Growth of Functions and Recurrence Relations

Chap 3, 4; Handout 3

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2019; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Asymptotic Notation

When we analyze the performance of algorithms for large input sizes we are concerned only with the rate of growth of the running time; this is determined by the dominant term of the expression for $T(n)$. For example if running time $T(n)$ is $T(n) = n^3 + 1000n^2 + 10000$, the rate of growth is cubic ; the cubic term n^3 determines the rate of growth not the asymptotically smallest quadratic or constant term. Those latter terms can be ignored. We are thus interested in how running time increases with problem size **in the limit** for large n. The **best algorithm** is the one with the smallest asymptotic growth, i.e. the (asymptotically) smallest running time.

We introduce **asymptotic notations** that we will use in the asymptotic analysis of algorithms in this course and to describe the asymptotic running time of introduced algorithms. Such notations allow us to extract information related to rate of growth from the functions that describe the running time of an algorithm of some problem size n. As the notation are asymptotic, the bounds on running time hold when problem size increases with out limit. Therefore for small problem sizes these conclusions may NOT hold.

For the previous example we will use the notation $T(n) = \Theta(n^3)$ or $T(n) \in \Theta(n^3)$ to denote that the most important term in the running time that determines the growth of function $T(n)$ is n^3, while all other terms can be ignored in the limit, for large n. This can read as "the order of growth of $T(n)$ is n^3" or "$T(n)$ is $\Theta(n^3)$. It should never be read literally however that "$T(n)$ is equal to n^3" : this is plain wrong. Note that the textbook prefers the former notation $T(n) = \Theta(n^3)$.

In describing the worst-case running time $T(n)$ of an algorithm, n is related to problem or input size and is **an integer**.
Such notation can be extended to the domain of real numbers as well. Let $\lg n$ be the base two logarithm of n.

- **Remember:** $2^{\lg n} = n$ and $a^{\lg n} = n^{\lg a}$ and $n^{1/\lg n} = 2$.
- **Notation:** $\lg^2 n = (\lg n)^2$ and $\lg^k n = (\lg n)^k$.
- **Fact:** In general $n^k > \lg^l n$ for any positive constant k, l and large enough n.
- **Fact:** $2^n > n^k$ for any positive constant k and large enough n.
- **Fact:** $n! > n^k$ for any constant k and large enough n.
- **Stirling's formula:** $n! \approx (n/e)^n \sqrt{2\pi n}$ for large enough n. Sometimes we write in asymptotic derivations $n! \approx (n/e)^n$ to derive that $\lg(n!) \approx n \lg n - n \lg e$.

$\forall$ stands for **for all**.
$\exists$ stands for **there exist(s)**.
$\in$ stands for **belong(s) to**.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Asymptotic notation: An introduction

We introduce symbols o (little oh), O (big oh), ω (little omega), Ω (big omega), and Θ (big theta, or plain theta) to denote asymptotically smaller, asymptotically at most (no greater), asymptotically larger, asymptotically at least (no smaller) and asymptotically equal respectively.

We would write $f(n) = ?(g(n))$, where ? is one of $o, O, \omega, \Omega, \theta$ to indicate the relationship between $f(n)$ and $g(n)$.
Do not forget: functions $f(n)$ and $g(n)$ are positive (or ... asymptotically positive) or non-negative and depend on natural or non-negative integer variable n. Or we can use (as some textbooks are using) $f(n) \in ?(g(n))$. The $?(g(n))$ denotes a set of functions. Therefore the relationship $f(n) = ?(g(n))$, is a set-membership relationship, where we use $=$ instead of $\in$, by **abusing** notation. Even if we abuse notation it should be borne in mind that the left hand side of $=$ is a function and the right hand side is a set of functions. If instead we use $\in$ we do not have these issues. Never use a $\leq$ or $\geq$ not even $\neq$. Thus $T(n) \leq \Theta(n)$ is NONSENSE. The $=$ of $=?$ is being used as a substitute for $\in$. Nothing else!

2.1 What does $f(n) = O(g(n))$ mean? It means that the growth of $f(n)$ is less than (slower) or equal to the growth of $g(n)$, or that $f(n)$ is asymptotically at most (no greater) than $g(n)$.

IT DOES NOT NECESSARILY MEAN that $f(n) \leq g(n)$. For example $f(n) = 10n$ and $g(n) = n$ have the same growth i.e. $10n = O(n)$, even if $10n \geq n$. It is also $n = O(10n)$.

2.2 What does $f(n) = \Omega(g(n))$ mean? It means that the growth of $f(n)$ is greater than (faster) or equal to the growth of $g(n)$, or that $f(n)$ is asymptotically at least (no smaller) than $g(n)$.

IT DOES NOT NECESSARILY MEAN that $f(n) \geq g(n)$. For example $f(n) = n$ and $g(n) = 10n$ have the same growth i.e. $n = \Omega(10n)$, even if $n \leq 10n$.

2.3 What does $f(n) = \Theta(g(n))$ mean? It means that the two functions have the same growth or $f(n)$ is asymptotically equal to $g(n)$.

Therefore $f(n) = n^2$ and $g(n) = 100n^2 + 10000n$ are such that $n^2 = \Theta(100n^2 + 10000n)$. We can also say $100n^2 + 10000n = \Theta(n^2)$. They both have the same asymptotic growth. They are asymptotically equal (i.e. they both are quadratic functions) even if one is clearly larger than the other. In an asymptotic comparison of $f(n), g(n)$ we do not care about low order terms such as $10000n$; moreover we do not care about multiplicative terms such as 100.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

2.4 What does $f(n) = o(g(n))$ mean? It means that the growth of $f(n)$ is slower than the growth of $g(n)$ or $f(n)$ is asymptotically smaller than $g(n)$.

2.5 What does $f(n) = \omega(g(n))$ mean? It means the growth of $f(n)$ is faster than the growth of $g(n)$ or $f(n)$ is asymptotically larger than $g(n)$.

When we **asymptotically compare** two functions we only care about the most important term. We do not care about low-order terms nor about multiplicative constants. Thus for $f(n) = n^2$ and $g(n) = 100n^2 + 10000n$, asymptotically comparing them means to asymptotically compare $f(n) = n^2$ and $g'(n) = 100n^2$ because $10000n$ is a low order term. Moreover it is equivalent to comparing $f(n) = n^2$ and $g''(n) = n^2$ as the 100 of $100n^2$ is a multiplicative constant term that can be ignored (i.e. use 1 instead of 100). Thus the asymptotic comparison of $f(n)$ and $g(n)$ is equivalent to that of $f(n) = n^2$ and $g''(n) = n^2$. It is obvious that the latter two functions are equal to each other. Thus $=$ for the latter functions is 'equivalent' to Θ for $f(n)$ and $g(n)$ i.e. $f(n) = \Theta(g(n))$. (It does not imply that $f(n)$ is equal to $g(n)$ just that $f(n)$ is asymptotically equal to $g(n)$.) Likewise $g(n) = \Theta(f(n))$.

Similarly a $\leq$ for $f(n)$ and $g(n)$ and their derivative functions would imply a O for $f(n)$ and $g(n)$ and not the other way around, a $\geq$ an Ω and a $<$ and $>$ would imply a o and a ω respectively.

Thus $f(n) = 100000000$ and $g(n) = n^2$ reduces to $f'(n) = 1$ and $g(n) = n^2$. It is (for positive n always) $f'(n) < g(n)$ for large enough n thus implying a $f(n) = o(g(n))$.

The 'eliminate' low-order terms and 'turn multiplicative constants into one' is a rule of thumb. A formal definition for O and Ω and Θ is given on the next page. This reminds you of the formal definition of a limit. In fact the formal definitions of o and ω on the following page use limits for large enough n i.e. as $n \to \infty$.

(c) Copyright A. Gerbessiotis.

Assumption: For this and the remainder, all functions used are asymptotically positive, and c_1, c_2, n_0 are POSITIVE and constant. In order to asymptotically compare $f(n)$ and $g(n)$ we can test inclusion of $f(n)$ into $\Theta(g(n))$, $\Omega(g(n))$, or $O(g(n))$ respectively. Or the other way around. The **formal definitions** for Θ, Ω, O are provided.

2.6 Θ **Definition.** For a function $g(n)$ we denote by $\Theta(g(n))$ the **set of all functions** $f(n)$ that have the following property.

$$\exists \text{ positive constants } \ c_1, c_2, n_0 : \ \ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \ \ \forall\ n \geq n_0.$$

We denote the asymptotic relationship between $f(n)$ and $g(n)$ by writing $f(n) \in \Theta(g(n))$ or the more common,

$$f(n) = \Theta(g(n)).$$

2.7 Ω **Definition.** For a function $g(n)$ we denote by $\Omega(g(n))$ the **set of all functions** $f(n)$ that have the following property.

$$\exists \text{ positive constants } \ c_1, n_0 : \ \ 0 \leq c_1 g(n) \leq f(n) \ \ \forall\ n \geq n_0.$$

We denote the asymptotic relationship between $f(n)$ and $g(n)$ by writing $f(n) \in \Omega(g(n))$ or the more common,

$$f(n) = \Omega(g(n)).$$

2.8 O **Definition.** For a function $g(n)$ we denote by $O(g(n))$ the **set of all functions** $f(n)$ that have the following property.

$$\exists \text{ positive constants } \ c_2, n_0 : \ \ 0 \leq f(n) \leq c_2 g(n) \ \ \forall\ n \geq n_0.$$

We denote the asymptotic relationship between $f(n)$ and $g(n)$ by writing $f(n) \in O(g(n))$ or the more common,

$$f(n) = O(g(n)).$$

We can also say respectively that $g(n)$ is an **asymptotic tight bound**, is an **asymptotic lower bound**, and is an **asymptotic upper bound** of $f(n)$. Or we can say that $f(n)$ is asymptotically equal, asymptotically at least (no smaller) and asymptotically at most (no greater) than $g(n)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Formal asymptotic comparison and definitions for o, ω. We present now the formal definitions for o and ω. They also imply an asymptotic comparison between $f(n)$ and $g(n)$ respectively.

2.9 o Definition. For two functions $f(n)$ and $g(n)$ we write $f(n) = o(g(n))$, if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

We then say that $f(n)$ is asymptotically smaller than $g(n)$ or the growth of $f(n)$ is smaller (slower) than the growth of $g(n)$.

2.10 ω Definition. For two functions $f(n)$ and $g(n)$ we write $f(n) = \omega(g(n))$, if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

We then say that $f(n)$ is asymptotically larger than $g(n)$ or the growth of $f(n)$ is greater (faster) than the growth of $g(n)$.
An alternative formulation of 2.6 for Θ. The limit in the two cases above is a 0 or ∞. A third case can be derived when the limit is a constant other than zero.

2.6′ Θ Definition. For two functions $f(n)$ and $g(n)$ we write $f(n) = \Theta(g(n))$, if and only if, for some constant $c > 0$,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c.$$

We then say that $f(n)$ is asymptotically equal to $g(n)$ or the growth of $f(n)$ is equal to the growth of $g(n)$.

2.6′, 2.9, 2.10 for $\mathbf{\Theta}, \mathbf{o}, \omega$. Thus in order to asymptotically compare $f(n)$ and $g(n)$ with o, ω, Θ we first find the limit $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ and then determine which case is applicable.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \Rightarrow f(n) = o(g(n)) \\ \infty & \Rightarrow f(n) = \omega(g(n)) \\ c \neq 0 & \Rightarrow f(n) = \Theta(g(n)) \end{cases}$$

The following results can then be derived.

2.11.1 Lemma 1 ($o \Rightarrow O$ implication). If $f(n) = o(g(n))$, then $f(n) = O(g(n))$.

2.11.2 Lemma 2 ($\omega \Rightarrow \Omega$ implication). If $f(n) = \omega(g(n))$, then $f(n) = \Omega(g(n))$.

2.11.3 Lemma 3 ($\Theta \Leftrightarrow \Omega \bigwedge O$ equivalence). $f(n) = \Theta(g(n))$, if and only if $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$.

Other results can also be derived similar to Lemma 1 and Lemma 2.

2.11.4 Lemma 4. $f(n) = o(g(n))$ if and only if $f(n) = O(g(n))$, and $f(n) \notin \Omega(g(n))$. (There is no way we can write $f(n) \neq \Omega(g(n))$ without violating what we have clearly stated on page 3 before 2.1!)

2.11.5 Lemma 5. $f(n) = \omega(g(n))$ if and only if $f(n) = \Omega(g(n))$, and $f(n) \notin O(g(n))$.

The following two results establish an `antisymmetry property`.

2.11.6 Lemma 6. $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

2.11.7 Lemma 7. $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

Lemma 8 is an alternative formulation of Lemma 3 using Lemma 7.

2.11.8 Lemma 8. $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Lemma 9 and Lemma 10 are complementary formulations to $2.6', 2.7, 2.8$.

2.11.9 Lemma 9. Let $\lim_{n\to\infty} f(n)/g(n)$ exists. $f(n) = O(g(n))$ if and only if $\lim_{n\to\infty} f(n)/g(n) \leq c$ for positive constant c.

2.11.10 Lemma 10. Let $\lim_{n\to\infty} g(n)/f(n)$ exists. $f(n) = \Omega(g(n))$ if and only if $\lim_{n\to\infty} g(n)/f(n) \leq c$ for positive constant c.

2.12.1 Remark 1. As $\Theta(g(n))$ is a set, the appropriate notation is $f(n) \in \Theta(g(n))$. We "abuse it" by writing $f(n) = \Theta(g(n))$ instead.

2.12.2 Remark 2. Note that $\Theta(g(n)) = f(n)$ is meaningless and wrong! The notation $f(n) \geq \Theta(g(n))$ is also meaningless and wrong! (Third time mentioned!)

2.12.3 Remark 3. If we write $\Theta(f(n)) = \Theta(g(n))$, this is to mean that **no matter how we choose a function on the left hand side, there is a way to choose a function on the right hand side to make equality to hold**.

We would write something like that when we simplify the growth of a function. For example $\Theta(n^3 + 100n^2 + 1000) = \Theta(n^3 + 100n^2) = \Theta(n^3)$.

2.12.4 Remark 4. Normally, the phrase **"the running time is $O(n^2)$"** is meaningless as the running time is expressed by a number and a unit of time (e.g. 6 seconds, 10 milliseconds). *This expression when used this way in this class means* that the worst-case running time (which is a function of n) is $\Theta(n^2)$, and by extension, no matter what a particular input of size n is chosen for each value of n, the running time on that set of input instances is $\Theta(n^2)$ or smaller, i.e. no worse than the worst-case running time.

2.12.5 Remark 5. A running time of $\Omega(g(n))$ for an algorithm means that no matter what the input of size n is, for each value of n, the running time of the algorithm will be at least $cg(n)$, for some constant c for large enough n. For example the **running time of insertion-sort** is $\Omega(n)$. Yet the **running time of bubble-sort** is $\Omega(n^2)$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

2.13.1 Example 1. Show that $n^2 - 2n = \Theta(n^2)$.
Proof of 2.13.1. We determine POSITIVE CONSTANTS c_1, c_2, n_0 so that for all $n \geq n_0$.

$$c_1 n^2 \leq n^2 - 2n \leq c_2 n^2$$

As $n^2 - 2n \leq n^2$ for all $n \geq 1$, we have shown that $n^2 - 2n \leq c_2\, n^2$, for $c_2 = 1$ and for all $n \geq n_1 = 1$. This is equivalent to showing that $n^2 - 2n = O(n^2)$ as well.
As $n^2/2 \leq n^2 - 2n$ for all $n \geq 4$, we have shown that $c_1 n^2 \leq n^2 - 2n$, for $c_1 = 1/2$ and $n \geq n_2 = 4$. This is equivalent to showing that $n^2 - 2n = \Omega(n^2)$ as well.

The n_0 in the definition is the largest of n_1 and n_2 i.e. $n_0 = 4$. Then, for $c_1 = 1/2$, $c_2 = 1$ and for all $n \geq n_0 = 4$, we have

$$c_1 n^2 \leq n^2 - 2n \leq c_2 n^2$$

Note: $c_1 = 1/2$, $c_2 = 1$ and $n_0 = 1000$ is also a correct answer to this problem. OUR OBJECTIVE IS TO FIND A SET OF SATISFYING CONSTANTS not THE BEST SET OF SATISFYING CONSTANTS.

2.13.2 Example 2. Since a constant is a degree 0 polynomial any constant c is such that $c = \Theta(1)$, $c = O(1)$, and $c = \Omega(1)$.

2.13.3 Example 3. Show that $1000 = O(1)$.
Proof of 2.13.3. There exists constant $c = 2000$ such that $1000 \leq c \cdot 1 = 2000 \cdot 1 = 2000$, for all $n \geq 1 = n_0$. $c = 1000$ works also. Be reminded: We don't need to find the best c or n_0.

2.13.4 Example 4. $n = \Omega(1)$, $2^{3 \lg n} = \Omega(n^2)$, $1000000 = \Omega(1)$, $n^3 = \Omega(n^2)$, $n = \Omega(n/\lg n)$, $n! = \Omega(n^{1000000})$, $1 = \Omega(1000000)$.

2.13.5 Example 5. Show that $2^n = \Omega(n)$.
Proof 2.13.5. Since $\lim 2^n/n = \infty$ as $n \to \infty$, we have $2^n = \omega(n)$. By Lemma 2 we have $2^n = \Omega(n)$ as well.

2.13.6 Example 6. Show that $n = O(n^3)$.
Proof 2.13.6. Since $\lim n/n^3 = 0$ as $n \to \infty$, we have $n = o(n^3)$. By Lemma 1 we have $n = O(n^3)$ as well.

2.13.7 Example 7. Show that $n^3 = \omega(n)$.
Proof 2.13.7. Since $\lim n^3/n = \infty$ as $n \to \infty$, we have $n^3 = \omega(n)$. The result can also be proved as a consequence of $n = o(n^3)$.

2.13.8 Example 8. Show that $n^{2/\lg n} = o(n)$.
Proof 2.13.8.

$$\frac{n^{2/\lg n}}{n} = \frac{(2^{\lg n})^{2/\lg n}}{n} = \frac{2^2}{n} = \frac{4}{n} \to 0$$

Thus $n^{2/\lg n} = o(n)$.

2.13.9 Example 9. Show that $n! = \omega(n)$.
Proof 2.13.9. From Stirling approximation formula, $n! \approx (n/e)^n$. Therefore, $\frac{n!}{n} \to \infty$. Thus $n! = \omega(n)$.

2.13.10 Example 10. Show that $n^3 = \Omega(2^{\lg n})$.
Proof 2.13.10. We have that $2^{\lg n} = n$. By an earlier example, we have $n^3 = \omega(n)$, and thus $n^3 = \Omega(n)$.

2.13.11 Example 11. Is $2^n = 2^{\Theta(n)}$? Be careful. The Θ is not on the right of the equal size. You are looking for trouble (or not)! (We sometimes write something similar to this because it is convenient.)

2.13.12 Example 12. Let me rephrase it: Is $2^n = \Theta(2^{\Theta(n)})$? A function $g(n)$ that is $\Theta(n)$ for the second Θ is $g(n) = 2n$. So let me rephrase the question: Is $2^n = \Theta(2^{2n})$? Well $2^n/2^{2n} \to 0$.

Note and Reminder: All limits are for $n \to \infty$ unless explicitly stated otherwise.

(c) Copyright A. Gerbessiotis.

We are coming back to discuss the running time of an algorithm now that we have introduced formally asymptotic notation.

2.14.1 Running time of an algorithm on a given input instance. The running time of an algorithm is expressed in terms of its problem size n and we usually use function $T(n)$ to describe it. The running time of InsertionSort on a sorted sequence (of n keys) is $T(n) = \Theta(n)$. When we talk about a **specific input** we can used it as a subscript and thus a more proper way to write so is to say $T_{\mathbf{Sorted}}(n) = \Theta(n)$. This is because by 1.22.3 the number of comparisons performed is $n-1$ for a sorted sequence of n keys. THe running time is proportional to the number of comparisons. Using asymptotic notation, the former or the latter is $\Theta(n)$. Thus $T_{\mathbf{Sorted}}(n) = \Theta(n)$. Likewise the running time of InsertionSort on a reverse-sorted sequence of n keys is $T_{\mathbf{Rev-sorted}}(n) = \Theta(n^2)$. This is because the number of comparisons performed for such an input is $n(n-1)/2$.

2.14.2 Running time of an algorithm. The running time of an algorithm is an expression that describes its performance on ANY and EVERY input. Thus we CANNOT SAY that the running time of InsertionSort is $T(n) = \Theta(n^2)$ because we know that for some inputs the running time would be linear (eg. sorted keys or almost sorted) i.e. **not** $\Theta(n^2)$ but "less than" $\Theta(n^2)$ or equivalently $o(n^2)$. Likewise we CANNOT SAY that the running time of InsertionSort is $T(n) = \Theta(n)$ because for a reverse-sorted sequence the running time would be "much more than $\Theta(n)$ i.e. $\omega(n)$ and in fact it would be quadratic i.e. $\Theta(n^2)$. Thus to describe the **running time of InsertionSort** we write $T(n) = O(n^2)$. This is because for some input the running time would be quadratic and in fact $Theta(n^2)$ but also for some other inputs it would be substantially less than quadratic (eg linear for a sorted sequence). By using $O(n^2)$ we are correct for ANY and EVERY INPUT INSTANCE, but not as accurate for the best case when performance is $\Theta(n)$ nor the worst-case when performance is $\Theta(n^2)$. That loss of accuracy is the penalty we pay to be able to say something for ALL INPUT INSTANCES. Likewise the running time of binary search (if key exists) on a sorted sequence of n keys is $T(n) = O(\lg n)$. This is because if we are lucky after one comparison we can find the key in $\Theta(1)$ time, otherwise we might be unlucky and would indeed need $\Theta(\lg n)$ time. Thus the use of $O(\lg n)$ covers all possibilities!

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

For a recursive algorithm like mergesort running time is described by a recurrence like the one shown below.

$$T(n) = T(n/2) + T(n/2) + \Theta(n) = 2T(n/2) + \Theta(n).$$

In order to solve this recurrence we need a boundary condition for the base case of the recurrence. The base case for the running time of merge-sort is the running time for an input of size 1 or in general, of some small constant size. In this case, we may assume that $T(1) = 1$, or $T(1) = \Theta(1)$ i.e. the sorting of a one-key sequence takes one step or constant time. The solution for this recurrence was claimed to be $T(n) = \Theta(n \lg n)$ (proof by induction or by using a recursion tree).

2.15.0.1 Remark 1. In the recurrence above, we ignored floors and ceilings. When we solve recurrences, we first remove floors/ceilings, show the claimed bounds and then check whether the existence of floors/ceilings would or could have made a difference. For the remainder of this course we assume that values like $n/2$ are always integer, and so the effect of the removal of floors/ceilings will not be discussed.

2.15.0.2 Remark 2. For boundary conditions we assume in general that $T(k) = \Theta(1)$, for some small constant k, as constant problem sizes can be solved in constant time. We use $T(k) = \Theta(1)$ instead of a more specific/explicit $T(k) = l$ (l some constant), as such a choice affects only constants in the expression for $T(n)$; i.e. growth rate is preserved. In general, **we shall ignore statements about boundary conditions**, because this simplifies the recurrence solution. If a boundary condition is explicitly given, the solution must be consistent with it.

We introduce **three methods** to solve recurrences.

- **Master** method: Provides a solution formula for recurrences of certain form such as $T(n) = aT(n/b) + f(n)$, for constant $a \geq 1$ and $b > 1$ and asymptotically positive $f(n)$.
- **RecursionTree/Iteration** method: Unfold recurrence by turning it into a sum.
- **Substitution or "guess and check"** method: Guess solution and then verify/check it (proof by induction).

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

The **master method** provides an easy solution method for a certain class of recurrences that are of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$ and $b > 1$ are constant and $f(n)$ is an asymptotically positive function.

2.15 Theorem. Let $T(n) = aT(n/b) + f(n)$, where $a \geq 1$ and $b > 1$ are constant and $f(n)$ is an asymptotically positive function. Then $T(n)$ is bounded as follows.

Case 1 If $f(n) = O(n^{\lg_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\lg_b a})$.

Case 2 If $f(n) = \Theta(n^{\lg_b a})$, then $T(n) = \Theta(n^{\lg_b a} \lg n)$.

Case 3 If $f(n) = \Omega(n^{\lg_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and for large n, then $T(n) = \Theta(f(n))$.

2.15.1 Note 1. Between any two cases there are gaps, i.e. there are cases not covered by this theorem. What happens if ϵ is not constant? For example consider $T(n) = 2T(n/2) + n \lg n$, $a = 2$, $b = 2$, $n^{\lg_b a} = n$ and $f(n) = n \lg n$. It falls between cases 2 and 3. **2.15.2 Note 2.** There is an alternative formulation of Case 2 (that will not be used)

Case 2′ If $f(n) = \Theta(n^{\lg_b a} \lg^k n)$, then $T(n) = \Theta(n^{\lg_b a} \lg^{k+1} n)$, where $k \geq 0$ is a non-negative constant. **2.15.3 Note 3.**

Sometimes the condition $a \geq 1$ is stated as $a > 0$.

2.16.1 Example 1. Note that $a^{\lg n / \lg b} = 2^{\lg a \lg n / \lg b} = n^{\lg a / \lg b}$. Consider $T(n) = 9T(n/3) + f(n)$. Then $a = 9$, $b = 3$ and $\lg_b a = 2$ i.e. $n^{\lg_b a} = \Theta(n^2)$. If $f(n) = n$, then $T(n) = \Theta(n^2)$. If however $f(n) = n^2$, then $T(n) = \Theta(n^2 \lg n)$.

2.16.2 Example 2. $T(n) = T(n/2) + f(n)$, where $a = 1$ and $b = 2$, and $n^{\lg_b a} = n^0 = 1$. If $f(n) = 1$, then $T(n) = \Theta(\lg n)$.
If $f(n) = \sqrt{n}$, then because $af(n/b) = a\sqrt{n/b} = 1\sqrt{n}/\sqrt{2} \leq \sqrt{1/2}\sqrt{n} = cf(n)$ for $c = \sqrt{1/2}$, we get that $T(n) = \Theta(\sqrt{n})$.

(c) Copyright A. Gerbessiotis.

2.17 Iteration Method / Recursion Tree Method. If the method is done graphically it is sometimes called the **recursion tree method** (see textbook, or the class proof of the running time of merge-sort). The traditional name is **iteration method**.

How. Expand the recurrence all the way down to the base case and sum up the residuals. In order to avoid problems with floors and ceilings and the use of identities such as that for all integers n, a, b, we have $\lfloor \lfloor n/a \rfloor /b \rfloor = \lfloor n/(ab) \rfloor$, we are going to make an assumption that n is a power of 2 and thus $n/2$, $n/4$, $n/8$ etc are all integer.

2.17.1 Solve $T(n) = 2T(n/2) + n, \qquad T(1) = 0.$

In order to compute $T(n)$ we need to establish $T(n/2)$. Likewise in order to compute $T(n/2)$ we need to establish $T(n/4)$. We iterate i iterations until $T(n/2^i)$ with the intent of making $n/2^i$ equal to the base case value 1. If $n/2^i = 1$ then we know that T(1)=0. For $n/2^i = 1$ we have $n = 2^i$ and solving for i we get $i = \lg n$.

$$
\begin{aligned}
T(n) &= 2T(n/2) + n = 2\overbrace{\left(2T(n/2^2) + n/2\right)}^{T(n/2)} + n \\
&= 2^2 T(n/2^2) + 2n = 2^2\overbrace{\left(2T(n/2^3) + n/2^2\right)}^{T(n/2^2)} + 2n \\
&= 2^3 T(n/2^3) + 3n \\
&\cdots \\
&= 2^i T(n/2^i) + i \cdot n \quad \textbf{Now, Substitute} \quad i = \lg n \\
&= 2^{\lg n} \cdot T(n/2^{\lg n}) + \lg n \cdot n = n \cdot T(1) + \lg n \cdot n = n \lg n
\end{aligned}
$$

- Keep track of number of iterations/depth of recursion tree.
- Keep track of sum of terms per iteration/level of recursion tree.
- Sometimes, the two previous steps and experience allow us to guess the solution correctly. We can then stop the solution with this method and switch to the substitution method instead.

2.17.2 Practice Exercise. Solve exactly $T(n) = 2T(n/2) + n$, $T(2) = 1$. Assume n is a power of 2.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

The previous one was straightforward and uncomplicated. Let's go back to the merge-sort recurrence we solved in class through the recursion tree method. (The one whose last term is $n-1$ not n referring to the maximum number of comparisons performed to merge two sorted sequences each of size $n/2$.)

2.17.3 Solve $T(n) = 2T(n/2) + n - 1, \qquad T(1) = 0.$

Again we shall terminate recursion for $T(n/2^i)$ when $n/2^i = 1$ and solving for i we shall get $i = \lg n$. We shall also need the geometric sequence result $2^0 + 2^1 + \ldots + 2^{i-1} = 2^i - 1$.

$$\begin{aligned}
T(n) &= 2T(n/2) + n - 1 \\
&= 2\left(2T(n/2^2) + n/2 - 1\right) + n - 1 = 2^2T(n/2^2) + 2n - 1 - 2 \\
&= 2^2\left(2T(n/2^3) + n/2^2 - 1\right) + 2n - 1 - 2 = 2^3T(n/2^3) + 3n - 2^0 - 2^1 - 2^2 \\
&\ldots \\
&= 2^iT(n/2^i) + i \cdot n - 2^0 - 2^1 - \ldots - 2^{i-1} = 2^iT(n/2^i) + i \cdot n - (2^i - 1) \\
&= 2^iT(n/2^i) + i \cdot n - (2^i - 1) \quad \textbf{Now, Substitute} \quad i = \lg n \\
&= 2^{\lg n} \cdot T(n/2^{\lg n}) + \lg n \cdot n - (2^{\lg n} - 1) \\
&= n \cdot T(1) + \lg n \cdot n - n + 1 = n \lg n - (n - 1)
\end{aligned}$$

The previous page recurrence was generating an additive term which was the same in each iteration: n. This one generates a term that is iteration dependent: $n - 2^{i-1}$ for iteration i. Thus for $i = 1$ we get $n - 1$, for $i = 2$ we get $n - 2$, and so on.

The answer for $T(n)$ with the $(-(n-1)$ low-order term) is the one we got through the recursion tree method as well.

(c) Copyright A. Gerbessiotis.

2.17.4 Solve exactly the recurrence $T(n) = 2T(n/2) + n$, $T(2) = 5$. Assume n is a power of 2.

Substituting $n/2$ for n in the recurrence we get. $T(n/2) = 2T((n/2)/2) + n/2 = 2T(n/2^2) + n/2$.
Substituting $n/4 = n/2^2$ for n we get. $T(n/2^2) = 2T((n/4)/2) + n/4 = 2T(n/2^3) + n/2^2$
Substituting $n/8 = n/2^3$ for n we get. $T(n/2^3) = 2T(n/2^4) + n/2^3$

Similarly we can substitute $n/2^4, \ldots, n/2^{i-1}, \ldots$ for n to resolve the 4, ..., $i-1$-st iteration all the way to the base case, and get similarly stated recurrences. We utilize all the derived recurrences to expand $T(n)$ by observing that $T(n)$ is $2T(n/2) + n$. Then we use the derivation for $T(n/2)$ to formulate the expression in terms of $T(n/2^2)$, then $T(n/2^2)$ is expressed in terms of $T(n/2^3)$ and so on ... in terms of $T(n/2^i)$. This goes on until we reach the based case of $n = 2$ since $T(2) = 5$ is given. We establish what value of i provides the base case by equating $T(2) = T(n/2^i)$ and solving for i. The unfolding of the recurrence stops then and all the terms unfolded get combined (summed). A closed form solution for $T(n)$ can then be found.

$$\begin{aligned} T(n) &= 2T(n/2) + n \;=\; 2(2T(n/2^2) + n/2) + n \;=\; 2^2T(n/2^2) + 2n \\ &= 2^2(2T(n/2^3) + n/2^2) + 2n \;=\; 2^3T(n/2^3) + n + 2n \;=\; 2^3T(n/2^3) + 3n \\ &= \ldots = 2^iT(n/2^i) + i \cdot n \end{aligned}$$

From the boundary condition $T(2) = 5$, we decide when to stop the unfolding of $T(n)$. The expansion ends at $n/2^i = 2$ since then $T(n/2^i) = T(2) = 5$. We solve for i by taking logarithms base two of both sides. Then we get that $\lg n - i = 1$ ie $i = \lg n - 1$. Then, for $i = \lg n - 1$, we get that.

$$\begin{aligned} T(n) &= 2T(n/2) + n = \ldots = 2^iT(n/2^i) + i \cdot n \\ &= 2^{\lg n-1}T(n/2^{\lg n-1}) + (\lg n - 1) \cdot n \\ &= (n/2)T(2) + (\lg n - 1) \cdot n = (n/2)5 + (\lg n - 1) \cdot n \\ &= (3n/2) + n \lg n \end{aligned}$$

Thus we have obtained the following solution to the recurrence: $T(n) = 3n/2 + n \lg n$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

2.17.5 Question 1. Is the obtained solution the correct one?

Answer: Yes, unless we missed something.

2.17.6 Question 2. How can we be sure?

Answer: Check the solution i.e. make sure that $T(n) = 3n/2 + n \lg n$ is such that
(a) $T(2) = 5$ (i.e. boundary condition can be verified), and
(b) $T(n) = 2T(n/2) + n$ (i.e. recurrence can be verified).

(2.17.6) Verify boundary condition.

$$T(2) = 3 \cdot 2/2 + 2 \lg 2 = 3 + 2 = 5.$$

Since $T(2) = 5$ indeed by the boundary condition, our solution satisfies the boundary condition.

(2.17.6) Verify recurrence. We obtained the solution

$$T(n) = 3n/2 + n \lg n.$$

Substituting $n/2$ for n we get that

$$T(n/2) = 3n/4 + (n/2) \lg n/2 = 3n/4 + (n/2)(\lg n - 1) = n/4 + (n/2) \lg n.$$

We start from the right hand side of the recurrence using the preceding equality.

$$2T(n/2) + n = 2(n/4 + n/2 \lg n) + n = 3n/2 + n \lg n = T(n)$$

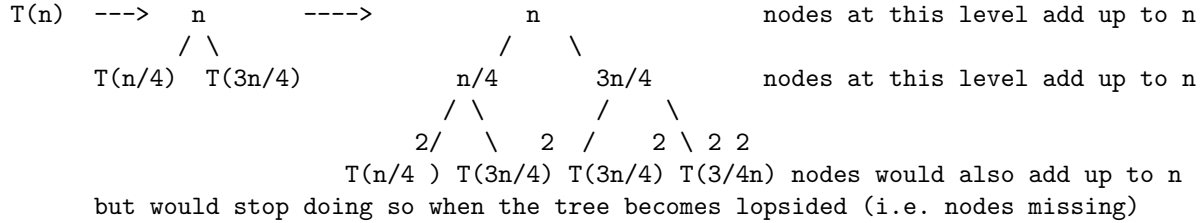
The last term is $T(n)$. We have thus proved that for $T(n) = 3n/2 + n \lg n$, we have that $T(n) = 2T(n/2) + n$, i.e. the recurrence is satisfied.

By (a) and (b) the solution satisfies both the recurrence and the boundary condition, i.e. it is indeed a solution to the recurrence.

Well, that was not the final example... There is one more in the appendix!

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

2.18 Recursion Tree Method. One example using that method involved the MergeSort recurrence solution in Subject 1. If the recurrence is complex then the iteration method will not work (easily). Moreover floors and ceilings will complicate things. Suppose $T(n) = T(n/4) + T(3n/4) + n$. Try to use a recursion tree. Start with a tree with one node labeled $T(n)$. Expand the $T(n)$ node by replacing it with a new node that is labeled with the non-recursive part of the recurrence i.e. n. Make that node have two children, the $T(n/4)$ and $T(3n/4)$ recursive parts of the recurrence. (If one had $2T(n/2)$ instead write it as $T(n/2) + T(n/2)$. Then expand similarly (using the recurrence) the two nodes labeled $T(n/4)$ and $T(3n/4)$. Expansion is shown. The recursion tree can be lopsided (Fibonacci case) or not. In this case it does not show after two iterations but it will be lopsided. The left-most path goes n, $n/4$, $n/4^2$ and so on. It reaches some constant value, say 1, after $n/4^i = 1$ i.e. $\lg(n)/2 = \log_4 n$ iterations and it is $\lg n/2$ levels deep (or hight). The righmost path it goes n, $3n/4$, $(3/4)^2 n$, and it reaches some constant value 1 after $(3/4)^i n = 1$ which solves for $i = \log_{4/3} n = \lg n / \lg(4/3)$. (Note that parentheses are dropped and thus the last expression should be read $\lg(n)/\lg(4/3)$.) The right path is longer/deeper/higher by maybe a factor of 4 or more. After i iterations the tree's height is i. The tree is lopsided (the Fibonacci sequence tree for the recursive solution was also lopsided). Deepest subtree is the rightmost one of depth $\lg n / \lg(4/3)$. Total time is thus $O(n \lg n)$. A Θ becomes too cumbersome to derive.



Note. The $T(n/4^2)$ and $T(3n/4^2)$ that appear in the third level of the tree in the figure above look awkward. And even more the $T((3^2/4^2)n)$ last term shown.

2.19 Substitution Method. It assumes that we can generate a good enough guess of the solution either by trial and error or by experience. Verification that the guess is indeed the solution takes place usually bu (strong) induction.

2.19.1 Solve the recurrence $T(n) = 2T(n/2)+n$ using the substitution (a.k.a. guess-and-check) method. (Implicit assumption is that $T(n)$ is nonnegative and defined for all positive n, or for arbitrarily large n).

Proof. By (strong) induction after we guess the answer. Since no boundary condition is given we can thus choose k and l constants greater than zero so that $T(k) = l$. We choose k, l in such a way to make the inductive proof as simple as possible. Let us choose $T(1) = 0$.

A. Guess Step: $T(n) = O(n \lg n)$. We guess $T(n) = O(n \lg n)$, i.e. $\exists$ pos. constant $c_2, n_0 : T(n) \leq c_2 n \lg n$, $\forall n \geq n_0$.

A.0 Check Step (Strong Induction). We shall prove our claim by using induction. Our $P(n)$, the proposition to be proven true would be $T(n) \leq c_2 n \lg n$ for arbitrarily large values of $n \geq n_0$, for some positive constant n_0.

A.1. Base Case of Induction. We show $P(1)$ is true is $T(1) \leq c_2 \cdot 1 \cdot \lg 1$. Since $T(1) = 0$ it is trivially true that $0 = T(1) \leq c_2 \cdot 1 \cdot \lg 1 = 0$ for all choices of a positive and constant c_2. Base case completed.

A.2. Inductive Step. $P(1) \wedge \ldots \wedge P(n/2) \wedge \ldots \wedge P(n-1) \Rightarrow P(n)$. We show that if $P(i)$ is true for all $i < n$ then we will show that $P(n)$ is also true. $P(n)$ is $T(n) \leq c_2 \cdot n \cdot \lg n$. If $P(i)$ is true for all $i < n$, since $n/2 < n$ then $P(n/2)$ is also true. Then by the inductive assumption for $i = n/2$ we have $T(n/2) \leq c_2(n/2) \lg (n/2)$. We then use the recurrence and utilize the inductive hypothesis for $T(n/2)$

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\overset{P(n/2):\ T(n/2) \leq c_2(n/2)\lg(n/2)}{=} 2 \overbrace{T(n/2)}^{} + n \\ &\leq 2c_2(n/2)\lg(n/2) + n = c_2 n \lg n - c_2 n + n \end{aligned}$$

To complete the inductive step we must show that the rightmost expression $c_2 n \lg n - c_2 n + n$ is at most $c_2 n \lg n$. For that to be true i.e $c_2 n \lg n - c_2 n + n \leq c_2 n \lg n$ we need $-c_2 n + n < 0$ i.e. $c_2 \geq 1$.

Thus if we choose a $c_2 = 1$ which is ≥ 1 and for the n_0 of the base case i.e. $n_0 = 1$ we have that

$$T(n) \leq n \lg n \quad \forall n \geq 1.$$

(c) Copyright A. Gerbessiotis.

Can we do better in the guessing game? Let us try a lower bound.

B. Guess Step: $T(n) = \Omega(n \lg n)$**.** We guess $T(n) = \Omega(n \lg n)$, i.e. $\exists$ pos. constant $c_1, n_0 : T(n) \geq c_1 n \lg n$, $\forall n \geq n_0$.

B.0 Check Step (Strong Induction). We shall prove our claim by using induction. The $P(n)$ is $T(n) \geq c_1 n \lg n$ for arbitrarily large values of $n \geq n_0$, for some positive constant n_0.

B.1. Base Case of Induction. We show $P(1)$ is true: $T(1) \geq c_1 \cdot 1 \cdot \lg 1$. Since $T(1) = 0$, it is true that $0 = T(1) \geq c_1 \cdot 1 \cdot \lg 1 = 0$ for all choices of a positive and constant c_1. Base case completed.

B.2. Inductive Step. $P(1) \wedge \ldots \wedge P(n/2) \wedge \ldots \wedge P(n-1) \Rightarrow P(n)$. We show that if $P(i)$ is true for all $i < n$ then we will show that $P(n)$ is also true. $P(n)$ is $T(n) \geq c_1 \cdot n \cdot \lg n$. If $P(i)$ is true for all $i < n$, since $n/2 < n$ then $P(n/2)$ is also true. Then by the inductive assumption for $i = n/2$ we have $T(n/2) \geq c_1(n/2) \lg (n/2)$. We then use the recurrence and utilize the inductive hypothesis for $T(n/2)$.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\qquad {\scriptstyle P(n/2):\ T(n/2) \geq c_1(n/2)\lg(n/2)} \\ &= 2 \qquad \overbrace{T(n/2)} \qquad +n \\ &\geq 2c_1(n/2)\lg(n/2) + n = c_1 n \lg n - c_1 n + n \end{aligned}$$

To complete the inductive step we must show that the rightmost expression $c_1 n \lg n - c_1 n + n$ is at least $c_1 n \lg n$. For that to be true i.e $c_1 n \lg n - c_1 n + n \geq c_1 n \lg n$ we need $-c_1 n + n > 0$ i.e. $c_1 \leq 1$.

Thus if we choose a $c_1 = 1$ which is ≤ 1 and for the n_0 of the base case i.e. $n_0 = 1$ we have that

$$T(n) \geq n \lg n \quad \forall n \geq 1.$$

Well we have just proven that $T(n) \geq n \lg n$. We also proved (previous page) that $T(n) \leq n \lg n$. Both of them prove that $T(n) = n \lg n$! (The last mark is an exclamation mark, not a factorial!)

Thus while our objective was to prove a tight asymptotic bound the derivation of the three constants n_0, c_1, c_2 allowed us to determine an exact bound.

Guessing completely an answer?

C. Guess Step : $T(n) = An\lg n + Bn + C$. Let us guess an answer that contains a log-linear, a linear and a constant term. Let's try to compute the constant values A, B and C. Only condition is that A should be positive.

C.1. Base Case utilization. We have $T(1) = 0$ substituting 1 for n in $T(n) = An\lg n + Bn + C$ we get $0 = T(1) = A \cdot 1 \cdot \lg 1 + B \cdot 1 + C$ i.e. $B + C = 0$.

C.2. Recurrence utilization. Since $T(n) = 2T(n/2) + n$ we have that

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ An\lg n + Bn + C &= 2(An/2\lg(n/2 + Bn/2 + C) + n \\ An\lg n + Bn + C &= An\lg n + (B - A + 1)n + 2C \end{aligned}$$

In the last equality the first term of both sides cancels out. We then equate the coefficients of n and the constant terms and solve for A, B, C also utilizing the base case $B + C = 0$.

The constant terms C and $2C$ should be equal to each other for $C = 0$. Since $B + C = 0$ we also have $B = 0$.

The linear term coefficients B and $(B - A + 1)$ should be equal to each other $B = B - A + 1$. This results to an $A = 1$. Thus the guessed function $T(n) = An\lg n + Bn + C$ gets resolved to having $A = 1$ and $B = C = 0$ leading to the same result with the iteratiion or previous substitution method. No need to explicitly do induction.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.20.1 Remark 1. In class we solve using the substitution method a recurrence of the form $T(n) = T(n/5) + T(7n/10) + n$. A swift and compressed proof is available in Subject 6 (around page 19) in the context of the analysis of the running time of worst-case Select (selection problem). A more generic and general proof for such a recurrence appears in a problem set, where $1/5$ and $7/10$ are replaced with generic constant a, b.

2.20.2 Remark 2. Let us try to guess the solutions of a variant where the linear term is replaced now with a constant term. What if we try to solve $T(n) = T(n/3) + T(2n/3) + 7$?

If we try to prove that $P(n) : T(n) \leq cn$ utilizing $P(n/3)$ and $P(2n/3)$ we end up with proving

$$T(n) \leq cn/3 + c2n/3 + 7 \leq cn + 7$$

not $T(n) \leq cn$. The $+7$ is causing us problems. How do we fix this **"tiny"** problem? By subtracting something smaller than the high order term from the solution i.e. try for a solution a $T(n) \leq cn - a$. ($cn - a$ MUST be positive as well i.e. $n > a/c$). We then get

$$T(n) \leq \overbrace{cn/3 - a}^{P(n/3)} + \overbrace{2cn/3 - a}^{P(2n/3)} + 7 \leq cn - a + (7 - a).$$

In order to prove that $T(n) \leq cn - a$ we must show that $T(n) \leq cn - a + (7 - a) \leq cn - a$ i.e. $7 - a \leq 0$. For the assumption to hold it suffices that $a \geq 7$.

2.20.3 Remark 3. What if we try to solve $T(n) = T(n/2) + T(n/3) + 10n$? Guess $T(n) = \Theta(n)$ (note that $1/2$ of $n/2$ plus $1/3$ of $n/3$ is $5/6 < 1$).

2.20.4 Remark 4. What if we try to solve $T(n) = T(2n/3) + T(n/3) + 10n$? Guess $T(n) = O(n \lg n)$ as $2/3 + 1/3 = 1$. $T(n) = \Theta(n \lg n)$ is the tightest possible bound.

2.20.5 Remark 5. What if we try to solve $T(n) = T(4n/3) + T(n/3) + 10n$? Guess $T(n) = O(n^\alpha)$ or $T(n) = \Theta(n^\alpha)$ for some constant $\alpha > 1$.

2.20.6 Remark 6. Useful Tricks: Change of variables might help. Remember to derive the correct boundary conditions when changing variables. $T(n) = T(\sqrt{n}) + \lg n$. Change variables $H(m) = T(2^m)$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Appendix
Example 1: Asymptotic Notation

2.21. *Use the definitions to show that* $n^5 - 25n = \Theta(n^5)$. We show first that $n^5 - 25 = O(n^5)$ and then that $n^5 - 25 = \Omega(n^5)$.

2.21.1 Case 1. Show that $n^5 - 25 = O(n^5)$.

For all $n \geq 1$ we have that

$$n^5 - 25 \leq n^5$$

Therefore there exist constant $n_2 = 1$ and $c_2 = 1$ such that $n^5 - 25 \leq c_2 n^5$ for all $n \geq n_2$. This proves the claim.

Technique 1. What we use in this proof is the fact that $n^2 \pm An \pm B$ is bounded above, for positive A, B, by $n^2 + An^2 + Bn^2 \leq (1 + A + B)n^2$.

2.21.2 Case 2. Show that $n^5 - 25 = \Omega(n^5)$

Technique 1 can not be used in this case. The next step is non-trivial. We bound $n^5 - 25$ from below by $n^5/2$. This is so as long as

$$\begin{aligned} n^5 - 25 &\geq n^5/2 \Leftrightarrow \\ n^5/2 &\geq 25 \Leftrightarrow \\ n^5 &\geq 50 \Leftrightarrow \\ n &\geq 3 \end{aligned}$$

We can do this as long as n is not zero; this is true since for all cases we assume that at least $n \geq 1$. Therefore there exist constant $n_1 = 3$ and $c_1 = 1/2$ such that $n^5 - 25 \geq n^5/2$ for all $n \geq n_1 = 3$. This proves the claim. Note that 3 is not the best possible constant. We don't need to find the best possible constant but THE EASIEST POSSIBLE!

In order to show that $n^5 - 25 = \Theta(n^5)$ we need to establish c_1, c_2 and n_0. c_1 and c_2 are $1/2$ and 1 respectively. $n_0 = \max(n_1, n_2) = 3$. For these values the problem is thus shown.

(c) Copyright A. Gerbessiotis.

Appendix
Example 2: Iteration method

2.22 *Use the iteration method to solve the recurrence,* $T(n) = 4T(n/2) + n$, $T(4) = 6$.

$$\begin{aligned}
T(n) &= 4T(n/2) + n \\
&= 4\left(4T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)\right) + n \\
&= 4^2 T\left(\frac{n}{2^2}\right) + \frac{4^1 n}{2^1} + n \\
&= 4^2\left(4T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)\right) + 2^1 n + 2^0 n \\
&= 4^3 T\left(\frac{n}{2^3}\right) + 2^2 n + 2^1 n + 2^0 n \\
&= \ldots \\
&= 4^i T\left(\frac{n}{2^i}\right) + 2^{i-1} n + \ldots + 2^1 n + 2^0 n \\
&= 4^i T\left(\frac{n}{2^i}\right) + n\left(2^i - 1\right)
\end{aligned}$$

The base case is $T(4) = 6$. We set $n/2^i = 4$, i.e. $n = 2^{i+2}$. If we solve this for i, we get $i = \lg n - 2$. Then $T(n/2^i) = T(4) = 6$. In addition, $2^i = n/4$, $4^i = 2^i \cdot 2^i = n/4 \cdot n/4 = n^2/16$. Therfore the formula for $T(n)$ becomes.

$$\begin{aligned}
T(n) &= 4^i T\left(\frac{n}{2^i}\right) + n\left(2^i - 1\right) \\
&= (n^2/16)T(4) + n(n/4 - 1) \\
&= 3n^2/8 + n^2/4 - n \\
&= 5n^2/8 - n
\end{aligned}$$

Note. It is easy to verify for example that $T(4) = 5 \cdot 4^2/8 - 4 = 6$. This can serve as a checking mechanism that the solution that you have established satisfies at least the base case.