Part II: Complexity Analysis of Recursive Algorithms

AUBURN UNIVERSITY

Hugh Kwon

Slides adapted from Dr. Debswapna Bhattacharya's class

Topics

1. Complexity Notations

   What is the technical meaning of Big-Oh (O)? How about the other notations (small-oh o, theta Θ, omega Ω and small-omega ω)?

2. Approximate Big-Oh analyses of non recursive algorithms
3. Detailed complexity analyses of nonrecursive algorithms
4. Approximate Big-Oh analyses of recursive algorithms
5. Detailed complexity analyses of recursive algorithms
   1. Characterizing recursive algorithms by developing recurrence relations
   2. Analyzing their complexity by solving recurrence relations

Approximate Big-Oh Analysis of Complexity Recursive Algorithms

1. **For each step that is not a loop, decide whether it has a constant cost or not. Ignore any recursive calls.**
    - If constant cost, use a cost of O(1) for the entire step.
    - If not constant cost, estimate maximum (worst case) cost as a function of n and state the appropriate Big-Oh complexity.
2. **For loops, start from the innermost loop and work outwards, updating the complexity as each loop is considered. For each loop:**
    - Estimate the maximum (worst case or upper bound) number of times the loop statement (for, while, etc.) will execute as a function of n and state the appropriate Big-Oh complexity.
    - Calculate the single execution cost of the loop body as the maximum of the costs of each step in the body.
    - Multiply the Big-Oh estimate of the number of times the loop will be executed with the Big-Oh estimate of the loop bod's cost.
3. **Complexity of a single execution of the algorithm is the same as the largest Big-Oh complexity of any step or loop after all estimations in steps (1) and (2) are done.**
4. **Now estimate the maximum (worst case or upper bound) number of recursive executions that will take place as a function of n <u>by drawing recursion trees for various input sizes and trying to find a pattern</u> (if possible; otherwise a detailed analysis is required!) and state the corresponding Big-Oh complexity.**
5. **Multiply the two Big-Oh complexities to obtain the overall complexity.**

Example: Find-Max

```
function find-max-1(A:array [i...j] of number)
k: number
if i=j then return A[i]
k= find-max(A:array [i+1...j])
if k>A[i] then return k else return A[i]
```

What is the estimate of the complexity of a single execution of the algorithm ?

What is the estimate of the maximum number of recursive executions that will take place as a function of n? What is the corresponding Big-Oh complexity?

What is the approximate overall Big-Oh complexity of this algorithm?

```
function fib (n: non-negative integer)
if n=0 or 1 then return 1
return fib(n-1)+fib(n-2)
```

What is the estimate of the complexity of a single execution of the algorithm ?

What is the estimate of the maximum number of recursive executions that will take place as a function of n? (Hint: What is the number of nodes in a binary tree of height n?) What is the corresponding Big-Oh complexity?

What is the approximate overall Big-Oh complexity of this algorithm?

Example: Merge-Sort

```
if p < r
    then q = ⌊(p+r)/2⌋
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```

What is the estimate of the complexity of a single execution of the algorithm ?

What is the estimate of the maximum number of recursive executions that will take place as a function of n? What is the corresponding Big-Oh complexity?

What is the approximate overall Big-Oh complexity of this algorithm?

What is the estimate of the complexity of a single execution of the Merge-Sort algorithm?

Since Merge-Sort calls Merge, you have to do an approximate analysis of the Merge algorithm.

We will do that in class and show that it is $O(n)$.

Merge(A,p,q,r)

$n_1 = q - p + 1$

$n_2 = r - q$

create array L[1 .. $n_1 + 1$] and R[1 ..$n_2 + 1$]

for $i = 1$ ***to*** n_1

    L[i] =A[$p + i - 1$]

for $j = 1$ ***to*** n_2

    R[j] =A[$q + j$]

  L[$n_1 + 1$] = ∞

  R[$n_2 + 1$] = ∞

Merge(A,p,q,r)

```
i = 1
j = 1
for k = p to r
    if L[ i ] ≤ R[ j ]
            then A[ k ] = L[ i ]
            i = i + 1
    else A[ k ] = R[ j ]
            j = j + 1
```

What is the estimate of the complexity of a single execution of the Merge-Sort algorithm ?

Step#	Big-Oh complexity
1 **if** $p < r$	O(1)
2 **then** $q = \lfloor (p+r)/2 \rfloor$	O(1)
3 MERGE-SORT(A,p,q)	recursive call
4 MERGE-SORT(A,q+1,r)	recursive call
5 MERGE(A,p,q,r)	O(n)

Estimate of the complexity of a single execution of Merge Sort = **O(n)**

What is the estimate of the maximum number of recursive executions of Merge-Sort that will take place as a function of n?
What is the corresponding Big-Oh complexity?

```
1 if p < r
2    then q=⌊(p+r)/2⌋
3         MERGE-SORT(A,p,q)
4         MERGE-SORT(A,q+1,r)
5         MERGE(A,p,q,r)
```

Draw recursion trees and see that Steps 3 & 4 together produce 2n-1 recursive executions when Merge Sort gets an input array of size n

So estimate of the maximum number of recursive executions that will take place as a function of n is 2n-1

What is the corresponding Big-Oh complexity? **O(n)**

What is the approximate overall Big-Oh complexity of this algorithm?

$$O(n)*O(n)=O(n^2)$$

Techniques for Detailed Complexity Analysis of Recursive Algorithms

- Why can’t we use the complexity calculation technique that we have been discussing for non-recursive algorithms?

- Because of recursion…

- So, what do we do?

- Develop a pair of equations (called “recurrences” or “recurrence relations”) that characterize the behavior of a recursive algorithms and solve those to obtain the algorithm’s complexity.

Recurrences

- What are these?

- A pair of equations giving T(n) of recursive algorithms in terms of the cost of recursive calls and the cost of other steps

Step 1: Develop Recurrence Relations

- How?
  1. Determine what the base case(cases) is (are).
  2. Determine steps that will be executed when the input size matches the base case(s).
  3. Calculate the complexity of those steps.
  4. Write the first part of the RR as T(base case input sizes) = what you calculated
  5. Determine steps that will be executed when the input size is n, different from the base case(s).
  6. Determine how many recursive calls (and with what input size in relation to the original input size of n) will be made.
  7. Calculate the complexity of all other steps, excluding the recursive calls.
  8. Write the second part of the RR as T(n) = T(input size for first recursive call) + T(input size for next recursive call) +…+complexity of all other steps.

Fib (n: non-negative integer)
1 if n==0 or n==1 then return 1
2 else return Fib(n-1)+Fib(n-2)

- $T(n)= 6$ when $n < 2$
- $T(n)=T(n-1)+T(n-2)+11$ when $n \geq 2$

A Recursive Divide & Conquer Algorithm

```
Find-Max-Recursive(A:array[i…j] of numbers)
if i==j then return A[i]
else
  mid=floor((i+j)/2)
  return max(Find-Max-Recursive(A[i…mid]),
  Find-Max-Recursive(A[mid+1…j])
```

- understand this algorithm
- can you draw a recursion tree for A=[1,0,-5,7,23]?
- develop and state its two recurrences

- What can you tell about a recursive algorithm from its recurrences?...A lot!

1. How many new recursive executions each execution will produce
2. What the input sizes for each of those executions are
3. What is the amount of work done by each recursive execution besides making those recursive calls
4. What the base cases are
5. How much work is done by the algorithm for base case inputs
6. Whether it is a D & C algorithm or an incremental algorithm
7. Estimation of the algorithm’s overall complexity based on the form of the equations

- The detailed analysis technique we studied for non-recursive algorithms does not fully work…

Let us try the step-by-step approach we've used for analyzing algorithms.

Step#	# of times	Cost
1 **if** $p < r$	1	3
2 **then** $q = \lfloor (p+r)/2 \rfloor$	1	6
3 MERGE-SORT(A,p,q)	1	???
4 MERGE-SORT(A,q+1,r)	1	???
5 MERGE(A,p,q,r)	1	$T(n) = 41.5\,n+12$

- Show that the detailed complexity of the Merge algorithm is given by $T(n) = 41.5n + 12$ under these assumptions:

1. $n_1=n_2=n/2$ where n is the size of the entire array
2. Step 3 is simply a declaration and has no execution cost

Merge(A,p,q,r)

$n_1 = q - p + 1$
$n_2 = r - q$
create array L[1 .. $n_1 + 1$] and R[1 .. $n_2 + 1$]
for $i = 1$ ***to*** n_1
    L[i] =A[$p + i - 1$]
for $j = 1$ ***to*** n_2
    R[j] =A[$q + j$]
L[$n_1 + 1$] = ∞
R[$n_2 + 1$] = ∞

Merge(A,p,q,r)

```
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        then A[k] = L[i]
            i = i + 1
    else A[k] = R[j]
        j = j + 1
```

How complex is Merge Sort?

Step#	# of times	Cost
1 **if** $p < r$	1	3
2 **then** $q = \lfloor (p+r)/2 \rfloor$	1	6
3 MERGE-SORT(A,p,q)	1	???
4 MERGE-SORT(A,q+1,r)	1	??? + 2
5 MERGE(A,p,q,r)	1	41.5 n+12

When the input is a base case (array size = 1 or 0), the cost of Merge Sort is 3.

Otherwise (array size is some n>1), it is 41.5n + 23 + cost of the two recursive calls.

Let the function T(n) stand for the cost of the algorithm when the input size is n.

Then we can say that

$T(n) = 3$ when $n \leq 1$ and

$T(n)=2T(n/2)+41.5n+23$ when $n>1$

There are the recurrences of Merge Sort.

Simplifying Merge Sort Recurrences

$T(n) = 3, n \le 1$ and

$T(n)=2T(n/2)+41.5n+23, \; n>1$

can be simplified as

$T(n) = 3, n \le 1$ and

$T(n)=2T(n/2)+41.5n, \; n>1$

why? since for large n 41.5n will eclipse the constant added cost of 23 (but note that we can ignore 23 only because of the presence of a larger term 41.5n – had that not been present in the recurrence, we should not ignore 23)

this can be further simplified as

$T(n) = c, n \le 1$ and

$T(n)=2T(n/2)+cn, n>1$ where $c > 3$ and $c > 41.5$

why? because we are looking at finding an upper bound, replacing the various constants with one constant c that is larger than any of them will help simplify the analysis, and we would still obtain the upper bound.

Note: text uses the notation $\Theta(1)$ for constants and $\Theta(n)$ for a constant multiple of n, i.e., cn.

- Recurrences (or Recurrence Relations) of Merge Sort:

$$T(n) = \begin{cases} c & if\, n = 0\, or 1 \\ 2T(n/2) + cn & if\, n > 1 \end{cases}$$

Recurrence relations of Merge Sort

$$T(n) = \begin{cases} c & if\, n = 1 \\ 2T(n/2) + cn & if\, n > 1 \end{cases}$$

$$\Downarrow \Uparrow$$

$$T(n) = \begin{cases} \Theta(1) & if \quad n = 1 \\ 2T(n/2) + \Theta(n) & if \quad n > 1 \end{cases}$$

1. Backward substitution method (not in text)
2. Forward substitution method (not in text)
3. Recursion-tree method (Section 4.4)
4. Master method (Section 4.5)

Method of backward substitutions

- Start with the recurrence relation for T(n), and repeatedly expand its right-hand side by substituting for the T terms.
- After several such expansions, look for and find a pattern that allows you to express T(n) as a closed-form formula.
- If such a formula is evident, check its validity by direct substitution into the recurrence relations.

Method of backward substitutions

$T(n)=T(n-1)+n;\ T(0)=1$

$T(n-1)=T(n-2)+(n-1)$

So $T(n)=T(n-2)+n+(n-1)$

$T(n-2)=T(n-3)+(n-2)$

So $T(n)=T(n-3)+n+(n-1)\ +(n-2)$

$T(n-3)=T(n-4)+(n-3)$

So $T(n)=T(n-4)+n+(n-1)\ +(n-2)+(n-3)$

Eventually, $T(n)=T(n-n)+n+(n-1)\ +(n-2)+(n-3)+\ldots+(n-(n-1))=T(0)+n+(n-1)+(n-2)+(n-3)+\ldots+1= 1+ n(n+1)/2$

Check:

LHS of recurrence $T(n) = 1+ n(n+1)/2 = n^2/2+n/2+1$ RHS $= T(n-1) + n = 1+ (n-1)n/2 + n = n^2/2+n/2+1$

- Start with the recurrence relation for T(base case), and repeatedly calculate non-base cases, e.g., T(1), T(2) etc.
- After several such calculations, look for and find a pattern that allows you to express T(n) as a closed-form formula.
- If such a formula is evident, <u>check its validity</u> by direct substitution into the recurrence relations.

Method of forward substitutions

$T(n)=T(n-1)+1;\ T(0)=1$

$T(1)=T(0)+1=1+1=2$

$T(2)=T(1)+1=2+1=3$

$T(3)=T(2)+1=3+1=4$

$T(4)=T(3)+1=4+1=5$

$T(5)=T(4)+1=5+1=6$

…

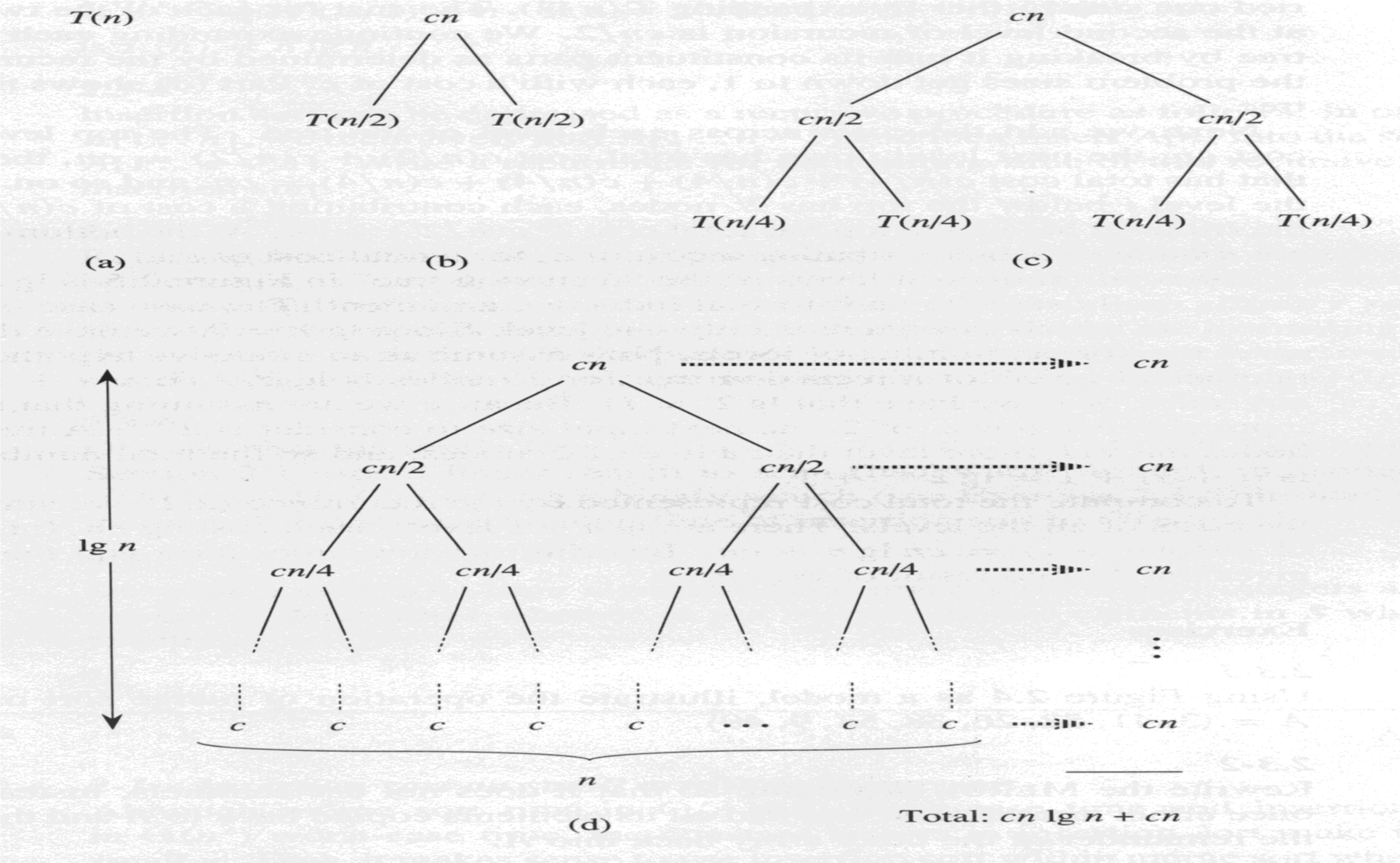
$T(n)=n+1$

Check:

$LHS = T(n) = n+1$

$RHS = T(n-1)+1 = n+1$

$$T(n) = cn\log n + cn = \Theta(n\log n)$$



Reading Assignment

- At this point, you should read Chapter 2 **section 2.3.2**, and finish reading any other parts of Chapter 2 that you haven't yet read.

- General form of divide-and-conquer algorithm recurrences

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + f(n) & \text{otherwise} \end{cases}$$

- Recursion tree method can be used to solve these kinds of recurrences
- But Master Method is more direct

No need to memorize Learn how to apply

If T(n)=aT(n/b) + f(n) (and T(base case)=some constant) and a and b are constants, then:

$$1: if \;\; f(n) = O(n^{(\log_b a)-\varepsilon}) \;\; for \;\; \varepsilon > 0 \;\; then \;\; T(n) = \Theta(n^{\log_b a})$$

$$2: if \;\; f(n) = \Theta(n^{\log_b a}) \;\; then \;\; T(n) = \Theta(n^{\log_b a} \lg n)$$

$$3: if \;\; f(n) = \Omega(n^{(\log_b a)+\varepsilon}) \;\; for \; \varepsilon > 0$$

$$and \; if \;\; af(n/b) \le cf(n) \;\; for \; some \; constant \; c < 1$$

$$then \;\; T(n) = \Theta(f(n))$$

- $T(n) = 9T(n/3) + n$

$a = 9, b = 3, f(n) = n$

$n^{\log_3 9} = n^2, \quad f(n) = O(n^{\log_3 9 - 1})$

$Case \quad 1 \Rightarrow T(n) = \Theta(n^2)$

- $T(n) = T(2n/3) + 1$

$a = 1, b = 3/2, f(n) = 1$

$n^{\log_{3/2} 1} = n^0 = 1 = f(n),$

$Case \quad 2 \Rightarrow T(n) = \Theta(\log n)$

- $T(n) = 3T(n/4) + n\log n$

$$a = 3, b = 4, f(n) = n\log n$$

$$n^{\log_4 3} = n^{0.793}, f(n) = \Omega(n^{\log_4 3+\varepsilon})$$

$Case \quad 3$

Check

$$af(n/b) = 3(\frac{n}{4})\log(\frac{n}{4}) \leq \frac{3n}{4}\log n = cf(n)$$

for $c = \frac{3}{4}$, and sufficiently large n

$\Rightarrow T(n) = \Theta(n\log n)$

- First develop the recurrences from the algorithm

- Then solve them using the most appropriate method

How do you know which method to apply?

- Recursion tree method and Master method: for Divide and Conquer algorithms that <u>divide</u> inputs by a constant <u>factor</u>

- Backward/Forward substitution method: for algorithms that <u>reduce</u> input by a constant <u>amount</u>. Recursion Tree method can also be applied.

- We have discussed several tools and techniques for mathematically determining the complexity of algorithms:
  - For non-recursive algorithms, calculate T(n) by adding up the (cost * # of executions) of each step
  - For recursive algorithms, develop the recurrence relations and solve them using a variety of techniques to obtain T(n)
  - Once you obtain an exact expression for T(n) [or through various approximations an upper or lower bound for T(n)] as a function of n, then you can determine the order of complexity of the algorithm.

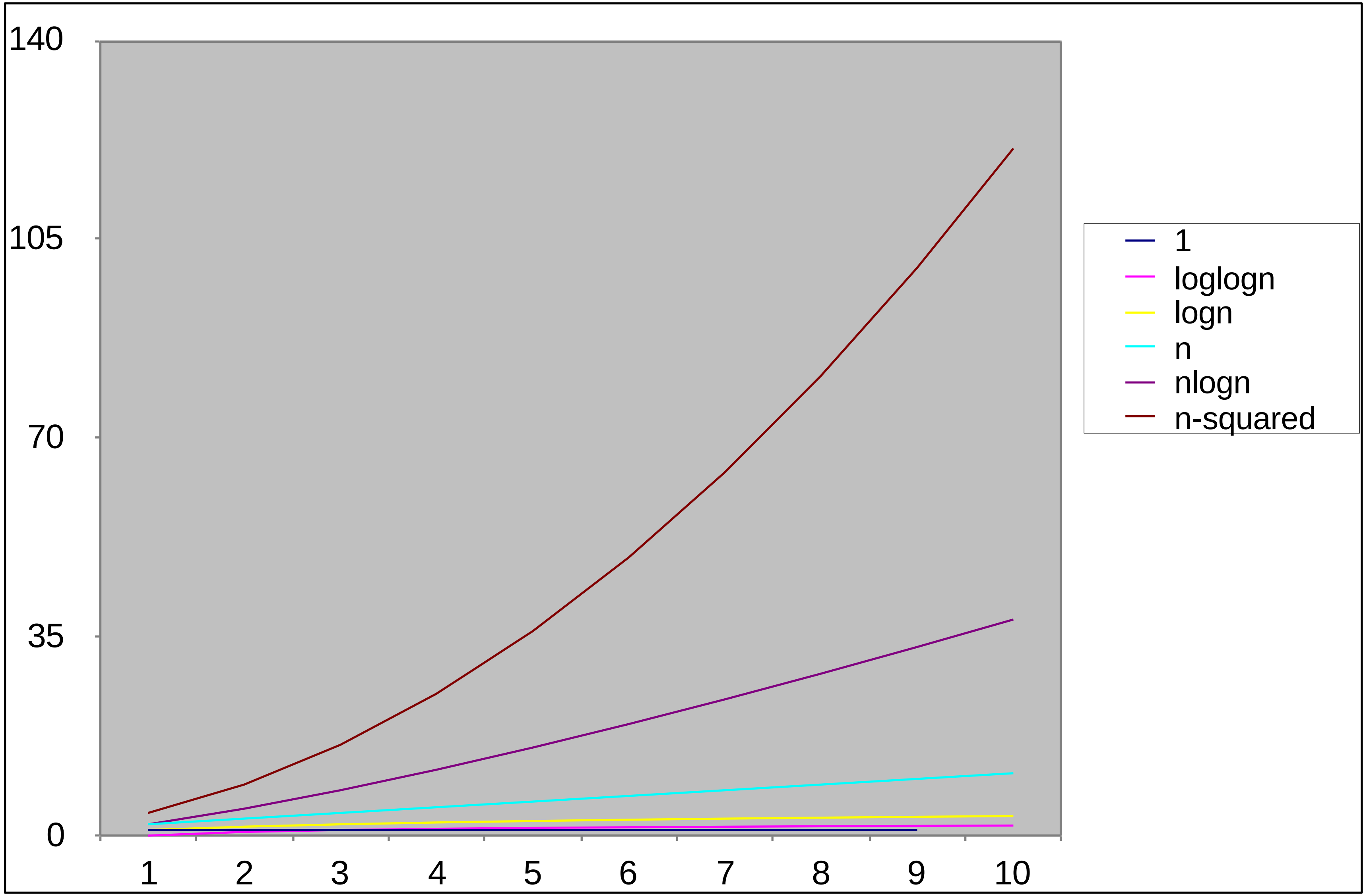
Chapter 4 Reading Assignments

- Omit Sections 4.1-4.3 & 4.6
- Read sections 4.4 & 4.5

Chapter 4 Thinking Assignments

- p. 93: 4.4-1 : 4.4-5 (ignore the “verify your answer” part)
- p. 97: 4.5-1, 4.5-3

- Another approach is to determine T(n) by plotting it as a graph of actual time taken by the algorithm versus input size by:
  - Coding the algorithm in a programming language
  - Randomly generating inputs of different sizes: typically, from small sizes up to 100K’s or millions
  - Running the program on each of these inputs and measuring the time taken using the system clock
  - Plotting time against input size
  - Determining the appropriate g(n) that fits this graph or provides an upper or lower bound (see next slide for examples of g(n))
  - This function g will then give you the order of complexity of the algorithm


140
105
70
35
0
1
2
3
4
5
6
7
8
9
10
1
loglogn
logn
n
nlogn
n-squared

AUBURN UNIVERSITY