Design and Analysis of Algorithms - CS 323
Lecture #3 – February 17, 2016

Notes by: Floyd Brown

Chapter 7 - Sorting Algorithms

Announcements

Sorting Terms:

Inversions – two elements out of order with respect to one another i<j.

Minimum :	= 0
Average:	= n(n-1)/4
Maximum:	= C(n,2) = n!/2!(n-2)!
Maximum	= C(n,2) = n(n-1)/2

Stable Sort – A sort that preserves the good order of non-inverted sort
Exchange – Swapping Elements

Internal sort – An **internal sort** is any data sorting process that takes place entirely within the main memory of a computer. **(Reference Chapter 7, p 223 of course text) & https://en.wikipedia.org/wiki/Internal_sort)**

We will focus on Internal Sort

Homework #2 Review

The first three questions pertain to the recurrence g(0) = 2, g(1) = 1, g(n) = g(n-1) + 2g(n-2)

1. Using the method of characteristic equations presented in class, derive the closed-form formula $g(n) = 2^n + (-1)^n$.
   Solution:
   How to setup a characteristic equation
   g(0)=2
   g(1)=1
   g(n)=g(n-1)+2g(n-2)

$g(n)-g(n-1)-2g(n-2)=0$
$X^n-X^{n-1}-2X^{n-2}=0$
$X^{n-2}(X^2-X-2)=0$ (Review foil factoring)
The solution is a linear combination of all three factors but 0 is not relevant

$$g(n)=a\cancel{0^n}+a2^n++b(-1)^n=a2^n+b(-1)^n$$

$(X2-X-2)=0$
$(X-2)(X+1)=0$
$X=2, X=-1$

For a linear homogenous solution use case $g(0)$
$g(0)=a2^0+b(-1)^0 = a+b =2 \quad a=2-b$ (i)
$g(1) = a2^1+b(-1)^1 = 2a-b = 1, 2a=1+b$ (ii)
combining (i) and (ii) $3a=3, a=1$
$b=2-a$ (from (i)), $b=1$

$g(n)=2^n+(-1)^n$

2. Using mathematical induction, prove that the formula $g(n) = 2^n + (-1)^n$ works for all cases $n \geq 0$.
Solution:
Base case:
$n=0$
$g(0) = 2^0 +-(1)^0=2$
$2 \geq 0$ ✔

Inductive Hypothesis
$g(k) = 2k +(-1)k$ for $0\leq k\leq n$
$g(k+1) = 2^{k+1}+(-1)^{k+1}$
$g(k+1) = g(k) + 2g(k-1)$
$= 2^k + (-1)^k + 2[2^{k-1} + (-1)^{k-1}]$
$= 2^k + (-1)^k + 2^k – 2 (-1)^k$
$= 2^{k+1} + (-1)^k + 2^k$
$-1^{(k-1)} = (-1)^{k+1}$

Question: What is Strong Induction?
Answer: Strong induction means we are assuming the hypothesis is true for all cases of n.

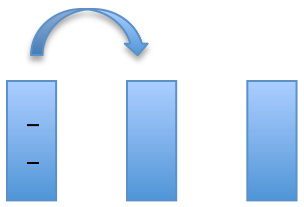
3. Using the logarithm-based approximation method discussed in class, estimate how many digits are in g(1000). Then use an on-line calculator to find out the actual number of digits in g(n).

Solution:
$g(n)=2^n+(-1)^n$
$g(1000) = 2^{1000} + (-1)^{1000}$

$\log_2 10 = \log_{10} 10/\log_{10} 2 \quad \rightarrow 2^n$
$(2^x)^{n/x} = 2(^{3.32})^{n/3.32}$
$n/3.32 = 1000/3.32 = 301$
$g(n) = 301$

The next four questions relate to using basic data structures for sorting.
(Do not use an array or other data structure besides the one indicated.)

4. Suppose n unsorted elements are in a **stack**. Describe how to sort the data, using auxiliary stacks as necessary. What is the time complexity of your algorithm?
Solution:
Sorting through a stack - The difficulty is that you don't know where max is on the stack. If the next value is bigger than the temp max, move it to tempmax and push it to stack. Iterate through each stack element and put that element at the top.



Is this **approach stable**? No, because existing order is not preserved. There is instability because the stack gets out of order.

What is the **time complexity** or order of magnitude? $O(n^2)$

Some students attempt to sort the stack with only one additional axillary stack. The challenge was in keeping track of element that is popped and stored in memory. But the time complexity may grow in managing the operation.

5. Suppose n unsorted elements are in a **queue**. Describe how to sort the data, using auxiliary queues as necessary. What is the time complexity of your algorithm?

Solution:
Wrap around queue is tracked with size or memory. Track with First variable you check. The **space complexity** is n.
Time complexity is (n-1) comparisons * n operations $=n^2$

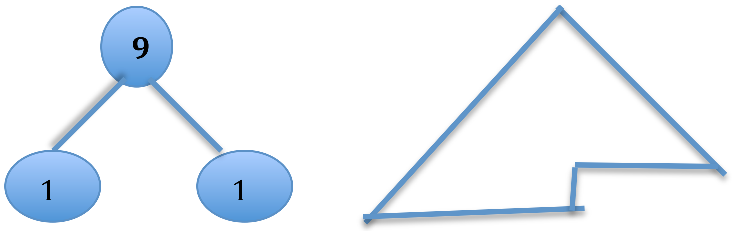
6. Suppose n unsorted elements are in a "binary search tree" (BST), defined as a binary tree in which the keys in the left and right children are respectively smaller and larger than the key in their parent node. Describe how to sort the data, using auxiliary BSTs as necessary. What is the time complexity of your algorithm?

   Solution:
   Traversing BST can recursively go parent, left child, right child
   Where left < right

   We do inorder traversal
   inorder(Left Sub Tree), root, inorder(Right Sub-Tree)
   with **time complexity** of O(n)

7. Suppose n unsorted elements are in a "heap", defined as a binary tree in which the keys in the left and right children are *both* larger than the key in their parent node. Describe how to sort the data, using auxiliary heaps as necessary. What is the time complexity of your algorithm?

   Solution:



   A **heap** is a binary tree where each node is a key and the root is the smallest value. The children of the parent nodes in a heap are bigger than the parent. A heap is balanced. All the levels are populated consecutively left to right except for the last level.
   We'll set the smallest element stored at the root. Use the restore heap propery. Promote one of the smaller child to the root. Then bubble all other values up. This is called heapify, move the rightmost element to fill empty spot.
   Time Complexity
   Removal O(1)
   Sort, Bubble up O(log n)
   O(n log n)

As heap gets smaller $\log n + \log(n-1) + \log(n-2) \approx \log n$

8. Given the various possible shapes of a binary tree, what are the best, average and worst case number of operations for “preorder” traversal? Why?

Best = $O(n)$
Average = $O(n)$
Worst = $O(n)$

Why are the Time complexities the same? You have to travers through every node once.

Lecture

Asymptotic Notation Review (Section 3.4 of course text)

O, big Oh $f(n) = O(g(n))$ iff there exist, n_o element N & c element R^+ such that for all $n \geq n_o$ $f(n) \leq c(g(n))$

Means f(n) is bounded by a multiple of g(n). The multiple of g(n) could be less than 1.

For example $1000n^2 = O(n^2+5)$ (i)
$1000\ n^2 + 3n + 10 = O(10n^2 + 400n + 40)$ (ii)

In order to argue that (i) is bigger than (ii) find n_o and find c, where $f(n) \leq c(g(n))$

Tip: Work with c then find n_o.

Note inifite choices
Pick c = 100
$100(10n^2+400\ n + 40) = 1000\ n^2 + 40000\ n + 4000$ (iii)

(iii) > (ii), pick n_o=1

Ω Big Omega – $f(n) = \Omega(g(n))$ iff there exist n_o element in N and c member of R+ such that for all $n \geq n_o$ $f(n) \geq c(g(n))$

$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))\ \&\ g(n) = O(f(n))$
$\Leftrightarrow f(n) = O(g(n)\ \&\ f(n) = \Omega(g(n))$

$f(n) = \Theta(g(n)$ iff there exist no element in N and C element in R+ such that for all $n \geq n_o$,
$c_2\, g(n) \leq f(n) \leq c_1\, g(n)$

example – looking back at $O(g(n))$ $\Omega(g(n)$
c1 = 100, what c2 to use?
$10n^2 + 400n+40 \leq c2\ (1000n^2 + 3n +10)$, c2=150

Sorting

How do you find the minimum and maximum of an array

```
Temp min =a[0]
For i=1 to n-1  //there are [n-1] comparison operations
If a [i] < tempmin
Tempmin = a[i]
```

If we need the min and max, how can we improve this algorithm by doing things simultaneously?

Two Search Techniques

Linear Search – Iterate through one element at a time through the array and compare values to the key.

Time Complexity Analysis

i) Best Case
   O(1) // one operation
ii) Average Case
   (the average depends on data. There n/2. (Average is affected by distribution if k exist in the array)
iii) Worst Case
   O(n) operations

Time (n) – Time complexity for a **linear search** (a **decrease and conquer**) algorithm where size of problem is decrease a little bit, (decreased by one), as we search for the key element….aka decrease by one and conquer

$T(1) = 1$ (i)
$T(n) = (n-1) +1$ (ii) //a problem of n-1
Worst time
How do we solve (i) and (ii)? Expand out (ii)

<u>Method 1</u>

T(n-1) +1 = [T(n-2)] +1 = T(n-3) + 1 +1 +1 ....=T(1) + n-1 = 1+ (n-1) =n.

<u>Method 2</u>

T(n) – T(n-1) =1
T(n-1) – T(n-2)=1
|
|
plug in n-1 for each n
|
|
T(2) – T91) =1
n-1+1=n

What are the left and right sides?
We have combination on the left
like a geometric series

Telescoping

The big equation collapse to
(n-1) 1 = n-1

You have an array that is already sorted.
Take the floor (integer below) or take the ceiling (Integer above).

$$\left\lceil \frac{\text{Low + high}}{2} \right\rceil$$

This is a decrease by conquer because the problem gets smaller using recursion for finding midpoints.

The implementation problem with binary search is that you are doing comparisons going left to right. There are two comparisons comparing the key in the position or compare index position of an array, did we go out of bounds in the array? We don't count the bounds because we are comparing data not an inexpensive index location. The key comparisons are more expensive to index comparisons.

A compare to method in Java returns -1, 0, 1 for an object. A smart comparison tells us if the value is to the left or right, the difference is an order of n comparison, -n, 2n (Binary search coming up)

Do simultaneous calculation examining if key is min or max to some somparative value.
Keep in mind, can we get away with one comparison for each operation?

T(1) = 1   //I operation & "T(1)" symbolizes one comparison
T(n) = T(n/2) +1 (i)   //"n/2 " represents " half-way (midpoint) of the array.
                              //1/2 data is looked at in the array we are sorting

How do we solve (i)? We could expand it out

Method 1

$T(n) = T(n/4) + 1 + 1$
$T(n) = T(n/8) + 1 + \quad 1$
|
|
$\lfloor \log 2\ n + 1 \rfloor$
$T(n) =$

Method 2

Use domain transformation of a function – use inputs Assume number of data elements is a power of 2.

$n=2^k \Rightarrow \quad T(n) = T(2^k)$
Transformation follows – Think of T(n) as a function of k T(k)

$T(n) = T(2^k) = S(k)$ //think in terms of function k
$S(k) + S(k-1) + \ldots + S(1) + S(0)$

$T(n/2) = T(2^{k-1}) = S(^{k-1})$

$S(0) = T(1) = 1$
$S(k) = S(k-1) + 1$
$S(k) = K + 1$
$T(2^k) = K + 1$
$T(n) = K + 1$

$T(n) =$

$\lfloor \mathrm{Log}_2\ n \rfloor$

$T(n) = k+1 \quad = \lfloor \log_2 n \rfloor \quad +1$

We cannot assume k is always power of two for every array.
What if m is not a power of two? We then add more elements to the array

Array of m.lenght()
$2^k<m<^{2k+1}$

We add more data to m that is not our search key…we make the array a power of 2 as a result. The work or time complexity is affected by the increase of these fillers as if we have
$2^6=64$ size array

but we have 65 element data set, double array to accommodate $2^7 = 128$ size array which is 63 more than 65 will increase the amount of actual work by 1.

What is T(1)?
T(1)=S(k)
S(k) = S(k-1) +1 …..we go to S(0), so +1

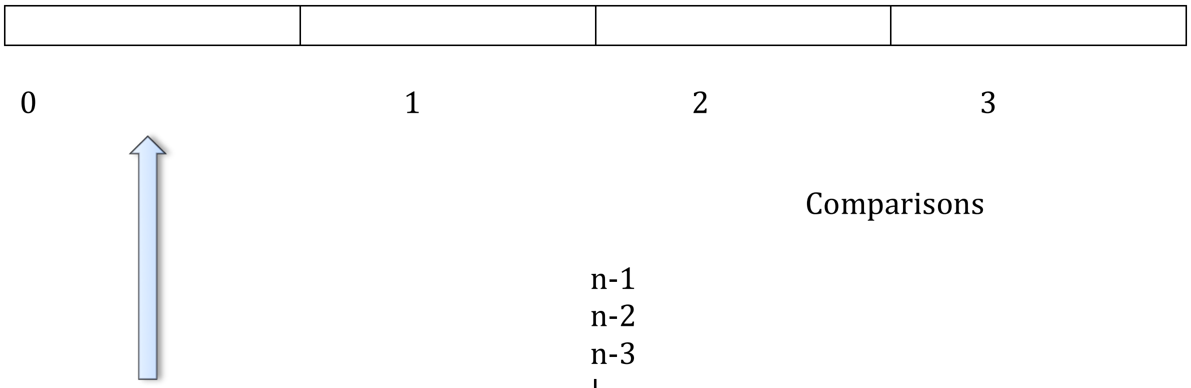
The impact of doubling the data is only one extra comparison.

Sorting

Bubble Sort

1) Compares adjacent elements
2) Swaps elements that are out of order

i-- ←



j=0 -> j=j++

Comparisons

n-1
n-2
n-3
|
|
0
total comparisons = n(n-1)2

```
for (i=n-1; i>0; i--){
        for (j=0; j< I; j++){
if (a[j] > a a[j+1]
temp = a[j]
a[j] = a[j+1] //swap
a[j-1] = temp
```

We are counting swaps and comparisons… swapping can be expensive if data is moved but inexpensive if only cost is moved to a pointer.

<u>Comparisons of Time Complexity of Bubble Sort</u>

Best:	n(n-1)/2
Avg:	n(n-1)/2
Worst:	n(n-1)/2

<u>Swap Complexity</u>

Best case:	0
Avg:	(1/2 elements in the array are swapped) n(n-1)/4
Worst:	n(n-1)/2

Note for homework: If you go through and not making swaps after going though the inner "for" loop, you can exit early. What is the probability it is initially sorted? i/n!

Place a conditional check if swap occurred after the inner for loop

Next class Selection, insertion and merge sort