Algorithm Analysis: Big O Notation

CMPT 225

Objectives

- Determine the running time of simple algorithms
  - Best case
  - Average case
  - Worst case
- Profile algorithms
- Understand O notation's mathematical basis
- Use O notation to measure running time

Algorithm Analysis

- Algorithms can be described in terms of
  - Time efficiency
  - Space efficiency
- Choosing an appropriate algorithm can make a significant difference in the usability of a system
  - Government and corporate databases with many millions of records, which are accessed frequently
  - Online search engines
  - Real time systems where near instantaneous response is required
    - From air traffic control systems to computer games

Comparing Algorithms

- There are often many ways to solve a problem
  - Different algorithms that produce the same results
    - e.g. there are numerous **sorting** algorithms
- We are usually interested in how an algorithm performs when its input is large
  - In practice, with today's hardware, **most** algorithms will perform well with small input
  - There are exceptions to this, such as the Traveling Salesman Problem

Measuring Algorithms

- It is possible to **count** the number of operations that an algorithm performs
  - By a careful visual walkthrough of the algorithm or by
  - Inserting code in the algorithm to count and print the number of times that each line executes (**profiling**)
- It is also possible to **time** algorithms
  - Compare system time before and after running an algorithm
    - E.g., in C++: `#include <ctime>`

Timing Algorithms

- It may be useful to time how long an algorithm takes to run
  - In some cases it may be *essential* to know how long an algorithm takes on some system
    - e.g. air traffic control systems
- But is this a good *general* comparison method?
- Running time is affected by a number of factors other than algorithm efficiency

Running Time is Affected By

- CPU speed
- Amount of main memory
- Specialized hardware (e.g. graphics card)
- Operating system
- System configuration (e.g. virtual memory)
- Programming language
- Algorithm implementation
- Other programs
- System tasks (e.g. memory management)
- …

Counting

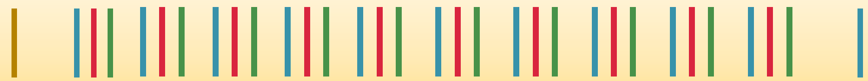
- Instead of *timing* an algorithm, *count* the number of instructions that it performs
- The number of instructions performed may vary based on
  - The size of the input
  - The organization of the input
- The number of instructions can be written as a cost function on the input size

A Simple Example

C++

```
void printArray(int *arr, int n){
  for (int i = 0; i < n; ++i){
    cout << arr[i] << endl;
  }
}
```

Operations performed on an array of length 10



declare and initialize *i*

perform comparison, print array element, and increment *i*:10 times

make comparison when *i* = 10

Cost Functions

- Instead of choosing a particular input size we will express a cost function for input of size **n**
- Assume that the running time, **t**, of an algorithm is proportional to the number of operations
- Express **t** as a function of **n**
  - Where **t** is the time required to process the data using some algorithm **A**
  - Denote a cost function as $\mathbf{t_A(n)}$
    - i.e. the running time of algorithm **A**, with input size **n**

A Simple Example

```
void printArray(int *arr, int n){
  for (int i = 0; i < n; ++i){
    cout << arr[i] << endl;
  }
}
```

Operations performed on an array of length **n**	1	3n	1
	declare and initialize *i*	perform comparison, print array element, and increment *i*: *n* times	make comparison when *i* = *n*

$t = 3n + 2$

Input Varies

- The number of operations usually varies based on the size of the input
  - Though not always, consider array lookup
- In addition algorithm performance may vary based on the **organization** of the input
  - For example consider searching a large array
  - If the target is the first item in the array the search will be very quick

Best, Average and Worst Case

- Algorithm efficiency is often calculated for three broad cases of input
  - Best case
  - Average (or "usual") case
  - Worst case
- This analysis considers how performance varies for *different* inputs of the *same* size

Analyzing Algorithms

- It can be difficult to determine the exact number of operations performed by an algorithm
  - Though it is often still useful to do so
- An alternative to counting all instructions is to focus on an algorithm's **barometer instruction**
  - The barometer instruction is the instruction that is executed the most number of times in an algorithm
  - The number of times that the barometer instruction is executed is usually proportional to its running time

Comparisons

- Let's analyze and compare some different algorithms
  - Linear search
  - Binary search
  - Selection sort
  - Insertion sort

Cost Functions for Searching

Searching

- It is often useful to find out whether or not a list contains a particular item
  - Such a search can either return true or false
  - Or the position of the item in the list
- If the array isn't sorted use **linear search**
  - Start with the first item, and go through the array comparing each item to the target
  - If the target item is found return true (or the index of the target element)

Linear Search

C++

```
int linSearch(int* arr, int n, int target){
    for (int i=0; i < n; i++){
        if(target == arr[i]){
            return i;
        }
    } //for
    return -1; //target not found
}
```

The function returns as soon as the target item is found

return -1 to indicate that the item has not been found

Linear Search Barometer Instruction

- Iterate through an array of **n** items searching for the target item
- The barometer instruction is equality checking (or *comparisons* for short)
  - `x == arr[i];`
  - There are actually two other barometer instructions, what are they?
- How many comparisons does linear search do?

Linear Search Comparisons

- Best case
  - The target is the first element of the array
  - Make 1 comparison
- Worst case
  - The target is not in the array or
  - The target is at the last position in the array
  - Make **n** comparisons in either case
- Average case
  - Is it (Best case + Worst case) / 2, so (n + 1) / 2?

Linear Search: Average Case

- There are two situations when the worst case arises
  - When the target is the last item in the array
  - When the target is not there at all
- To calculate the average cost we need to know how often these two situations arise
  - We can make assumptions about this
  - Though any these assumptions may not hold for a particular use of linear search

Assumptions

- Assume that the target is not in the array ½ the time
  - Therefore ½ the time the entire array has to be searched
- Assume that there is an equal probability of the target being at any array location
  - If it is in the array
  - That is, there is a probability of $1/n$ that the target is at some location i

Cost When Target Not Found

- Work done if the target is **not** in the array
  - n comparisons
  - This occurs with probability of 0.5

Cost When Target Is Found

- Work done if target is in the array:
  - 1 comparison if target is at the 1st location
    - Occurs with probability $1/n$ (second assumption)
  - 2 comparisons if target is at the 2nd location
    - Also occurs with probability $1/n$
  - i comparisons if target is at the i^{th} location
- Take the weighted average of the values to find the total expected number of comparisons (E)
  - $E = 1*1/n + 2*1/n + 3*1/n + \ldots + n * 1/n$ *or*
  - $E = (n + 1) / 2$

Average Case Cost

- Target is **not** in the array: n comparisons
- Target **is** in the array $(n + 1) / 2$ comparisons
- Take a weighted average of the two amounts:
  - $= (n * ½) + ((n + 1) / 2 * ½)$
  - $= (n / 2) + ((n + 1) / 4)$
  - $= (2n / 4) + ((n + 1) / 4)$
  - $= (3n + 1) / 4$
- Therefore, on average, we expect linear search to perform $(3n + 1) / 4$ comparisons*

  *recall the assumptions we made about ½ not in array, uniform distribution if in array

Searching Sorted Arrays

- If we sort the target array first we can change the linear search average cost to around *n* / 2
  - Once a value equal to or greater than the target is found the search can end
    - So, if a sequence contains 8 items, on average, linear search compares 4 of them,
    - If a sequence contains 1,000,000 items, linear search compares 500,000 of them, etc.
- However, if the array is sorted, it is possible to do **much better** than this

Binary Search Sketch

Search for 32

Guess that the target item is in the middle, that is index = 15 / 2 = 7

value	07	11	15	21	29	32	44	45	57	61	64	73	79	81	86	92
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The array is sorted, and contains 16 items indexed from 0 to 15

Binary Search Sketch

Search for 32

45 is greater than 32 so the target must be in the lower half of the array

Repeat the search, guessing the mid point of the lower subarray (6 / 2 = 3)

value	07	11	15	21	29	32	44	45	57	61	64	73	79	81	86	92
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Everything in the upper half of the array can be ignored, halving the search space

Binary Search Sketch

Search for 32

21 is less than 32 so the target must be in the upper half of the subarray

Repeat the search, guessing the mid point of the new search space, 5

The target is found so the search can terminate

value	07	11	15	21	29	32	44	45	57	61	64	73	79	81	86	92
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The mid point = (lower subarray index + upper index) / 2

Binary Search

- Requires that the array is sorted
  - In either ascending or descending order
  - Make sure you know which!
- A **divide and conquer** algorithm
  - Each iteration divides the problem space in half
  - Ends when the target is found or the problem space consists of one element

Binary Search Algorithm

C++

```
int binSearch(int * arr, int n, int target){
    int lower = 0;
    int upper = n - 1;
    int mid = 0;
    while (lower <= upper){
        mid = (lower + upper) / 2;
        if(target == arr[mid]){
            return mid;
        } else if(target > arr[mid]){
            lower = mid + 1;
        } else { //target < arr[mid]
            upper = mid - 1;
        }
    } //while
    return -1; //target not found
}
```

Index of the last element in the array

Note the if, else if, else

Analyzing Binary Search

- The algorithm consists of three parts
  - Initialization (setting lower and upper)
  - While loop including a return statement on success
  - Return statement which executes when on failure
- Initialization and return on failure require the same amount of work regardless of input size
- The number of times that the while loop iterates depends on the size of the input

Binary Search Iteration

- The while loop contains an if, else if, else statement
- The first if condition is met when the target is found
  - And is therefore performed at most once each time the algorithm is run
- The algorithm usually performs 5 operations for each iteration of the while loop
  - Checking the while condition
  - Assignment to mid
  - Equality comparison with target
  - Inequality comparison
  - One other operation (setting either lower or upper)

Binary Search: Best Case

- In the best case the target is the midpoint element of the array
  - Requiring one iteration of the while loop

Binary Search: Worst Case

- What is the worst case for binary search?
  - Either the target is not in the array, or
  - It is found when the search space consists of one element
- How many times does the while loop iterate in the worst case?

Analyzing the Worst Case

- Each iteration of the while loop halves the search space
  - For simplicity assume that n is a power of 2
    - So $n = 2^k$ (e.g. if $n = 128$, $k = 7$)
  - The first iteration halves the search space to $n/2$
  - After the second iteration the search space is $n/4$
  - After the kth iteration the search space consists of just one element, since $n/2^k = n/n = 1$
    - Because $n = 2^k$, $k = \log_2 n$
  - Therefore at most $\log_2 n$ iterations of the while loop are made in the worst case!

Average Case

- Is the average case more like the best case or the worst case?
  - What is the chance that an array element is the target
    - 1/*n* the first time through the loop
    - 1/(*n*/2) the second time through the loop
    - … and so on …
- It is more likely that the target will be found as the search space becomes small
  - That is, when the while loop nears its final iteration
  - We can conclude that the average case is more like the worst case than the best case

Binary Search vs Linear Search

n	(3n+1)/4	$\log_2(n)$
10	8	3
100	76	7
1,000	751	10
10,000	7,501	13
100,000	75,001	17
1,000,000	750,001	20
10,000,000	7,500,001	24

Simple Sorting

Simple Sorting

- As an example of algorithm analysis let's look at two simple sorting algorithms
  - Selection Sort and
  - Insertion Sort
- Calculate an approximate cost function for these two sorting algorithms
  - By analyzing how many operations are performed by each algorithm
  - This will include an analysis of how many times the algorithms' loops iterate

Selection Sort

- Selection sort is a simple sorting algorithm that repeatedly finds the smallest item
  - The array is divided into a sorted part and an unsorted part
- Repeatedly swap the first unsorted item with the smallest unsorted item
  - Starting with the element with index 0, and
  - Ending with last but one element (index $n - 1$)

Selection Sort

23	41	33	81	07	19	11	45	find smallest unsorted - 7 comparisons
07	41	33	81	23	19	11	45	find smallest unsorted - 6 comparisons
07	11	33	81	23	19	41	45	find smallest unsorted - 5 comparisons
07	11	19	81	23	33	41	45	find smallest unsorted - 4 comparisons
07	11	19	23	81	33	41	45	find smallest unsorted - 3 comparisons
07	11	19	23	33	81	41	45	find smallest unsorted - 2 comparisons
07	11	19	23	33	41	81	45	find smallest unsorted - 1 comparison
07	11	19	23	33	41	45	81	

Selection Sort Comparisons

Unsorted elements	Comparisons to find smallest
n	n-1
n-1	n-2
…	…
3	2
2	1
1	0
	$n(n-1)/2$

Selection Sort Algorithm

C++

```
void selectionSort(int *arr, int n){
    for(int i = 0; i < n-1; ++i){
        int smallest = i;
        // Find the index of the smallest element
        for(int j = i + 1; j < n; ++j){
            if(arr[j] < arr[smallest]){
                smallest = j;
            }
        }
        // Swap the smallest with the current item
        int temp = arr[i];
        arr[i] = arr[smallest];
        arr[smallest] = temp;
    }
}
```

outer loop
n-1 times

inner loop body
$n(n-1)/2$ times

Selection Sort Cost Function

- The outer loop is evaluated n-1 times
  - 7 instructions (including the loop statements)
  - Cost is $7(n\text{-}1)$
- The inner loop is evaluated $n(n-1)/2$ times
  - There are 4 instructions but one is only evaluated some of the time
  - Worst case cost is $4(n(n-1)/2)$
- Some constant amount (k) of work is performed
  - e.g. initializing the outer loop
- Total cost: $7(n\text{-}1) + 4(n(n-1)/2) + k$
  - Assumption: all instructions have the same cost

Selection Sort Summary

- In broad terms and ignoring the actual number of executable statements selection sort
  - Makes $n*(n-1)/2$ comparisons, regardless of the original order of the input
  - Performs $n-1$ swaps
- Neither of these operations are substantially affected by the organization of the input

Insertion Sort

- Another simple sorting algorithm
  - Divides array into sorted and unsorted parts
- The sorted part of the array is expanded one element at a time
  - Find the correct place in the sorted part to place the 1st element of the unsorted part
    - By searching through all of the sorted elements
  - Move the elements after the insertion point up one position to make space

Insertion Sort

23	41	33	81	07	19	11	45	treats first element as sorted part
23	41	33	81	07	19	11	45	locate position for 41 – 1 comparison
23	33	41	81	07	19	11	45	locate position for 33 – 2 comparisons
23	33	41	81	07	19	11	45	locate position for 81 – 1 comparison
07	23	33	41	81	19	11	45	locate position for 07 – 4 comparisons
07	19	23	33	41	81	11	45	locate position for 19– 5 comparisons
07	11	19	23	33	41	81	45	locate position for 11– 6 comparisons
07	11	19	23	33	41	45	81	locate position for 45 – 2 comparisons

Insertion Sort Algorithm

C++

```
void insertionSort(int *arr, int n){
    for(int i = 1; i < n; ++i){
        int temp = arr[i];
        int pos = i;
        // Shuffle up all sorted items > arr[i]
        while(pos > 0 && arr[pos - 1] > temp){
            arr[pos] = arr[pos - 1];
            pos--;
        } //while
        // Insert the current item
        arr[pos] = temp;
    }
}
```

outer loop
n-1 times

inner loop body
how many times?

min: just the test for each outer loop iteration, n

max: i – 1 times for each iteration, n * (n – 1) / 2

Insertion Sort Cost

Sorted Elements	Worst-case Search	Worst-case Shuffle
0	0	0
1	1	1
2	2	2
…	…	…
n-1	n-1	n-1
	$n(n-1)/2$	$n(n-1)/2$

Insertion Sort Best Case

- The efficiency of insertion sort **is** affected by the state of the array to be sorted
- In the best case the array is already completely sorted!
  - No movement of array elements is required
  - Requires *n* comparisons

Insertion Sort Worst Case

- In the worst case the array is in reverse order
- Every item has to be moved all the way to the front of the array
  - The outer loop runs n-1 times
    - In the first iteration, one comparison and move
    - In the last iteration, n-1 comparisons and moves
    - On average, n/2 comparisons and moves
  - For a total of $n * (n\text{-}1) / 2$ comparisons and moves

Insertion Sort: Average Case

- What is the average case cost?
  - Is it closer to the best case?
  - Or the worst case?
- If *random* data are sorted, insertion sort is usually closer to the worst case
  - Around n * (n-1) / 4 comparisons
- What is average input for a sorting algorithm in any case?

O Notation

Algorithm Summary

- Linear search: $3(n + 1)/4$ – average case
  - Given certain assumptions
- Binary search: $\log_2 n$ – worst case
  - Average case similar to the worst case
- Selection sort: $n((n - 1) / 2)$ – all cases
- Insertion sort: $n((n - 1) / 2)$ – worst case
  - Average case is similar to the worst case

Algorithm Comparison

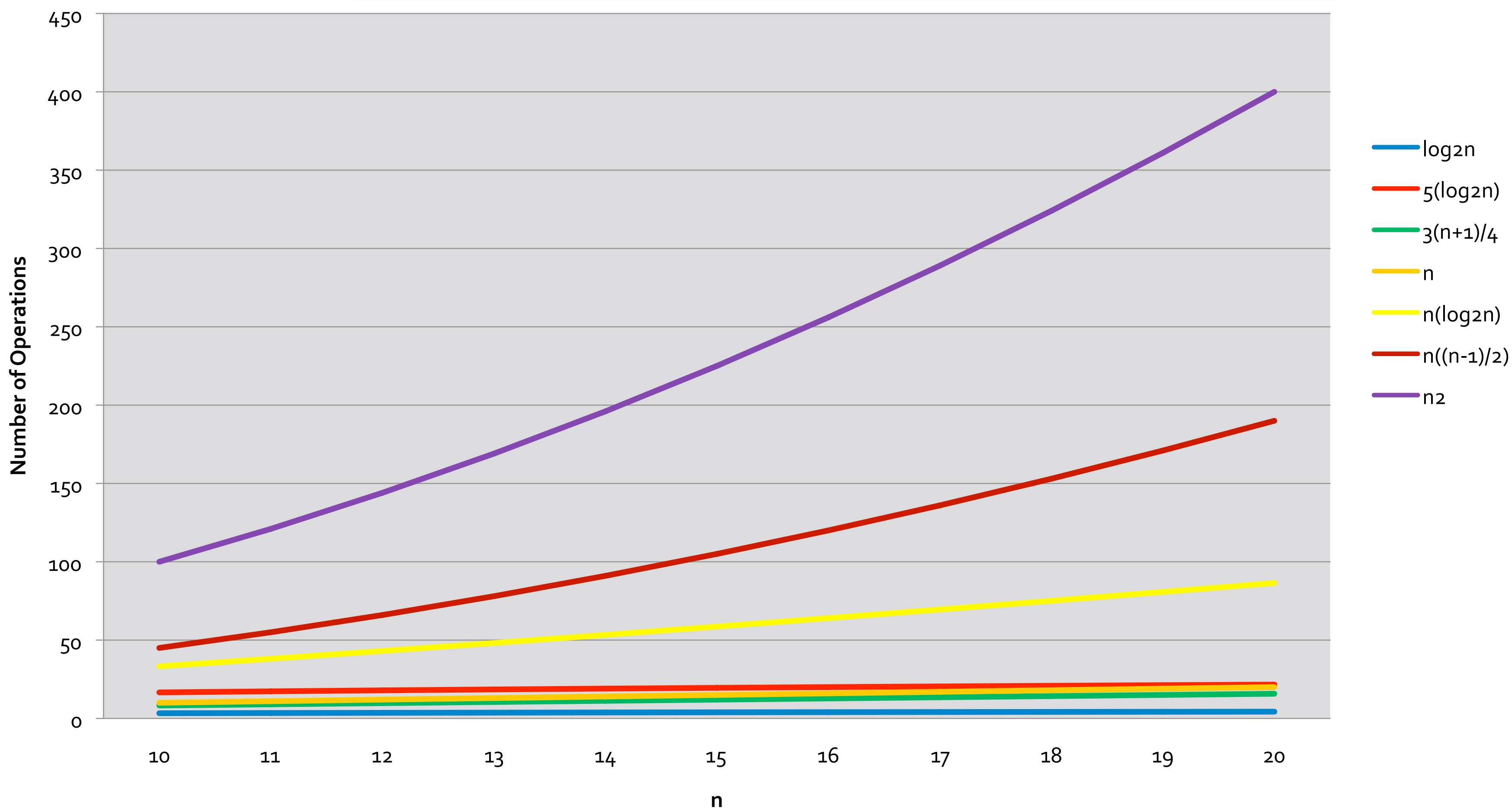
- Let's compare these algorithms for some arbitrary input size (say n = 1,000)
  - In order of the number of comparisons
    - Binary search
    - Linear search
    - Insertion sort best case
    - Quicksort (next week) average and best cases
    - Selection sort all cases, Insertion sort average and worst cases, Quicksort worst case

Algorithm Growth Rate

- What do we want to know when comparing two algorithms?
  - The most important thing is how quickly the time requirements increase with input size
  - e.g. If we double the input size how much longer does an algorithm take?
- Here are some graphs …

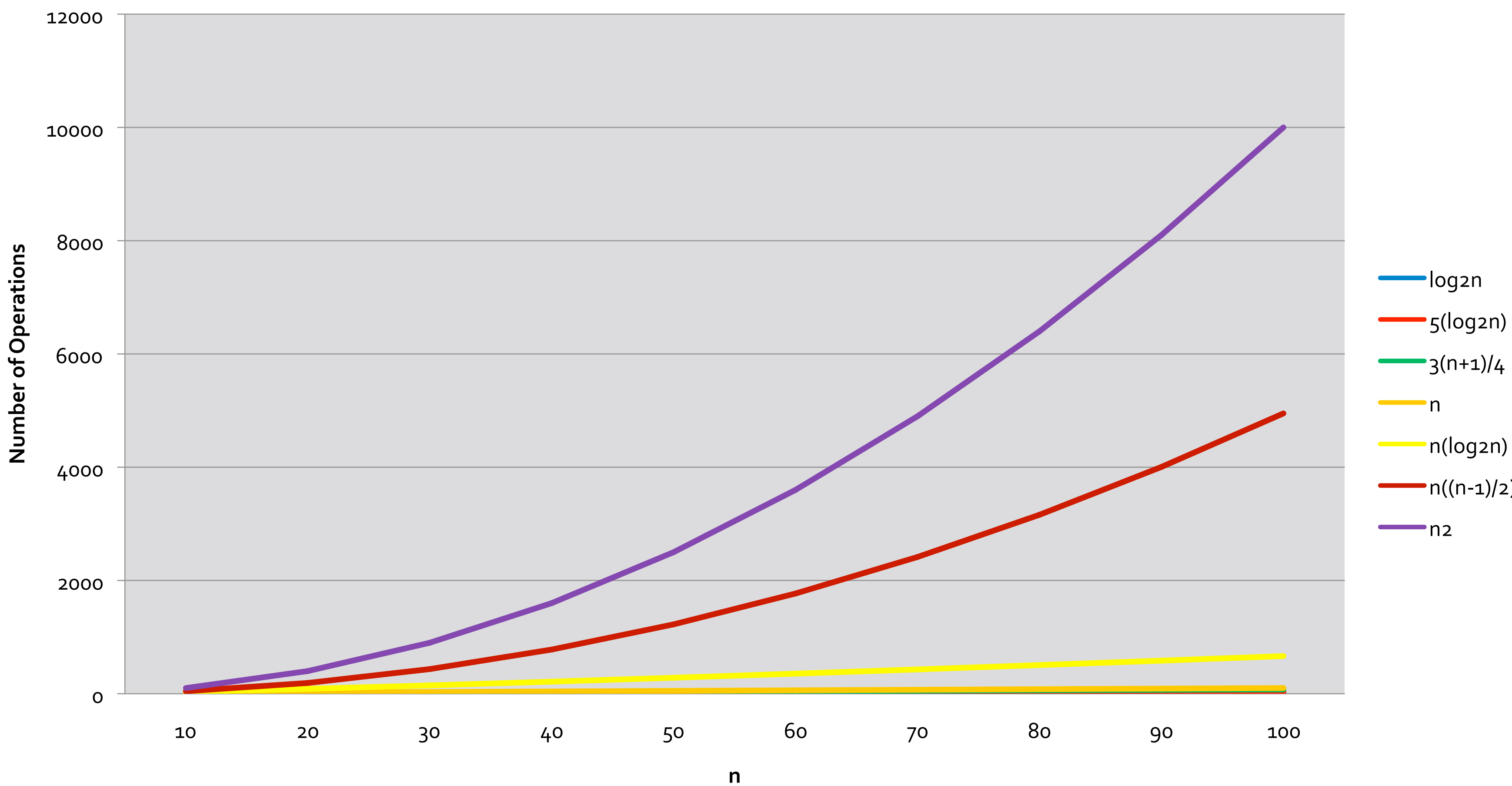
Small *n*

Hard to see what is happening with n so small …



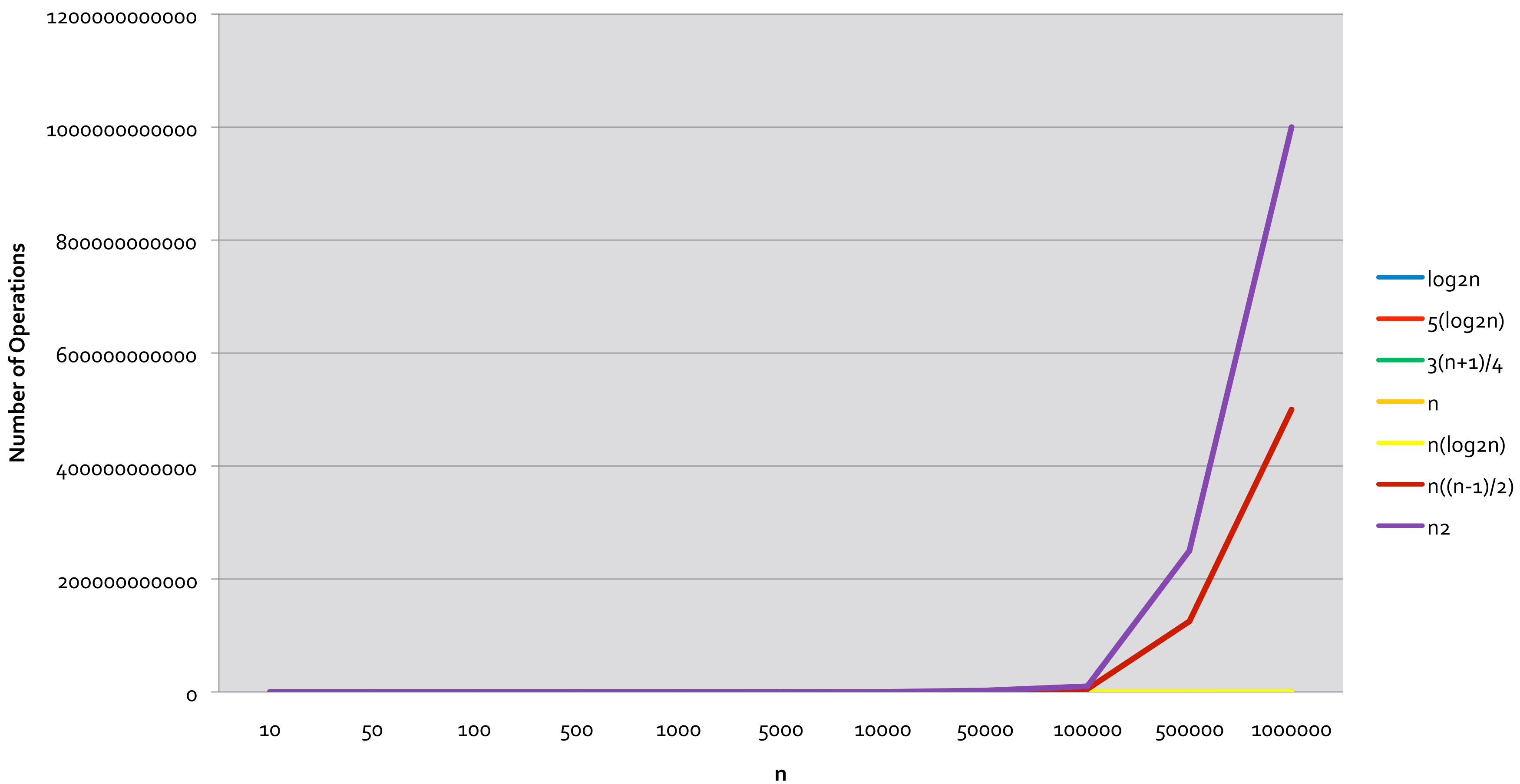
Not Much Bigger *n*

n^2 and n(n-1)/2 are growing much faster than any of the others



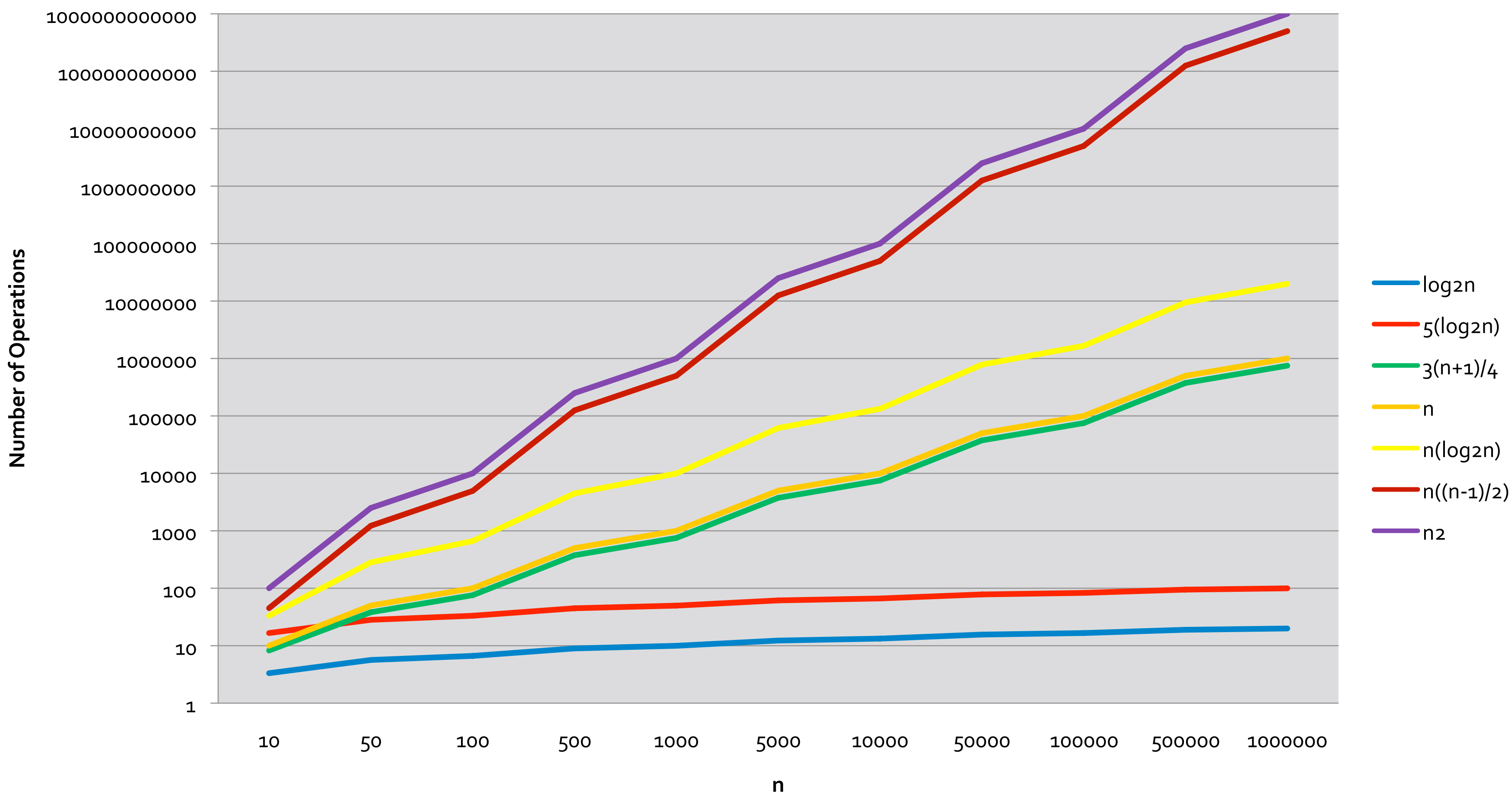
n from 10 to 1,000,000

Hmm! Let's try a logarithmic scale …



n from 10 to 1,000,000

Notice how **clusters** of growth rates start to emerge



O Notation Introduction

- Exact counting of operations is often difficult (and tedious), even for simple algorithms
  - And is often not much more useful than estimates due to the relative importance of other factors
- **O Notation** is a mathematical language for evaluating the running-time of algorithms
  - O-notation evaluates the growth rate of an algorithm

Example of a Cost Function

- Cost Function: $t_A(n) = n^2 + 20n + 100$
  - Which term in the funtion is most important (dominates)?
- It depends on the size of *n*
  - $n = 2$, $t_A(n) = 4 + 40 + \underline{100}$
    - The constant, 100, is the dominating term
  - $n = 10$, $t_A(n) = 100 + \underline{200} + 100$
    - $20n$ is the dominating term
  - $n = 100$, $t_A(n) = \underline{10{,}000} + 2{,}000 + 100$
    - n^2 is the dominating term
  - $n = 1000$, $t_A(n) = \underline{1{,}000{,}000} + 20{,}000 + 100$
    - n^2 is the dominating term

Big O Notation

- O notation approximates a cost function that allows us to estimate growth rate
  - The approximation is usually good enough
    - Especially when considering the efficiency of an algorithm as **n** gets very large
- Count the number of times that an algorithm executes its **barometer instruction**
  - And determine how the count increases as the input size increases

Why Big O?

- An algorithm is said to be ***order*** $f(n)$
  - Denoted as $O(f(n))$
- The function $f(n)$ is the algorithm's growth rate function
  - If a problem of size n requires time proportional to n then the problem is $O(n)$
    - i.e. If the input size is doubled then the running time is doubled

Big O Notation Definition

- An algorithm is *order* $f(n)$ if there are positive constants k and m such that
  - $t_A(n) \le k * f(n)$ for all $n \ge m$
  - If so we would say that $t_A(n)$ is $O(f(n))$
- The requirement $n > m$ expresses that the time estimate is correct if n is sufficiently large

Or In English…

- The idea is that a cost function can be approximated by another, simpler, function
  - The simpler function has 1 variable, the data size n
  - This function is selected such that it represents an *upper bound* on the value of $t_A(n)$
- Saying that the time efficiency of algorithm A $t_A(n)$ is $O(f(n))$ means that
  - A cannot take more than $O(f(n))$ time to execute, and
  - The cost function $t_A(n)$ *grows at most as* fast as $f(n)$

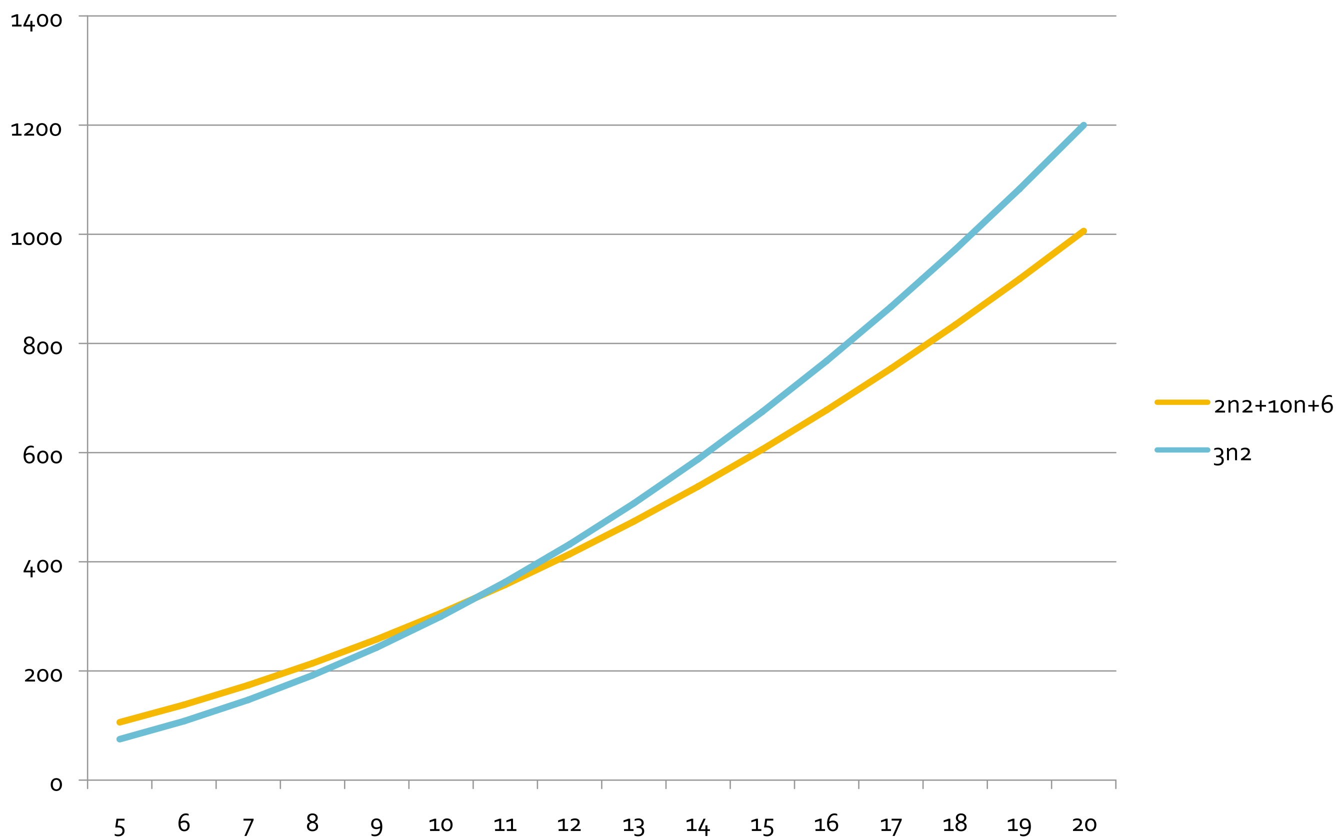
Big O Example

- Consider an algorithm with a cost function of $3n + 12$
  - If we can find constants m and k such that:
  - $k * n \geq 3n + 12$ for all $n \geq m$ then
  - The algorithm is $O(n)$
- Find values of k and m so that this is true
  - $k = 4$, and
  - $m = 12$ then
  - $4n \geq 3n + 12$ for all $n \geq 12$

Another Big O Example

- Consider an algorithm with a cost function of $2n^2 + 10n + 6$
  - If we can find constants m and k such that:
  - $k * n^2 \geq 2n^2 + 10n + 6$ for all $n \geq m$ then
  - The algorithm is $O(n^2)$
- Find values of k and m so that this is true
  - $k = 3$, and
  - $m = 11$ then
  - $3n^2 \geq 2n^2 + 10n + 6$ for all $n \geq 11$

And Another Graph



The general idea is …

- When using Big-*O* notation
- Instead of giving a precise formulation of the cost function for a particular data size
- Express the behaviour of the algorithm as the data size *n* grows very large so ignore
  - lower order terms and
  - constants

O Notation Examples

- All these expressions are $O(n)$:
  - n, $3n$, $61n + 5$, $22n - 5$, …
- All these expressions are $\mathbf{O}(\mathbf{n}^2)$:
  - n^2, $9n^2$, $18n^2 + 4n - 53$, …
- All these expressions are $O(n \log n)$:
  - $n(\log n)$, $5n(\log 99n)$, $18 + (4n - 2)(\log (5n + 3))$, …

Arithmetic and O Notation

- $O(k * f) = O(f)$ if k is a constant
  - e.g. $O(23 * O(\log n))$, simplifies to $O(\log n)$
- $O(f + g) = \max[O(f), O(g)]$
  - $O(n + n^2)$, simplifies to $O(n^2)$
- $O(f * g) = O(f) * O(g)$
  - $O(m * n)$, equals $O(m) * O(n)$
  - Unless there is some known relationship between m and n that allows us to simplify it, e.g. $m < n$

Typical Growth Rate Functions

- $O(1)$ – **constant** time
  - The time is independent of **n**, e.g. list look-up
- $O(\log n)$ – **logarithmic** time
  - Usually the log is to the base 2, e.g. binary search
- $O(n)$ – **linear** time, e.g. linear search
- $O(n*\log n)$ – e.g. quicksort, mergesort (next week)
- $O(n^2)$ – **quadratic** time, e.g. selection sort
- $O(n^k)$ – **polynomial** (where k is some constant)
- $O(2^n)$ – **exponential** time, very slow!

Note on Constant Time

- We write $O(1)$ to indicate something that takes a constant amount of time
  - e.g. finding the minimum element of an *ordered* array takes $O(1)$ time
    - The min is either at the first or the last element of the array
- **Important**: constants can be huge
  - So in practice $O(1)$ is not *necessarily* efficient
  - It tells us is that the algorithm will run at the same speed no matter the size of the input we give it

Worst, Average and Best Case

- The *O*-notation growth rate of some algorithms varies depending on the input
- Typically we consider three cases:
  - **Worst case**, usually (relatively) easy to calculate and therefore commonly used
  - **Average case**, often difficult to calculate
  - **Best case**, usually easy to calculate but less important than the other cases

O Notation Running Times

- Linear search
  - Best case: $O(1)$
  - Average case: $O(n)$
  - Worst case: $O(n)$
- Binary search
  - Best case: $O(1)$
  - Average case: $O(\log n)$
  - Worst case: $O(\log n)$

O Notation Running Times

- Selection sort
  - Best Case: $O(n^2)$
  - Average case: $O(n^2)$
  - Worst case: $O(n^2)$
- Insertion sort
  - Best case: $O(n)$
  - Average case: $O(n^2)$
  - Worst case: $O(n^2)$

Summary

Summary

- Analyzing algorithm running time
  - Record actual running time (e.g. in seconds)
    - Sensitive to many system / environment conditions
  - Count instructions
  - Summarize coarse behaviour of instruction count
    - O Notation

  - Note that all are parameterized by *problem size* ("n")
  - Analyze best, worst, "average" case

Summary

- Sorting algorithms
  - Insertion sort
  - Selection sort
- Running times of sorting algorithms

Readings

- Carrano Ch. 9