0 Syllabus, other information

0.1 Syllabus

The syllabus is up on dropbox.

1 Chapter 1 Introduction (& Review)

1.1 Introduction

This course is about Algorithms and Data Structures (and the Algorithms that operate on those Data Structures...). The purpose of this chapter is a fast review of stuff you may have learned in CSCE 146 or some equivalent course.

1.2 Review of Asymptotic Notation for Time and Space Complexity

Algorithmic efficiency can be described multiple ways. One way is by actually timing them on sample inputs (Empirical Analysis). Surely we want the fastest/most efficient algorithm we can have but this tends to vary by computer and architecture and compiler and compiler options and ... Accordingly, we will prefer "Big-O" descriptions of *algorithms*, which, while quite coarse in the sense that they only tell us the way run times (or memory usage) grow as the input size grows, it will hold for basically any regular computer – i.e. a regular one with enough RAM or even disk space. This big-picture approach makes results from the 1970s, or earlier, still relevant today. Using this approach will get you programs/software having implementations that only modestly slower than a fully "tweaked" highly optimized implementation. To be fair, they might have a runtime that's 5-10 times as slow (but with some profiling you can identify and then improve the most used regions of code (90/10 rule) to get much closer) but if your approach follows the one from this class, it will (almost) never blow up in your face. "Blow up in your face", here, means having an unexpected, undesirable, and (hopefully) unnecessary runtime or memory usage that makes your software *unusable* for practical reasons; i.e. a "correct" algorithm doesn't mean it is a "good" or "useful" one. Even if you do not end up being a programmer, this is a cornerstone of how software like databases are built so you will likely be using the results of this sort of approach, even if only indirectly.

1.3 Review of Major Data Structures

1.3.1 Arrays

A collection of fixed-size items contiguously allocated. Items in the array are indexed by an integer as so A[i]. Arrays may be 0-indexed or 1-indexed meaning the index of the first item will be at index 0 or 1, respectively. The book will often, subtly, indicate which indexing is used for the Algorithm inputs as A[0..n-1] or A[1..n], again respectively. Some algorithms are easier to express with one indexing scheme or another.

Advantages:

- Its main advantage is that it can be indexed in $O(1)$ time because to find an item by index requires only a multiplication/shift (index*size) plus one addition to add the address of the first item. It is very efficient if an algorithm indexes "randomly".
- Replacing and swapping items by index is efficient.
- Adding to end is efficient, assuming a pointer/index is kept to keep track of the last or next to be occupied index.
- For sequential access (beginning to end, e.g.) it makes good use of the memory cache.

Disadvantages:

- The size of the array is generally fixed at instantiation. If the amount of data required grows beyond a fixed size, the array may need to be re-allocated. We will often use std::vector for this, which is a dynamically allocated array (will resize itself automatically).
- Inserting/Deleting may involve shifting many items, worst case $O(n)$
- When dealing with basic number values, may have no clear way to tell if a value at some index is empty or represents a value (not an issue for pointer types if array is fully initialized). Sometimes there may be a special value indicating the end of valid input, e.g. '\0' in C-Style strings. For an array of positive integers, the value -1 might be sufficient to mark the end.

1.3.2 Linked Lists

The basic idea for a linked list is to store, with each value, a pointer to the next value. Code using a linked list will store a pointer to the first item (or NULL if none exists at that time). The last item's pointer in the list will be NULL as well.

Advantages:

- Insertion/Deletion at an index, given a pointer to the node at that index is proportional to the amount of data to be inserted.
- No max size (within OS/HW limits).

Disadvantages:

- General indexing is O(n) in time. Note: There's nothing preventing one from storing, e.g. a pointer to the last node in the Linked List.
- For large amounts of small data, the cost in memory for the pointers may be prohibitive.
- For sequential access, the nodes may "hop around" in cache, slowing things down.

1.3.3 Doubly-Linked Lists

These are linked lists that store pointers to both the node before and after a node. Good if the list needs to be accessed in either direction at the memory cost of more pointers.

1.3.4 Graphs

- Graphs are $G =< V, E >$, i.e. tuples of sets of vertices and edges.
  - V is the set of vertices, $|V|$ is the number of vertices. Nodes are *adjacent* if there is an edge between them.
  - E is the set of edges, $|E|$ is the number of edges. Edges have two endpoint vertices, which can be stated as the edge being *incident* upon those two vertices.
- Weighted and Unweighted Graphs

- Directed and Undirected Graphs
- Path: An ordered sequence of vertices from one vertex to another such that there exists an edge between every pair of adjacent nodes (and obeys direction constraints for directed graphs). I can also be represented by a sequence of edges such that adjacent pairs of edges are incident upon the same vertices. A *simple* path is a path that visits each vertex or edge at most once.
- Cycle: A simple path between a vertex and itself.
- (Self-)Loop : An edge from a node to itself.
- Circuit: A cycle where vertices may be repeated
- Hamiltonian Circuit: A circuit that visits *every* vertex in a graph exactly one time.
- Hamiltonian Path: A path that visits *every* vertex in a graph exactly one time.
- Connected Graphs: There exists an *path* between every pair of vertices.
- Complete Graphs: There exists an *edge* between every pair of vertices. May be represented by $K_{|V|}$. Generally for undirected graphs.
- Dense Graphs and Sparse Graphs
- Representations
  - Adjacency List
  - Adjacency Matrix

  How efficient are these? Sparse Graphs (e.g. for a large graph there are an average of 4 edges per vertex or $|E| \approx 4|V|$. Dense/Complete graphs, especially weighted ones? What if we need to look up edges (for weights/etc.) quickly?

1.3.5 Trees

A tree is often meant as a *rooted* tree. There is some node in the tree the is the "root" of the entire tree. It may have 0, 1, or many "children". Often, we assume no duplicates for trees (or can store duplicates in the same node with a counter or list...). Generally, each parent stores pointers to each of its children.

- Descendant: A node is a descendant of an ancestor node if there is a path from the ancestor to the descendant node (assuming tree is "directed" parent to child).
- Leaf: A node with no children.
- n-ary tree: a tree where there are at most n children of every node.
- Binary Tree: a 2-ary tree.
- Binary *Search* Tree: a binary tree such that every node is greater than its "left" child (and descendants) and less than its right child (and those descendants). This enables a faster form of searching, $O(h)$, where h is the height of a tree.
- (Essentially) Complete: Means that every layer, except possibly the last is full ("packed").
- Height: The length of the path to the deepest node in a tree from the root.
- Depth: The length of the path to a *particular node* in a tree from the root.
- Balanced: A term indicating that the height of the tree isn't (fuzzy) "too far" from the minimum, based on the size of the tree (the number of nodes). .
- First Child-Next Sibling Tree: Each nodes maintains two pointers. One to a list of it's children. Another to its right sibling.
- Free Tree: An acyclic, connected graph with $n = |V|$ vertices and $n-1$ edges, generally undirected.
- Forest: A graph consisting entirely of free tree subgraphs.

1.3.6 A connected, undirected graph with $|V|$ vertices, having $|V| - 1$ edges is a free tree.

A free tree is a connected, acyclic graph, so we need only show that the tree is acyclic (since we are assuming it is connected). We do this with (weak) induction on n = $|V|$, the number of vertices in an connected, graph $G =< V, E >$. We are going to prove that if there is a connected, acyclic (induced) subgraph of G and then note that it took all the edges in $|E| = n-1$ to build G, so there cannot be a cycle.

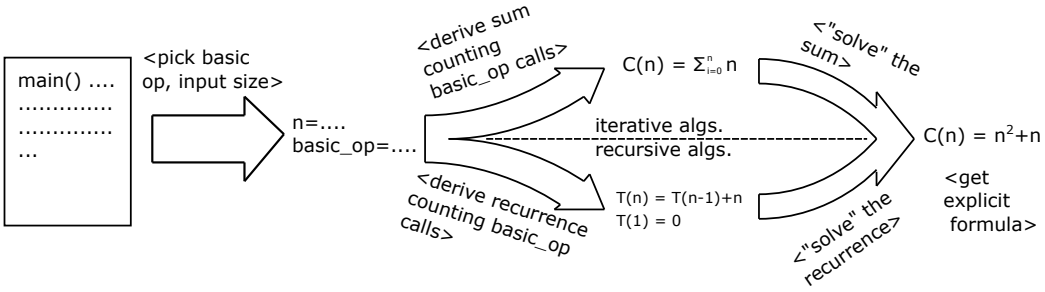
Inductive step:

- I.H.-Assume a connected graph $G_0 =< V, E >$ with n vertices and n-1 edges is acyclic.
- I.C.-Show that a connected graph $G_1 =< V_1, E_1 >$ with $n + 1$ vertices and n edges and G_0 as a subgraph is acyclic.
- G_1 is connected and, since there is one vertex in G_1 not in G_0, it must have one edge between this vertex and a node in (its own copy of) the G_0 subgraph. The addition of this new node and its single edge to connect it to G_0 cannot create a cycle since the cycle would have to go "through", using two edges, this new node and it has only one edge. The edge is not a self-loop, either. There are no other edges in E_1.

1.3.7 Heaps

2 Ch.2: Fundamentals of the Analysis of Algorithm Efficiency

2.1 The Big Picture



Our "pipeline" for analyzing algorithms is:

1. Pick the input size and basic operation.
2. Based on whether iterative or recursive algorithm
   - If Iterative: derive sum describing number of basic op calls and solve for explicit formula.
   - Else if Recursive: derive recurrence relation describing number of basic op calls and solve for explicit formula.
3. Given an explicit formula, one can:
   - Easily describe the order of growth by the dominant term.
   - Compare to other algorithms, given functions describing their order of growth, using limits.

2.2 Analysis of Sequential Search (7,8), *baby steps*

What is the input size? It must be the n, the *size* of the array to be searched. It's n in the algorithm, but you need to state what n is.

What is the basic operation? We must compare the key to the items in the array and this is in the loop. Looking back at slide 3, we can see that, since this is comparison-based searching we must use comparison. We can also observe that the comparison is essential for the algorithm (the loop *seems* essential, but it's not; imagine a recursive function that has the key, a

pointer to the end and a pointer to an item in the array). The loop, however, suggests that we will be doing iterative analysis.

Are there separate best/worst/average cases? Yes.

The *worst* case is either the item is not in the array or it is the last item. The entire array has to be compared to the key to know if it is the last item or not in the array. The algorithm goes item-by-item until the end, so that is definitely $O(n)$ (it grows linearly with the size of the input). We will soon just assume you can eyeball this but let's take our time and actually get there mathematically. Here's the algorithm:

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Note that the algorithm goes item by item (each iteration of the loop). So, each loop iteration has one, 1, comparison except, perhaps the last one, if that **and** is short circuit, which is reasonable since that might cause a segfault in real code. How many iterations are there? If the last item is the key value, it will be n iterations since, for each value that i becomes there is one iteration, $|\{0, 1..n-1\}|$, or $n+1$ if not in it, but note that we never reach the key comparison in the latter case so that *last* iteration has *zero comparisons*, so we can ignore it.

The case where the last item is the key is

$$1 + 1 + 1 + \cdots + 1, \tag{1}$$

which is just the sum of n 1s.

The case where the item is not in the array *could* be represented in a painfully meticulous manner as

$$1 + 1 + 1 + \cdots + 1 + 0, \tag{2}$$

or just n 1s followed by a single 0. Adding zero doesn't change the answer so it is the same as the case where the last item is in the array. There's nothing

magical about sums, *they're just another notation for addition when there are patterns.*

The sum of n ones is certainly n but there is also a sum to represent this, and a closed form as well. The sum is

$$\Sigma_{i=1}^{n} 1 \tag{3}$$

and there's a closed form for this (which you will use, repeatedly)

$$\Sigma_{i=l}^{u} 1 = u - l + 1 \tag{4}$$

Which we can then solve as $n - 1 + 1 = n$. This is just a trivial example of how to use closed forms and a reminder for what a sum represents. As an aside, we kind of took a roundabout way to get there... you can just take a loop and convert it to a sum directly (just look at the bounds):

$$\Sigma_{i=0}^{n-1} 1 \tag{5}$$

which, plugging into the closed form is $(n - 1) - 0 + 1$, or n. Thankfully, we found that there are as many numbers in $\{0, 1..n - 1\}$ as there are from $\{1, 2..n\}$.

So, in the worst case, there will be n comparisons.

The *best* case is if the item is the first item in the array. Note that the input size doesn't matter. This means we do a constant amount of work, simply checking the first item and then returning 0, the index of the first item. We represent this with $O(1)$, which means the cost does not grow with the input size.

The *average* case is often the most problematic because the nebulous nature of the word "average". What is the *average* probability of key being at a particular index? What is the *average* probability of the item even being in the array? There are often *reasonable* choices (guesses) but they require *assumptions*. For example, it is reasonable to assume that item to be found is at any index with equal probability, or *uniformly distributed*, i.e. $probability\_at\_index\_i = \frac{1}{n}$. This is an assumption. Users may search for things more recently added, probably near the end of the array, more than at the beginning. The second question, i.e. how often are items that are searched for in the input array, has no really obvious answer, so we'll just parameterize our answer in p, which is the probability $p \in [0, 1], p \in \mathbb{R}$ that the item being searched for is in the array.

So, under the assumption that the key to be searched for is uniformly distributed across the array and parameterizing with p, defined immediately above, we can begin the analysis. The probability is just the a weighted average, by p, of two possibilities, i.e. $C_{in}(n)$ and $C_{not}(n)$, the costs for the possibilities of the key being in the array or not, respectively, so

$$C_{avg}(n) = p \cdot C_{in}(n) + (1-p) \cdot C_{not}(n) \tag{6}$$

is the cost. Note that the $(1-p)$ is the probability of the key *not* being in the array, derived from the fact that the probabilities must sum to 1.

The cost of not being in the array is same as the worst case, so that's easy, i.e.

$$C_{avg}(n) = p \cdot C_{in}(n) + (1-p) \cdot \mathbf{n}. \tag{7}$$

Now, the issue is that we don't yet know what $C_{in}(n)$ is. So, looking at it like a conditional probability, roughly speaking under the assumption that the item is in the array, the probability of each position, $i = 0..n-1$ has a cost (in number of comparisons) of $i+1$, and each has a probability of $\frac{1}{n}$ This corresponds to a sum of the form

$$C_{in}(n) = \Sigma_{i=0}^{n-1}[i+1] \cdot \frac{1}{n} \tag{8}$$

Now, the fraction $\frac{1}{n}$ does not vary with the sum, so we can pull it out, (it's a variable in our problem but not in the sum)

$$C_{in}(n) = \frac{1}{n}\Sigma_{i=0}^{n-1}[i+1]. \tag{9}$$

Note that i goes from 0 to $n-1$ but you can obviously see that this is the same as

$$C_{in}(n) = \frac{1}{n}\Sigma_{i=1}^{n} i, \tag{10}$$

using the closed form

$$\Sigma_{i=1}^{n} i = \frac{(n)(n+1)}{2} \tag{11}$$

to yield

$$C_{in}(n) = \frac{1}{n}\frac{(n)(n+1)}{2} \tag{12}$$

and simplified to

$$C_{in}(n) = \frac{(n+1)}{2}, \tag{13}$$

which makes sense: if it's in the array and the probability of the key being at any position is the same for all positions, we expect to have to search about half of the array. (Note that the closed form n, i is different than the $C_{in}(n)$'s n, i. Be careful with this.)

So now we have a solution for the average cost:

$$C_{avg}(n) = p \cdot \frac{(n+1)}{2} + (1-p) \cdot n. \tag{14}$$

which can be rearranged directly to the slide/book solution :

$$C_{avg}(n) = \frac{p(n+1)}{2} + n \cdot (1-p) \tag{15}$$

Note that we can carry this a little further

$$C_{avg}(n) = \frac{np+p}{2} + \frac{2n-2np}{2} \tag{16}$$

$$C_{avg}(n) = \frac{np+p+2n-2np}{2} \tag{17}$$

$$C_{avg}(n) = \frac{2n-np+p}{2} \tag{18}$$

$$C_{avg}(n) = n - \frac{np}{2} + \frac{p}{2} \tag{19}$$

which behaves as expected, remembering that $p \in [0..1]$ and is obviously linear in n; this form is a little closer to an obvious polynomial considering the values p can take on so it is a little easier to analyze. This expression for $C_{avg}(n)$ ranges from n, worst case when $p = 0$ to the uniformly distributed case of Eqn. 13, when $p = 1$. Note that this conclusion only applies under the assumptions we made. What if there are duplicates in the array?

Show $n - \frac{np}{2} + \frac{p}{2} \in O(n)$

$$\begin{aligned}
n - \frac{np}{2} + \frac{p}{2} &\leq n + \frac{p}{2} && n \geq 0, p \geq 0 \\
&\leq n + \frac{p}{2} && \frac{p}{2} \text{ is at a } \textit{maximum}, \frac{1}{2} \\
&\leq 2 \cdot n && \forall n \geq \mathbf{1} \textit{ not } 0 \\
& c = 2, n_0 = 1 \quad \text{Note: any } c > 1 \text{ works if } n_0 \text{ is large enough.}
\end{aligned}$$

Show $n - \frac{np}{2} + \frac{p}{2} \in O(n)$, using above values *with weak induction.*

$$\begin{aligned} f(n) &= n - \frac{np}{2} + \frac{p}{2} \\ g(n) &= n \\ c &= 2 \\ n_0 &= 1 \\ S(n) &= \text{“}f(n) \leq c \cdot g(n), \forall n \geq n_0\text{”} \end{aligned}$$

Base Case:

$$\begin{aligned} f(n) \leq g(n) &= 1 - \frac{1 \cdot p}{2} + \frac{p}{2} \leq 2 \cdot 1 && n_0 = 1, c = 2, p \in [0..1] \\ &= 1 - \frac{p}{2} + \frac{p}{2} \leq 2 \cdot 1 \\ &= 1 \leq 2 \cdot 1 && True \end{aligned}$$

Inductive Step: In the inductive step, a subproof, we show that if $S(k)$ is true for some k, then $S(k+1)$ is true. The *inductive hypothesis* is that $S(k)$ is true, i.e.

$$I.H. = k - \frac{kp}{2} + \frac{p}{2} \leq 2 \cdot k \qquad k > 1, p \in [0..1]$$

Prove S(k+1) holds, i.e.:

$$\begin{aligned} (k+1) - \frac{(k+1)p}{2} + \frac{p}{2} &\leq 2 \cdot (k+1) \\ (k+1) - \frac{kp+p}{2} + \frac{p}{2} &\leq 2k + 2 \\ k - \frac{kp}{2} + \frac{p}{2} + 1 - \frac{p}{2} &\leq 2k + 2 && I.H. \\ 1 - \frac{p}{2} &\leq 2 && \text{look at remaining terms} \\ 1 - \frac{p}{2} &\leq 2 && p \in [0..1], \text{ specifically } p \geq 0 \\ 1 &\leq 2 && \text{always true} \end{aligned}$$

Basically, the last several steps are showing that the remaining terms will not ever cause the left hand side to be greater than the right hand side. In particular, the term $-\frac{p}{2} \leq 0$ and equals zero at $p = 0$.

2.3 Establishing order of growth using the *definition* (16)

Show $10n \in O(n^2)$

Well, we're supposed to use the definition of "big O", which is

Definition: $f(n)$ is in $O(g(n))$, $f(n) \in O(g(n))$, if order of growth of $f(n) \leq$ order of growth of $g(n)$ (within constant multiple of $g(n)$), i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \leq cg(n) \text{ for every n } \geq n_0 \tag{20}$$

Now, this can be proved formally with induction, and we'll do that later but lets argue semi-formally (this argument really is just sloppy induction where the "hard" part corresponds to the inductive step and picking n_0 is the base case). So, to show $10n \in O(n^2)$, we need to find a c and n_0 that satisfies the above definition. Note that $f(n)$ is the function that we are trying to prove the order of and $g(n)$ is the simple function representing the order. This is easiest with a chain of inequalities:

$$\begin{aligned} 10n &\leq c \cdot n^2 & \\ &\leq 10n^2 & \text{for } n \geq 1 \\ &= cn^2 & \text{so } c = 10, n_0 = 1 \end{aligned}$$

Note that you can argue this differently, i.e.

$$\begin{aligned} f(n) &\leq cg(n) & \text{the general definition} \\ 10n &\leq cn^2 & \\ 10n &\leq c \times n \times n & c = 1 \\ 10n &\leq n \times n & \forall n \geq 10, \text{ so } n_0 = 10 \end{aligned}$$

Note that in this example the values of c and n_0 are not fixed, and they won't be. The main thing is that you argue that it holds for all n past some point, $n \geq n_0$. *Simply giving a c and n_0 is not an answer.*

Show $\mathbf{5n + 20 \in O(n)}$

$$\begin{aligned} 5n + 20 &\leq 5n + 20n && \text{for } n \geq 1, \textit{ not} \text{ true at } 0 \\ &= 25n && \text{so } c = 25, n_0 = 1 \end{aligned}$$

Note that this one is at least a little interesting, since $5n+20$ and n are of the same order of growth, since they are polynomials of the same degree. Let's do it a different way, observing that since $f(n)$ has an additional positive lower order term, i.e. $+20$, $c > 5$. Let's pick $c = 10$ to make the algebra clean.

$$\begin{aligned} f(n) &\leq cg(n) && \text{the general definition} \\ 5n + 20 &\leq 10n && \text{picked } c = 10, \text{ discussed above} \\ 20 &\leq 5n \\ 4 &\leq n && \text{so } c = 10, n_0 = 4 \end{aligned}$$

Extra example: Show $\mathbf{n^3 - n^2 + n + 1 \in O(n^3)}$

$$\begin{aligned} n^3 - n^2 + n + 1 &\leq n^3 + n + 1 && -n^2 \leq 0 \text{ for all } n \geq 0 \\ &\leq n^3 + n^3 + n^3 && n \geq \mathbf{1} \\ &\leq 3n^3 && \text{So, } c = 3, n_0 = 1 \end{aligned}$$

The actual values of c, n_0 aren't that interesting. Pick ones that make the algebra clear and make it less likely for you to make a mistake.

Extra example, round 2: Show $\mathbf{n^3 - n^2 + n + 1 \in \Omega(n^3)}$ Well, the definition of "big-Ω" is:

Definition: $f(n)$ is in $\Omega(g(n))$, $f(n) \in \Omega(g(n))$, if order of growth of $f(n) \geq$ order of growth of $g(n)$ (perhaps within constant factor of $g(n)$), i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \geq cg(n) \text{ for every n } \geq n_0 \tag{21}$$

$$
\begin{aligned}
n^3 - n^2 + n + 1 &\geq n^3 - n^2 && n + 1 > 0 \text{ for all } n \geq 0 \\
&\geq \frac{1}{2}n^3 + \frac{1}{2}n^3 - n^2 \\
&\geq \frac{1}{2}n^3 + \mathbf{\frac{1}{2}} \cdot \mathbf{n} \cdot n^2 - n^2 && \frac{1}{2} \cdot n \geq 1, \forall n \geq 2 \\
&\geq \frac{1}{2}n^3 && \text{So, } c = \frac{1}{2}, n_0 = 2
\end{aligned}
$$

Pitfall, $n \to \infty$: Show $\mathbf{n + n \log_2 n \in O(n)}$ (a false statement) This is why you need to reason a little carefully, the inequality must hold as n goes to ∞. Logarithms grow very slowly, but they still grow, i.e. you cannot ignore them. This example also demonstrates why I don't zero out terms.

$$
\begin{aligned}
n + n \log_2 n &\leq c \cdot n && \textbf{at } n = 1, \text{ for } c = 1 \\
n + n \cdot \mathbf{0} &\leq c \cdot n && \textbf{at } n = 1, \text{ for } c = 1
\end{aligned}
$$

2.4 Establishing order of growth using limits (18)

$10n$ **vs.** n^2

$$
\begin{aligned}
&\lim_{n\to\infty} \frac{10n}{n^2} && \text{simplify} \\
&= \lim_{n\to\infty} \frac{10}{n} && \text{top} \to 10, \text{ bottom} \to \infty \\
&= \lim_{n\to\infty} \frac{10}{n} \to 0
\end{aligned}
$$

So the conclusion is the bottom, n^2, grows faster than the top, $10n$.

$n(n+1)/2$ **vs.** n^2

$$\begin{aligned}
&\lim_{n\to\infty} \frac{\frac{n(n+1)}{2}}{n^2} \\
&= \lim_{n\to\infty} \frac{n(n+1)}{2n^2} \\
&= \lim_{n\to\infty} \frac{(n^2+n)}{2n^2} \\
&= \lim_{n\to\infty} \frac{(n^2+n)}{2n^2} \cdot \frac{\frac{1}{n^2}}{\frac{1}{n^2}} \\
&= \lim_{n\to\infty} \frac{1+\frac{1}{n}}{2} \\
&= \lim_{n\to\infty} \frac{1+\frac{1}{n}}{2} \to \frac{1}{2} \qquad \text{since } \frac{1}{n} \to 0
\end{aligned}$$

So, the conclusion, since the limit converged on a nonzero constant, is that they are of the same order of growth.

2.5 L'Hôpital's rule and Stirling's formula (19)

Use a limit to compare $\log_{\mathbf{b}} \mathbf{n}, \mathbf{b} > \mathbf{1}$ and n

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log_b n}{n} &= \lim_{n\to\infty} \frac{(\log_b n)'}{n'} \qquad L'H \\
&= \lim_{n\to\infty} \frac{\frac{1}{n \ln b}}{1} \\
&= \lim_{n\to\infty} \frac{1}{n \ln b} \to 0
\end{aligned}$$

Therefore, the denominator, n grows faster than $\log_b n$. You use L'Hôpital's when you have an indeterminate form, generally for us meaning the top and the bottom are both going to infinity. Always simplify before applying L'Hôpital's.

Use a limit to compare $2^{\mathbf{n}}$ and n! using Stirling's Formula,

$\mathbf{n! \approx (2\pi n)^{1/2}(\frac{n}{e})^n}$

$$\begin{aligned}
\lim_{n\to\infty}\frac{n!}{2^n} &\approx \lim_{n\to\infty}\frac{(2\pi n)^{1/2}(\frac{n}{e})^n}{2^n} && \text{Stirling's}\\
&= \lim_{n\to\infty}\frac{(2\pi n)^{1/2}(\frac{n}{e})^n}{2^n}\cdot\frac{(\frac{1}{2})^n}{(\frac{1}{2})^n} && \text{times one}\\
&= \lim_{n\to\infty}\frac{(2\pi n)^{1/2}(\frac{n}{\mathbf{2}e})^n}{\mathbf{1}} \to \infty
\end{aligned}$$

Notice that a little algebra and this problem is trivialized.

2.6 Orders of growth of some important functions (20)

All logarithmic functions $\log_a n$ **belong to the same class** $\Theta(\log n)$ **no matter what the logarithm's base** $a > 1$ **is** Stated a little differently, all $\log_a n, \forall a > 1$ functions grow within a constant of one another. Just set up the limit for two logarithmic functions $f(n) = \log_a n, a > 1$ and $g(n) = \log_b n, b > 1$ and see. Use the following property:

$$\log_a x = \frac{\log_b x}{\log_b a} \qquad \text{pg. 475, property 7}$$

$$\begin{aligned}
\lim_{n\to\infty}\frac{f(n)}{g(n)} &= \lim_{n\to\infty}\frac{\log_a n}{\log_b n}\\
&= \lim_{n\to\infty}\frac{\frac{\log_b n}{\log_b a}}{\log_b n} && \text{Using Eqn. } \mathbf{??}\\
&= \lim_{n\to\infty}\frac{\frac{1}{\log_b a}}{1}\\
&= \lim_{n\to\infty}\frac{1}{\log_b a} \to \frac{1}{\log_b a} && \text{since there is no } n \text{ growing}
\end{aligned}$$

This converges to a nonzero constant; therefore, all logs with bases $a > 1, b > 1$ grow within a constant multiple of one another.

All polynomials of the same degree k belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \mathbf{...} + a^0 \in \Theta(n^k)$

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{a_k n^k + a_{k-1} n^{k-1} + ... + a^0}{n^k} && n^k \text{ is the term w/ greatest order} \\
&= \lim_{n\to\infty} \frac{a_k n^k}{n^k} + \frac{a_{k-1} n^{k-1}}{n^k} + ... + \frac{a^0}{n^k} \\
&= a_k + 0 + ... + 0 \to a_k
\end{aligned}$$

This shows that only the leading term matters as $n \to \infty$.

Exponential functions a^n have different orders of growth for different a's

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{a^n}{b^n} \\
&= \lim_{n\to\infty} (\frac{a}{b})^n && \text{simple algebra} \\
&= \lim_{n\to\infty} (\frac{a}{b})^n \to \infty && \text{if } a > b \\
&= \lim_{n\to\infty} (\frac{a}{b})^n \to 0 && \text{if } a < b
\end{aligned}$$

The last two lines state that the exponent with the greater base grows faster.

2.7 Limit Examples (22)

$$\begin{aligned}
\lim_{n\to\infty}\frac{f(n)}{g(n)} &= \lim_{n\to\infty}\frac{2^n}{n!} && \textit{without} \text{ Stirling's} \\
&= \lim_{n\to\infty}\frac{2\times 2\times ...\times 2}{1\times 2\times ...\times n} \\
&= \lim_{n\to\infty}\frac{2}{1}\times\frac{2}{2}\times\frac{2}{3}\times ...\frac{2}{n-1}\times\frac{2}{n} \\
&< \lim_{n\to\infty}\frac{2}{1}\times\frac{2}{2}\times(\frac{2}{3})^{\mathbf{n-2}} && n>2 \\
&< \lim_{n\to\infty}(\frac{2}{3})^{\mathbf{n-2}} && n>2 \\
&= \lim_{n\to\infty}(\frac{2}{3})^{\mathbf{n-2}}\to 0
\end{aligned}$$

The final product goes to 0.

$$\begin{aligned}
\lim_{n\to\infty}\frac{f(n)}{g(n)} &= \lim_{n\to\infty}\frac{3^n}{2^n} && \text{did generically, above.} \\
&= \lim_{n\to\infty}(\frac{3}{2})^n\to\infty
\end{aligned}$$

Therefore 3^n grows faster than 2^n.

Well, we know logarithms grow slower than linear functions, but what

about the *squares* of logarithms?

$$\begin{aligned}
\lim_{n\to\infty}\frac{f(n)}{g(n)} &= \lim_{n\to\infty}\frac{n}{\ln^2 n} && \text{indeterminate form, top/bottom} \to \infty \\
&= \lim_{n\to\infty}\frac{n'}{(\ln^2 n)'} && \text{L'Hôpital's} \\
&= \lim_{n\to\infty}\frac{1}{2n^{-1}\ln n} && \text{bottom: chain rule} \\
&= \lim_{n\to\infty}\frac{n}{2\ln n} && \text{indeterminate form} \\
&= \lim_{n\to\infty}\frac{n'}{(2\ln n)'} && \text{L'Hôpital's} \\
&= \lim_{n\to\infty}\frac{1}{2n^{-1}} \\
&= \lim_{n\to\infty}\frac{n}{2} \to \infty
\end{aligned}$$

So the top, n, grows faster than the bottom, $\ln^2 n$. Not only do degree-1 polynomials grow faster than logs, they grow faster than the squares of logs. Yes, you can apply L'Hôpital's multiple times.

$$\begin{aligned}
\lim_{n\to\infty}\frac{f(n)}{g(n)} &= \lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}} && \text{indeterminate form, top/bottom} \to \infty \\
&= \lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'} && \text{L'Hôpital's} \\
&= \lim_{n\to\infty}\frac{\frac{1}{n\ln 2}}{\frac{1}{2}n^{-\frac{1}{2}}} \\
&= \lim_{n\to\infty}\frac{\frac{2}{n\ln 2}}{n^{-\frac{1}{2}}} \\
&= \lim_{n\to\infty}\frac{2\cdot n^{\frac{1}{2}}}{n\ln 2} \\
&= \lim_{n\to\infty}\frac{2}{n^{\frac{1}{2}}\ln 2} \\
&= \lim_{n\to\infty}\frac{2}{\sqrt{n}\ln 2} \to 0
\end{aligned}$$

So, $\log_2 n$ grows slower than square root. You can parameterize the above argument in *any positive* power of n and find that all logs grow slower than any positive (even fractional) power of n.

2.8 Spring 2019 Quiz 2 Solution

1. Show that $n \log_2 n + n - 2 \in O(n \log_2 n)$ using the definition of "big-O", $O()$-notation.

It helps to label the functions:

$$f(n) = n \log_2 n + n - 2$$
$$g(n) = n \log_2 n$$

Remember what you are trying to prove/argue: There exists some $c > 0$, $n \geq 0$ such that $f(n) \leq cg(n)$ for all n past some point called n_0. Give the c and the n_0.

Doing this by a chain of inequalities going *up*, $f(n) \to g(n)$:

$$\begin{aligned} n \log_2 n + n - 2 &< n \log_2 n + n && \text{always/everywhere} \\ &\leq n \log_2 n + n \log_2 n && \forall n \geq 2, \log_2 2 = 1 \\ &= 2n \log_2 n && \text{so } \mathbf{c = 2, n_0 = 2} \end{aligned}$$

Sometimes, not this time really, you can go the opposite direction more easily. Chaining *down* $g(n) \to f(n)$:

$$\begin{aligned} cn \log_2 n &= (c-1)n \log_2 n + n \log_2 n && \text{"eliminate" first term} \\ &= (c-2)n \log_2 + n \log_2 n + n \log_2 n && \text{new term "eliminates" } n \text{ if } n \geq 2 \\ &= (c-2)n \log_2 + n \log_2 n + n \log_2 n && -2 \text{ doesn't need to be eliminated, } c = 2 \\ &= (2-2)n \log_2 + n \log_2 n + n \log_2 n && \text{plug in } c = 2 \\ &= n \log_2 n + n \log_2 n && \text{simplify} \\ &> n \log_2 n + n && \text{most of } f(n) \\ &> n \log_2 n + n - 2 && \text{now, for } \mathbf{c = 2, n_0 = 2} \end{aligned}$$

Another way is to just start with the definition of $O()$-notation and figure out the constraints. I *used to* show students this way but it has got many pitfalls. It does help demonstrate some features, however.

$$
\begin{aligned}
f(n) &\leq c \times g(n) && \underline{\text{Prove holds: }} \forall n \geq n_0 \\
n \log_2 n + n - 2 &\leq c \times n \log_2 n \\
n - 2 &\leq (c-1) n \log_2 n && c \geq 1 \text{ as } n \to \infty \\
n &\leq (c-1) n \log_2 n + 2 && \text{which holds even at } n = 1 \text{ if...} \\
n &\leq (c-1) n \log_2 n + 2 && c = 2
\end{aligned}
$$

So, we find that even $\mathbf{c = 2, n_0 = 1}$ works. What if that "-2" was a "-1" or more? Try playing around with a graphing calculator; there are some online ones.

2. Compare the order of growth of $n \times n!$ and $n! \ln n$ using limits.

Remember:

$$\frac{d}{dx} \ln x = \frac{1}{x}$$

Also remember to simplify every chance you get.

$$
\begin{aligned}
\lim_{n \to \infty} \frac{n \times n!}{n! \ln n} &= \lim_{n \to \infty} \frac{n \times n!}{n! \ln n} \\
&= \lim_{n \to \infty} \frac{n}{\ln n} && \text{indeterminate form} \\
&= \lim_{n \to \infty} \frac{n'}{(\ln n)'} && \text{L'Hôpital's} \\
&= \lim_{n \to \infty} \frac{1}{\frac{1}{n}} \\
&= \lim_{n \to \infty} n \to \infty && \text{original top grows faster}
\end{aligned}
$$

So, the conclusion is that $n \times n!$ grows faster than $n! \ln n$. **Always state the conclusion**.

2.9 Spring 2019 Quiz 3 Solution

1. Show that $n \log_2 n + n - 2 \in \Omega(n \log_2 n)$ using the definition of "big-Ω", $\Omega()$-notation.

It helps to label the functions:

$$f(n) = n \log_2 n + n - 2$$
$$g(n) = n \log_2 n$$

Remember what you are trying to prove/argue: There exists some $c > 0$, $n \geq 0$ such that $f(n) \geq cg(n)$ for all n past some point called n_0. Give the c and the n_0.

Doing this by a chain of inequalities going *down*, since we're doing big-Ω, $f(n) \to g(n)$:

$$\begin{aligned} n \log_2 n + n - 2 &\geq n \log_2 n - 2 && \text{always/everywhere, } n > 0 \\ &\geq \frac{1}{2} n \log_2 n && \forall n \geq 4, 4 \cdot 2 - 2 \geq \frac{1}{2} \cdot 4 \cdot 2 \\ & && \text{...so } \mathbf{c = \frac{1}{2}, n_0 = 4} \end{aligned}$$

Remember: the values you find are not that interesting but you have to argue that they hold as $n \to \infty$. $n_0 = 4$ was picked simply because it makes the calculation for the $\log_2 n$ easy ($4 = 2^2$).

A different strategy was suggested by one of your peers, so here it is:

$$\begin{aligned} n \log_2 n + n - 2 &\geq n \log_2 n && n \geq 2 \\ & && \text{because when } n \geq 2, n - 2 \geq 0 \\ & && \text{so } \mathbf{c = 1, n_0 = 2} \end{aligned}$$

(this one is actually a little better, I think. It's a little less "turn the crank" than my approach). If you wanted to be a little more formal then you might go with the following:

$$n \log_2 n + n - 2 \geq c \cdot n \log_2 n \qquad n - 2 \in O(n) \text{ and } \forall n \geq 2, n - 2 \geq 0$$
$$n - 2 \geq (c-1) \cdot n \log_2 n \qquad \text{true everywhere } n \geq 2 \text{ if } c \leq 1$$
$$\text{(... logs go negative for } n < 1)$$
$$\text{so } \mathbf{c = 1, n_0 = 2}$$
$$\text{more generally } \mathbf{0 < c \leq 1, n_0 \geq 2}$$
$$\text{with this argument}$$

(if you were really picky you could then do analysis on $n-2 \in O(n \log n)$. Do you see why?)

2. Compare the order of growth of $n \log_3 n$ and $n \log_2 n$ using limits. State, clearly, your conclusion.

Remember:

$$\frac{d}{dx} \log_b x = \frac{1}{x \ln b}$$

Also remember to simplify every chance you get.

$$\begin{aligned}
\lim_{n\to\infty} \frac{n \log_3 n}{n \log_2 n} &= \lim_{n\to\infty} \frac{\log_3 n}{\log_2 n} && \text{indeterminate form} \\
&= \lim_{n\to\infty} \frac{\log_3' n}{\log_2' n} && \text{L'Hôpital's} \\
&= \lim_{n\to\infty} \frac{\frac{1}{n \ln 3}}{\frac{1}{n \ln 2}} \\
&= \lim_{n\to\infty} \frac{n \ln 2}{n \ln 3} \\
&= \lim_{n\to\infty} \frac{\ln 2}{\ln 3} \to \frac{\ln 2}{\ln 3} && \text{constants "converge" to constants}
\end{aligned}$$

So, since it converged on a constant, both functions have the same order of growth.

ALGORITHM *MaxElement*$(A[0..n-1])$
//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] > maxval$
    $maxval \leftarrow A[i]$
return *maxval*

2.10 Example 1: Maximum element (27)

Input Size:n,*the number of items in the array*
Basic Operation: $>$
Different Best/Worst/Average Cases?: No, you have to examine every item.
Analysis: Iterative algorithm, so do iterative analysis, a sum. One comparison per loop. Use bounds of for loop for sum. (The purpose of the analysis is to take a sum (or recurrence relation, for a recursive algorithm) and "solve"/simplify it to a simple formula that can be easily analyzed for order of growth, i.e. so we can classify as Big-$O(\_), \Theta(\_), \Omega(\_)$.)

$$C(n) = \Sigma_{i=1}^{n-1} 1 \qquad = (n-1) - 1 + 1 = n - 1$$

What's an upper bound on the order of growth? ($\mathbf{O}(\cdot)$): (note, I did not say prove) It's obviously $O(n)$.

2.11 Example 2: Element uniqueness problem (28)

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **return false**
return true

Input Size:n,*the number of items in the array*
Basic Operation: =, comparing items
Different Best/Worst/Average Cases?: Yes.

- **Best Case:** The first two items are duplicates. Input size doesn't matter so, $O(1)$.
- **Worst Case:** There are no duplicates. Analyzed below.
- **Average Case:** This is another one, like sequential search, where it's hard to do average case. You have to make many assumptions about the probabilities of duplicate keys, i.e. is there one key that might be duplicated, does every key have a possible duplicate... etc. The *one* interesting thing to think about, to me, is whether this **brute force** algorithm is *completely* useless compared to the more elegant **pre-sorting** strategy where you sort with a good algorithm, $O(n \log n)$, and then check *adjacent* pairs, $O(n)$. If duplicates are *common*, then you might *expect*, for the average case, to have very few iterations and it might actually be $O(n)$.

Worst Case Analysis: Copying the sum bounds from the loop bounds...

$$
\begin{aligned}
C(n) &= \Sigma_{i=0}^{n-2}\Sigma_{j=i+1}^{n-1}1 \\
&= \Sigma_{i=0}^{n-2}(n-1)-(i+1)+1 \\
&= \Sigma_{i=0}^{n-2}(n-1)-i \\
&= \Sigma_{i=0}^{n-2}(n-1)-\Sigma_{i=0}^{n-2}i && \text{split the sum} \\
&= (n-1)\Sigma_{i=0}^{n-2}1-\Sigma_{i=0}^{n-2}i && \text{drive left to closed form} \\
&= (n-1)((n-2)-0+1)-\Sigma_{i=0}^{n-2}i && \text{solve left} \\
&= (n-1)(n-1)-\Sigma_{i=0}^{n-2}i && \text{simplify left} \\
&= (n-1)(n-1)-\Sigma_{i=1}^{n-2}i && \text{adjust right's bound (x+0=x)} \\
&= (n-1)^2-\frac{(n-1)(n-2)}{2} && \text{simplify left and solve right} \\
&= \frac{2(n-1)(n-1)}{2}-\frac{(n-1)(n-2)}{2} && \text{common factor,}(n-1)\text{, combine} \\
&= \frac{2(n-1)(n-1)-(n-1)(n-2)}{2} && \text{combining...} \\
&= \frac{(2(n-1)-(n-2))(n-1)}{2} && \text{combining...} \\
&= \frac{(2n-2-n+2)(n-1)}{2} && \text{combining...} \\
&= \frac{n(n-1)}{2} && \text{combined.}
\end{aligned}
$$

Ugh. The fact that we end up with something that looks like a closed form, specifically

$$\mathbf{\Sigma_{i=1}^{n} i} = \frac{\mathbf{n(n+1)}}{\mathbf{2}}, \tag{22}$$

suggests we missed something. Looking at the code we can tell that every item is compared to every item after it. The first item is compared to the $n-1$ items after it, the second item to the $n-2$ after it, and so forth to the second to last item, which compared to the single, 1, item after it... that's just the sum

$$\Sigma_{i=1}^{n-1} i$$

written backwards and addition is commutative so this is equivalent. You can write this as

$$\Sigma_{i=1}^{n-1} n - i$$

as well.
What's an upper bound on the order of growth? ($\mathbf{O}(\cdot)$): It's obviously $O(n^2)$. Just change to polynomial form

$$\frac{n(n-1)}{2} = \frac{n^2 - n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

if this is not obvious.

2.12 Example 3: Matrix multiplication (30)

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $C[i, j] \leftarrow 0.0$
    for $k \leftarrow 0$ **to** $n-1$ **do**
      $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

Input Size:n,*the size of one side of the* square *matrices*
Basic Operation: multiplication, $\times$
Different Best/Worst/Average Cases?: No, and n-length column must be multiplied by an n-length row (element-wise, so n multiplies) for each position in the array, of which there are $n \times n$ of them.

Analysis:

$$
\begin{aligned}
C(n) &= \Sigma_{i=0}^{n-1}\Sigma_{j=0}^{n-1}\Sigma_{k=0}^{n-1}1 \\
&= \Sigma_{i=0}^{n-1}\Sigma_{j=0}^{n-1}(n-1)-0+1 \\
&= \Sigma_{i=0}^{n-1}\Sigma_{j=0}^{n-1}n \\
&= \Sigma_{i=0}^{n-1}n\Sigma_{j=0}^{n-1}1 && n \text{ doesn't vary in the sum} \\
&= \Sigma_{i=0}^{n-1}n \times n \\
&= n^2\Sigma_{i=0}^{n-1}1 \\
&= n^2 \times n \\
&= n^3
\end{aligned}
$$

What is the, $O(f(n))$, order of growth?: n^3 is obviously $O(n^3)$.

2.13 Example 4: Gaussian elimination (32)

Algorithm *GaussianElimination*(*A*[0..*n*-1,0..*n*])
//Implements Gaussian elimination of an *n*-by-(*n*+1) matrix *A*
for *i* ← 0 **to** *n* - 2 **do**
    for *j* ← *i* + 1 **to** *n* - 1 **do**
        for *k* ← *i* **to** *n* **do**
            A[*j*,*k*] ← *A*[*j*,*k*] - *A*[*i*,*k*] * *A*[*j*,*i*] / *A*[*i*,*i*]

Find the efficiency class and a constant factor improvement.

Input Size:n,*the number of rows in the matrix* (There is one extra column in an augmented matrix).
Basic Operation: Division, /
Different Best/Worst/Average Cases?: No, this version is like matrix multiplication. You do the same thing every time, no matter the input. (*This is a simplified version of the algorithm that doesn't do row swapping when a pivot goes to zero and, also doesn't deal with numerical stability issues.*) The algorithm works by "eliminating" (subtracting) a multiple, which varies row to row, of every row but the last from all rows beneath them.

Analysis:

$$
\begin{aligned}
C(n) &= \Sigma_{i=0}^{n-2}\Sigma_{j=i+1}^{n-1}\Sigma_{k=i}^{n}1 \\
&= \Sigma_{i=0}^{n-2}\Sigma_{j=i+1}^{n-1}n-i+1 && i \text{ varies in the } \textit{outer} \text{ sum} \\
&= \Sigma_{i=0}^{n-2}(n-i+1)\Sigma_{j=i+1}^{n-1}1 \\
&= \Sigma_{i=0}^{n-2}(n-i+1)((n-1)-(i+1)+1) \\
&= \Sigma_{i=0}^{n-2}(n-i+1)(n-1-i-1+1) \\
&= \Sigma_{i=0}^{n-2}(n-i+1)(n-i-1) \\
&= \Sigma_{i=0}^{n-2}(n^2-ni-n-ni+i^2+i+n-i-1) \\
&= \Sigma_{i=0}^{n-2}(n^2-ni-ni+i^2-1) \\
&= \Sigma_{i=0}^{n-2}(n^2-2ni+i^2-1) \\
&= \Sigma_{i=0}^{n-2}(n^2-1-2ni+i^2) \\
&= \Sigma_{i=0}^{n-2}(n^2-1)-\Sigma_{i=0}^{n-2}2ni+\Sigma_{i=0}^{n-2}i^2 \\
&= (n-1)(n^2-1)-\Sigma_{i=0}^{n-2}2ni+\Sigma_{i=0}^{n-2}i^2 \\
&= (n^3-n-n^2+1)-\Sigma_{i=0}^{n-2}2ni+\Sigma_{i=0}^{n-2}i^2 \\
&= (n^3-n^2-n+1)-2n\Sigma_{i=0}^{n-2}i+\Sigma_{i=0}^{n-2}i^2 \\
&= (n^3-n^2-n+1)-2n\Sigma_{i=\mathbf{1}}^{n-2}i+\Sigma_{i=0}^{n-2}i^2 \\
&= (n^3-n^2-n+1)-2n\frac{(n-2)(n-1)}{2}+\Sigma_{i=0}^{n-2}i^2 \\
&= (n^3-n^2-n+1)-2n\frac{(n^2-3n+2)}{2}+\Sigma_{i=0}^{n-2}i^2 \\
&= (n^3-n^2-n+1)-n(n^2-3n+2)+\Sigma_{i=0}^{n-2}i^2 \\
&= (n^3-n^2-n+1)-(n^3-3n^2+2n)+\Sigma_{i=0}^{n-2}i^2 \\
&= n^3-n^2-n+1-n^3+3n^2-2+\Sigma_{i=0}^{n-2}i^2 \\
&= 2n^2-n-1+\Sigma_{i=0}^{n-2}i^2 && \text{applying closed form} \\
&= 2n^2-n-1+\frac{(n-2)(n-1)(2(n-2)+1)}{6} \\
&= 2n^2-n-1+\frac{(n^2-3n+2)(2n-3)}{6} \\
&= 2n^2-n-1+\frac{(2n^3-6n^2+4n-3n^2+9n-6)}{6} \\
&= \frac{12n^2-6n-6}{6}+\frac{2n^3-9n^2+13n-6}{6} \\
&= \frac{2n^3+3n^2+7n-12}{6}
\end{aligned}
$$

What's an upper bound on the order of growth? ($\mathbf{O}(\cdot)$): $O(n^3)$.
Constant Factor Improvement: Note the basic operation, division, happens in the last term in the innermost loop:

$$A[j,k] \leftarrow A[j,k] - A[i,k] * \mathbf{A[j,i]}/\mathbf{A[i,i]}.$$

The division can be pulled out of the innermost loop and only calculated once per row elimination, so $O(n^2)$ divisions. This changes the basic operation to multiplication, since that is the next-most expensive, but not the order of growth since that multiplication was executed exactly as many times as the division.

2.14 Example 5: Counting binary digits (33)

ALGORITHM *Binary*(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
return *count*

This example looks like iterative analysis is appropriate (you can try to force it and get something reasonable) but this is a lot less obvious than the previous examples.
Input Size:n, in particular n's *value*. n's magnitude is what will control the number of loop iterations. Every iteration "chops off" one bit.
Basic Operation: / or $\lfloor n/2 \rfloor$, the latter might just be a right shift. Note that the algorithm might not be running on a computer.
Different Best/Worst/Average Cases?: No. Since we carefully described the input size, we can note that there is only one possible problem for each input size, i.e. only one natural number for each value. Remember, for best/worst/average cases, you assume a fixed large n and need to reason about the inputs of size n. When you're done, probably with recursive analysis, you can note that the input sizes can be grouped by the largest power of 2 less than or equal to n, but by then you're basically done anyway.

Analysis: Iterative algorithm, so *try* to do iterative analysis, a sum. In the previous examples, we could just copy the loop variables, generally from for-loops. We can't do that here. We'll do this later with the recursive analysis but the strategy that is relatively obvious is to just, by hand, try a number, e.g. $156 \to 78 \to 39 \to 19 \to 9 \to 4 \to 2 \to 1$. Note that the value (basically the input size) goes down by about half every time. Depending on how comfortable you are with logarithms, this may make it obvious that the runtime is $O(\log_2 n)$. Most students are more comfortable with powers so just write it backwards, $1 \to 2 \to 4 \to 9 \to 19 \to 39 \to 78 \to 156$, and note that the value approximately doubles every time, exponential growth. The inverse of exponential is logarithm. This is not rigourous... This algorithm is really just the iterative version of a recursive algorithm:

$$\begin{aligned} A(n) &= A(\lfloor n/2 \rfloor) + 1 && n > 1 \\ A(1) &= 1 && n == 1 \\ A(0) &= 1 && n == 0, \text{ (0 does require a bit)} \end{aligned}$$

Also, note that the basic operation is preserved but the loop counter variable disappeared. Basic operations are essential to the algorithm.
What's an upper bound on the order of growth? ($\mathbf{O}(\cdot)$): It's $O(\log_2 n)$, which is the same as $O(\log n)$. We generally omit the base of the logarithm since they are all the same order of growth for bases greater than one.

2.15 Example 1: Factorial (36)

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

Input Size:n, the *value/magnitude* of n determines the run time, the number of *recursive calls*. We will assume that multiplication is a constant time operation in this course. Sometimes the value of n will have an effect on an *individual* recursive call's runtime.

Basic Operation: Multiplication $*$ or $\times$. Picking the basic operation for recursive algorithms is generally easy since they often very strongly resemble the terse definitions and there are so few operations to consider.
Different Best/Worst/Average Cases?: No, this algorithm counts down from the number $n \to 0$, the base case (also called the "stopping condition"). We call it a base case because induction is quite easy to apply to many proofs regarding recursive algorithms. Induction goes from the base case to higher numbers, recursion generally the opposite.
Analysis: *Recursively* defined operation, so we need a **recurrence relation describing the number of basic operation calls; we don't use the recursively defined algorithm directly.** This is the analogue of converting iterative logic (for-loops, while-loops) to sums. It just seems a little odd that the recurrence relation we define looks *awfully* close to the recursively defined algorithm, which may be plain code or just a recursive definition (which *can* be more or less directly copied for some programming languages like Haskell to yield working code.

So, here's the algorithm, as given, again:

ALGORITHM $F(n)$
//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

which could be alternately defined like so

$$F(n) = F(n-1) \times \mathbf{n} \qquad n > 0$$
$$F(n) = 1 \qquad n = 0$$

...and following is the recurrence relation describing the number of basic operation calls, i.e. multiplies. It is called $M(n)$ because it is not calculating the n^{th} Fibonacci number, but, rather, the number of *multiplications* that the $F(n)$ would make.

$$M(n) = M(n-1)+\mathbf{1} \qquad n > 0$$
$$M(n) = 0 \qquad n = 0$$

Note that the base case does not have any multiplies, so it is just 0. Now that we have $M(n)$, we need to solve it. The objective, like with sums, is to get a simple expression that can be directly classified into $O(g(n))$ or whatever we need. We do this differently than sums, of course. More complicated ones may have a "sum on the side". We'll see this later and we'll show this in a toy way here. We are going to use "backwards substitution", covered in an appendix in your book, to do this. The advantage of this is that it tends to reveal the pattern more clearly than "forward substitution". Backwards substitution works *down/backwards* to the base case. Forward substitution works *up/forwards* to any/all inputs. To make it even more confusing, "backwards substitution" runs in the same direction or manner as the definition being used to calculate a result. Forward would work in the opposite direction, the same as induction would to prove some fact about the input but opposite of the actual application of the definition/algorithm. This slightly convoluted nomenclature might be confusing so I just wanted to get our bearings straight.

The basic idea of backwards substitution is to substitute the values into the recursive part of the definition, $M(n) = M(n-1) + 1$ above, and do this until the pattern becomes obvious. Once you see the pattern you "fast-forward" to the base case (generally one or two levels above it).

So, solving $M(n)$ to get a simple expression with no recursive definitions or sums:

Following is the definition, i.e. what you substitute into:

$$\text{Definition: } M(n) \qquad = M(n-1) + 1$$

Use the above part of the recursive definition and go until the pattern is obvious. (Note adjacent bolds and underlines, alternating.)

$$
\begin{aligned}
M(n) &= \mathbf{M(n-1)} + 1 \\
&= \underline{\mathbf{M(n-2)}} + \mathbf{1} + 1 && \textbf{substitute } n-1 \textbf{ in above} \\
&= \underline{\mathbf{M(n-3)} + 1} + 1 + 1 && \underline{\text{substitute } n-2 \text{ in above}} \\
&= \underline{\mathbf{M(n-4)}} + \mathbf{1} + 1 + 1 + 1 && \textbf{substitute } n-3 \textbf{ in above} \\
&= \underline{\mathbf{M(n-5)} + 1} + 1 + 1 + 1 + 1 && \underline{\text{substitute } n-4 \text{ in above}} \\
&\ldots \\
&= \underline{\mathbf{M(2)}} + \mathbf{1} + 1 + 1 + \ldots + 1 + 1 && \textbf{substitute } 3 \textbf{ in above (not shown)} \\
&= \underline{\mathbf{M(1)} + 1} + 1 + 1 + \ldots + 1 + 1 && \underline{\text{substitute } 2 \text{ in above}} \\
&= \underline{\mathbf{M(0)}} + \mathbf{1} + 1 + 1 + \ldots + 1 + 1 && \textbf{substitute } 1 \textbf{ in above} \\
&= \underline{0} + 1 + 1 + 1 + \ldots + 1 + 1 && \underline{\text{substitute } \textbf{Base Case}}
\end{aligned}
$$

Informally, there are $n+1$ calls to $F(n)$ since each call except the last decrements by one the current value and makes a recursive call with that value, so there is a call for every value of in range, inclusively, 0 to n. There is one multiply for each use of the *recursive* definition and zero for the *base case*, of which there is only one.

So, perhaps anticlimactically,

$$M(n) = n \qquad , n \geq 0 \ .$$

One last comment about backwards substitution: It's not a proof but it suggests what you might want to prove, if you need to, and is a reasonable argument if you don't. (Valid proofs are quite strong statements but proof techniques like induction don't really tell you *what* to prove, so you would likely use both backward substitution and induction *together* for recursive algorithms if you don't have a good idea what you want to prove.)

What's an upper bound on the order of growth? ($\mathbf{O}(\cdot)$): $M(n)$ is obviously $O(n)$.

2.16 Example 2: The Tower of Hanoi Puzzle (41)

Input Size:n,*the number of disks*

Basic Operation: Moving a disk is the only operation... so it's the basic operation. It is also essential to completing the task.
Different Best/Worst/Average Cases?: No.
Analysis: This one is a little vague: the algorithm is what we're analyzing – we're just counting moves. The goal for the player is to move the stack of disks on the left peg to the right peg. Each disk may only be moved between pegs and never placed on top of a smaller disk. So, it's necessary to move the largest disk, permanently, to the rightmost peg, and then move the second smallest to the rightmost peg, etc. So, the strategy is to move all the smaller disks to the middle, auxiliary, peg, move the largest to the right peg and then move the entire remaining stack to the right peg. $M(n)$ is a move operation. The base case is moving the smallest disk in a pile, M(1).

$$\begin{aligned} M(n) &= M(n-1) + 1 + M(n-1) && n > 1 \\ &= 2M(n-1) + 1 && \text{combine the moves of } n-1 \text{ disks} \\ M(1) &= 1 && n = 1 \end{aligned}$$

Solve the above recurrence relation.

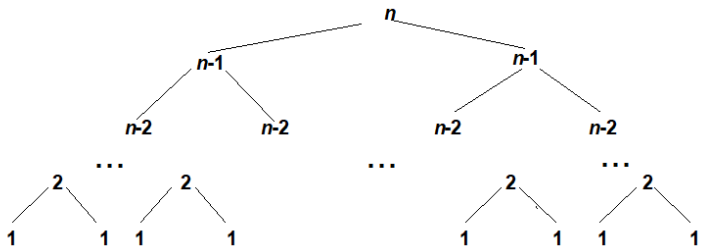
$$
\begin{aligned}
M(n) &= 2M(n-1)+1 \\
&= 2(2M(n-2)+1)+1 && \text{substitute} \\
&= 4M(n-2)+2+1 && \text{simplify} \\
&= 4(2M(n-3)+1)+2+1 && \text{substitute} \\
&= 8M(n-3)+4+2+1 && \text{simplify} \\
&= 8(2M(n-3)+1)+4+2+1 && \text{substitute} \\
&= 16M(n-3)+8+4+2+1 && \text{simplify} \\
&= 16M(n-3)+2^3+2^2+2^1+2^0 && \text{the pattern} \\
&= \cdots && \text{fast forward down to base cases} \\
&= 2^{n-3}2M(3)+2^{n-4}+\cdots+2^1+2^0 && n-1 \text{ recursive layers} \\
&= 2^{n-3}(2M(2)+1)+2^{n-4}+\cdots+2^1+2^0 && \text{substitute} \\
&= 2^{n-2}2M(2)+2^{n-3}+2^{n-4}+\cdots+2^1+2^0 && \text{simplify} \\
&= 2^{n-2}(2M(1)+1)+2^{n-3}+2^{n-4}+\cdots+2^1+2^0 && \text{substitute} \\
&= 2^{n-1}M(1)+2^{n-2}+2^{n-3}+2^{n-4}+\cdots+2^1+2^0 && \text{simplify} \\
&= 2^{n-1}\times 1+2^{n-2}+2^{n-3}+2^{n-4}+\cdots+2^1+2^0 && \text{substitute } \textbf{Base Case} \\
&= 2^{n-1}+2^{n-2}+2^{n-3}+2^{n-4}+\cdots+2^1+2^0 && \text{simplify}
\end{aligned}
$$

$$
\begin{aligned}
M(n) &= \Sigma_{i=0}^{n-1}2^i && \text{use closed form, pg. 476, Eqn. 5} \\
&= 2^n-1
\end{aligned}
$$

(note: I solved this a little differently than the slides.)

Consider the root problem and the tree figure: to move n disks you have to move the $n-1$ disks above, corresponding to the left subtree, move the disk (+1 in the recursive definition), and move the $n-1$ disks on top of the disk just moved, the right sub tree. To move $n-1$ disks you have to... <goes recursive, obviously>

The base cases also correspond to single moves, so we can just count the number of nodes in this tree and get our answer (*this does not work in general*). A term that we use when looking at these algorithms is "layer",



where a "layer" is all the nodes at the *same* depth in the tree. From top to bottom all the "layers" have exactly twice as many children in each child layer versus its corresponding parent layer. So, a piece of the pattern is successive powers of 2, i.e. $1, 2, 4, 8, \ldots$. The next question is, "How many powers of two?" Well, we start at n and go down to 1, so n powers of two, starting at 0, so $1, 2, 4, 8, \ldots, 2^{n-1} = 2^0, 2^1, 2^2, 2^3, \ldots, , 2^{n-1}$, which is a pattern, which should suggest there may be a closed form for this (appendix in book, referenced above).

What's an upper bound on the order of growth? ($\mathbf{O}(\cdot)$): It's obviously $O(2^n)$.

2.17 Example 3 – Solving Recurrence Relations Using Backward Substitutions (45)

For this one, we are just solving the recurrence, and forgetting about the algorithm. We'll still describe the algorithm's growth in $O()$ notation.

The recurrence:

$$\begin{aligned} T(n) &= T(n-1) + 2 && n > 1 \\ T(1) &= 2 && n = 1 \end{aligned}$$

The "solution", using backward substitution:

$$
\begin{aligned}
T(n) &= \mathbf{T(n-1)} + 2 \\
&= \underline{\mathbf{T(n-2)}} + \mathbf{2} + 2 && \textbf{substitute} \\
&= \underline{\mathbf{T(n-3)} + 2} + 2 + 2 && \underline{\textit{substitute}} \\
&= \mathbf{T(n-4)} + \mathbf{2} + 2 + 2 + 2 && \textbf{substitute} \\
&\ldots && \text{pattern 2 per recursive layer} \\
&= \mathbf{T(2)} + 2 + \cdots + 2 + 2 && \text{fast forward } \{n, n-1, \ldots, 2\} \\
&= \underline{\mathbf{T(1)}} + \mathbf{2} + 2 \cdots + 2 + 2 && \textbf{substitute} \\
&= \underline{2} + 2 + 2 \cdots + 2 + 2 && \underline{\text{substitute Base Case}} \\
&= 2 + \Sigma_{i=2}^{n} 2 \\
&= 2 + 2\Sigma_{i=2}^{n} 1 \\
&= 2 + 2(n - 2 + 1) \\
&= 2 + 2n - 2 \\
\mathbf{T(n)} &= \mathbf{2n}
\end{aligned}
$$

Obviously $2n \in O(n)$. Again, remember that $2n \in O(n^d), \forall d \geq 1$, and many other functions, e.g. $2n \in O(n^n)$. Since $O()$-notation is an *upper bound*, the lowest true upper bound is the most meaningful. Saying that $2n \in O(n^n)$ is like a weatherman predicting a temperature between $-273.15K$ and $10000K$ and that it may sleet, rain, snow, hail or be perfectly clear. It's almost entirely useless. Saying $2n \in O(n^2)$ is more like me writing down a test grade like "70 ± 30".

2.18 Example 4 – Solving Recurrence Relations Using Backward Substitutions (46)

Another one where we are just solving the recurrence and forgetting about the algorithm. We'll still describe the algorithm's growth in $O()$ notation.

The recurrence:

$$
\begin{aligned}
T(n) &= T(n-1) + 2\underline{n} && n \geq 1 \\
T(0) &= 2 && n = 0
\end{aligned}
$$

The "solution", using backward substitution:

$$
\begin{aligned}
T(n) &= \mathbf{T(n-1)} + 2n \\
&= \underline{\mathbf{T(n-2)}} + \mathbf{2(n-1)} + 2n && \textbf{substitute} \\
&= \underline{\mathbf{T(n-3)} + 2(n-2)} + 2(n-1) + 2n && \underline{\textit{substitute}} \\
&= \mathbf{T(n-4)} + \mathbf{2(n-3)} + 2(n-2) + 2(n-1) + 2n && \textbf{substitute} \\
&\ldots && \text{pattern: multiples of 2} \\
&= \mathbf{T(1)} + 2(2) + \cdots + 2(n-1) + 2n && \text{fast forward } \{n, n-1, \ldots, 2\} \\
&= \underline{\mathbf{T(0)}} + \mathbf{2(1)} + 2(2) + \cdots + 2(n-1) + 2n && \textbf{substitute} \\
&= \underline{2} + 2(1) + 2(2) + \cdots + 2(n-1) + 2n && \underline{\text{substitute Base Case}} \\
&= 2 + \Sigma_{i=1}^{n} 2i \\
&= 2 + 2\Sigma_{i=1}^{n} i \\
&= 2 + 2\frac{n(n+1)}{2} \\
&= 2 + n(n+1) \\
&= 2 + n^2 + n \\
\mathbf{T(n)} &= \mathbf{n^2 + n + 2}
\end{aligned}
$$

The most *meaningful* way to describe this in $O()$-notation is that $T(n) \in O(n^2)$ or $n^2+n+2 \in O(n^2)$. This example, and the Tower of Hanoi examples are more typical problems for recurrences. Often the "layer costs" are not fixed (sometimes they are like for Example 3 or binary search) but this is the other, somewhat more challenging, common case. You often have to do a sum to solve the pattern (patterns are often present with the recursively defined functions that recurrences represent). Also, this is an "advertisement" for backwards substitution: working with just the recursive cases when you from large n down to the base cases makes the recursive pattern clear. Working the other direction, "forward substitution" tends to have the base cases add some confounding terms.

2.19 Example 5 – Solving Recurrence Relations Using Backward Substitutions (47)

Another one where we are just solving the recurrence and forgetting about the algorithm. We'll still describe the algorithm's growth in $O()$ notation. This one has the problem size going down by a *half*, not a fixed amount, every time. A common trick is to assume the function is "smooth", which actually has a formal definition and make the number a power of some sort. Since we are dividing by two, a power of two makes sense here. You do not have to prove a function is smooth. I will tell you when you can use it.

The recurrence:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 2n && n > 1 \\ T(1) &= 2 && n = 1 \end{aligned}$$

Assumption: $n = 2^k, k \in \mathbb{N}$

The "solution", using backward substitution:

$$\begin{aligned} T(n) &= \mathbf{T}(\frac{\mathbf{n}}{\mathbf{2}}) + 2n \\ &= \underline{\mathbf{T}(\frac{\mathbf{n}}{\mathbf{4}})} + \mathbf{2}\frac{\mathbf{n}}{\mathbf{2}} + 2n && \textbf{substitution} \\ &= \underline{\mathbf{T}(\frac{\mathbf{n}}{\mathbf{8}}) + 2\frac{n}{4}} + 2\frac{n}{2} + 2n && \underline{\textit{substitution}} \\ &= \mathbf{T}(\frac{\mathbf{n}}{\mathbf{16}}) + \mathbf{2}\frac{\mathbf{n}}{\mathbf{8}} + 2\frac{n}{4} + 2\frac{n}{2} + 2n && \textbf{substitution} \\ &\cdots \\ &= \mathbf{T}(\mathbf{2}) + 2(4) + 2(8) + \cdots + 2\frac{n}{4} + 2\frac{n}{2} + 2n \\ &= \underline{\mathbf{T}(\mathbf{1})} + \mathbf{2}(\mathbf{2}) + 2(4) + 2(8) + \cdots + 2\frac{n}{4} + 2\frac{n}{2} + 2n && \textbf{substitute} \\ &= \underline{\mathbf{2}} + \mathbf{2}(\mathbf{2}) + 2(4) + 2(8) + \cdots + 2\frac{n}{4} + 2\frac{n}{2} + 2n && \underline{\text{substitute Base Case}} \\ &\cdots \end{aligned}$$

Looks like the pattern is successive powers of two, so can use the following equation (note the lower bound)

$$\Sigma_{i=0}^{n} 2^i = 2^{n+1} - 1 \tag{23}$$

Correcting the lower bound with base case.

$$= 2(1) + 2(2) + 2(4) + 2(8) + \cdots + 2\frac{n}{4} + 2\frac{n}{2} + 2n$$
$$= 2(2^0) + 2(2^1) + 2(2^2) + 2(2^3) + \cdots + 2\frac{n}{4} + 2\frac{n}{2} + 2n$$

Well, if $n = 2^k$, then we can evenly divide by $2 \log_2 n$ times.

$$= \Sigma_{i=0}^{\log_2 n} 2i$$
$$= 2\Sigma_{i=0}^{\log_2 n} i$$
$$= 2(2^{\log_2 n+1} - 1)$$
$$= 2(2^{\log_2 n+1} - 1)$$
$$= 2(2 \cdot 2^{\log_2 n} - 1)$$
$$= 2(2 \cdot n - 1)$$
$$= 4n - 2$$

(the slides do this with change of variable)

3 Ch.3: Brute Force

3.1 Brute Force (2)

List of various strategies for calculating a^n(preview of upcoming chapters). We're assuming $n \geq 0$ and $n \in \mathbb{N}$, i.e. it is a natural number.

3.1.1 Brute Force

"Brute Force" generally, vaguely, means to do a very straight forward approach, often using the definition. a^n is just n as multiplied together, so this is just

$$a^n = \Pi_{i=1}^{n} a$$

which does correctly handle the a^0 case.

3.1.2 Decrease by a constant

"Decrease by as constant" means to reduce your problem to a smaller (by a constant) subproblem and then extend the solution to solve the original problem. There's no advantage compared to brute force for this example, but it does yield a different approach.

$$\begin{aligned} a^0 &= 1 \\ a^n &= a \times a^{n-1} \qquad\qquad n > 0 \end{aligned}$$

One thing is that this is a little easier to translate to code than the Brute Force approach (recursive or iterative).

3.1.3 Decrease by a constant *factor*

"Decrease by as constant factor" means to reduce your problem to a smaller subproblem (by a constant factor, generally 2)and then extend the solution to solve the original problem. This actually yields a pretty big speedup.

$$
\begin{aligned}
a^0 &= 1 \\
a^n &= (a^{\frac{n}{2}})^2 && n > 0, \text{ if } n \text{ is even} \\
a^n &= a \times (a^{\lfloor \frac{n}{2} \rfloor})^2 && n > 0, \text{ if } n \text{ is odd}
\end{aligned}
$$

3.1.4 Divide and Conquer

The canonical divide and conquer strategy is to split the problem into subproblems, solve the subproblems, and then combine the solutions to the subproblems into a solution to the original problem. For this problem, it just corresponds to doing the $n-1$ multiplications in a different order for no real improvement *but* if you apply this approach one might notice a lot of repeated work, especially calculating a^2 or a^3 so, if one caches/remembers the result one could actually do this more efficiently (this is called **memoization**).

$$
\begin{aligned}
a^0 &= 1 \\
a^1 &= a \\
a^n &= a^{\lfloor \frac{n}{2} \rfloor} \times a^{\lceil \frac{n}{2} \rceil}, && n > 1
\end{aligned}
$$

4 Ch.4: Decrease and Conquer

4.1 Exponentiation Example

See pg.44.

4.2 Euclid's GCD Constant Factor Proof

The slides state "One can prove that the size, measured by the second number, decreases at least by half after two consecutive iterations." How to prove this? Let us write the algorithm here as a recursive function with the constraint that $m > n$:

$$\textbf{Euclid}(\mathbf{m},\mathbf{n}) \begin{cases} Euclid(n, m \mod n), & \text{if } n > 0 \\ m & \text{otherwise.} \end{cases}$$

Looking at this, we merely need to remember the general *mod* property that for z, where $z = x \mod y$ is guaranteed to be less than y, specifically, $z \in \{0 \dots y-1\}$.

$$\begin{aligned} Euclid(m,n) &= Euclid(n, m \mod n) \\ &= Euclid(n,q) \qquad\qquad q = m \mod n \end{aligned}$$

Now, the result of the first mod operation is $q = m \mod n$ and $q \in \{0 \dots n-1\}$. It helps to split this into two cases: If $q \in \{0 \dots \lfloor \frac{n}{2} \rfloor\}$, the we are done in one "iteration" of Euclid's. If not, then we can use the fact that $q \in \{\lceil \frac{n}{2} \rceil \dots n-1\}$, continuing the above.

$$Euclid(n,q) = Euclid(q, n \mod q) \qquad q \in \{\lceil \frac{n}{2} \rceil \dots n-1\}$$

$$Euclid(n,q) = \begin{cases} Euclid(q, n-q) & \text{if } q > \frac{n}{2} \text{ or } n \text{ is odd} \\ Euclid(q,0) & \text{if } n \text{ is even and } q = \frac{n}{2} \end{cases}$$

So, because $q \geq \frac{n}{2}$ and the second number in this last case is either 0 or $n-q$, the second number "decreases by at least a half", as we were supposed to prove.

ALGORITHM *Quickselect*$(A[l..r], k)$
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow$ *LomutoPartition*$(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ *Quickselect*$(A[l..s-1], k)$
else *Quickselect*$(A[s+1..r], k - 1 - s)$

4.3 Quickselect (23)

Let's start chapter 2-style analysis:
Input Size:n,*the number of items in the array*
Basic Operation: $<$, inside Lomuto Partition
Different Best/Worst/Average Cases?: Yes, best case you find the partition $s = k - 1$ in the first round. Worst case you only eliminate one item every round, so $O(n^2)$, since Lomuto Partition is $O(n)$, really $n-1$, since you compare the pivot to every other item once as the basic operation. What does the average case look like? To do this formally, we'd need expectations but let's do some back of the envelope analysis... which I think is fairly convincing. The issue that drives the run time to worst case or best case is the quality of the partition. The best case is where you find the perfect split every round . The worst is that you have really unlucky splits and only remove one (or some *constant* number of items every round of Quickselect). Both seem unlikely as $n \to \infty$ so let's do this a little pessimistically. Let's just assume that Lomuto Partitioning yields a $9:1$ split every time and that the k^{th} item is in the larger portion every time. So, we end up with something like the following. We're also dropping the l, r notation and just looking at the number of items, i.e. $A[n]$ instead of $A[l..r]$ i.e. $n = r - l + 1$.
Analysis: Recursive Algorithm $n-1$ comparisons per recursive call (Lomuto Partitioning cost).

$$C(A[1], k) = 0 \qquad \text{no comparisons in one item case}$$
$$C(A[n], k) = C(\frac{9n}{10}) + n - 1$$

Just remember that this is (grossly) informal but gives an intuition for why this is algorithm is linear (what are the odds of a sequence of splits this

bad or worse?). Also note that we're just going to reasonably assume that C(n) is non-negative. This is easiest if we just assume each recursive call has n cost, not $n-1$, which is an over approximation, which is fine, so

$$\begin{aligned} C(1) &= 0 \\ C(n) &= C(\frac{9n}{10}) + \mathbf{n} \qquad\qquad \text{make just n} \end{aligned}$$

This, with backward substitution, is going to yield

$$C(n) = n + (\frac{9}{10})n + (\frac{9}{10})^2 n + (\frac{9}{10})^3 n \cdots$$

but this is just $n\Sigma_{i=0}^{n}(\frac{9}{10})^i$ which is just n times a geometric series, which has a nice closed form for numbers less than one implied in the book and available elsewhere (look at equation 5 on page 476 and take $n \to \infty$). So for the $\Sigma_{i=0}^{n}(\frac{9}{10})^i$ part of the cost, we're just going to assume n goes to infinity, since it converges on a constant.

$$\Sigma_{i=0}^{\infty}(\frac{9}{10})^i = \frac{1}{1-\frac{9}{10}} \qquad\qquad = 10$$

so the total cost is just $10n$. This "analysis" roughly approximates one for Quicksort found in [1], which also has a formal version. We may do Quicksort a little more carefully.

What's an upper bound on the order of growth? ($\mathbf{O}(\cdot)$): It's obviously $O(n)$.

4.4 Interpolation Search (24)

The basic idea of interpolation search is that you assume a linear distribution of the keys to search a sorted array more efficiently. This is a somewhat stronger assumption than the simply sorted one in Binary Search. The basic idea, compared to binary search, is that you look at the end points and then *interpolate* between them. This is analogous to how one might search a dictionary: If you are looking for someone with last name "Voltaire", you look towards the end. There's this equation, used to pick the next index to search at:

$$x = l + \left\lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \right\rfloor$$

Since this is *linear* interpolation, we should be able to parse this without too much help, so let's do so: l, r, and x are *indices*, basically $x - values$ in the standard $y = mx + b$ equation. $A[l], A[r]$, and v correspond to the y values (these are the keys at some index) so the end*points* are $(l, A[l])$ and $(r, A[r])$. While I'm sure you can work backwards to use the $y = mx + b$ formula, probably having to find b, it seems much more useful to think about this from another point of view considering the slope m. What is the slope? Well, "rise over run" slope for the endpoints will be $\frac{A[r]-A[l]}{r-l}$. What we really want is the *index* of the item we're searching for. If you think of this with vectors then you're just adding a vector to the left point that corresponds to a fraction of the $\mathbf{l} - \mathbf{r}$ difference.

Note that I'm switching to vector notation so

$$\mathbf{l} = [l, A[l]]^T$$

and

$$\mathbf{r} = [r, A[r]]^T$$

so

$$\mathbf{r} - \mathbf{l} = [r - l, A[r] - A[l]]^T$$

Under the assumption that the distribution of the keys between $\mathbf{l}$ and $\mathbf{r}$ is linear, then the key we're searching for should be at some fraction of the $\mathbf{r}-\mathbf{l}$ difference vector equal to the ratio $d = \frac{v-A[l]}{A[r]-A[l]}$ added to $\mathbf{l}$.

So, the vector $\mathbf{v}$ corresponding to $[x, v]^T$ should be

$$\mathbf{v} = \mathbf{l} + d \cdot (\mathbf{r} - \mathbf{l})$$

or

$$[x, v]^T = [l, A[l]]^T + d \cdot ([r, A[r]]^T - [l, A[l]]^T)$$

so to solve the top (indices), we solve the bottom (values):

$$v = A[l] + d \cdot (A[r] - A[l])$$

rearranging for the single unknown, d:

$$d = \frac{v - A[l]}{A[r] - A[l]}$$

Now, using d to solve for the unknown index x (solving the top).

$$
\begin{aligned}
x &= l + d(r - l) \\
x &= l + \frac{v - A[l]}{A[r] - A[l]} \cdot (r - l) && \text{plug in } d \\
x &= l + \frac{(v - A[l])(r - l)}{A[r] - A[l]} && \text{simplify to form} \\
x &= l + \left\lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \right\rfloor && x \in \mathbb{N} \ (l \text{ already natural})
\end{aligned}
$$

Side note: Often, when working with points, instead of single numbers, you end up solving systems of equations. In this case we had two equations, corresponding to the x and y components of the vectors ("top" and "bottom", really x and $A[x]$). You can even think of $A[x]$ as a mapping (function) x to y, i.e $y = A[x]$ is the analog of $y = f(x)$.

Analysis for this one is difficult but the reward is substantial: On (uniformly distributed) random keys you expect the average run time to be $O(\log_2 \log_2 n) + 1$ comparisons, which grows *very* slowly. It is probably the slowest growing function we will encounter that actually grows. (What's the slowest growing we've encountered, which doesn't grow? Any function $f(n) \in O(1)$) If the distribution is actually linear, then you expect maybe two iterations). The worst case is actually $O(n)$. How?

References

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.