

CS 162 Notes (Winter 2017)

Justin Lubin

Contents

1	Asymptotic Analysis	5
1.1	Big-O Notation	5
1.1.1	Definition	5
1.1.2	Assumption: Constant Operations (Approximations)	5
1.1.3	Non-Constant Operations	5
1.2	Reducing Big-O Expressions	6
1.2.1	What to Eliminate	6
1.2.2	Examples	6
1.3	Linear Search	6
1.3.1	Code	6
1.3.2	Analysis	6
1.4	Binary Search	7
1.4.1	Code	7
1.4.2	Analysis	7
1.5	Methods of Asymptotic Analysis	7
1.5.1	Expansion Method	7
1.5.2	Substitution Method	8
1.6	The Towers of Hanoi	9
1.6.1	Gameplay	9
1.6.2	Algorithmic Solution	9
1.7	Mergesort	10
1.7.1	Description	10
1.7.2	Example	10
1.7.3	Analysis	10
2	Algorithm Correctness	13
2.1	Key Parts of an Algorithm	13
2.2	Proving Correctness	13
2.2.1	Basic Example	13
2.2.2	Loop Invariants	14
2.2.3	Strong Induction	14
2.2.4	Mergesort	14
3	Data Structures I	17
3.1	Dictionaries	17
3.1.1	Description	17

3.1.2	Implementations	17
3.1.3	Better Implementations	18
3.2	Trees	18
3.2.1	Terminology	18
3.3	Binary Trees	18
3.3.1	Description	18
3.3.2	Binary Tree Traversal	19
3.4	AVL Trees	20
3.4.1	Motivation	20
3.4.2	Example	21
3.4.3	Maintaining the AVL Condition	21
3.5	Hashtables	23
3.5.1	Description	23
3.5.2	Example Hash Function	23
3.5.3	Collisions	23
3.5.4	Load Factor	24
3.5.5	Linear Probing	24
3.5.6	Quadratic Probing	24
3.5.7	Double Hashing	25
3.5.8	Universal Hash Functions	25
3.5.9	Perfect Hash Functions	26
3.6	Union-Find Data Structure	26
3.6.1	Mathematical Definitions	26
3.6.2	The Union-Find Algorithm	26
3.6.3	Union-Find Operations	27
3.6.4	A Cute Application of Union-Find	27
3.6.5	Up-Tree	28
3.7	Priority Queue	29
3.7.1	Binary Min-Heap	29
3.7.2	Array Representation of Binary Trees	30
3.7.3	Fast Way to Build Heap	31

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 substitution 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
=> 5 3 1 6
    => 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 pseudocode 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 $A[l \dots m]$ and $A[m + 1 \dots r]$ 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 $n = 1 \dots k$ 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 $m - l + 1 = (k + 1) / 2 \leq k$ elements, so by the inductive hypothesis, `mergesort` holds. Hence, $A[l \dots m]$ is sorted.
2. For the second recursive call, we have that `mergesort` is sorting $r - (m + 1) + 1 = r - m = (k + 1) / 2 \leq k$ elements, so by the inductive hypothesis, `mergesort` holds. Hence, $A[m + 1 \dots r]$ 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 (guaranteed to be balanced)
3. B-trees (guaranteed 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 tree, 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 the 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 recursively be defined as $\max(\text{height}(\text{child1}), \text{height}(\text{child2})) + 1$; 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 table size.

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 deletion, 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 quadratic 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 g 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 $\{S_1, S_2, \dots, S_n\}$ 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 (yassssssssss)

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 $\text{find} \in O(\log n)$.
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 $\text{union} \in O(1)$.

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 $\text{find} \in O(\log n)$, 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. Propagate 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:

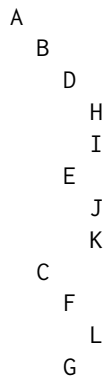
1. Put the node in the bottom right of the tree
2. “Percolate up” (do the swapping thing described in the previous paragraph but upward)

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:



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 $i * 2 = i \ll 1$
- To get the right child, go to index $i * 2 + 1 = i \ll 1 + 1$
- To get the parent, go to index $i / 2 = i \gg 1$

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.