CS 410 Summer 2000
Solutions to Homework 3

1. *Consider the following 3 sets of data:*

   (a) 1 2 3 4 5

   (b) 5 4 3 2 1

   (c) 3 1 4 2 5

   Show step by step how each of these data sets will be sorted by each of insertionsort, mergesort, and heapsort, respectively. Show what gets changed after the execution of each step. Use the algorithms as detailed in CLR. Create a table which gives the total number of array element comparisions made in each case.

   Insertion sort: For each of the three data sets, the following table shows the ordering of the array before and after each change, and the comparisons made while the array is in that configuration (order). Each comparison is shown as a pair. For example (1,2) indicates that the value 1 is compared with the value 2.

Algorithm	**Data Set**	**Array**	**Comparisons**
Insertion sort			
	(a)	1 2 3 4 5	(1,2), (2,3), (3,4), (4,5)
			Total: 4 comparisons are made
	(b)	5 4 3 2 1	(5,4)
		4 5 3 2 1	(5,3)
		4 5 5 2 1	(4,3)
		4 4 5 2 1	
		3 4 5 2 1	(5,2)
		3 4 5 5 1	(4,2)
		3 4 4 5 1	(3,2)
		3 3 4 5 1	
		2 3 4 5 1	(5,1)
		2 3 4 5 5	(4,1)
		2 3 4 4 5	(3,1)
		2 3 3 4 5	(2,1)
		2 2 3 4 5	
		1 2 3 4 5	
			Total: 10 comparisons are made
	(c)	3 1 4 2 5	(3,1)
		3 3 4 2 5	
		1 3 4 2 5	(3,4), (4,2)
		1 3 4 4 5	(3,2)
		1 3 3 4 5	(1,2)
		1 2 3 4 5	(4,5)
			Total: 6 comparisons are made

Heap sort: the following table contains the same information as the table above for insertion sort. Comparisons preceded by the initials "bh" indicate this array configuration was considered during *heapsort*'s call to *buildheap*.

Algorithm	**Data Set**	**Array**	**Comparisons**
Heap sort			
	(a)	1 2 3 4 5	bh: (2,4), (2,5)
		1 5 3 4 2	bh: (1,5), (1,3)
		5 1 3 4 2	bh: (1,4), (1,2)
		5 4 3 1 2	
		2 4 3 1 5	(2,4), (2,3)
		4 2 3 1 5	(2,1)
		1 2 3 4 5	(1,2), (1,3)
		3 2 1 4 5	
		1 2 3 4 5	(1,2)
		2 1 3 4 5	
		1 2 3 4 5	
			Total comparisons = 12
	(b)	5 4 3 2 1	bh: (4,2), (4,1), (5,4), (5,3)
		1 4 3 2 5	(1,4), (1,3)
		4 1 3 2 5	(1,2)
		4 2 3 1 5	
		1 2 3 4 5	(1,2), (1,3)
		3 2 1 4 5	
		1 2 3 4 5	(1,2)
		2 1 3 4 5	
		1 2 3 4 5	
			Total comparisons = 10
	(c)	3 1 4 2 5	bh: (1,2), (1,5)
		3 5 4 2 1	bh: (3,5), (3,4)
		5 3 4 2 1	bh: (3,2), (3,1)
		1 3 4 2 5	(1,3), (1,4)
		4 3 1 2 5	
		2 3 1 4 5	(2,3), (2,1)
		3 2 1 4 5	
		1 2 3 4 5	(1,2)
		2 1 3 4 5	
		1 2 3 4 5	
			Total comparisons = 11

Merge sort: For each of the three data sets, the following table shows the calls to *merge-sort* and the calls to *merge*, each with the parameters they are called with (*merge-sort* is called with an array, and *merge* is called with two arrays). Each call to *merge* is annotated with the resulting array (OK, this part **was** redundant), as well as the comparisons made to produce the merged array. Comparisons are shown as in the previous tables.

Algorithm	**Data Set**	**Subroutine call**	**Array**	**Comparisons**
Merge sort				
	(a)	1 2 3 4 5		
		merge-sort(1,2,3)		
		merge-sort(1 2)		
		merge-sort(1)		
		merge-sort(2)		
		merge({1}, {2})	1 2	(1,2)
		merge-sort(3)		
		merge({1,2}, {3})	1 2 3	(1,3), (2,3)
		merge-sort(4,5)		
		merge-sort(4)		
		merge-sort(5)		
		merge({4},{5})	4 5	(4,5)
		merge({1,2.3}, {4,5})	1 2 3 4 5	(1,4), (2,4), (3,4)
				Total comparisons=7
	(b)	5 4 3 2 1		
		merge-sort(5 4 3)		
		merge-sort(5 4)		
		merge-sort(5)		
		merge-sort(4)		
		merge({5}, {4})	4 5	(5,4)
		merge-sort(3)		
		merge({4,5}, {3})	3 4 5	(4,3)
		merge-sort(2,1)		
		merge-sort(2)		
		merge-sort(1)		
		merge({2},{1})	1 2	(2,1)
		merge({3,4,5}, {1,2})	1 2 3 4 5	(3,1), (3,2)
				Total comparisons= 5
	(c)	3 1 4 2 5		
		merge-sort(3,1,4)		
		merge-sort(3,1)		
		merge-sort(3)		
		merge-sort(1)		
		merge({3}, {1})	1 3	(3,1)
		merge-sort(4)		
		merge({1,3}, {4})	1 3 4	(1,4), (3,4)
		merge-sort(2,5)		
		merge-sort(2)		
		merge-sort(5)		
		merge({2},{5})	2 5	(2,5)
		merge({1,3,4}, {2,5})	1 2 3 4 5	(1,2), (3,2), (3,5), (4,5)
				Total comparisons= 8

Summary of the number of comparisons:

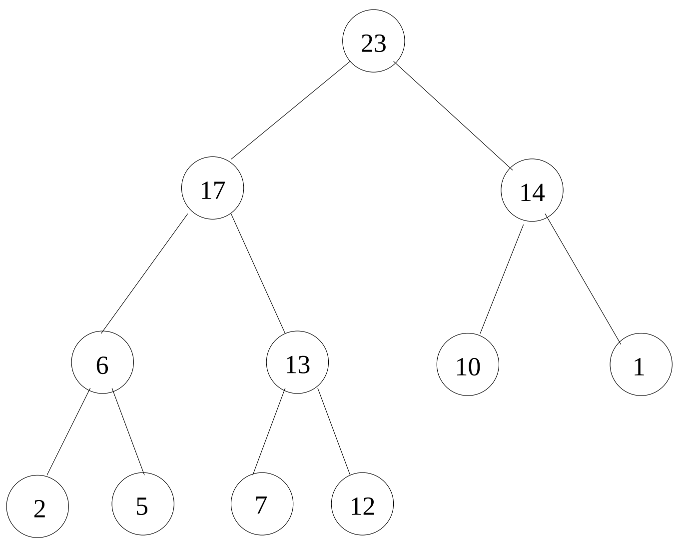
Algorithm	**Data Set:**	**(a)**	**(b)**	**(c)**
Insertion sort		4	10	6
Heapsort		12	10	11
Mergesort		7	5	8

Interesting observations: Although heapsort is asymptotically much faster than insertion sort, on small data sets it actually performs more comparisons (due to a larger constant).

For data with 5 distinct values (as we had here), both heapsort and mergesort actually do fewer comparisons on reversed data than on ordered data. For heapsort, this will occur for all input sizes because there is less work involved in building a heap (the data is already a valid heap). For mergesort, it is because each time there is an odd number of values combined between the two arrays in a call to merge, the second array will be smaller, and in the case of reversed input all of the smaller values will be in the smaller sized array, so this array will become empty more quickly.

2. *Is the sequence* $\langle 23, 17, 14, 6, 13, 10, 1, 2, 5, 7, 12 \rangle$ *a heap?*

   The given sequence is a heap, as shown below.



3. *In the* Partition *subroutine of* Quicksort*, it would be problematic if the pointer j (which starts at the back) ended pointing to the last element of the array, since then the two "subarrays" formed would be of size n and 0, respectively. Explain carefully why this can never happen with the pseudocode given.*

   (refer to *Partition* algorithm on page 154)

   Summary: Need to show *Partition* won't terminate with $j == r$.
   In the first iteration, pointer i always stops at p, i.e. at the pivot element which is in the first position. If pointer j passes position r (going to the left) it will never go back. If in the first pass pointer j stops at position r, then items in positions p and r (which are distinct positions) will be swapped, after which pointer j will begin the next iteration at position $r - 1$.

(a) When a call to $Partition(A, p, r)$ is made we have $p < r$ (p is strictly less than r).

(b) Initially we have $i = p - 1$ and $j = r + 1$

(c) When line 7 is executed we have $i == p$ and hence line 8 is true and control flow comes out of the repeat-until loop. So, the front pointer ends up pointing to the pivot element in the first position.

(d) If $j < r$ (back pointer is forward of last position) then we have nothing to prove. Therefore case $j == r$ (back pointer ends first iteration of *partition* pointing at last element).
Since $i == p$ and $p < r$, line 10 is executed, swapping the first and last elements, and control goes back to line 5 and $j = j - 1$ is executed moving the back pointer forward, and hence $j < r$

4. *CLR 8.3 on page 169.*

(a) proof of correctness (by induction)

(base case) easy to see that it sorts correctly when the length of input is 1 and 2

Induction Hypothesis (IH): Stooge-sort sorts correctly for input of any size between 1 and $n - 1$.
(induction step) Consider input of size $n : A = (a_1, a_2, ...a_n)$.
Consider line 6 (stooge-sort(A,1,2n/3)). since $2n/3 < n$, by induction hypothesis the 2n/3 numbers $B = (a_1, a_2, ..., a_2n/3)$ are sorted after the call to stoogesort. Now in the first one-half elements of B (the first one-third elements of A) can never be a part of the last one-third elements in the sorted order of A, since we know that the last one-half elements of B are all larger than these. Therefore the second call to stooge sort on line 7 involving the last one-half elements of sorted array B and the last one-third of A fixes the last one-third part of A correctly. Finally, the third call to stoogesort on line 8 fixes the first two-thirds.
Hence stoogesort correctly sorts the input data.

(b) Time taken by stooge sort is (not the worst or best but actual time)

$T(n) = 1 + 3T(2n/3)$
$b = 3/2 \quad a = 1 \quad f(n) = 1$
Therefore case (1) of master theorem holds. Hence $T(n) = \Theta(n^{log_{1.5}(3)})$

(c) since $1.5^2 < 3$, it follows that $log_{1.5}(3) > 2$, so that $T(n) > n^2$. The worst case time of quicksort is n^2, insertion sort is n^2, merge sort is $n \lg n$, heapsort is $n \lg n$, bubble sort is n^2 and selection sort is n^2. Therefore stooge-sort is inferior to all of these. The profs need to be dismissed!

5. *CLR 9.1-1*

The smallest possible depth is $n - 1$. Intuitive reason : we have to make n-1 comparisons to check whether the input is already sorted.

proof :

claim : given any n elements, we have to make at least n-1 comparisons to find the correct sorted order.

proof : Consider a graph where each node is an element, and edges are placed between elements which have been compared. A disconnected graph indicates multiple blocks of elements where nothing in one group has been compared to anything in another, so we cannot possibly know their relative order, so the graph must at a minimum be connected, so there must be at least n-1 edges. This shows the smallest depth is $\geq n-1$ comparisons.

Smallest depth is ¡= n-1 because (for example) insertion sort will sort sorted input in n-1 comparisons.

So, the smallest possible depth of a leaf in a decision tree for a sorting algorithm is exactly n-1.

6. *CLR 9.1-3*

   Claim : If there are n leaves in a binary tree then the height of the tree is at least $\lceil \lg n \rceil$.

   Proof : Let "h" be the height of the binary tree. If there is a leaf at a height less than "h" then 1 more leaf can be added to the tree by adding 2 children to that leaf (that node ceases to be a leaf.) Continuing these argument we see that max num of leaves in a tree of height h is equal to number of nodes(leaves) at height "h" equal to 2^h. Therefore $n \leq 2^h \Rightarrow h \geq \lceil \lg n \rceil$.

   Returning to the original problem : On the contrary assume that there is a comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. This means that for $n!/2$ of the inputs no more than $c * n$ (c is a const) comparisons are needed, which implies that there are $n!/2$ leaves at height $<= c * n$ in the decision tree for that sorting method. Chop off the tree beyond height $c * n$. The number of leaves is $>= n!/2$. Therefore by the prev claim height $>= lg(n!/2)$.

   Now we claim $\lg(n!/2)$ is not $<= c * n$.

   $\lg(n!/2) = \lg(n!) - \lg 2 > \lg(n/e)^n - 1$ (by Stirling's approximation) $= n\lg(n/e) - 1$, which is not $O(n)$ since $\lg(n/e)$ is not constant.

   So, $\lg(n!/2)$ is not $<= c * n$. Hence a contradiction.

   What about a fraction of 1/n? 1/n * n! = (n-1) !
   As above, lg (n-1)! = nlg n (approximately). Therefore ans is NO.

   What about a fraction of $1/2^n$?

   $lg(n!/2^n) = nlgn - n = nlgn$. Therefore ans is NO.

7. *Demonstrate your understanding of linear time sorting algorithms:*

   (a) *Using CLR Figure 9.2 as a model, illustrate the operation of Counting-Sort on the array* $A = \langle 7, 1, 7, 4 \rangle$.

   A={7,1,7,4} C={0,0,0,0,0,0,0}
   C={1,0,0,1,0,0,2}
   C={1,1,1,2,2,2,4}
   B={-,4,-,-} C={1,1,1,1,2,2,4}

B={-,4,-,7} C={1,1,1,1,2,2,3}
B={-,4,7,7} C={1,1,1,1,2,2,2}
B={1,4,7,7} C={0,1,1,1,2,2,2}

(b) *Using CLR Figure 9.3 as a model, illustrate the operation of Radix-Sort on the following list of English words:*
BOX, COW, BAR, FOX, EAR.

```
box          bar          bar          bar
cow    s3    ear    s2    ear    s1    ear
bar ------> cow ------> cow ------> box
fox          box          box          cow
ear          fox          fox          fox
```

s3 = sort on third (last) character
s2 = sort on second (middle) character
s1 = sort on first character

(c) *Using CLR Figure 9.4 as a model, illustrate the operation of Bucket-Sort on the array* $A = \langle .71, .58, .41, .17, .14 \rangle$.

bucket	range	keys
b0	[0,.20)	.17, .14
b1	[.20,.40)	
b2	[.40,.60)	.58, .41
b3	[.60,.80)	.71
b4	[.80,1)	

ans=.14,.17,.41,.58,.71

8. *Give an algorithm which sorts any array of input size 5 with no more than 7 comparisions. ($\lceil \lg 5! \rceil = 7$, hence 7 comparisions are needed.) This algorithm (not others) should be given in decision tree format.*

let the 5 elements be a_1, a_2, b_1, b_2, c

1. sort a_1, a_2. let the answer be d_1, d_2. (i.e if $a_1 <= a_2$ then $d_1 = a_1, d_2 = a_2$ else $d_1 = a_2, d_2 = a_1$)

Total comparisons=1

2. sort b_1, b_2. let the answer be e_1, e_2.

Total comparisons=2

3.1 compare d_1, e_1

Total comparisons=3

case $d_1 <= e_1$

3.2 compare e_1, c

Total comparisons=4

case $e_1 <= c$ we have $d_1 <= e_1 <= c$ and $d_1 <= d_2$ and $e_1 <= e_2$

to insert e_2 into $\{d_1, e_1, c\}$ we need one comparison Total comparisons=5 and to insert d_1 into $\{\{d_1, e_1, c\} \bigcup e_2\}$ we need 2 comparisons. therefore Total comparisons=7

case $e_1 > c$ compare d_1 and c Total comparisons=5

case $d_1 <= c$

we have $d_1 <= c <= e_1 <= e_2$ to insert d_2 into this list we need 2 comparisons. therefore Total comparisons=7

case $d_1 > c$

we have $c < d_1 <= e_1 <= e_2$

again to insert d_2 into this list we need 2 comparisons. therefore Total comparisons=7

All remaining cases follow same argument !

9. *Consider a 2 dimensional array A[1..m,1..n]. Each row of the array is sorted in nondecreasing order and also each column is sorted in nondecreasing order. Given an element x, give an algorithm which detects whether x is in A in time O(m+n).*

The idea is that by starting in the top right corner (equivalently, the bottom left corner), you always know which direction you should go. Say the element in the top right corner is a, and the element you're seeking is x. Then x is either equal to that a (so you've found it), less than a (so you can eliminate the entire last column), or greater than a (so you can eliminate the entire first row).

```
row=1
col=n

/* while there are rows and columns which have not yet been eliminated */
while ((row<=m) and (col >= 1)) do
begin
   /* check for element */
   if (x==a[row,col]) then
   begin
      print ("x" is present)
      exit
   end
   else
   begin
      if (x<a[row,col]) then
      begin
 /* eliminate this column from consideration */
         col=col-1
         /* since colms are in nondecreasing order , x cannot
         be in column "col" as all elements are > x  */
      end
      else
      begin
 /* eliminate this row from consideration */
         row=row + 1
         /* since rows are in nondecreasing order , x cannot
         be in row "row" as all elements are < x */
      end
```

```
    end
end
print ("x is not present")
```

Time reqd : in each execution of "while" loop either "row" is incremented or "col" is decremented. hence the loop is executed atmost m+n times. work done within the loop is of const time. hence O(m+n)

proof of correctness: it is obvious that the algorithm will not declare "x" to be present when it is not!
suppose x=a[i,j].
case 1 : col reaches j before row reaches i
in this case $x > A[row, col]$ is true and hence row gets incremented. this happens repeatedly till $row == i$ and hence we succeed.

case 2 : row reaches i before col reached j
in this case $x < A[row, col]$ is true and hence col gets decremented. this happens repeatedly till col==j and hence we succeed.

10. *You have m arrays $A_1, A_2, ...A_m$. A_i has n_i elements. $n_1 + n_2 + ... + n_m = n$. The numbers from 1..n are distributed among these "m" arrays. sort all the "m" arrays in time O(n).*

```
/* use array is_in[1..n] to record in which array the number "k"
(1<=k<=n) is present.
  i.e if k is in array A_i then is_in[k]=i */

for i=1 to m do
begin
/* process array A_i */
for j= 1 to n_i do
begin
is_in[A_i[j]]=i
  /* recording that A_i[j] is in array i */
end
end

/* use "m" index variables index[1], ...index[m] */
/* index[1]=index[2]=..=index[m]=1 */
/* i.e each index pointer is pointing to the first location of each
   array */

for (i=1 to n do)
begin
k=is_in[i]
/* find in which array "i" is in */

A_k[index[k]]=i
/* place "i" in array "k" at the right position

++index[k]
/* move the index pointer of array k to point to the
```

```
   next position */
end
```

Time needed. The time need for the first "for" loop (which included the nested "for" loop) is O(n). the second "for" loop takes O(n) time. hence total time needed is O(n).

11. *CLR 12.4-1.*

    let the hash table be h[0..10]

    linear probing

```
{-,-,-,-,-,-,-,-,-,-,10}    h'(10)=10
{22,-,-,-,-,-,-,-,-,-,10}   h'(22)=0
{22,-,-,-,-,-,-,-,31,-,10}  h'(31)=8
{22,-,-,-,4,-,-,-,31,-,10}  h'(4)=4
{22,-,-,-,4,15,-,-,31,-,10} h'(15)=4
{22,-,-,-,4,15,28,-,31,-,10} h'(28)=6
{22,-,-,-,4,15,28,17,31,-,10} h'(17)=6
{22,88,-,-,4,15,28,17,31,-,10} h'(88)=0
{22,88,-,-,4,15,28,17,31,59,10} h'(59)=4
```

    quadratic probing

```
h(k,0) = h'(k)

        {-,-,-,-,-,-,-,-,-,-,10}        h'(10)=10
        {22,-,-,-,-,-,-,-,-,-,10}       h'(22)=0
        {22,-,-,-,-,-,-,-,31,-,10}      h'(31)=8
        {22,-,-,-,4,-,-,-,31,-,10}      h'(4)=4

h'(15)=4 h(15,1)= 4 + 1 + 3 = 8
h(15,2)=4 + 1*2 + 3*2*2  = 18 = 7

{22,-,-,-,4,-,-,15,31,-,10}
        {22,-,-,-,4,-,28,15,31,-,10}    h'(28)=6

h'(17)=6 h(17,1)= 6+ 1 + 3=10
h(17,2)=6+ 1*2 + 3*2*2  = 20 = 9

        {22,-,-,-,4,-,28,15,31,17,10}

h'(88)=0 h(88,1)= 0 + 1 + 3 = 4
h(88,2)=0 + 1*2 + 3*2*2 = 14 = 3

{22,-,-,88,4,-,28,15,31,17,10}

h'(59)=4 h(59,1)= 4 + 1 + 3 =8
h(59,2)= 4 + 1*2 + 3*2*2 = 18=7
h(59,3)= 4 + 1*3 + 3*3*3 = 34 1

{22,59,-,88,4,-,28,15,31,17,10}
```

    double hashing

```
$h_1(k)=h'(k)$
therefore $h(k,0)=h'(k)$

        {-,-,-,-,-,-,-,-,-,-,10}        $h'(10)=10$
        {22,-,-,-,-,-,-,-,-,-,10}       $h'(22)=0$
        {22,-,-,-,-,-,-,-,31,-,10}      $h'(31)=8$
        {22,-,-,-,4,-,-,-,31,-,10}      $h'(4)=4$

$h'(15)=4 h(15,1) = 4 + 1*5 = 9$

{22,-,-,-,4,-,-,-,31,15,10}
{22,-,-,-,4,-,28,-,31,15,10} $h'(28)=6$

$h'(17)= 6 h(17,1) = 6 + 1*7 = 13=2$

{22,-,17,-,4,-,28,-,31,15,10}

$h'(88)=0  h(88,1)= 0 + 1*8 = 8$
$h(88,2)=0 + 2*8 =16 = 5$

{22,-,17,-,4,88,28,-,31,15,10}

$h'(59)=4 h(59,1)=4+1*9= 13=2$
$h(59,2)=4+2*9= 22=0$
$h(59,3)=4+3*9= 31=9$
$h(59,4)=4+4*9=40= 7$

{22,-,17,-,4,88,28,59,31,15,10}
```