CSE 247 Exam II Study Guide
Compiled by Tony De La Nuez

RECCURENCES
	//Best resource I've found: http://www.radford.edu/~nokie/classes/360/recurrence.eqns.revised.html
	//I summarize most of the relevant material for the exam below.
Performance of Non-Recursive programs

```
for(int i = 0; i < n; i++){
	counter++;
}
```

Performance:
T(n) = θ(n)
	This is what we call a non-recursive routine. The program above takes a number n, and increments the counter n times.
	The operation performed inside of the loop is trivial, does not add any significant terms to the time complexity.

Example 2:
```
for(int i = 0; i < n; i++){
	for(int j = 0; j < n; j++){
		counter++;
	}
}
```

Performance:
T(n) = θ(n^2)
	There is now a nested loop. Looking at the outer loop, we can tell from the values of i (0, 1, 2, ... , n) that it iterates n times.
	Looking at the inner loop: j assumes values (0, 1, 2, ..., n) as well, meaning it also iterates n times.
	Since the inner loop acts INSIDE of the outer loop, we will MULTIPLY the iterations of each loop because for EACH iteration of the outer loop,
	the inner loop must complete its entire iteration. Therefore, the counter will be incremented O(n^2) times.

Question: If we add another for loop INSIDE the inner loop (j loop) that has the iteration (for(int k = 0; k < n ;k++)), what will the time complexity of the loop be?
Answer: θ(n^3)

For more information on nested loops, look at this handout:
// http://isites.harvard.edu/fs/docs/icb.topic600780.files/loop_analysis.pdf

Moving on,

Performance of Recursive Programs

	A recurrence defines T(n) in terms of T for smaller values. T(n) is the time complexity of the algorithm.
	//	https://en.wikipedia.org/wiki/Time_complexity

Example of Recurrences:
T(n) = 2(T(n/2))
T(n) = T(n-1) + 6

		//from https://www.cs.sfu.ca/~ggbaker/zju/math/recurrence.html
		Explanation of Divide-and-Conquer Relations
		"Binary search: takes O(1) time in the recursive step, and recurses on half the list. Its running time is f(n) = f(n/2) + 1"
		"Mergesort: takes O(n) time in the recursive step, and recurses on both halves of the list. It's running time is f(n) = 2f(n/2) + n"
		Binary Search on n objects: O(nlogn)
		Mergesort on n objects: O(nlogn)

How to Reason About Recurrences:
	T(n) = T(n-1) + 6
This recurrence is telling us that the time to run up to n is equal to the time at (n-1) + 6.
We call this a recurrence because each step is calculated based on the results of a step before it. There is a RECURRENCE RELATION between step T(n) and T(n-1).

Further, in this equation, the time taken to go from T(n-1) to T(n) is the difference between the two.
	T(n) - T(n-1) = T(n-1) + 6 - T(n-1)
	 = 6

In order to find the value of T(k) at a particular k, there must be INITIAL CONDITIONS.
For example, looking at the Fibonacci Sequence:
0,1,1,2,3,5,8,13...

The first terms are 0 and 1, each term after is the sum of the two previous terms.
We can define the recurrence:
	F(0) = 0
	F(1) = 1
	F(n) = F(n-1) + F(n-2)	for n >= 2

F(0) and F(1) are the initial conditions that we plug in to find any given value.
We must start at F(0) and work our way up to F(k) or start at F(k) and work our way down using the initial conditions, or we can solve the recurrence and evaluate the resulting expression
Generally, the initial conditions are the value of the recurrence for the smallest values of n.
Since many recurrences have multiple solutions, the initial conditions determine which solution applies.

Evaluating a Recurrence:
	Substituting values (THIS IS NOT THE SUBSTITUTION METHOD):
	Ex.
		Find the value of T(n) = 2(T(n-1)) + n	for n = 3, IC: T(1) = 4
	Ans.
		Start at T(1) (given initial condition)
		T(1) = 4
		Solve T(2)
		T(2) = 2(T(2 - 1)) + 2
		T(2) = 2(T(1)) + 2
		T(2) = 2(4) + 2
		T(2) = 10
		T(3) = 2(T(2)) + 3
		T(3) = 2(10) + 3
		T(3) = 23

Solving a Recurrence (IMPORTANT):
	To solve a recurrence, we must find the CLOSED FORM for it.
	Simply put, a closed form for T(n) defines T(n) without using T.
			Given T(n-1) + 1
				Closed form: T(n) = n

	Techniques to solve recurrences:
	*GUESS AND CHECK
		*FORWARD SUBSTITUTION
		*BACKWARD SUBSTITUTION
	*CHARACTERISTIC EQUATION
	*MASTER METHOD

Guess and Check, Forward Substitution (Backward Substitution is on the top link, much more tedious for this example)
Steps:
	1. Look for a pattern
	2. Guess
	3. Informal Check
	4. If closed form satisfies open and IC, informally correct
	5. If need to prove formally correct, use induction

	Using example T(n) = T(n-1) + 1, IC T(1) = 2
	1. T(1) = 2
		T(2) = T(1) + 1 = 2 + 1 = 3
		T(3) = 3 + 1 = 4
		T(4) = 4 + 1 = 5
		T(5) = 5 + 1 = 6
	2. Hopefully you have picked up on the pattern, T(n) = n + 1. This is our guess.
	3. T(n) = T(n-1) + 1 = [(n-1) + 1] + 1, Open form satisfied
		= n + 1
		T(1) = 1 + 1 = 2, IC satisfied
	In case you are confused about what happened in step 3:
		We are using T(n) = n + 1. When we see T(n-1) we just substitue n-1 in for n, yielding (n-1) + 1 for T(n-1), then we add the extra 1 because
		we are solving for T(n).

Solving Recurrences with the Substitution Method
1. Make a guess for the form of the solution, prove by induction
	Example: T(n) = 2(T(n/2)) + n
		Because of the n/2, we guess some sort of log function.
		Guess T(n) <= cnlogn for some constance c (T(n) = O(nlogn))

```
                Proof:
                - Base Case: show that our guess holds for some base case
                - Assume holds for (n/2):
                        T(n/2) <= c(n/2) log(n/2)
                - Prove it holds for n:
                        T(n) = 2(T(n/2)) + n
                                <= 2(c(n/2)log(n/2)) + n    (by substitution)
                                 = cnlog(n/2) + n
                                 = cnlogn - cnlog2 + n
                                 = cnlogn - cn + n      (since log2 is just a constant)

        Example from Lecture 4 Notes:
        //https://docs.google.com/presentation/d/1nZLEFGgmpqXEzbTxT722ciKyIEJpZEE9EKKMnCNF8R0/edit#slide=id.g17ece0a364_0_74
        Binary Search
        T(n) = T(n/2) + c
        Showed T(n) = θ(log n)
        Verify by substitution T(n) = O(n^2)
                - T(n) <= k n^2
                - T(n/2) <= k(n/2)^2
                - T(n/2) <= kn^2 /4
                        Substituting:
                        - T(n) = T(n/2) + c <= kn^2 /4 + c  <= kn^2 ?    (plugging in T(n/2))
                                                                kn^2  + 4c <= 4kn^2 ?    (mult by 4)
                                                                        4c <= 3kn^2 ?    (subtract kn^2 from each side)
                                                                        Yes. Can choose k = 10c or k = 4/3 c


        More information on substitution:
        //http://www.cs.cornell.edu/courses/cs3110/2014sp/recitations/24/using-the-substitution-and-master-method.html

COOKIE CUTTER MASTER METHOD:
Solving recurrences of the form:
        T(n) = aT(n/b) + θ(n^d)
        T(1) = c

        n = b^k, for positive integer k
        Conditions:
        a >= 1, b >= 2, c > 0, d >= 0 are constants

With the Master Method you ask: which term dominates?

Solutions using Master Method:
T(n) =  θ(n^d),            a < b^d          CASE 1
T(n) =  θ(n^d log n),      a = b^d          CASE 2
T(n) =  θ(n ^ (log_b(a))), a > b^d          CASE 3

Replace "T(n) = " with "T(n) <= " or "T(n) >= " to get O and Ω performance

Examples:
T(n) = 16T(n/4) + 5n^3
        - a = 16, b = 4, d = 3
        - 16 < 4 ^ 3, Case 1.
        T(n) = θ(n^3)

GENERAL MASTER METHOD:
Solving recurrence of the form:
        T(n) = aT(n/b) + f(n)

T(n) = θ(f(n)),                                         f(n) = Ω(n^(log_b(a)+ϵ)) and a*f(n/b) <= c*f(n) for some constant c<= 1 and large n     CASE 1
T(n) = θ(n^(log_b(a)) lg n),                    f(n) = θ(n^(log_b(a)))
        CASE 2
T(n) = θ(n^(log_b(a))),                                 f(n) = O(n^(log_b(a)+ϵ))
                CASE 3

Assume ϵ > 0
Case 1 can also be written as n^(log_b(a) + ϵ) = O(f(n))

Examples:
T(n) = 2T(n/2) + n
        a = 2, b = 2, f(n) = n, log_b(a) = lg 2 = 1
        Case 2
        T(n) = θ(nlgn)

T(n) = 8T(n/2) + n
        a = 8, b=2, f(n) = n, log_b(a) = log_2(8) = 3
        Case 3
        T(n) = θ(n^3)


RECURSION TREES
Recursion trees are used to calculate  the amount of work expressed by a recurrence equation
They are useful for visualing what happens when a recurrence is iterated and can be a good method of guessing.

//http://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec20-master/lec20.html

I can't easily include images in a text document, so go to this link and look at page 3:
//https://math.dartmouth.edu/archive/m19w03/public_html/Section5-1.pdf

In summary:
T(n) = aT(n/b) + g(n)

We keep track of the problem size, draw the tree, and keep track of work done.
- Each level of the diagram represents a level of iteration of the recurrence.
For each level, we determine three things:
        - Number of subproblems
        - Size of each subproblem
        - Total work done

We also need to determine how many levels there are in the recursion tree.
The tree has a vertex representing the initial problem and one representing each problem we need to solve.
Each non-leaf vertex has "a"(constant from formula) children.
The vertices are divided into levels corresponding to sub problems of the same size, which we write to the left.
On the right, we write the total amount of work done at that level by an algorithm whose work is described by the recurrence (not including recursive call work).
The bottom level of the tree represents the final stage of iterating the recurrence; this is the BASE CASE.
```