CSCI 311 Q1 Review

Week 1

1. http://www.ecst.csuchico.edu/~judyc/1516S-csci311/notes/01-intro.html -> *Algorithms & Efficiency*
2. Read Cormen 1.1-1.2

Week 2

1. http://www.ecst.csuchico.edu/~judyc/1516S-csci311/notes/02-insertionSort.html -> *Insertion Sort*
2. http://www.ecst.csuchico.edu/~judyc/1516S-csci311/notes/05-analysis.html -> *Analysis*
3. Read Cormen 2.3, 3.1 and 3.2

week 3

1. http://www.ecst.csuchico.edu/~judyc/1516S-csci311/notes/06-asymptotic.html -> *Merge Sort*
2. Cormen 7.4??

Notes from lecture, week 1

Terminology

- **algorithm:** well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output.
- **problem statement:** specifies in general terms the desired input/output relationship.
- **instance:** the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem
- **correctness:** for every input instance, the algorithm halts with the correct output.
- **data structure:** a way to store and organize data in order to facilitate access and modifications.

Example

- a problem statement (sorting)

```
Input: A sequence of n numbers {a[1], a[2],...,a[n]}
Output: A permutation (reordering) {a'[1], a'[2],...,a'[n]} such that a'[1] <= a'[2] <= ... <=a'[n].
```

- An instance of the problem

```
[31, 41, 59, 26, 41, 58]
```

- The solution

```
{26, 31, 41, 41, 58, 59}
```

Effiency

Growth	n=100	n=1000	n=1x10^6	n=1x10^9
f(n)=lg^n	≈6.644	≈9.966	≈19.932	≈29.897
f(n)=n	≈100	≈1000	≈1000000	≈1000000000
f(n)=n^2	≈10000	≈1000000	≈1 X 10^12	≈1 X 10^18

f(n)=n^3	≈1000000	1000000000	≈1 X 10^18	≈1 X 10^27
f(n)=2^n	≈1.26... x 10^30	≈blah	≈blah	≈blah
f(n)=n!	≈9.33 X 10^157	≈stupidBig	≈galaxySpanning	≈mindAltering

Notes from Cormen, week 1:

1.1 & 1.2

What are algorithms? Why is the study of algorithms worthwhile? What is the role of algorithms relative to other technologies used in computers?

Algorithm: Any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. i.e. An algorithm is a sequence of computation steps that transform the input into the output

Merge sort vs Insertion sort: For the sorting of 100 million numbers, insertion sort takes more than 23 days, while merge sort takes under four hours. In general, as the problem size increases, so does the relative advantage of merge sort.

Example problems:

1.2-2 Suppose we are comparing implementations of insertion sort and merge sort on the same machine, For inputs of size n, insertion sort runs in 8n^2 steps, while merge sort runs in 64nlgn steps. For which values of n does insertion sort beat merge sort?

1.2-3 What is the smallest value of n such that an algorithm whose running time is 100n^2 runs faster than an algorithm whose running time is 2^n on the same machine?

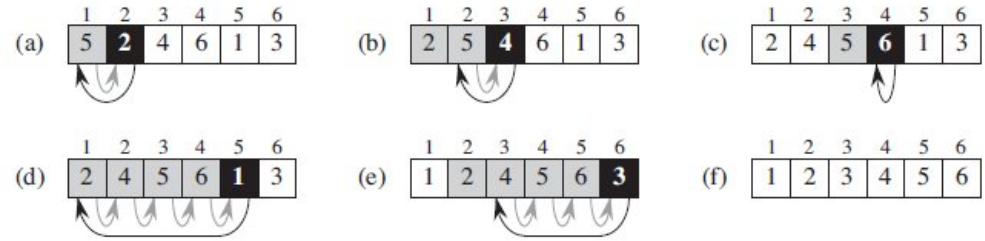
Notes from lecture, week 2

Terminology

- **insertion sort:** an efficient algorithm for sorting a small number of elements by inserting each in sorted order, shifting elements where necessary.
- **keys:** the numbers being sorted are also called keys
- **pseudocode:** whatever expressive method is most clear and concise to specify a given algorithm - could be English, could be close to a programming language syntax, could throw in a math equation even.
- **loop invariant:** a property that holds true immediately before and immediately after each iteration of a loop - used to argue correctness.

Insertion Sort Algorithm

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```



note

- stepping through insertion sort on A = <5, 2, 4, 6, 1, 3>
- array indices on top (they start at 1!)
- black is the key
- gray are elements compared with the key
- arrows show elements

Showing Algorithm Correctness Using a Loop Invariant

a loop invariant is a property that holds true immediately before and immediately after each iteration of a loop - note that this says nothing about its truth or falsity part way through an iteration.

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

This is an example of how to use a loop invariant to argue correctness of the insertion sort algorithm:

Statement: the loop invariant - the subarray to the left of j is always in sorted order

- initialization
  - before the first loop iteration, j = 2
  - the subarray is just one element, A[1], which is sorted
- maintenance
  - a loop iteration shifts elements left of j, A[j-1], A[j-2],..., to the right one position until it finds the proper position for A[j] and inserts it.
  - when j is incremented to j+1, A[1...j-1] is in sorted order
- termination
  - the loop terminates when j > A.length = n
  - so j = n+1 when the loop terminates
  - so A[0..j-1] = A[1..n] (the entire array!) is in sorted order

Correctness proven

Analysis of Insertion Sort

typically we are looking at computational time, although you can also analyze other things. Computational time taken depends on the input, both size and content (the order matters).

We will define running time of a program as a function of the size of the input. definition of input size depends on the problem: number of inputs, number of bits, number of nodes, etc.

Running time is based on the number of steps the algorithm takes - we assume each line of pseudocode executed takes constant time

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

In the nested loop, tj indicates the number of times the given line executes for loop iteration j, you can add these up.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1) .$$

In the best case we assume that the array is already sorted, then tj = 1 during every iteration of the loop.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

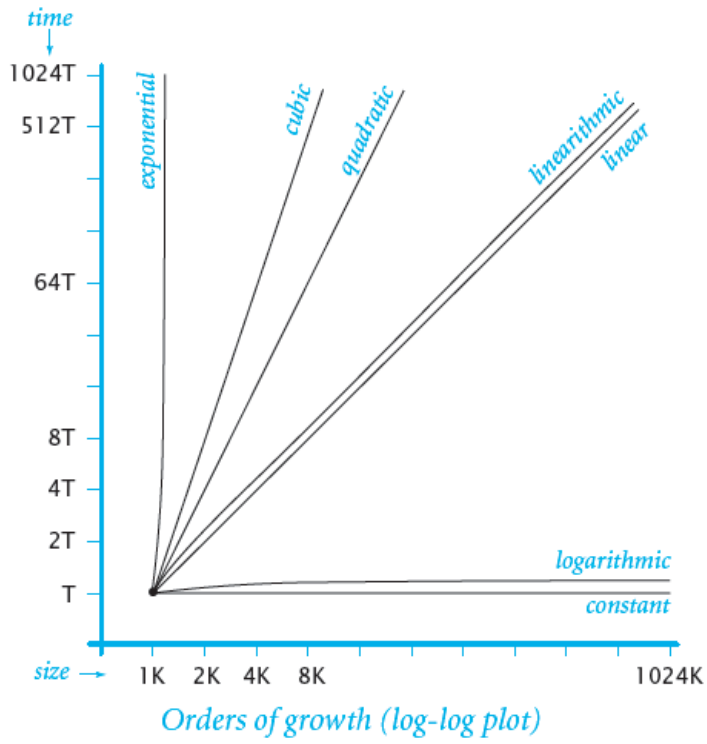
Order of Growth (Big O)

we are typically most interested in the worst case analysis, it gives us an upper bound as the algorithm will never perform worse than this. Some algorithms frequently encounter their worst case input, often the average case is just as bad as the worst case. It is more complicated to derive a formula for the average case, what is an average input?

in addition, we are only interested in the order of the first term of the T(n). We are generally interested in knowing what happens as n gets large, the lower order terms become insignificant.

The worst case running time for insertion sort is O(n^2) - "order n squared". We consider one algorithm to be more efficient than another if its worst case running time has a lower order of growth. **O(1)** is more efficient than **O(n)** is more efficient than **O(n log2 n)** is more efficient than **O(n^2)** is more efficient than **O(n^3)** etc.

growth rate	name	T(2N) / T(N)
1	constant	1
log N	logarithmic	~ 1
N	linear	2
N log N	linearithmic	~ 2
N^2	quadratic	4
N^3	cubic	8
2^N	exponential	T(N)



Orders of growth (log-log plot)

Practice

illustrate insertion sort on A = < a1, a2, a3, a4, a5, a6 > (randomly select six integers for the ai)

write down the entire array after every modification to the array, keep track of changes to key as well

Notes from Cormen, week 2:

2.1 & 2.2

insertion sort: The algorithm sorts the input numbers **in place**; it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when the insertion sort procedure is finished.

Exercises

2.1-: Illustrate the operation of insertion sort on the array A = {31, 41, 59, 26, 41, 58}

2.1-2: rewrite the insertion sort procedure to sort into non-increasing instead of non-decreasing order.

2.2-1: Express the function n^3/1000 - 100n^2 -100n + 3 in terms of Θ-notation.

Notes from lecture, week 3

Terminology

- **input size:** How many of a specific type of input - usually designated 'n' - type depends on algorithm.
- **running time:** The number of primitive steps taken in the execution of an algorithm.
- **worst case running time:** The worst running time of an algorithm on any input of size n.
- **order of growth:** the leading term of the running time formula ignoring the coefficient - a measure of the growth of running time as the input size grows

Divide and Conquer Algorithm

Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

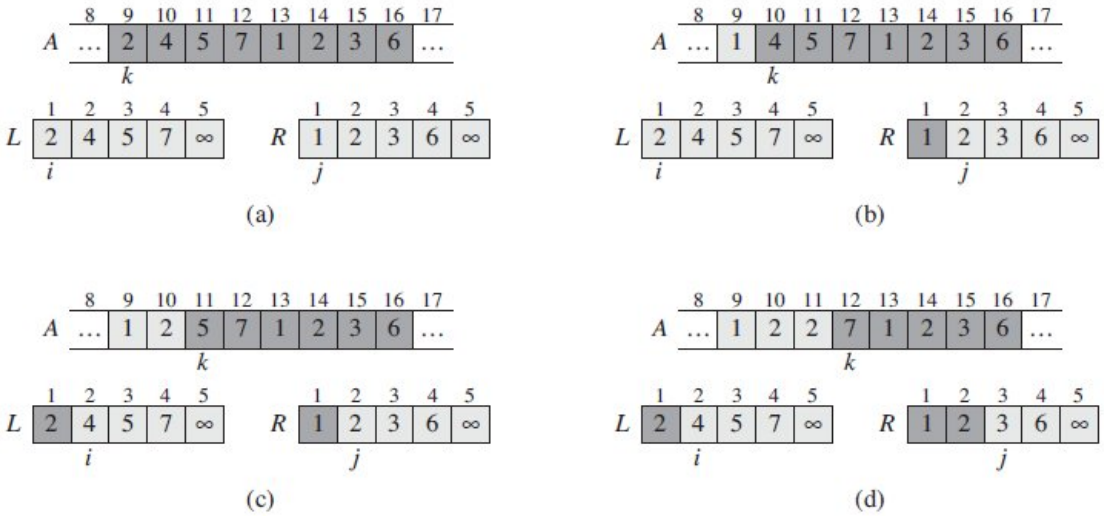
Merge Sort

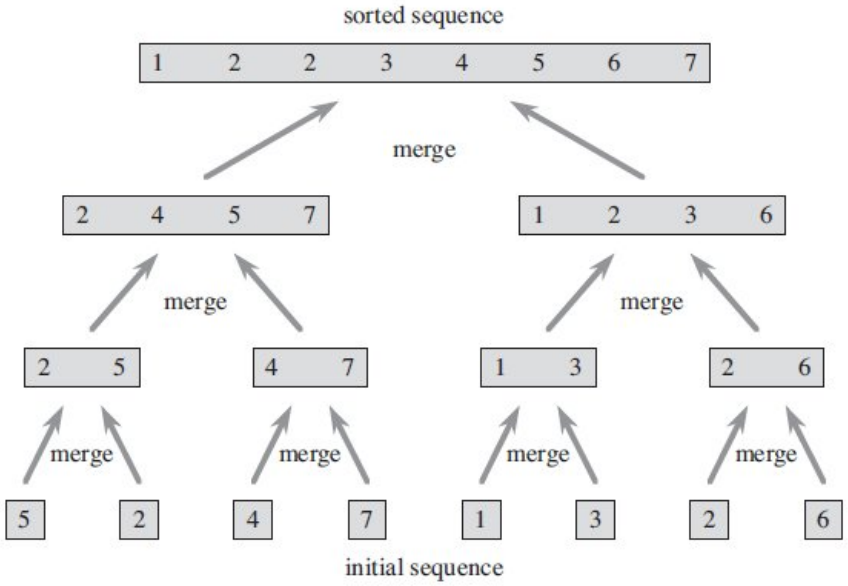
A is the entire sequence to be sorted. P, q, and r are indices - reminder: indexing starts at 1 in this pseudocode. A[p..r] is the subsequence to be sorted

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

The merging of two sorted sequences is the key part of the algorithm

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```





Correctness of Merge Sort

At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Initialization: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p..k-1]$ is empty. This empty subarray contains the $k-p=0$ smallest elements of L and R, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \le R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p..k-1]$ contains the $k-p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p..k]$ will contain the $k-p+1$ smallest elements. Incrementing k (in the **for** loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

Termination: At termination, $k = r+1$. By the loop invariant, the subarray $A[p..k-1]$, which is $A[p..r]$, contains the $k-p=r-p+1$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. The arrays L and R together contain $n_1+n_2+2 = r-p+3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

Analysis of Merge Sort

Terminology

- **recursive algorithm:** To solve a given problem, the algorithm calls itself recursively one or more times to deal with closely related subproblems
- **recurrence equation:** Describes the running time on a problem of size n in terms of the running time on smaller inputs
- **recursion tree:** A diagram that looks like an upside-down tree showing each level of recursive calls - can be used to visualize running time

Analysis of Divide and Conquer Algorithms

This approach works on many recursive (like divide and conquer) algorithms. A recursive algorithm is one that calls itself we can define its run time using a recurrence equation define the run time on input of size n in terms of the run time on smaller inputs we have mathematical tools to solve recurrences.

let T(n) be the run time on input of size n. If n is small the solution will take constant time. We divide the problem into subproblems of size n/b (a=b=2 in merge sort). It takes T(n/b) to solve one subproblem and aT(n/b) to solve all of them. Suppose it takes D(n) time to divide the problem and C(n) time to combine solutions then here is the recurrence equation

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

If we apply this algorithm to merge sort specifically we get:

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

- merge sort is a recursive algorithm
- we define T(n) as a recurrence equation
- assume n is a power of 2 and the divide step results in two subsequences of size n/2
- divide: divide step takes constant time, so D(n) = θ(1)
- conquer: recursive solve of two subproblems takes 2T(n/2)
- combine: the merge step is linear in n or C(n) = θ(n)
- D(n) + C(n) = θ(1) + θ(n) = θ(n)
- This simplifies to θ(n lg n) for the worst case

Asymptotic Notation

the primary use is to describe the running times of algorithms can also be used to characterize functions of other things like memory requirements asymptotic notation is a way to describe the behavior of functions in the limit we're studying asymptotic efficiency describes the growth of functions as a function of the size of the problem we focus on what is important by abstracting away low-order terms and constant factors all of the functions are assumed to be asymptotically positive gives us a way to compare "efficiency" of functions:

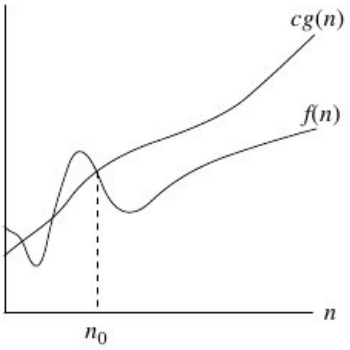
Cormen: When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the asymptotic efficiency of algorithms. Usually an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

O ≈ ≤ big Omicron or "big O" notation Ω ≈ ≥ big Omega Θ ≈ = big Theta o ≈ > little omicron or "little o" notation ω ≈ < little omega

big Omicron	big Omega	big Theta	little omicron	little omega	
O ≈ ≤	Ω ≈ ≥	Θ ≈ =	o ≈ >	ω ≈ <	
upper bound (includes worst case)	lower bound (includes best case)	upper and lower bound (is in between two other cases, i.e. tight bound)	less than (but not best case)	greater than (but not worst case)	

O-notation

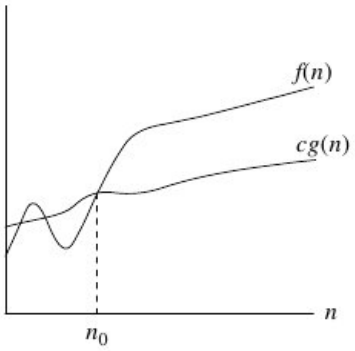
$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Ω-notation

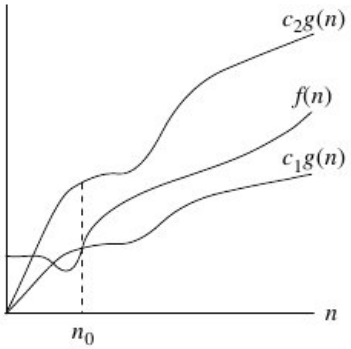
$\Omega(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that
$0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Θ-notation

$\Theta(g(n)) = \{f(n) :$ there exist positive constants c_1, c_2, and n_0 such that
$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Accumulated Vocab

- **algorithm:** well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output.
- **problem statement:** specifies in general terms the desired input/output relationship.
- **instance:** the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem
- **correctness:** for every input instance, the algorithm halts with the correct output.
- **data structure:** a way to store and organize data in order to facilitate access and modifications.
- **insertion sort:** an efficient algorithm for sorting a small number of elements by inserting each in sorted order, shifting elements where necessary.
- **keys:** the numbers being sorted are also called keys
- **pseudocode:** whatever expressive method is most clear and concise to specify a given algorithm - could be English, could be close to a programming language syntax, could throw in a math equation even.
- **loop invariant:** a property that holds true immediately before and immediately after each iteration of a loop - used to argue correctness.
- **input size:** How many of a specific type of input - usually designated 'n' - type depends on algorithm.
- **running time:** The number of primitive steps taken in the execution of an algorithm.
- **worst case running time:** The worst running time of an algorithm on any input of size n.

- **order of growth:** the leading term of the running time formula ignoring the coefficient - a measure of the growth of running time as the input size grows
- **recursive algorithm:** To solve a given problem, the algorithm calls itself recursively one or more times to deal with closely related subproblems
- **recurrence equation:** Describes the running time on a problem of size n in terms of the running time on smaller inputs
- **recursion tree:** A diagram that looks like an upside-down tree showing each level of recursive calls - can be used to visualize running time