

Recurrence Notes

Inexperienced programmers tend to believe that a working program is a good program. Hopefully, as programmers become more experienced, they become less concerned with the binary it-works/it's-still-broken mentality and more concerned with the quality of algorithm they use. Quality is a vague word, and there are many factors that make one algorithm preferable to another. Sometimes one programmer's quality is another's hack. One factor that is universally accepted as desirable, however, is how fast an algorithm runs. Thus, it seems like a good idea to be able formally quantify "how fast" an algorithm works.

The obvious way to answer to the question "How fast does such-and-such a program run?" is to use a stopwatch. Something like the UNIX `time` command gives us a more precise answer that does not involve factors such as a person's attention span and reflexes. However, there are various possible objections to this easy answer. The time required by a program is a function of the input, so presumably we have to time several instances of the command and extrapolate the result. Some programs, however, behave fine for *most* inputs, but sometimes take a very long time; how do we report (indeed, how can we be sure to notice) such anomalies? What do we do about all the inputs for which we have no measurements? How do we validly apply results gathered on one machine to another machine?

The trouble with measuring raw time is that the information is precise, but limited: the time for *this* input on *this* configuration of *this* machine. On a different machine whose instructions take different absolute or relative times, the numbers don't necessarily apply. Indeed, suppose we compare two different programs for doing the same thing on the same inputs and the same machine. Program A may turn out faster than program B. This does *not* imply, however, that program A will be faster than B when they are run on some other input, or on the same input, but some other machine. Determining how programs run on average is often quite complicated. You can learn all about it and many other useful things in CompSci 130 (plug).

Another characteristic assumption in the study of *algorithmic complexity* (i.e., the time or memory consumption of an algorithm) is that we are interested in *typical* behavior of an idealized program over the entire set of possible inputs. Idealized programs, of course, being ideal, can operate on inputs of any possible size, and most "possible sizes" in the ideal world of mathematics are extremely large. Therefore, in this kind of analysis, it is traditional not to be interested in the fact that a particular algorithm does very well for small inputs, but rather to consider its behavior "in the limit" as input gets very large. For example, suppose that one wanted to analyze algorithms for computing π to any number of decimal places. I can make *any* algorithm look good for inputs up to, say, 1,000,000 by simply storing the first 1,000,000 digits of π in an array and using that to supply the answer when 1,000,000 or fewer digits are requested. If you paid any attention to how my program performed for inputs up to 1,000,000, you could be seriously misled as to the cleverness of my algorithm. Therefore, when studying algorithms, we look at their *asymptotic behavior*—how they behave as they input size goes to infinity.

Big-Oh review

The *order of growth* of an algorithm is measured using Big-O notation. An algorithm is $O(f(n))$ if there are constants c and n_0 such that $n \geq n_0, T(n) \leq cf(n)$. ($T(n)$ is the actual running time of the program for an input of size n .) The algorithm is said to have a *growth rate* of $f(n)$, or to run in $O(f(n))$ time.

Thus, if we say an algorithm is $O(n^2)$, then we mean that if n is sufficiently large (greater than n_0) then $T(n) \leq cn^2$. This definition is accurate, although a bit lax. Virtually any algorithm we will be interested in will be $O(n^n)$. Usually, when we say an algorithm runs in $O(f(n))$ time, we mean that $f(n)$ is a tight upper bound on the growth rate.

Some important rules for order arithmetic are:

¹Notes are collected directly from the various wonderful handouts of Mike Cleron, Nick Parlante, Paul Hilfinger, John Davis, and Owen Astrachan.

1. $O(f(n) + g(n)) = O(\max(f(n), g(n)))$
2. $O(c * f(n)) = O(f(n))$
3. If the running time of one code fragment is $O(f(n))$ and the running time of another code fragment is $O(g(n))$, then the running time of their product is $O(f(n)g(n))$. This rule is useful for nested loops or nested recursion.

General rules for calculating complexity

1. All basic statements in C++ (assignment, testing conditionals, etc.) run in constant time. In big-O notation, that is written $O(1)$.
2. The complexity of an algorithm will be determined by the complexity of the most frequently executed statement. This is a result of the addition rule. If one statement is executed n^3 times and another is executed n times, clearly the n^3 statement will dominate.
3. The most frequently executed can usually be found in the innermost loops or inside recursive functions.

Other bounds

We have big-Oh, but that's only part of the ball game. There's also big-Oh's friends little-Oh, Omega, and Theta.

- **Big Omega** $\Omega(f(n))$:
 $T(n) \in \Omega(f(n))$ if there are constants, $c, N_0 > 0$ such that $|T(n)| \geq c|f(n)|$ for all $n \geq N_0$.
- **Theta** $\Theta(f(n))$:
 $T(n) \in \Theta(f(n))$ if and only if $T(n) \in O(f(n))$ AND $T(n) \in \Omega(f(n))$
- **Little-Oh** $o(f(n))$:
 $T(n) \in o(f(n))$ if and only if $T(n) \in O(f(n))$ AND $T(n) \notin \Omega(f(n))$

The scorecard on our order of growth bounds is in Table 1. Table 2 gives a few common examples of orders that we deal with and their containment relations.

Notation	Properties	Bound	Loosely analogous relation
$O(f(n))$	$T(n) \leq cf(n)$	upper	\leq
$\Omega(f(n))$	$T(n) \geq cf(n)$	lower	\geq
$\Theta(f(n))$	$c_1f(n) \leq T(n) \leq c_2f(n)$	upper & lower	$=$
$o(f(n))$	$T(n) < cf(n)$	strict upper	$<$

Table 1: Different types of order notations

Table 2 shows some common examples of orders that we deal with and their containment relations.

1 Examples

Linear search

Let's apply all of this to a particular program. Here's a tail-recursive linear search for seeing if a particular value is in a sorted array:

```

/* True iff X is one of A[k]...A[A.length-1]. Assumes A is in
 * increasing order, k >= 0. */
boolean isIn(int[] A, int k, int X) {
    if (k >= A.size || A[k] > X)
        return false;
    else if (A[k] == X)
        return true;
    else
        return isIn(A, k+1, X);
}

```

$f(n)$	Is contained in	Is not contained in
$1, 1 + 1/n$	$O(10000), O(\sqrt{n}), O(n),$ $O(n^2), O(\lg n), O(1 - 1/n)$ $\Omega(1), \Omega(1/n), \Omega(1 - 1/n)$ $\Theta(1), \Theta(1 - 1/n)$ $o(n), o(\sqrt{n}), o(n^2)$	$O(1/n), O(e^{-n})$ $\Omega(n), \Omega(\sqrt{n}), \Omega(\lg n), \Omega(n^2)$ $\Theta(n), \Theta(n^2), \Theta(\lg n), \Theta(\sqrt{n})$ $o(100 + e^{-n}), o(1)$
$\log_k n, \lfloor \log_k n \rfloor,$ $\lceil \log_k n \rceil$	$O(n), O(n^\epsilon), O(\sqrt{n}), O(\log_{k'} n)$ $O(\lfloor \log_{k'} n \rfloor), O(n/\log_{k'} n)$ $\Omega(1), \Omega(\log_{k'} n), \Omega(\lfloor \log_{k'} n \rfloor)$ $\Theta(\log_{k'} n), \Theta(\lfloor \log_{k'} n \rfloor),$ $\Theta(\log_{k'} n + 1000)$ $o(n), o(n^\epsilon)$	$O(1)$ $\Omega(n^\epsilon), \Omega(\sqrt{n})$ $\Theta(\log_{k'}^2 n), \Theta(\log_{k'} n + n)$
$n, 100n + 15$	$O(.0005n - 1000), O(n^2),$ $O(n \lg n)$ $\Omega(50n + 1000), \Omega(\sqrt{n}),$ $\Omega(n + \lg n), \Omega(1/n)$ $\Theta(50n + 100), \Theta(n + \lg n)$ $o(n^3), o(n \lg n)$	$O(10000), O(\lg n),$ $O(n - n^2/10000), O(\sqrt{n})$ $\Omega(n^2), \Omega(n \lg n)$ $\Theta(n^2), \Theta(1)$ $o(1000n), o(n^2 \sin n)$
$n^2, 10n^2 + n$	$O(n^2 + 2n + 12), O(n^3),$ $O(n^2 + \sqrt{n})$ $\Omega(n^2 + 2n + 12), \Omega(n), \Omega(1),$ $\Omega(n \lg n)$ $\Theta(n^2 + 2n + 12), \Theta(n^2 + \lg n)$	$O(n), O(n \lg n), O(1)$ $o(50n^2 + 1000)$ $\Omega(n^3), \Omega(n^2 \lg n)$ $\Theta(n), \Theta(n \cdot \sin n)$
n^p	$O(p^n), O(n^p + 1000n^{p-1})$ $\Omega(n^{p-\epsilon}),$ $\Theta(n^p + n^{p-\epsilon})$ $o(p^n), o(n!), o(n^{p+\epsilon})$	$O(n^{p-1}), O(1)$ $\Omega(n^{p+\epsilon}), \Omega(p^n)$ $\Theta(n^{p+\epsilon}), \Theta(1)$ $o(p^n + n^p)$
$2^n, 2^n + n^p$	$O(n!), O(2^n - n^p), O(3^n), O(2^{n+p})$ $\Omega(n^p), \Omega((2 - \delta)^n), \Omega(n2^n)$ $\Theta(2^n + n^p)$ $o(n2^n), o(n!), o(2^{n+\epsilon}), o((2 + \epsilon)^n)$	$O(n^p), O((2 - \delta)^n)$ $\Omega((2 + \epsilon)^n), \Omega(n!)$ $\Theta(2^{2^n})$

Table 2: Some examples of order relations. In the above, $\epsilon > 0, 0 \leq \delta \leq 1, p > 1$, and $k, k' > 1$.

This is essentially a loop. As a measure of its complexity, let's define $C_{\text{isIn}}(N)$ as the maximum number of instructions it executes for a call with $k = 0$ and $A.\text{length} = N$. By inspection, you can see that such a call will execute the first `if` test up to $N + 1$ times, the second and third up to N times, and the tail-recursive call on `isIn` up to N times. With one compiler¹, each recursive call of `isIn` executes at most 14 instructions before returning or tail-recursively calling `isIn`. The initial call executes 18. That gives a total of at most $14N + 18$ instructions. If instead we count the number of comparisons `k >= A.length`, we get at most $N + 1$. If we count the number of comparisons against `X` or the number of fetches of `A[0]`, we get at most $2N$. We could therefore say that the function giving the largest amount of time required to process an input of size N is either in $O(14N + 18)$, $O(N + 1)$, or $O(2N)$. However, these are all the same set, and in fact all are equal to $O(N)$. Therefore, we may throw away all those messy integers and describe $C_{\text{isIn}}(N)$ as being in $O(N)$, thus illustrating the simplifying power of ignoring constant factors.

Again, this bound is a worst-case time. For all arguments in which `X <= A[0]`, the `isIn` function runs in constant time. That time bound—the *best-case* bound—is seldom very useful, especially when it applies to so atypical an input.

Giving an $O(\cdot)$ bound to $C_{\text{isIn}}(N)$ doesn't tell us that `isIn` *must* take time proportional to N even in the worst case, only that it takes no more. In this particular case, however, the argument used above shows that the worst case is, in fact, proportional to N , so that we may also say that $C_{\text{isIn}}(N) \in \Omega(N)$. Putting the two results together, $C_{\text{isIn}}(N) \in \Theta(N)$.

In general, then, asymptotic analysis of the space or time required for a given algorithm involves the following.

¹a version of gcc with the `-O` option, generating SPARC code for a Sun Sparcstation IPC workstation.

- Deciding on an appropriate measure for the *size* of an input (e.g., length of an array or a list).
- Choosing a representative quantity to measure—one that is proportional to the “real” space or time required.
- Coming up with one or more functions that bound the quantity we’ve decided to measure, usually in the worst case.
- Possibly summarizing these functions by giving $O(\cdot)$, $\Omega(\cdot)$, or $\Theta(\cdot)$ characterizations of them.

Quadratic example

Here is a bit of code for sorting integers:

```
void sort(int[] A) {
    for (int i = 1; i < A.size(); i++) {
        int x = A[i];
        int j;
        for (j = i; j > 0 && x < A[j-1]; j--)
            A[j] = A[j-1];
        A[j] = x;
    }
}
```

If we define $C_{\text{sort}}(N)$ as the worst-case number of times the comparison $x < A[j-1]$ is executed for $N = A.\text{length}$, we see that for each value of i from 1 to $A.\text{length}-1$, the program executes the comparison in the inner loop (on j) at most i times. Therefore,

$$\begin{aligned} C_{\text{sort}}(N) &= 1 + 2 + \dots + N - 1 \\ &= N(N - 1)/2 \\ &\in \Theta(N^2) \end{aligned}$$

This is a common pattern for nested loops.

Analyzing Recursive Programs: Recurrence Relations

A *recurrence relation* expresses the cost of an algorithm $T(n)$ in terms of the cost of the recursive calls it generates. It includes information about how many recursive calls are generated at each level of the recursion, how much of the problem is solved by each recursive call, and how much work is done at each level.

For example, the factorial program shown here:

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

The recurrence relation can be written as:

$$\begin{aligned} T(0) &= a \\ T(n) &= T(n - 1) + b \quad \text{if } n > 1 \end{aligned}$$

The recurrence above means that the cost of computing $T(0)$ is simply a constant a (the time it takes to do a conditional and return “1”), and the the cost of computing $T(n)$ where $n > 1$ is a constant b plus the cost of $T(n - 1)$. The constant b represents how much time it takes to set up the function call, do the condition, and then evaluate the expression. The term $T(n - 1)$ represents the cost of the recursive call `factorial(n-1)`.

Solving Recurrence Relations by Repeated Substitution

The recurrence relation contains a lot of information, but we can't determine the complexity of `factorial` just by looking at its recurrence relation. We would like to produce an expression for $T(n)$ that does not involve T on the right side of the equal sign. In math-speak, we would like a closed form expression for $T(n)$. There is a vast array of techniques used to solve recurrence relations. We will begin our study of this topic by using the most intuitive approach, which is to keep expanding the recurrence until we find a pattern. To solve the factorial recurrence, we begin by repeatedly expanding the right side of the recurrence relation. We know that $T(n) = T(n-1) + b$. Therefore, it must be true that $T(n-1) = T(n-2) + b$ (we obtained this by substituting $n-1$ for n in the original recurrence). Since we now have a value for $T(n-1)$, we can substitute this value for $T(n-1)$ in the original expression. We will do this a few times to get a feel for it:

$$\begin{aligned} T(n) &= T(n-1) + b \\ &= T(n-2) + 2b && \text{because } T(n-1) = T(n-2) + b \\ &= T(n-3) + 3b && \text{because } T(n-2) = T(n-3) + b \\ &= T(n-4) + 4b && \text{because } T(n-3) = T(n-4) + b \end{aligned}$$

At this point, a pattern should be obvious. After i substitutions, we will have an expression of the form

$$T(n) = T(n-i) + ib$$

We can verify this expression by induction on i , but we won't. The general expression of $T(n)$ in terms of i is very useful, since we can now substitute a value of i that will give us a final answer. Remember that $T(n) = T(n-i) + ib$ holds for *any* value of i , so we can pick any i that will give us a closed form answer. In this case, we choose $i = n$, so

$$\begin{aligned} T(n) &= T(0) + n * b \\ &= a + n * b \\ &= O(n) \end{aligned}$$

Example Recurrences

1. Compute the complexity of this function which inserts an element into a binary tree

```
TreeNode insert(TreeNode t, int item)
{
    if (t == null)
        return new TreeNode(item, null, null);
    if (item < t.info)
        t->left = insert(t.left, item);
    else
        t->right = insert(t.right, item);
}
```

First, we write the recurrence relation in terms of n , the number of nodes in the tree:

$$\begin{aligned} T(1) &= a \\ T(n) &= T(n/2) + b \quad \text{if } n > 1 \end{aligned}$$

We are assuming the tree is balanced, so that each recursive call solves half the problem by discarding half the tree. (This assumption is where the $T(n/2)$ term comes from.) Note that even though there are two recursive calls in this

procedure, only one of them will ever be executed because they are protected from each other by an `else`. Once we have the recurrence relation, we can solve it in the usual manner:

$$\begin{aligned}
 T(n) &= T(n/2) + b \\
 &= T(n/4) + 2b \\
 &= T(n/8) + 3b \\
 &= T(n/16) + 4b \\
 &= T(n/2^i) + ib
 \end{aligned}$$

We want to progress to our base case where $n/2^i = 1$. Equivalently $2^i = n$, and if we take the base two logarithm (\lg) of both sides, we get $\lg 2^i = \lg n$. Thus, we should choose $i = \lg n$.

$$\begin{aligned}
 T(n) &= T(1) + b * \lg n \\
 &= a + b * \lg n \\
 &= O(\log n)
 \end{aligned}$$

What would happen if the tree were not balanced? Specifically, what if the tree had degenerated to a list where every subtree was nil? In this case, each recursive call would bring us only one step closer to the answer as opposed to cutting the search space in half. The recurrence relation for this case would be:

$$\begin{aligned}
 T(0) &= a \\
 T(n) &= T(n-1) + b \quad \text{if } n > 1
 \end{aligned}$$

We have already solved this recurrence relation and found it to be $O(n)$.

2. Remember the problem of converting a tree to a list, preserving the ordering of elements

```

void tree2ListHelpL (TreeNode t, ListNode first)
// pre: first points to first node of list built-so-far from nodes in t
//      t points to a binary search tree
// post: first points to first node of list built from
//       all nodes in t
// idea: think of first as the top of a stack of nodes onto which nodes
//       from the tree are being pushed
{
    if (t != 0)
    {
        Tree2ListHelpL(t->right, first); // right subtree into list
        first = new ListNode(t->info, first); // add root to front of list
        Tree2ListHelpL(t->left, first);
    }
}

```

The recurrence relation for this code is:

$$\begin{aligned}
 T(1) &= a \\
 T(n) &= 2T(n/2) + b \quad \text{if } n > 1
 \end{aligned}$$

The solution, as we might expect, works out to $O(n)$:

$$\begin{aligned}
 T(n) &= 2T(n/2) + b \\
 &= 4T(n/4) + 3b \\
 &= 8T(n/8) + 7b \\
 &= 16T(n/16) + 15b \\
 &= 2^i T(n/2^i) + (2^i - 1)b
 \end{aligned}$$

Choose $i = \lg n$

$$\begin{aligned}
 T(n) &= 2^{\lg n} T(1) + (2^{\lg n} - 1) * b \\
 &= n * a + (n - 1) * b \\
 &= O(n)
 \end{aligned}$$

3. In this example, we see what happens if there is more than one recursive call at each level, but each call does not solve very much of the problem. This procedure is a solution to the classic Towers of Hanoi problem.

```

void move(int from, int to, int aux, int numDisks)
// pre: numDisks on peg from,
// post: numDisks moved to peg to
{
    if (numDisks == 1)
        System.out.println(from + " to " + to);
    else
    {
        move(from, aux, to, numDisks-1);
        move(from, to, aux, 1);
        move(aux, to, from, numDisks-1);
    }
}

```

When we write the recurrence relation for this code, we should notice that we are making two calls at each level. However, each level brings us only one step closer to the solution. Intuition tells us that this should result in an exponential number of calls.

$$\begin{aligned}
 T(1) &= a \\
 T(n) &= 2T(n-1) + b \quad \text{if } n > 1
 \end{aligned}$$

$$\begin{aligned}
 T(n) &= 2T(n-1) + b \\
 &= 4T(n-2) + 3b \\
 &= 8T(n-3) + 7b \\
 &= 16T(n-4) + 15b \\
 &= 2^i T(n-i) + (2^i - 1)b
 \end{aligned}$$

Choose $i = n - 1$

$$\begin{aligned} T(n) &= 2^{n-1}T(1) + (2^{n-1} - 1) * b \\ &= 2^{n-1} * a + (2^{n-1} - 1) * b \\ &= O(2^n) \end{aligned}$$

4. Given a binary tree, is it a search tree?

```
boolean isBST(Tree t)
// postcondition: returns true if t represents a binary search
//                tree containing no duplicate values;
//                otherwise, returns false.
{
    if (t == null) return true;           // empty tree is a search tree

    return valsLess(t.left,t.info)      &&
           valsGreater(t.right,t.info) &&
           isBST(t.left)                 &&
           isBST(t.right);
}
```

Assume that `valsLess` and `valsGreater` both run in $O(n)$ time for an n node tree. The recurrence relation for this code is:

$$\begin{aligned} T(1) &= a \\ T(n) &= 2T(n/2) + 2O(n) + b \quad \text{if } n > 1 \\ &= 2T(n/2) + O(n) \end{aligned}$$

so plugging and chugging we get:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &= 16T(n/16) + 4n \\ &= 2^i T(n/2^i) + in \end{aligned}$$

Choose $i = \lg n$

$$\begin{aligned} T(n) &= 2^{\lg n} T(1) + n * \lg n \\ &= n * a + n * \lg n \\ &= O(n \log n) \end{aligned}$$

5. Consider a function with the following form.

```
int boom(int M, int X)
{
```

```

if (M == 0)
    return H(X);
return boom(M-1, Q(X)) + boom(M-1, R(X));
}

```

and suppose we want to compute $C_{\text{boom}}(M)$ —the number of times Q is called for a given M in the worst case. If $M = 0$, this is 0. If $M > 0$, then Q gets executed once in computing the argument of the first recursive call, and then it gets executed however many times the two inner calls of `boom` with arguments of $M - 1$ execute it. In other words,

$$\begin{aligned} C_{\text{boom}}(0) &= 0 \\ C_{\text{boom}}(i) &= 2C_{\text{boom}}(i-1) + 1 \end{aligned}$$

A little mathematical massage:

$$\begin{aligned} C_{\text{boom}}(M) &= 2C_{\text{boom}}(M-1) + 1, \text{ for } M \geq 1 \\ &= 2(2C_{\text{boom}}(M-2) + 1) + 1, \text{ for } M \geq 2 \\ &\vdots \\ &= \underbrace{2(\cdots(2(0+1)+1)+1)\cdots+1}_M \\ &= \sum_{0 \leq j \leq M-1} 2^j \\ &= 2^M - 1 \end{aligned}$$

and so $C_{\text{boom}}(M)$ is $O(2^M)$.

6. Consider now a subprogram that contains *two* recursive calls.

```
int mung(int [] A, L, U);
{
    if (L >= U)
        return false;
    else {
        int m = (L+U)/2;
        mung(A, L, m);
        mung(A, m+1, U);
    }
}
```

We can approximate the arguments of both of the internal calls by $N/2$ as before, ending up with the following approximation, $C_{\text{mung}}(N)$ to the cost of calling `mung` with argument $N = U - L + 1$ (we are counting the number of times the test in the first line executes).

$$\begin{aligned} C_{\text{mung}}(1) &= 3 \\ C_{\text{mung}}(i) &= 1 + 2C_{\text{mung}}(i/2), \quad i > 1 \text{ a power of 2.} \end{aligned}$$

So,

$$\begin{aligned} C_{\text{mung}}(N) &= 1 + 2(1 + 2C_{\text{mung}}(N/4)), \quad N > 2 \text{ a power of 2.} \\ &\vdots \\ &= 1 + 2 + 4 + \dots + N/2 + N \cdot 3 \end{aligned}$$

This is a sum of a geometric series $(1 + r + r^2 + \dots + r^m)$, with a little extra added on. The general rule for geometric series is

$$\sum_{0 \leq k \leq m} r^k = (r^{m+1} - 1)/(r - 1) = (r \cdot r^m - 1)/(r - 1)$$

so, taking $r = 2$,

$$C_{\text{mung}}(N) = 4N - 1$$

or $C_{\text{mung}}(N)$ is $O(N)$.

Helpful Hints

The Big Five recurrence relations

We, in general, just want you to be able to reason about recursive functions just using the recurrence relations below. You are responsible for being able to solve other recurrence relations, but, the most crucial thing is to be able to see a recursive function, derive the relation, and then the “Big Five” recurrence relations give you the big Oh.

Recurrence	Algorithm	Big-Oh solution
$T(n) = T(n/2) + O(1)$	Binary Search	$O(\log n)$
$T(n) = T(n - 1) + O(1)$	Sequential Search	$O(n)$
$T(n) = 2T(n/2) + O(1)$	Tree Traversal	$O(n)$
$T(n) = T(n - 1) + O(n)$	Selection sort (or other n^2 sorts)	$O(n^2)$
$T(n) = 2T(n/2) + O(n)$	Mergesort (average case Quicksort)	$O(n \log n)$

Say you are given running times for various programs for various size inputs, how can you determine the order of growth for each function? Here are some rules that you can use a *rough* guideline.

Constant $O(1)$

Symptoms increase size of input by arbitrary amount $k \rightarrow$ running time does not change

Examples adding ints, prepending onto a list, finding element in hash table (?)

Logarithmic $O(\log_k n)$

Symptoms multiply size by $k \rightarrow$ running time increases by constant amount

Examples Finding element in binary tree, binary search

Linear $O(n)$

Symptoms multiply size by $k \rightarrow$ running time increases by factor of k

Examples Finding element in unsorted vector, inorder traversal of binary tree

Quadratic $O(n^2)$

Symptoms multiply size by $k \rightarrow$ running time increases by a factor of k^2

Examples Given n points in 2 dimensions, find closest two points

Cubic $O(n^3)$

Symptoms multiply size by $k \rightarrow$ running time increases by k^3

Examples Given n points in 2 dimensions, determine if any three form a straight line

Exponential $O(k^n)$

Symptoms add constant amount to problem size \rightarrow running time increases by factor of k

Examples n queens, a whole lot of interesting problems