Pseudocode:

- Discuss how an algorithm works without being encumbered with actual implementation details in a language agnostic way.
- Should give enough detail to undersand, analyze and implement the algorithm.

Conventions: indices start at 1 not 0, notation like ∞ instead of constants like Integer.MAX VALUE, shortcuts for simple functions (e.g. swap) to simplify pseudocode, often use $\leftarrow$ instead of = to avoid ambiguity, indentation specifies scope

Proofs: A deductive argument showing a statement is true based on previous knowledge (axioms). They are important/useful because they allow us to be sure that something is true and in algs, they allow us to prove properties of algorithms.

Techniques $\rightarrow$ (counter)example, enumeration, by cases, by inference (aka direct proof), trivially, contrapositive, contradiction, induction (strong and weak)

Proof by Induction (Weak):

1. Base case: prove some starting case is true
2. Inductive case: Assume some event is true and prove the next event is true
   (a) Inductive hypothesis: Assume the event is true (usually k or k-1)
   (b) Inductive step to prove: What you're trying to prove assuming the inductive hypothesis is true
   (c) Proof of inductive step

Example:

- Prove that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- (Base Case, n = 1) $\sum_{i=1}^{1} i = 1$ and $\frac{1(1+1)}{2} = 1$, so true for n = 1.
- (Inductive case, n = k' + 1) We begin by assuming true for n = k': $\sum_{i=1}^{k'} i = \frac{k'(k'+1)}{2}$. Next we must prove that $\sum_{i=1}^{k'+1} i = \frac{(k'+1)(k'+2)}{2}$:

$$\begin{aligned}\sum_{i=1}^{k'+1} i &= (k'+1) + \sum_{i=1}^{k'} i \\ &= (k'+1) + \frac{k'(k'+1)}{2} \\ &= \frac{2(k'+1)}{2} + \frac{k'(k'+1)}{2} \\ &= \frac{(k'+2)(k'+1)}{2}\end{aligned}$$

Since we proved the statement is true for n = 1 and for n = k' + 1 (assuming it is true for n = k') we have shown that the statement is true $\forall$ n $\geq$ 1.

Selection sort: sorts arrays by repeatedly finding the minimum element from the unsorted part and putting it at the beginning.

Insertion sort: builds the final sorted list one item at a time by inserting items from the unsorted part one item at a time.

Weak vs. Strong Induction: Weak: inductive hypothesis only assumes it holds for some step (e.g., kth step); Strong: inductive hypothesis assumes it holds for all steps from the base case up to k.

Loop Invariant: A statement about a loop that is true before the loop begins and after each iteration of the loop; upon termination of the loop, the invariant should help you show something useful about the algorithm. Example for insertion sort: at the start of each iteration of the for loop, the subarray A[1..j - 1] is the sorted version of the original elements of A[1..j - 1].

Asymptotic Notation: talk about the computational cost of algorithms by focusing on essential parts and ignoring irrelevant details. Precisely calculating the actual steps is tedious and not generally useful since different operations take different amounts of time (even from run to run, things such as caching, etc. cause variations). We want to identify categories of algorithmic run times.

Big-O Notation (Upper Bound): $O(g(n))$ = { $f(n)$: there exists positive c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ }

Omega (Lower Bound): $\Omega(g(n))$ = { $f(n)$: there exists positive c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }

Theta (Upper and Lower Bound): $\Omega(g(n))$ = { $f(n)$: there exists positive c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ }

Vocab: Worst-case: what is the worst the running time of the algorithm can be? Best-case: what is the best the running time of the algorithm can be? Average-case: given random data, what is the running time of the algorithm? Note: Don't confuse this with O, Ω and Θ. The cases above are situations, asymptotic notation is about bounding particular situations.

Proving bounds - find constants that satisfy inequalities: Show that $5n^2 - 15n + 100$ is $\Theta(n^2)$. Step 1: Prove $O(n^2)$. Find positive constants c and n_0 such that $5n^2 - 15n + 100 \leq cn^2$ for all $n > n_0$. Let c = 100, n_0 = 10. Step 2: Prove $\Omega(n^2)$. Find positive constants c and n_0 such that $cn^2 \leq 5n^2 - 15n + 100$ for all $n > n_0$. Let c = 1, n_0 = 5. Since f(n) $\in O(n^2)$ and f(n) $\in \Omega(n^2)$, f(n) $\in \Theta(n^2)$.

Disproving Bounds: Is $5n^2$ = O(n)? No, it's not. But how do we prove this? We begin by assuming it's true. In other words, we assume that there exists some c and n_0 such that $5n^2 \leq cn$ for all $n > n_0$. This is a contradiction since this would mean that $5n \leq c$. Since n is increasing, no such constant exists.

Basic Hierarchy: $O(1) \rightarrow O(\log n) \rightarrow O(n) \rightarrow O(n \log n) \rightarrow O(n^2) \rightarrow O(2^n) \rightarrow O(n!)$; Examples (same order): add two 32 bit numbers, binary search, find element in linked list; sort list with merge sort, double nested loops, enumerate all possible subsets, enumerate all permutations.

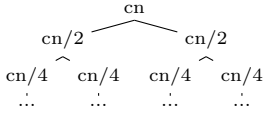
Rules of Thumb:

- Multiplicative constants can be omitted: $14n^2$ becomes n^2; $7 \log n$ becomes $\log n$.
- Lower order functions can be omitted: $5 + n$ becomes n; $n^2 + n$ becomes n^2.
- n^a dominates n^b if a > b: n^2 dominates n, so $n^2 + n$ becomes n^2; $n^{1.5}$ dominates $n^{1.4}$.
- a^n dominates b^n if a > b: 3^n dominates 2^n.
- Any exponential dominates any polynomial: 3^n dominates n^5; 2^n dominates n^c.
- Any polynomial dominates any logarithm: n dominates $\log n$ or $\log \log n$; n^2 dominates $n \log n$; $n^{1/2}$ dominates $\log n$.
- Note: Do not omit lower order terms of different variables e.g. $(n^2 + m)$ does not become n^2.

Divide and Conquer:

- Divide: Break problem into smaller sub-problems
- Conquer: Solve sub-problems. Generally, involves waiting for the problem to be small enough that it is trivial to solve (i.e. 1 or 2 items)
- Combine: Given results of solved sub-problems, combine them to generate a solution for complete problem

Merge Sort: T(n) = { c if n is small & 2T(n/2) + cn otherwise }



We observe that each layer sums exactly to cn. Next, we need to figure out the number of layers to figure out the total cost. Solving $\frac{n}{2^d} = 1$ for d, we get $\log_2 n$ = d. Thus our total work is cn $\cdot \log_2 n$, which is bounded by $\Theta(n \log n)$.

Reccurence: A function that is defined with respect to itself on smaller inputs. Recurrences are often easy to define because they mimic program's structure. They do not directly express the computational cost, however. To remove self-recurrence and find a more understandable form for the function, we have three approaches:

- Substitution method: when you have a good guess, prove it's correct.
- Recursion-tree method: If you don't have a good guess, use recursion tree and solve with substitution method. For this method, we draw out the cost at each level of recursion and sum up the cost of the levels of the tree.
- Master method: solutions for recurrences of form: T(n) = aT(n/b) + f(n).

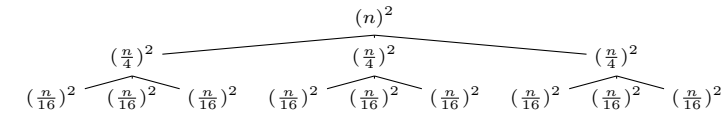
Substitution example: T(n) = T(n/2) + d. Halves the amount of work and then does a constant amount of work. Looks like binary search, so we guess T(n) = $O(\log_2 n)$. To prove this guess, we use induction. We begin by assuming T(k) = $O(\log_2 k)$ for all k < n. Thus, using the definition of big-O, we know that T(n/2) $\leq c \log_2(n/2)$. We now need to show that T(n) = $O(\log_2 n)$. Cont'd on next page.

$$\begin{aligned}T(n) &= T(n/2) + d \\ &\leq c \log_2(n/2) + d \\ &\leq c \log_2 n - c \log_2 2 + d \\ &\leq c \log_2 n - c + d\end{aligned}$$

Now the question becomes does some constant c' exist such that T(n) $\leq c' \log_2 n$. The answer is yes if c > d and if not, we just let c' = d. For our inductive proof, we don't really have to worry about our base case since it's bounded by $\Theta(1)$ which is in $O(\log n)$.

Note for substitution: If we pick a guess that is a much too generous bound (e.g. if we guess $O(n^2)$ for something that can be bounded by $O(n)$), we will end up with something very easy to prove like $cn \geq 2$.

Recursion tree method example: $T(n) = 3T(n/4) + n^2$.



From this tree, we observe that the cost of each layer is $(\frac{3}{16})^d n^2$. Next, we need to figure out the total depth of the tree. Rewriting $\frac{n}{4^d} = 1$ gives us $\log_4 n = d$. Thus we get that T(n) = $n^2 + (\frac{3}{16})n^2 + (\frac{3}{16})^2 n^2 + ... + (\frac{3}{16})^d n^2 = n^2 \sum_{i=0}^{log_4 n} (\frac{3}{16})^i < n^2 \sum_{i=0}^{\infty} (\frac{3}{16})^i = \frac{1}{1-3/16} n^2 = O(n^2)$. We would continue by verifying this solution using the substitution method.

Master Method: Provides solutions to all recurrences of the form $T(n) = aT(n/b) + f(n)$. There are three cases:

- If $f(n) = O(n^{\log_b a - \epsilon})$ for $\epsilon > 0$, then T(n) = $\Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a})$, then T(n) = $\Theta(n^{\log_b a} \log n)$.
- If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$ and $af(n/b) \leq cf(n)$ for $c < 1$ then $T(n) = \Theta(f(n))$.

Master Method Example: $T(n) = 16T(n/4) + n$. This means a = 16, b = 4 and $f(n)$ = n. Next we compute $n^{\log_b a} = n^{\log_4 16} = n^2$. We are in the first case since $f(n) = O(n^{2-\epsilon})$ for $\epsilon = 1$. Thus $T(n) = \Theta(n^2)$.

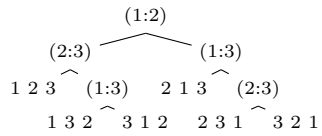
Another Master Method Example: $T(n) = T(n/2) + 2^n$. This means a = 1, b = 2 and $f(n) = 2^n$. Next we compute $n^{\log_b a} = n^{\log_2 1} = n^0$. We are in the third case since $f(n) = 2^n = \Omega(n^{0+\epsilon})$ and $af(n/b) = 2^{n/2} \leq cf(n)$ for c = 1/2. This is because for c = 1/2, the right side is $\frac{1}{2} 2^n = 2^{-1} 2^n = 2^{n-1}$. Thus $T(n) = \Theta(2^n)$.

Partition Function: Partitions a list of elements on one element such that after the function is run, the element we are partitioning on is in the correct location. The function operates in place. The running time of partition is $\Theta(n)$.

```
Quicksort(A, p, r)
    if p < r
        q <- Partition(A, p, r)
        Quicksort(A, p, q - 1)
        Quicksort(A, q + 1, r)
```

```
Partition(A, p , r)
    i <- p - 1
    for j <- p to r - 1
        if A[j] <= A[r]
            i <- i + 1
            swap A[i] and A[j]
    swap A[i + 1] and A[r]
    return i + 1
```

Quicksort: Worst case running time $T(n) = T(n-1) + \Theta(n)$ which is $\Theta(n^2)$. This happens when the list is sorted or reverse sorted. The best case running time is $T(n) = T(n/2) + \Theta(n)$ which is $\Theta(n \log n)$. This happens when we have random data. We can avoid the worst case by injecting randomness into the data.
Comparison-based sorting: Sorted order is determined based only on a comparison between input elements i.e. A[i] < A[j] or A[i] > A[j] or A[i] = A[j] or A[i] <= A[j] or A[i] >= A[j]. To show that we can't do better than $O(n \log n)$ for a comparison-based sorting approach, we use a decision-tree model.



How many leaves in the decision tree? Leaves must include all permutations of the input. Thus if the input size is n, there are n! leaves. The worst case number of comparisons is the height of the tree. A complete binary tree has 2^h leaves. Thus $2^h \geq$ n! $\implies$ h $\geq \log n!$ $\implies$ h = $\Omega(n \log n)$. Thus, we can't really do better than $n \log n$ for comparison based sorting.
Calculating the median: One approach we might take is to sort the list and then pick the n/2 element which has a runtime of $\Theta(n \log n)$. This feels like we are doing a lot of extra work, though! To get the k-th element (or the median) by sorting, we're finding all the k-th elements at once. However, we just want one!
Selection problem: Finding the k-th smallest element is called the selection problem. We can use the partition function and then break the problem in half.

```
Selection(A, k, p, r)
    q <- Partition(A,p,r)
    relq = q-p+1
    if k = relq
        Return A[q]
    else if k < relq
        Return Selection(A, k, p, q-1)
    else (k > relq)
        Return Selection(A, k-relq, q+1, r)
```

where A is the array of data, k is the index we're looking for and p and r are the bounds we're looking for (initially 1, len(A)). In the best case, we partition in the right spot and the running time is $\Theta(n)$. In the worst case, each call to partition only reduces our search by 1. In this case our recurrence is $T(n) = T(n-1) + \Theta(n)$ which is $O(n^2)$. This happens when the list is sorted. If we get a good partition half of the time, on average we need two attempts before we get a good partition. If we roll in the cost of the bad partitions, our recurrence becomes $T(n) = T(3n/4) + O(n)$ which is just $T(n) = T(3n/4) + \Theta(n)$ in this case, which is bounded by $\Theta(n)$.
Data Structures: Data strucutres are a way of storing data that facilitates particular operations. Dynamic set operations: For a set S:

- Search(S,k) – Does k exist in S?
- Insert(S,k) – Add k to S
- Delete(S,x) – Given a pointer/reference, x, to an element, delete it from S
- Min(S) – Return the smallest element of S
- Max(S) – Return the largest element of S

Array: Sequential locations in memory in linear order. Elements are accessed via index. Search(S, k) = O(n). Insert(S, k) = $\Theta(1)$ if we leave extra space, $\Theta(n)$. InsertIndex(S, k) = $\Theta(n)$. Delete(S, x) = $\Theta(n)$. Min(S) = $\Theta(n)$. Max(S) = $\Theta(n)$.
Linked List: Elements are arranged linearly. An element in the list points to the next element in the list. Search(S, k) = O(n). Insert(S, k) = $\Theta(1)$. InsertIndex(S, k) = $O(n)$ or $\Theta(1)$ if at index. Delete(S, x) = $O(n)$. Min(S) = $\Theta(n)$. Max(S) = $\Theta(n)$.
Doubly Linked List: Elements are arranged linearly. An element in list points to the next element and previous element in the list. The back link allows us to have $\Theta(1)$ deletion.
Stack: LIFO. Can implement with an array or a linked list. Empty – check if stack is empty = $\Theta(1)$. Pop – removes the top element from the list = $\Theta(1)$. Push – add an element to the list = $\Theta(1)$. Array: more memory efficient. Linked list: don't have to worry about "overflow". Other options? ArrayList (expandable array): compromise between two, but not all operations are $\Theta(1)$.
Queue: FIFO. Can implement with an array or a a linked list. Array: keep head an tail indices, add to one and remove form the other. Linked list: keep a head and tail reference, add to the tail, remove from the head. Operations: Empty = $\Theta(1)$. Enqueue - add element to end of queue = $\Theta(1)$. Dequeue - remove element from the front of the queue = $\Theta(1)$.

Properties of Logarithms:

- $\log_b a^n = n \log_b a$
- $\log_b a = \frac{\log_c a}{\log_c b}$
- $a^{\log_b n} = n^{\log_b a}$
- $\log_c(ab) = \log_c a + \log_c b$
- $\log_c(a/b) = \log_c a - \log_c b$

Exponent Properties:

- $a^{xy} = (a^x)^y$
- $a^{x+y} = a^x a^y$
- $a^{x-y} = \frac{a^x}{a^y}$

Summation Properties:

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- $\sum_{i=0}^{\infty} a(r)^n = \frac{a}{1-r}$ for $|r| < 1$

Proofs of Log Properties:

- Proof that $\log_b a^n = n \log_b a$. $b^x = a \implies b^{xn} = a^n \implies \log_b b^{xn} = \log_b a^n \implies xn = \log_b a^n \implies n \log_b a = \log_b a^n$.
- Proof that $\log_b a = \frac{\log_c a}{\log_c b}$. $b^x =$ a $\implies \log_c b^x = \log_c a \implies x \log_c b = \log_c a \implies$ x $= \frac{\log_c a}{\log_c b} \implies \log_b a = \frac{\log_c a}{\log_c b}$.
- Proof that $a^{log_b n} = n^{log_b a}$. Let x = $\log_b n$ and let y = $\log_b a$. Then $b^x =$ n and $b^y =$ a. So, $a^{\log_b n} = a^x$. Since $a = b^y$, we have $a^{\log_b n} = (b^y)^x = b^{xy}$. Similarly, $n^{\log_b a} = (b^x)^y = b^{xy}$. Therefore, $a^{\log_b n} = n^{\log_b a}$.

Asymptotics:

1. 2^{n+1} is $O(2^n)$. True (use $2^{n+1} = 2 \cdot 2^n$)
2. 2^{2n} is $O(2^n)$. False (proof by contradiction, use $2^{2n} = (2^n)^2$).
3. $3n^2 \log_2 n + 16n$ is $O(n^3)$. True ($3n^2 \log_2 n + 16n \leq 3n^3 + 16n^3 \leq 19n^3$).
4. $25 \log_2 8n^{10}$ is $O(\log_{10} n)$. True ($25 \log_2 8n^{10} = 25(\log_2 8 + \log_2 n^{10}) = 25(3 + 10 \log_2 n)$).
5. $8^{\log_2 n}$ is $O(n^3)$. True (use $8^{\log_2 n} = n^{\log_2 8} = n^3$).

Ranking Functions: O(1) = 47. $O(n^{1/3}) = n^{1/3} + \log_5 n$. $O(n^{1/2}) = \sqrt{n}$. O(n) = n. O(nlogn) = $n \log_4 n$ and $\log_2 n!$. $O(n^2) = n^2$ and $4^{log_2 n}$. $O(n^{log n}) = n^{\log_2 \log_2 n}$ and $(\log_2 n)^{log_2 n}$. $O(1.5^n) = (\frac{3}{2})^n$. $O(2^n) = 2^n$. O(n2^n) = $n2^n$. $O(e^n) = e^n$. O(n!) = n!. O((n+1)!) = (n+1)!.
Solving Recurrences:

- T(n) = 4T(n/2) + cn. We can use the master method for this recurrence. We know that $a = 4$, $b = 2$, and $f(n) = cn$. We see that $\log_b a = \log_2 4 = 2$. From here, we see that we are in the first case since $f(n) = cn = O(n^{2-\epsilon})$ where $\epsilon = 1$. Thus we have $T(n) = \Theta(n^2)$.
- T(n) = T(n - 1) + n. We can write the sum of the work done at each of the layers as: $\sum_{i=0}^{n-1} n - i = \sum_{i=1}^{n} i$. This is simply an arithmetic series. By A.2 in the textbook (p. 1146), the bound of this is $T(n) = \Theta(n^2)$.
- T(n) = T(n-1) + 1/n. We can write the sum of the work done at each of the layers as: $\sum_{i=0}^{n-1} \frac{1}{n-i} = \sum_{i=1}^{n} \frac{1}{n}$. By the Harmonic Series in Section A.7 of the textbook (p. 1147), the tight bound is $T(n) = \Theta(\log n)$.
- T(n) = T(9n/10) + n. We can use the master method to solve this recurrence. In this case, $a = 1$, $b = 10/9$ and $f(n) = n$. We begin with: $\log_b a = \log_{10/9} 1 = 0$. From here, it is evident that we are in the third case since $f(n) = n = \Omega(n^{0+\epsilon})$ where $\epsilon = 1$ and $9n/10 < cn$ where c = 19/20. Thus, $T(n) = \Theta(n)$.
- $T(n) = 2T(3n/4) + \sqrt{n}$. We move forward with master method; $a = 2$, $b = 4/3$, and $f(n) = \sqrt{n}$. We compute $\log_b a = \log_{4/3} 2$. From here, we can see that we are in the first case since $f(n) = n^{1/2} = O(n^{log_b a - \epsilon}) = O(n^{\log_{4/3} 2 - \epsilon})$ for $\epsilon = \log_{4/3} 2 - 1/2$. Thus $T(n) = \Theta(n^{\log_{4/3} 2})$.
- $T(n) = 3T(n/3) + n \log n$. We move forward with master method; $a = 3$, $b = 3$, and $f(n) = n \log n$. We compute $\log_b a = \log_3 3 = 1$. From here, we can see that we are in the third case since $f(n) = n \log n = \Omega(n^{log_b a + \epsilon}) = \Omega(n^{1+\epsilon})$ for $\epsilon = 0.1$ and $af(n/b) = 3 \cdot \frac{n}{3} \log \frac{n}{3} \leq cf(n) = c \cdot n \log n$ for $c < 1$. This is true since if we rewrite our equation, we get $\frac{\log n - \log 3}{\log n} \leq$ c and we can thus pick a $c < 1$. Thus $T(n) = \Theta(n \log n)$

Recursion tree analysis for T(n) = 3T(2n/3) + c: We observe that the total cost of the first layer is c, the total cost of the second layer is $3c$ and the total cost of the third layer is $9c$. If we generalize this, we get that the cost of the layer at depth d is $c3^d$. We next need to figure out the depth of the tree. We compute: $1 = \frac{n}{(3/2)^d} \implies n = (3/2)^d \implies \log_{(3/2)} n = d$.

$$T(n) = \sum_{i=0}^{log_{(3/2)} n} c3^i \qquad \textbf{(Sum of cost at each layer)}$$

$$= c \sum_{i=0}^{log_{(3/2)} n} 3^i \qquad \textbf{(Pulling out constant)}$$

$$= c(\frac{3^{log_{(3/2)} n} - 1}{2}) \qquad \textbf{(Geometric sum; appendix A.5 of text)}$$

$$= \frac{c}{2}(n^{log_{(3/2)} 3} - 1) \qquad \textbf{(Pulling out 1/2)}$$

$$= \Theta(n^{\log_{(3/2)} 3}) \qquad \textbf{(Applying bounds)}$$