CS 10: Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson
260 (255) Sudikoff

Day 7 – Lists Part 2

Agenda

1. Growing array list implementation
2. Orders of growth
3. Asymptotic notation
4. List analysis
5. Iteration

Last time we implemented a List with a linked list, now we will use an array

Arrays

- Ordered set of elements of fixed total length
- Each element in array is of fixed length
- Can’t easily add/remove elements
- Indexed starting at 0 (Matlab starts at 1) to n-1
- Random access to elements (linked list required march down list to find desired element)
- Easy to get/set any element in an array
- One big chunk of memory
- In Java, access arrays with square brackets []

```
int[] numbers = new int[10]; //array of int 0..9 (NOT 10!)
for (int i=0;i<10;i++) {
    numbers[i] = i*2;  //set each element to i*2
}
```

The trick to using an array to represent a List is to grow the array size when needed

Using an array to implement List

- Allocate array of starting `maxSize` (say 10 items)
- Add items as required
- If `size` grows to `maxSize` then
  - Allocate larger array (say 2 times current `maxSize`)
  - Copy items from old array into new array
  - Set array instance variable to new array
  - (old array will be garbage collected)
- `add()`/`remove()` may require moving elements to make or close hole in array

With the growing trick, we can implement the List interface with an array

GrowingArray.java

- Create *array* of <T> to hold elements, size=0, and initial capacity=10
- Constructor, new the array to allocate space
- *size()* – return size variable as in linked list
- *add(int idx, T item)*
  - Check idx bounds
  - If size == array.length
    - Create new array 2 times larger than size
    - Copy elements from old array to new
    - Set *array* to new array
  - Loop backward from last to idx to move elements right one space
  - Set *array[idx] = item*
  - Increment *size*

With the growing trick, we can implement the List interface with an array

GrowingArray.java

- *remove(int idx)*
  - Check *idx* bounds
  - Loop from item 0 to *idx*-1, move items left one space
- *get(int idx)*
  - Check *idx* bounds
  - Return *array[idx]*
- *set(int idx, T item)*
  - Check *idx* bounds
  - Set *array[idx] = item*
- *toString()*
  - Return String representation of List
- Notice how fast get/set are in relation to linked list where we had to march down the list to get/set the element we wanted

Agenda

1. Growing array list implementation
2. Orders of growth
3. Asymptotic notation
4. List analysis
5. Iteration

Often run-time will depend on the number of elements an algorithm must process

Consider an array of length n

- Returning the first element takes a constant amount of time, irrespective of the number of elements in the array
- Binary search runs in log(n) time
- Sequential search runs in time proportional to n
- Many sorting algorithms run in time proportional to n^2 (think of "round-robin" tournament)
  - Given array with {1,2,3,4,5}
  - Compute:

    1+1, 2+1, … 5+1
    1+2, 2+2, … 5+2
    1+3, 2+3, … 5+3
    1+4, 2+4, … 5+4
    1+5, 2+5, … 5+5

    n rows and n columns means $n * n = n^2$ operations

Often run-time will depend on the number of elements an algorithm must process

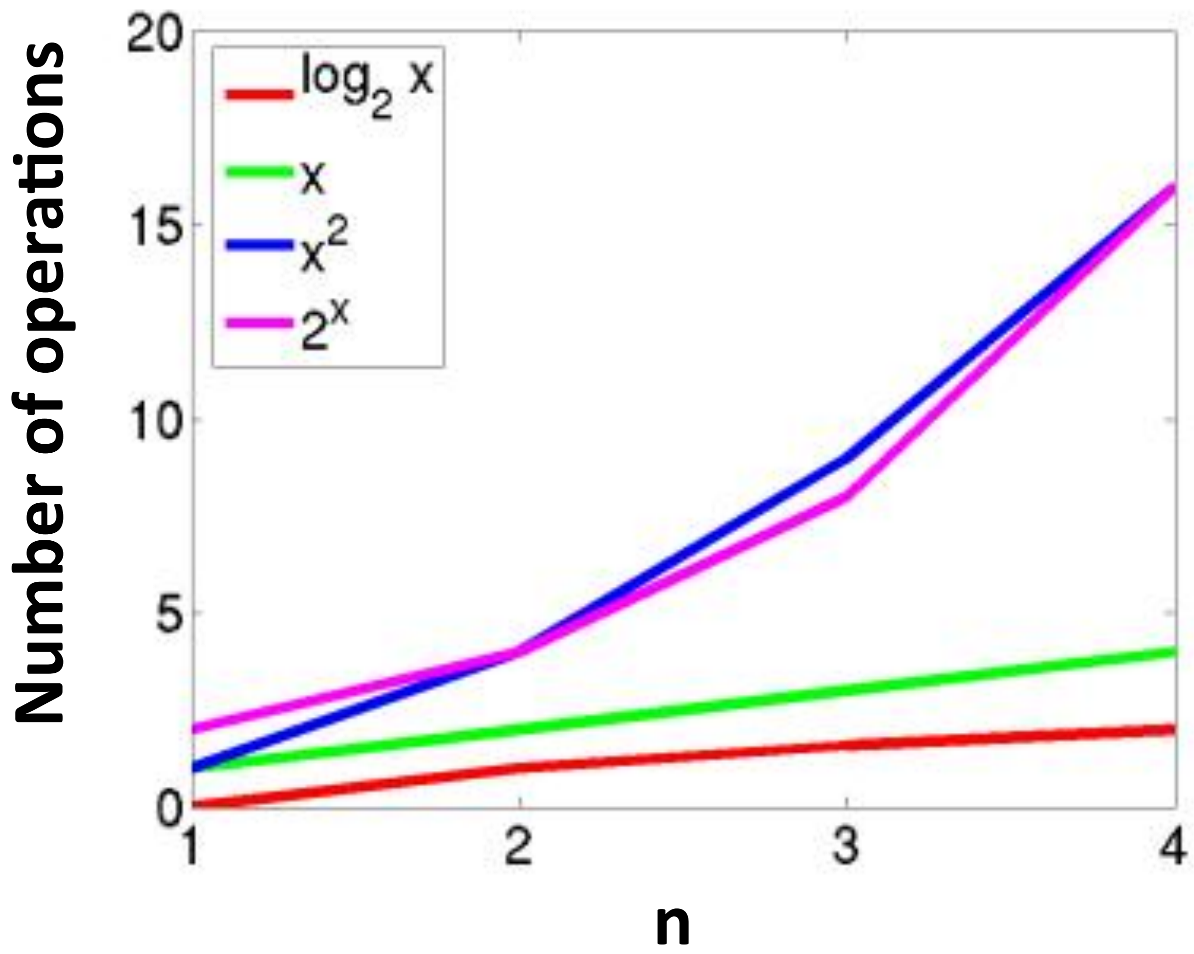
Consider an array of length n

- Computing all possible combinations of items runs in 2^n time

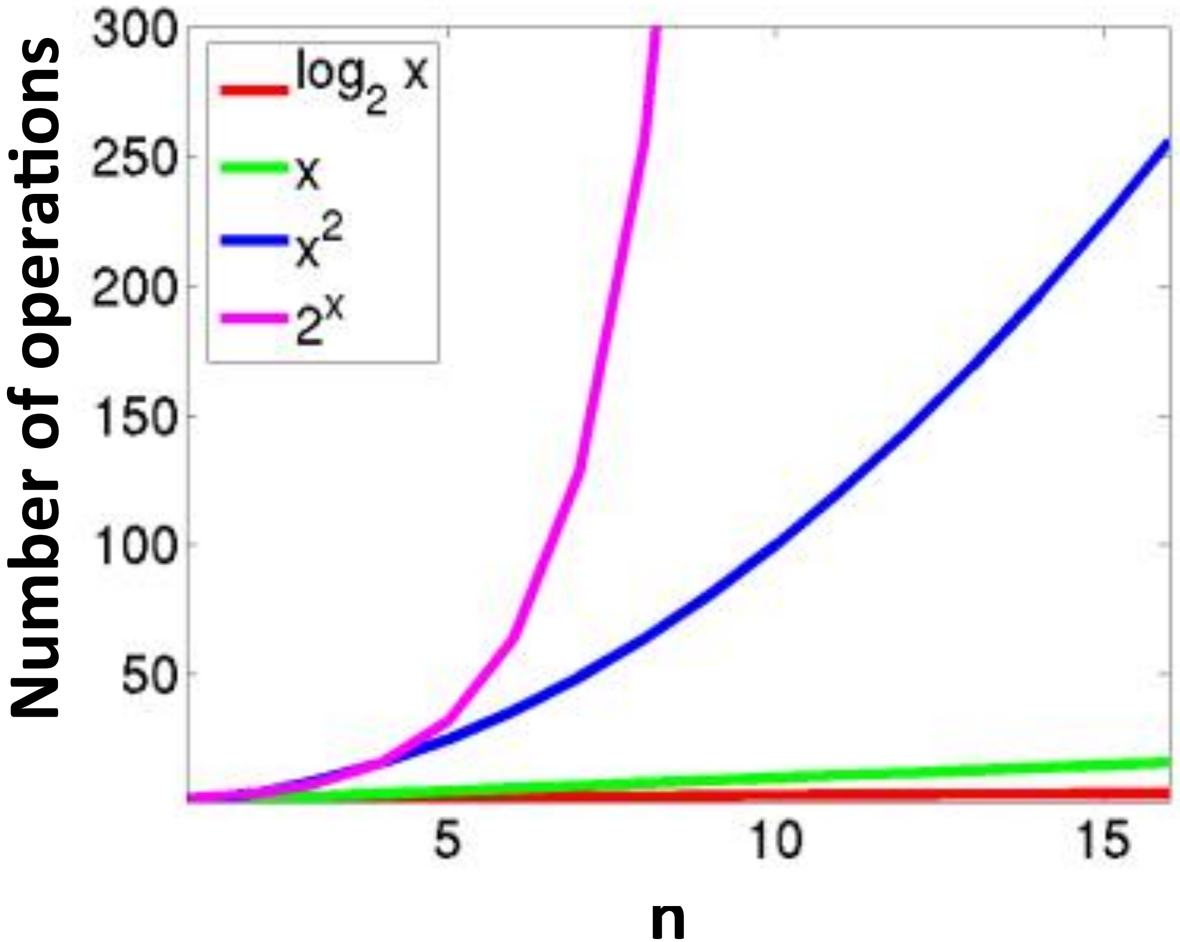
  1) 1,2,3,4,5
  2) 1+2,1+3,1+4,1+5,2+3,2+4,2+5,3+4,3+5,4+5
  3) 1+2+3,1+2+4,1+2+5, …
  4) 1+2+3+4, …
  5) 1+2+3+4+5, …

- Think of all possible moves in chess

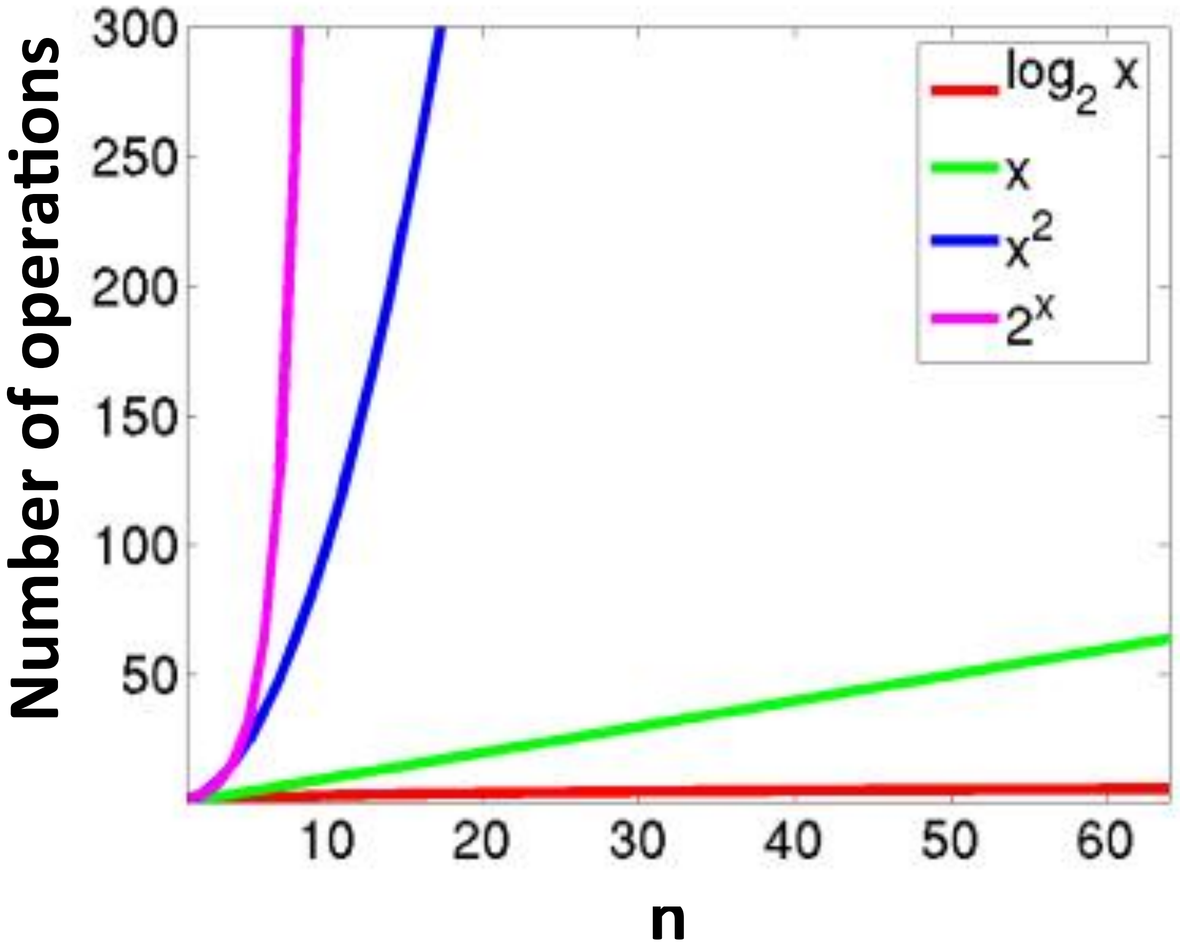
For small numbers of items, run time does not differ by much



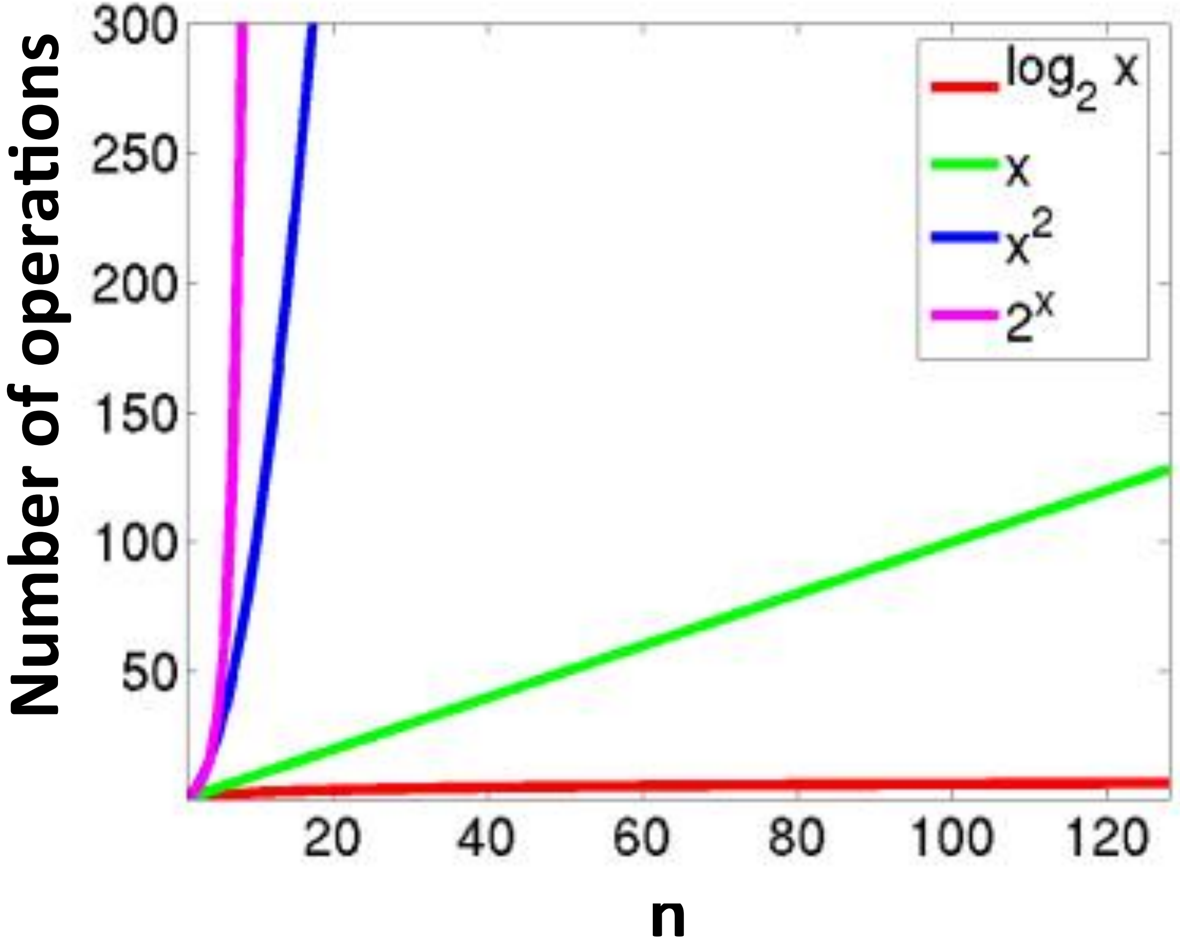
As *n* grows, number of operations between different algorithms begins to differ



Even with only 60 items, there is a large difference in number of operations



Eventually, even with speedy computers, some algorithms become impractical



Sometimes complexity can hurt us, sometimes it can help us

Hurts us
Can't brute force chess
algorithm 2^n

Helps us
Can't crack password
algorithm 2^n

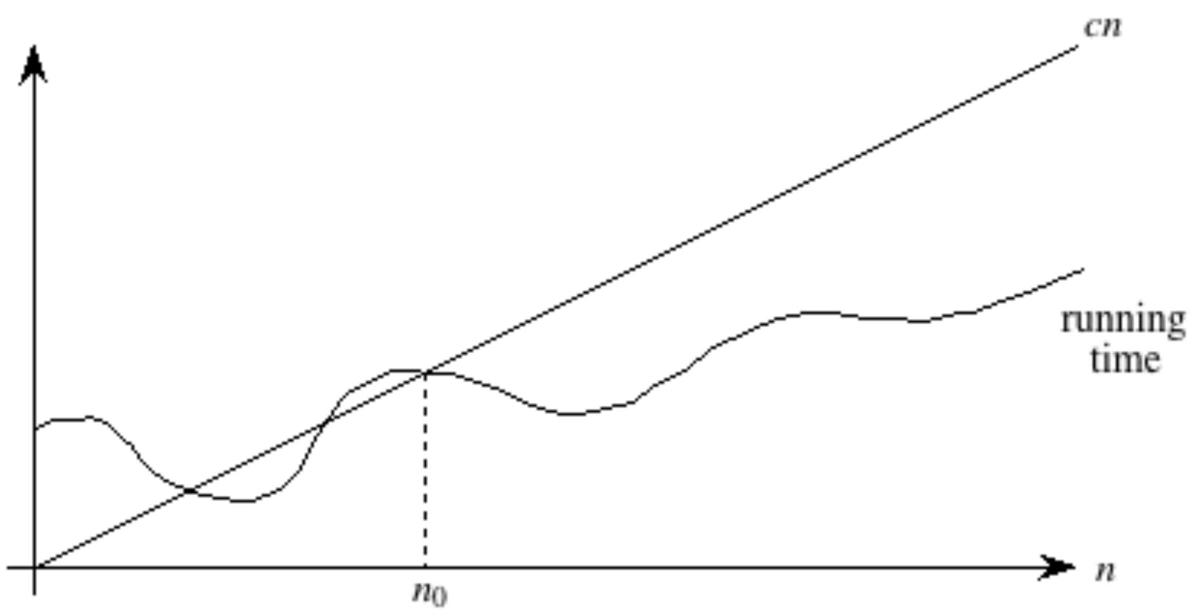
Images: thechessstore.com; studyoffice.org

Agenda

1. Growing array list implementation
2. Orders of growth
3. Asymptotic notation
4. List analysis
5. Iteration

Computer scientists describe upper bounds on orders of growth with "Big Oh" notation

O gives an asymptotic upper bounds

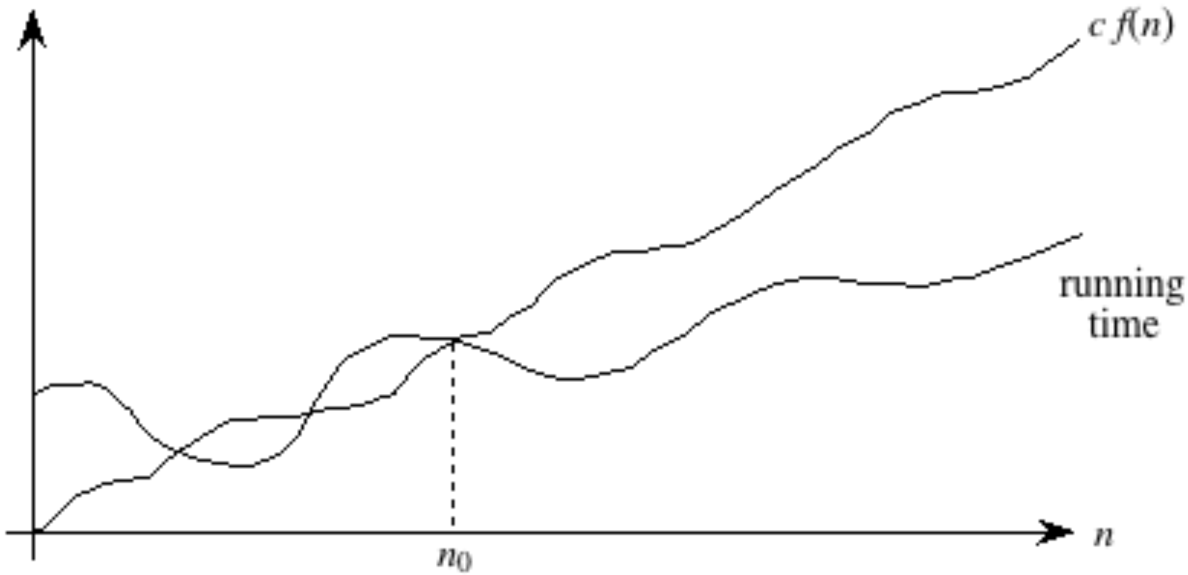


Run time is O(n) if there exists constants n_0 and c such that:

- $\forall n \geq n_0$
- run time of size n is at *most* cn, upper bound
- O(n) is the worst case performance for large n, but actual performance could be better
- O(n) is said to be "linear" time
- O(1) means constant time

We can extend Big Oh to any, not necessarily linear, function

O gives an asymptotic <u>upper</u> bounds



Run time is O(f(n)) if there exists constants n_0 and c such that:

- $\forall n \geq n_0$
- run time of size n is at <u>*most*</u> cf(n), upper bound
- O(f(n)) is the worst case performance for large n, but actual performance could be better
- f(n) can be a non-linear function such as n^2

We focus on upper bounds (worst case) for a number of reasons

Reasons to focus on worst case

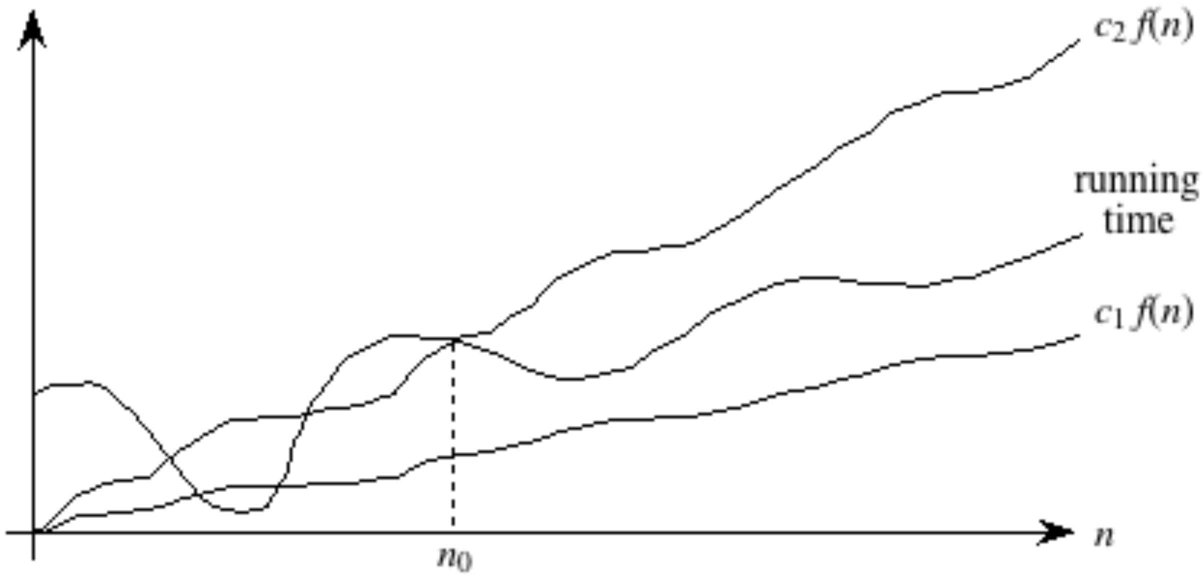
- Worst case gives upper bound on *any* input
- Gives a guarantee that algorithm never takes any longer
- We don't need to make an educated guess and hope that running time never gets much worse

Why not average case instead of worst case?

- Seems reasonable (sometimes we do)
- Need to define what *is* the average case: search example
  - Video database might return most popular items first, so might find popular items before obscure items
  - In cases like linear search, might find item half way (n/2)
  - Sometimes never find what you are looking for (n)
- Average case often about the same as worst case

Run time can also be Ω (Omega), where run time grows at least as fast

Ω gives an asymptotic <u>lower</u> bounds

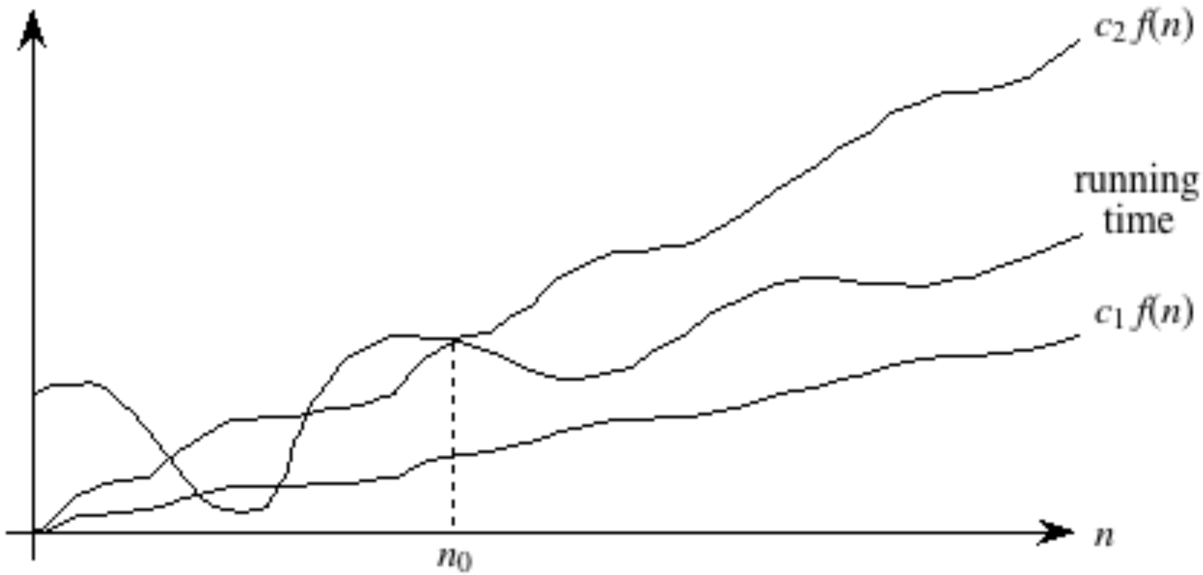


Run time is Ω(n) if there exists constants n_0 and c_1 such that:

- $\forall n \geq n_0$
- run time of size n is at *<u>least</u>* c_1n, lower bound
- Ω(n) is the best case performance for large n, but actual performance can be worse

We use Θ (Theta) for tight bounds when we can define O and Ω

Θ gives an asymptotic tight bounds



Run time is Θ(n) if there exists constants n_0 and c_1 and c_2 such that:

- $\forall\, n \geq n_0$
- run time of size n is at *least* c_1n and at most c_2n
- Θ(n) gives a tight bounds, which means run time will be within a constant factor
- Generally we will use either O or Θ, called asymptotic notation

We ignore constants and low-order terms in asymptotic notation

Constants don't matter, just adjust c_1 and c_2

- Constant multiplicative factors are absorbed into c_1 and c_2
- Example: $1000n^2$ is $O(n^2)$ because we can choose c_1 and c_2 to be 1000 (remember bounded by $c_1 n$ and $c_2 n$)
- Do care in practice – if an operation takes a constant time, O(1), but more than 24 hours to complete, can't run it everyday

Low order terms don't matter either

- If $n^2+1000n$, then choose $c_1 = 1$, so now $n^2 + 1000n \geq c_1 n^2$
- Now must find c_2 such that $n^2 + 1000n \leq c_2 n^2$
- Subtract n^2 from both sides and get $1000n \leq c_2 n^2 - n^2 = (c_2-1)n^2$
- Divide both sides by $(c_2-1)n$ gives $1000/(c_2-1) \leq n$
- Pick $c_2 = 2$ and $n_0 = 1000$, then $\forall n \geq n_0$, $1000 \leq n$
- So, $n^2 + 1000n \leq c_2 n^2$, try with n=1000 get $n^2 + 1000^2 = 2*n^2$
- In practice, we simply ignore constants and low order terms

Agenda

1. Growing array list implementation
2. Orders of growth
3. Asymptotic notation
4. List analysis
5. Iteration

Linked list is O(n), Growing array is O(1) based on amortized analysis

Linked list

- `add/remove/get/set` from front of list, O(1) constant time
- `add/remove/get/set` not at front, might have to march down entire list to find item we want, O(n)
- So worst case is O(n)

Growing array

- `get/set` O(1)
- `add` might cause 2*n memory allocation and copy operation, O(n), or have to move subsequent items O(n)
- `remove()` first element causes all elements to move left to fill hole, O(n)
- Linked list looks better, but is it?

Amortized analysis shows growing array is actually only O(1)!

Amortized analysis

- Imagine for each `add` operation, we charge 3 "tokens", not 1
- One token pays for the current `add`, and two go "in the bank"
- After *n* `add` operations, we will have *2n* tokens in the bank (say *n=10*, then 20 tokens in the bank)
- We will then have to grow the array size by *2n*, and copy *n* items to the new array (last *n* positions in new array are empty)
- We charge *n* tokens to copy the *n* items to the new array (e.g., 10 tokens subtracted from 20 leaves 10 tokens and 10 empty spots)
- So, already "paid" for the empty spaces by charging the 2 extra tokens – one token paid for the copy, one for the empty space
- In the end, we have O(3) for each `add` operation which is O(1)
- Java ArrayList expands 3/2 times, but same result with 4 tokens

Agenda

1. Growing array list implementation
2. Orders of growth
3. Asymptotic notation
4. List analysis
5. Iteration

Its so common to march down a list of items that Java makes it easy with iterators

Traditional for loop

```
for (int i=0;
     i<blobs.size();
     i++) {
   blobs.get(i).step();
}
```

Comments

- i serves no real purpose, don't really care what its value is at any point
- i is reset every time, doesn't keep track of where it was last
- Could lead to $O(n^2)$

Iterator

```
For (Blob b : blobs) {
   b.step();
}
```

Comments

- Easier to read?
- Keeps track of where it left off
- Iterator has two main methods:
  - `hasNext()` can advance?
  - `next()` do advance

We can add our own iterator to the List we previously created

SimpleIterator.java

- Interface for own iterator

SimpleIList.java

- Note the I in the class title
- Same as previous List interface, but adds iterator as public method *newIterator()*

ISinglyLinked.java

- Creates new class called *IterSinglyLinked* that implements *SimpleIterator* from SimpleIList.java
- Instance variable *curr* tracks current position
- Otherwise same as linked list version in SinglyLinked.java

IGrowingArray.java

- Similar to ISinglyLinked.java, but with array implementation

We can add our own iterator to the List we previously created

IterTest.java

- Commented out with Linked list or Growing array
- Add items to 2 different lists of whichever list type is not commented out
- Prints elements using an index (still can do that)
- Checks to see if each element is equal (ugly syntax)
- Prints elements using iterator and *hasNext(), next()* methods
- Checks to see if two lists are equal using iterators
- Run with Linked list
- Run with Growing array
- Book has a fancier version