Introduction to Analysis of Algorithms

Objectives

- Understand the concept of analyzing algorithms in terms of efficiency
- Determine the big-Oh time complexity of mathematical equations
- Compute the complexity of simple looped algorithms
- Compare the complexities of different collection-based operations

Introduction to Analysis of Algorithms

- One aspect of software quality is the efficient use of *computer resources*

- We want to analyze algorithms with respect to "execution time" to determine their efficiency
  - Called *time complexity* analysis
  - Note that we are not actually measuring execution time, but rather examining the number of operations performed.

Time Complexity

- Analysis of time taken is based on:
  - Problem size (e.g. number of items to sort)
  - Primitive operations (e.g. comparison of two values)
- What we want to analyze is the relationship between
  - The size of the problem, n
  - And the time it takes to solve the problem, t(n)   x, f(x).
    - Note that t(n) is a function of n, so it depends on the size of the problem

Time Complexity Functions

- This t(n) is called a ***time complexity function***
- What does a time complexity function look like?
  - Example of a time complexity function for some algorithm:

    $t(n) = 15n^2 + 45n$

    - See the next slide to see how t(n) changes as n gets bigger!

Example: $15n^2 + 45\ n$

No. of items **n**	**$15n^2$**	**45n**	**$15n^2 + 45n$**
1	15	45	60
2	60	90	150
5	375	225	600
10	1,500	450	1,950
100	150,000	4,500	154,500
1,000	15,000,000	45,000	15,045,000
10,000	1,500,000,000	450,000	1,500,450,000
100,000	150,000,000,000	4,500,000	150,004,500,000
1,000,000	15,000,000,000,000	45,000,000	15,000,045,000,000

Comparison of Terms in $15n^2 + 45n$

- When n is small, which term is larger? 45n.
- But, as n gets larger, note that the $15n^2$ term grows more quickly than the 45n term
- We say that the n^2 term is *dominant* in this expression.

looks for a large data set.

Big-Oh Notation

- We require a measurement of the time complexity of an algorithm that is ***independent*** on any implementation details (programming language and computer that will execute the algorithm).
- i.e. we generally only care about **number of operations performed**, not how long it takes since that varies from one machine to another.

It will be effected by the hardware of the computer.

Big-Oh Notation

- The key issue is the ***asymptotic complexity*** of the function or ***how it grows as n increases***
  - This is determined by the ***dominant term*** in the growth function (the term that increases most quickly as *n* increases)
  - Constants become irrelevant as *n* increases since we want a characterization of the time complexity of an algorithm that is ***independent*** of the computer that will be used to execute it. Since different computers differ in speed by a constant factor, constant factors are ignored when expressing the asymptotic complexity of a function.

Big-Oh Notation

- The asymptotic complexity of the function is referred to as the *order of* the function, and is specified by using ***Big-Oh notation.***
  - ***Example***: ***$O(n^2)$*** means that the time taken by the algorithm grows like the n^2 function as n increases

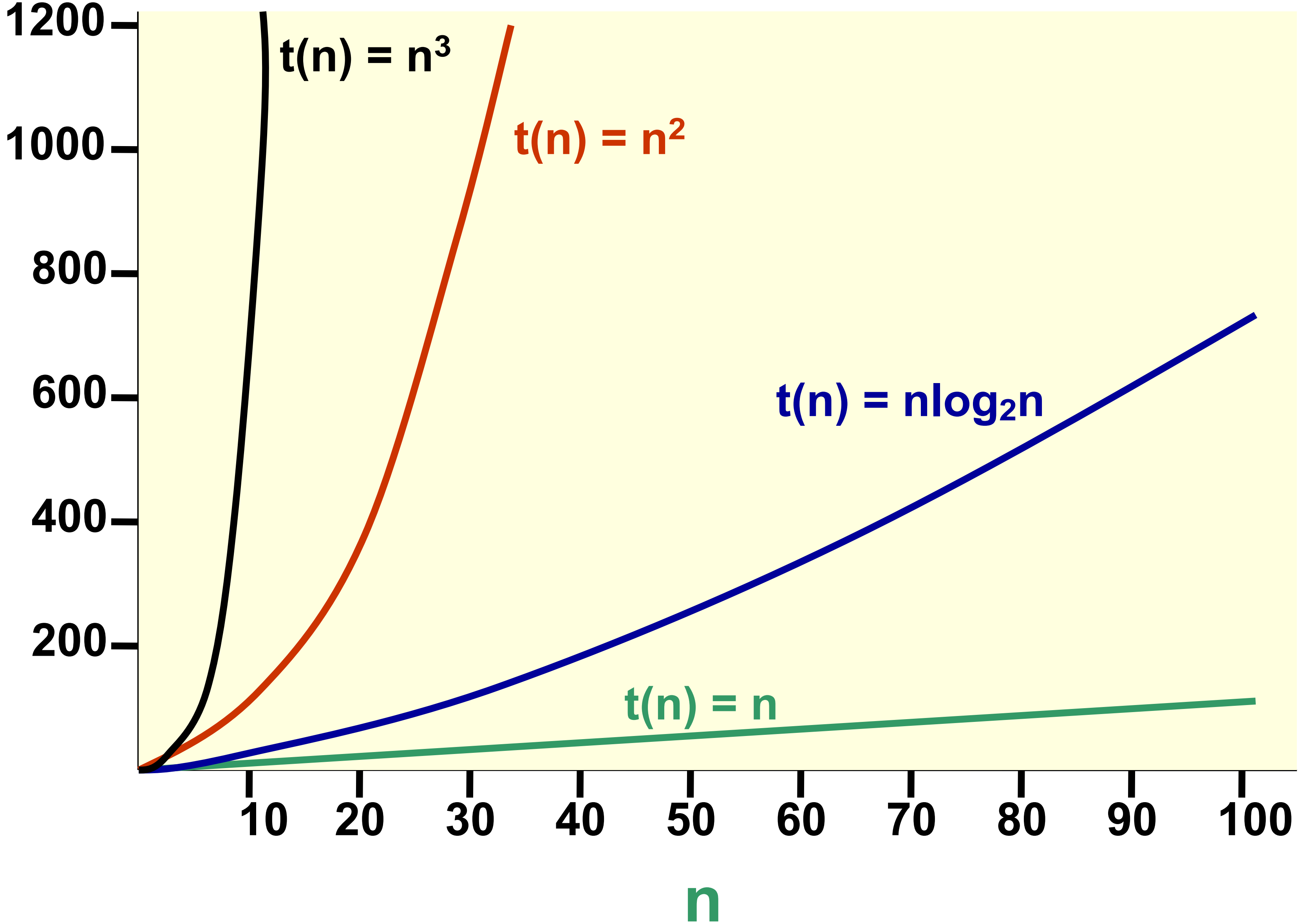
    dominant term.
  - ***O(1)*** means constant time, regardless of the size of the problem

10 operations => O(1).

Some Growth Functions and Their Asymptotic Complexities

Growth Function	Order
$t(n) = 17$	$O(1)$
$t(n) = 20n - 4$	$O(n)$
$t(n) = 12n * \log_2 n + 100n$	$O(n*\log_2 n)$
$t(n) = 3n^2 + 5n - 2$	$O(n^2)$
$t(n) = 2^n + 18n^2 + 3n$	$O(2^n)$

Comparison of Some Typical Growth Functions



Exercise: Asymptotic Complexities

Growth Function	Order
$t(n) = 5n^2 + 3n$	? $O(n^2)$
$t(n) = n^3 + \log_2 n - 4$	? $O(n^3)$
$t(n) = \log_2 n * 10n + 5$	? $O(n \cdot \log_2 n)$
$t(n) = 3n^2 + 3n^3 + 3$	? $O(n^3)$.
$t(n) = 2^n + 18n^{100}$	? $O(2^n)$.

Determining Time Complexity

- Algorithms frequently contain sections of code that are executed over and over again, i.e. *loops*
- Analyzing loop execution is vital in determining time complexity.

Analyzing Loop Execution

- A loop executes a certain number of times (say n), so the time complexity of the loop is n times the time complexity of the body of the loop
- *Example*: what is the time complexity of the following loop, in Big-O notation?

```
x = 0;
for (int i=0; i<n; i++)
        x = x + 1;
```

O(n)

- ***Nested loops***: **the body of the outer loop includes the inner loop**
- ***Example***: **what is the time complexity of the following loop, in Big-O notation? Read the next set of notes from the course's webpage to see how the time complexity of this algorithm and the algorithms in the following pages are computed.**

```
for (int i=0; i<n; i++) {
        x = x + 1;
        for (int j=0; j<n; j++)
            y = y – 1;
}
```

$O(n^2)$.

outer loop · inner loop

n · n.

n^2 times.

More Loop Analysis Examples

```
x = 0;
for (int i=0; i<n; i=i+2) {
        x = x + 1;
}
```

$\frac{1}{2}n$ times => $O(n)$.

```
x = 0;
for (int i=1; i<n; i=i*2) {
        x = x + 1;
}
```

2 4 8 16 ……

$\log_2 n$ => $O(\log_2 n)$.

More Loop Analysis Examples

```
x = 0;
for (int i=0; i<n; i++)
   for (int j = i; j < n; j ++) {
        x = x + 1;
   }
```

n

n

$O(n^2)$.

$n=3$:

i	j		
0	0	1	2
1	1	2	
2	2		

$n(n+1)/2$.

Best-Case vs Worst-Case

- **Our time complexities are usually based on the worst-case or most common case.**
- **What does it mean to consider a "*best case scenario*" or "*worst case scenario*" for time complexity?**
- **We have to think about all different cases for the algorithms to determine if some require additional operations.**
- **Best case means the number of operations is at its least. Worst case means the number of operations is at its most.**

Best-Case vs Worst-Case

- Suppose you are moving and there are *n* boxes and your laptop is contained in one of them. You need to search them until you find it.
  - Worst-case: The last box you look at contains the laptop so this is O(n).
  - Best-case: The first box you look at contains the laptop so this is just O(1).
  - If the box is anywhere between, it would be considered O(n).
    normal case
    on average, ½ n => O(n).
  - In general, we would say this problem is O(n).

Analysis of Collection Operations

- **Stack operations are generally efficient, because they all work on only one end of the collection. How do they compare to Queue operations?**

- **Which are more efficient: array implementations or linked list implementations?**

- **What are the time complexities of the various List add() methods?**

Analysis of Stack Operations

- n is the number of items on the stack
- push operation for ArrayStack:
  - O(1) if array is not full (why?) constant operations.
  - What would it be if the array is full? *(worst case)* full → expandCapacity() : need to transfer arrays. n operations O(n)
- push operation for LinkedStack:
  - O(1) (why?) always just adding new nodes.
- pop operation for each? O(1).
- peek operation for each? O(1). in both cases.

Analysis of Queue Operations

- **n** is the number of items on the queue
- **enqueue** operation:
  - O(1) for LinkedQueue just adding.
  - O(1) or O(n) for ArrayQueue expand capacity.
  - O(1) or O(n) for CircularArrayQueue expandCapacity() is called.
- **dequeue** operation:
  - O(1) for LinkedQueue
  - O(n) for ArrayQueue all items after the dequeued item move one position forward.
  - O(1) for CircularArrayQueue just incrementing front, no need to move items.
- **first** operation for each?

O(1) just return the first element.

Analysis of the List add() Operations

- **n** is the number of items in the list
- **OrderedList add()**:
  a comparsions. to each elements. to keep the List ordered. b shifts backward
  - **O(n)** (Linked), **O(n)** (Array)
    a+b = n
- **UnorderedList addToFront()**:
  to front
  - **O(1)** (Linked), **O(n)** (Array)
    shifted backwards. probably expand capacity() : 2n
    O(n)
- **UnorderedList addToRear()**:
  have a rear pointer   no rear pointer.
  expand capacity.
  - **O(1)** or **O(n)** (Linked), **O(1)** or **O(n)** (Array)
    shifting
    to get to the rear, n traversals.   space is enough.
- **UnorderedList addAfter()**:
  - **O(n)** (Linked), **O(n)** (Array)
    shifts backward and probably expand capacity
    traversal to find the target element.
- **IndexedList add()**:
  - **O(n)** (Linked), **O(n)** (Array)
    traverse to get to the index
    shift backward and the probably shift backwards.