can just keep an integer that stores how far away you currently are from the root as the search algorithm progresses, then return this integer when you are done.

4.4 Minimum Spanning Tree

4.4.1 Definition

Given an unweighted graph $G = (V, E)$, a spanning tree $G' = (V', E')$ is a **connected** graph such that $E' \subseteq E$ and $|E'| = V - 1$. For a weighted graph, this definition holds, but it is also required that the sum of the weight of the edges is minimized.

4.4.2 Algorithm Overviews

Prim's algorithm is related to Dijsktra's algorithm (both are based on expanding cloud of known vertices and use priority queues), but cannot handle weighted graphs as Dijsktra's algorithm can.

4.4.3 Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm that uses a priority queue with union-find to find a minimum spanning tree.

The idea of Kruskal's algorithm is to grow a forest out of edges that do not create a cycle. Then, pick an edge with the smallest weight. As such, Kruskal's algorithm is called an *edge-based* greedy algorithm. It works on a weighted graph, so it tries to minimize the sum of the weights of the edges.

Here is some pseudocode:

```
Kruskal(G):
    // It is better to use a priority queue and just keep track of the minimum
    // element
    sort the edges in G by weight

    // Union-Find
    put each node in its own subset

    while output size < |V| - 1:
        consider the next smallest edge (u, v)
        if find(v) != find(u):
            output(u, v)
            union(find(u), find(v))
```

Here is an analysis of Kruskal's algorithm (no priority queue):

- Sort edges: $O(|E| \log |E|)$
- Iterate through edges and do union-find: $O(|E| \log^* |E|) \approx O(|E|)$

Here is an analysis with using a priority queue:

- Use priority queue for keeping track of minimum edges: $O(|E|)$
- Iterate for each edge and delete minimum: $O(|E| \log |E|)$
  - Iterating through edges: $O(|E|)$
  - Delete minimum: $O(\log |E|)$

Although both of these algorithms are $O(|E| \log |E|)$, the second one is usually used because the "expensive" part is in the loop where we may terminate early, whereas we always have to do the entirety of the expensive part for the first one.

4.4.4 Dijsktra's Algorithm

Dijsktra's algorithm gives you the shortest path to a particular node. In fact, it actually gives you the shortest path to *all* the nodes, because it relies on the fact that to get to a particular node, you need to find the shortest path to each of the nodes before it.

Here is some info about the algorithm:

- Grow a set of nodes whose shortest distance has been computed
- Nodes not in set have a "best distance so far"
- Use a pariority queue for efficiency
- Greedy algorithm (ocally optimized choices)
  - But it turns out that it is actually also *globally* optimal (not usually the case for greedy algorithms, but it is for this one)

And here's the pseudocode:

```
Dijkstra(V, E, vStart):
    for v in V:
        v.cost = infinity
        v.known = false

    vStart.cost = 0

    while there exist unknown nodes:
        b = find unknown node with smallest cost
        b.known = true
        for each edge (b, a) in E:
            if not a.known:
                if b.cost + weight(b, a) < a.cost:
                    a.cost + b.cost + weight(b, a)
                    a.path = b
```

4.4.5 Correctness of Dijsktra's Algorithm

All "known" vertices have correct shortest path (this is true initially, and it is stays true by of induction). A really important fact is that if we mark a vertex as known, then we *will not* find a shorter path later. From this, we can prove via contradiction that there cannot be a shorter path than the one that Dijsktra gives us.

4.4.6 Asymptotic Analysis of Dijsktra's Algorithm

Here are the running times for the various parts of the algorithm:

- Setting initial cost of nodes: $O(|V|)$.
- The while loop takes $O(|V^2| + |E|) = O(|V|^2)$.
  - Looping through all unknown nodes: $O(|V|)$.
    - Finding minimum element for each loop iteration: $O(|V|)$.
      - Unless we use a priority queue… (see next analysis)
  - Looping through the edges: $O(|E|)$ overall.

Thus, Dijsktra's algorithm is $O(|V|^2)$. ***UNLESS…*** we use a priority queue!

4.4.7 Dijsktra's Algorithm with a Priority Queue

Here's a slightly modified version of the algorithm:

```
Dijkstra(V, E, vStart):
    for v in V:
        v.cost = infinity
        v.known = false
```

```
vStart.cost = 0

build heap with all vertices

while H not empty:
    b = H.deleteMin()
    b.known = true

    for each edge (b, a) in E:
        if not a.known:
            newcost = b.cost + weight(b, a)
            oldcost = a.cost
            if newcost < oldcost:
                // reorder a in H by decreasing its cost
                H.decreaseKey(a, newcost - oldcost)

                a.path = b
```

And here is the new asymptotic analysis:

- Setting initial cost of nodes: $O(|V|)$.
- Building heap: $O(|V| \log |V|)$.
- The while loop takes $O(|V| \log |V| + |E| \log |V|)$.
  - Looping through all unknown nodes: $O(|V|)$.
    - Getting minimum element from heap for each loop iteration: $O(\log |V|)$.
  - Looping through the edge and decreasing key: $O(|E| \log |V|)$ because `decreaseKey` $\in O(\log |V|)$.

Thus, Dijkstra's algorith (with a priority queue) is $O(|V| \log |V| + |E| \log |V|)$.

4.4.8 Density vs Sparsity

If there are not many edges in a graph, then it is said to be *sparse*. Alternatively, if there are many edges in a graph, it is said to be *dense*. If your graph is sparse you will want to use the priority queue implementation, but if its dense, you will want to use the $O(|V|^2)$ implementation. However, note that most graphs are sparse, so the priority queue implementation is more commonly used.

Chapter 5

Sorting

5.1 Definition

A *sorting algorithm* will take in an array and return it in sorted order. It is important to note that there is no "best" sorting algorithm for every single purpose. Having a sorted array may be useful when you want to pre-process some data for future operations, or if you want to get the first (or last) k elements of the array.

5.2 Specification

A sorting algorithm will take an array A of records (where there is a key and a value in each record), and a comparison function. The effect of the algorithm will be to reorganize the elements of A such that if $i < j$, then $A[i] \leq A[j]$. Note also that the output array must have the exact same data as the input array, but just in a different order.

5.3 Variations

There are a couple of variations that you might want in a sorting algorithm.

- You can have it operate on linked lists instead of arrays.
- You can have a *stable* sort, in which ordering ties between elements are broken by their original positions in A.
- You can have an *in-place* sort that uses $O(1)$ auxiliary space.
- You can use an *external sort* if the array is too big for memory.

5.4 Categories of Sorting Algorithms

- Simple algorithms (can still be useful because of low constants): $O(n^2)$

  - Insertion sort
  - Selection sort
  - Shell sort
  - Bubble sort
- Fancier algorithms: $O(n \log n)$
  - Heapsort
  - Mergesort
  - Quicksort
- Specialized sorts (for when you have a specific range of data): $O(n)$
  - Radix sort
  - Bucket sort

Note that it is proven that any comparison sort (general-purpose sort) is proven to be in $\Omega(n \log n)$).

5.5 Insertion Sort

The general idea of insertion sort is, at step k, put the k-th element in the correct position among the first k elements. Therefore, the loop invariant is that, at loop index i, the first i elements are sorted.

The best case for insertion sort is when the array is already sorted. In this case, insertion sort is $O(n)$.

The average and the worst cases for insertion sort are both $O(n^2)$.

5.6 Selection Sort

The general idea of selection sort is, at step k, find the smallest element among the not-yet-sorted elements and put it at position k. The loop invariant is that at loop index i, the first i elements are the i smallest elements of A in sorted order (they will therefore not be touched for the rest of the algorithm).

The best, average, and worst cases for selection sort are all $O(n^2)$ because the minimum of the array ($O(n)$) needs to be found for each element ($O(n)$).

5.7 Heapsort

Here is the pseudocode for heapsort:

```
HeapSort(A):
    build a heap with each A[i]
    for i = 0 to length(A - 1):
        A[i] = deleteMin()
```

This is $O(n \log n)$. With this implementation heapsort is not in-place, but there is an array trick to make it in-place.

This exact same idea can be applied to AVL trees to get an AVL sort.

5.8 Divide and Conquer

Before we move on to the recursive sorts, here is a general strategy for divide and conquer algorithms (such as quicksort and mergesort):

1. Divide problem into two smaller parts
2. Indepdently solve the parts (possibly via recursion and parallelism)
3. Combine the two parts into a solution

5.9 Mergesort

This was already done in class. See previous section on mergesort.

5.10 Quicksort

Here is the pseudocode for quicksort:

```
Quicksort(A):
    pick a pivot
    divide elements into everything <  the pivot and
                         everything >= the pivot
    recursively sort the pieces
```

So, how do you pick the pivot?

- Any choice is correct, but some are slow, so we want to pick a good pivot.
- One good strategy is to pick the median of the elements with the lowest, middle, and highest indices of the array. This will get a good approximation of the distribution of the data in the array.