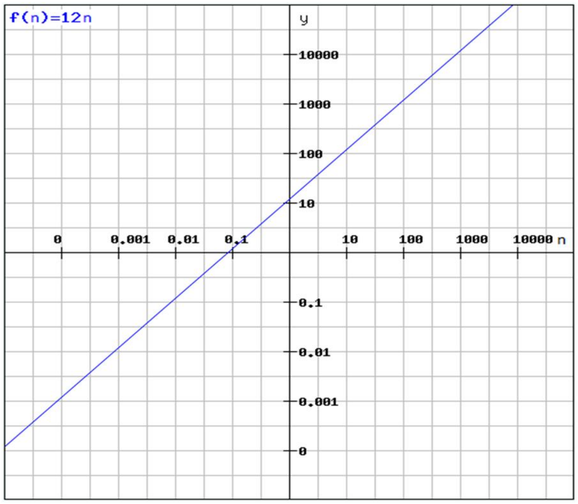
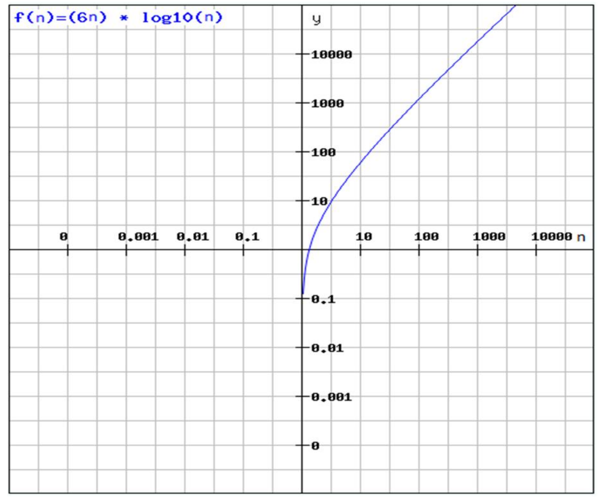
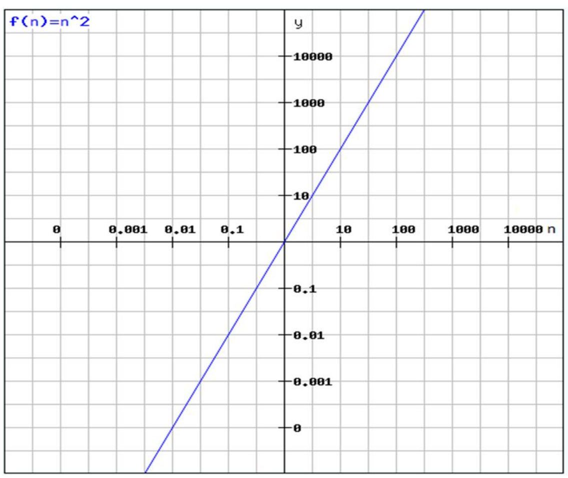
Name: Walid Sultan Aly Ahmed Id: 984627

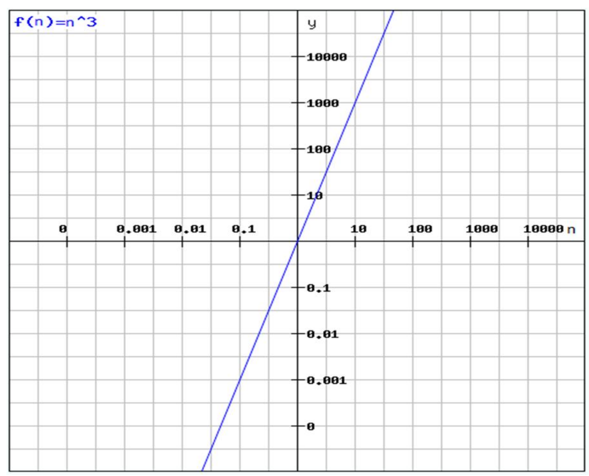
Assignment 1

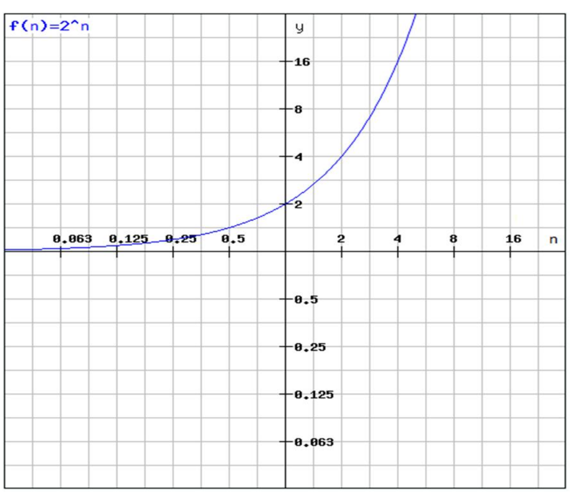
R-1.1











R-1.2

$10n \log n <= n^2$

$10 \log n <= n$

$n_o = 10$

R-1.6

4^n

2^n

n^3

$n^2 \log n$

$4^{\log n}$

$2n \log^2 n$

$4n^{3/2}$

$n \log n$

$5n$

$n^{1/2}$

$\log \log n$

$1/n$

R-1.10

Algorithm Loop1(n)

s ← 0	1
for *i* ← 1 **to** *n* **do**	n
s ← *s* + *i*	n

Algorithm Loop1 runs in O(n) time

R-1.14

Algorithm Loop5(n)

s ← 0	1
for *i* ← 1 **to** n^2 **do**	n^2
for *j* ← 1 **to** i **do**	$n^2 (n^2+1)/2$
s ← *s* + *i*	$n^2 (n^2+1)/2$

Algorithm Loop5 runs in $O(n^4)$ time

Proof

$\mathrm{Log}_b\ x^a = a \log_b x$

let $\mathrm{Log}_b\ x^a = y$

$b^y = x^a$

$b^{y/a} = x$

$\log_b b^{y/a} = \log_b x$

$y/a = \log_b x$

$y = a \log_b x$

<u>**Assignment 2**</u>

R-2.1

Algorithm insertBefore(p, e)

Create new node v

v.element ← e

v.next ← p {link v to its successor}

v.prev ← p.prev {link v to its predecessor}

(p.prev).next ← v {link p old predecessor to its new successor}

p.prev ← v {link p to its predecessor}

return v

Algorithm insertFirst(e)

firstPosition ← L.first() {get the position of the first element in the list}

firstNode ← insertBefore(firstNode, e)

return firstNode

Algorithm insertLast (e)

Create new node v

lastPosition ← L.last()

v.element ← e

v.prev ← lastPosition {link v to its predecessor}

lastPosition.next ← v {link lastPosition to its new successor}

return v

C-2.1

Algorithm findMiddle(L)

```
{Input: L is a doubly linked list}

{output: middle node of L}

h ← L.header                  1

t ← L.trailer                 1

while h ¬ = t do              n/2
    h ← L.after(h)            n/2
    t ← L.before(t)           n/2
return h                      1
```

The running time for findMiddle(L) is O(n)

C-2.2

Algorithm enqueue(o)

```
S1.push(o)                              1
```

Algorithm dequeue()

```
If S2.Empty() then                      1
    While ¬ S1.isEmpty() do             n
        S2.push(S1.pop())               2n
Return S2.pop()                         1
```

The running time of enqueue is O(1)
The running time of dequeue is O(n)

C-2.3

Algorithm push(o)

```
Q1.enqueue(o)                           1
```

Algorithm pop()

```
While Q1.size()>1 do                    n
    Q2.enqueue(Q1.enqueue())            2n
e ← Q1.dequeue()                        1
tmp ← Q2                                1
Q2 ← Q1                                 1
Q1← tmp                                 1
Return e                                1
```

The running time of enqueue is O(1)
The running time of dequeue is O(n)

C-2-4

```
Algorithm permuteNumbers(s)
    {Input sequence s}
    {output sequence containing permutations of s}

    create new sequence permutedList
    create new sequence permutedListInner

    t ← skipFirstElement(s)            {copy all of the elements in s except the first one to t}

    if s.Size()>1 then
        permutedListInner ← permuteNumbers (t)
    else
        permutedListInner.addLast(t)

    for each permutation in permutedListInner
        for i←0 to  s.size()-1 do
            singlePermutation ← copy(permutation)
            singlePermutation.addAtRank(i, s.first())
            permutedList.add(singlePermutation)

    return permutedList

Algorithm skipFirstElement (s)
    {Input sequence s}
    {copy all of the elements in s except the first one to t}

    Create new sequence t
    For i←1 to s.size()-1 do
        t.addLast(s. elemAtRank(i))

    return t
```

C-2-5

Algorithm size()
    Return (N-f +t) mod N

Algorithm isEmpty()
    return (f = t)

Algorithm insertFront(o)
    If size() = N-1 then
        Throw vectorFullException()
    else
        f ← (f-1) mod N
        V[f] ← o

Algorithm deleteFront()
    If isEmpty() then
        Throw vectorEmptyException()
    else
        f ← (f+1) mod N
        V[f] ← null

Algorithm insertLast(o)
    If size() = N-1 then
        Throw vectorFullException()
    else
        t ← (t+1) mod N
        V[t] ← o

Algorithm deleteLast()
    If isEmpty() then
        Throw vectorEmptyException()
    else
        t ← (t-1) mod N
        V[t] ← null

Algorithm elementAtRank(r)
    If r<0 V r > size() then
        Throw outOfIndexException()
    Else
        Pos ← (N-f +r) mod N
        Return V[pos]

Assignment 3

R-2.7

Algorithm root()

    Return S.elemAtRank(1)

Algorithm parent(v)

    If p(v) mod 2 > 0
        Return S.elemAtRank((p(v)-1) /2)
    Else
        Return S.elemAtRank(P(v)/2)

Algorithm leftChild(v)

    Return S.elemAtRank(2p(v))

Algorithm rightChild(v)

    Return S.elemAtRank(2p(v) + 1)

Algorithm isInternal(v)

    Return ((2 p(v)+ 1)< (S.size()-1) ∧ (leftChild(v) ¬ = null V rightChild(v) ¬ = null))

Algorithm isExternal(v)

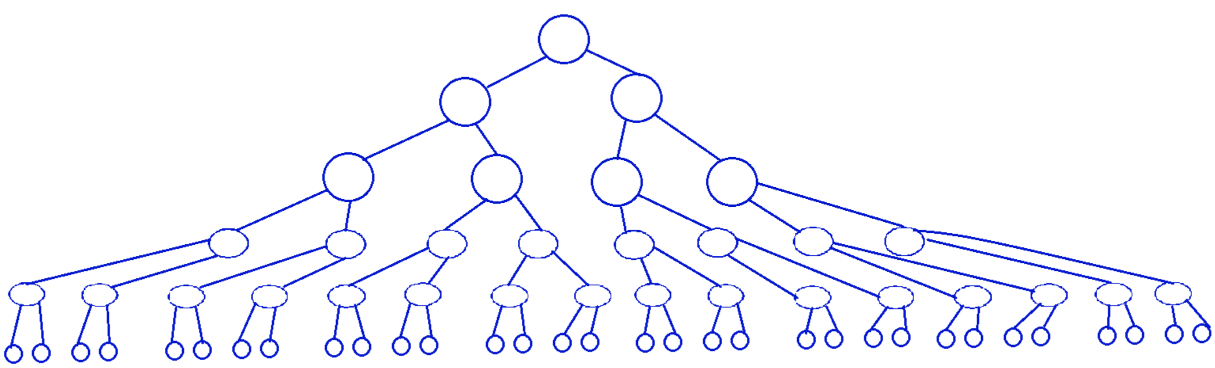
    Return ¬ isInternal(v)

Algorithm isRoot(v)

    Return v= root()

R-2.8

a)

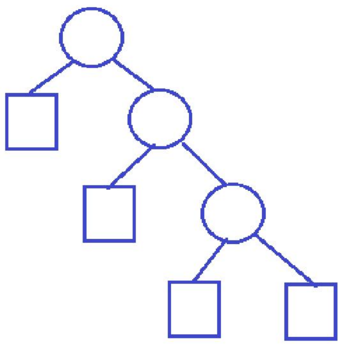


b) $e = i+1$

$i >= h$

$e >= h+1$

The minimum number of external nodes is h +1



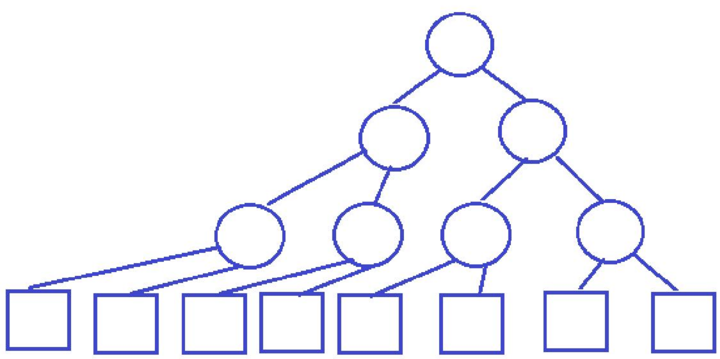
c)

$I <= 2^h - 1$

$I = e - 1$

$e <= 2^h$

The maximum number of external nodes for a binary tree is 2^h



d) $h <= I <= 2^h - 1$

$h+1 <= e <= 2^h$

$2h+1 <= n <= 2^{h+1} - 1$

$n >= 2h+1 \quad \wedge \quad n <= 2^{h+1} - 1$

$h <= (n-1)/2 \quad \wedge \quad h >= \log(n+1) - 1$

$\log(n+1) - 1 <= h <= (n-1)/2$

e) $\log(n+1) - 1 = (n-1)/2$

$\log(n+1) = (n+1)/2$

The above relation is true for:

$n+1=2$ or $n+1 = 4$

$n=1$ and $h=0$

$n=3$ and $h=1$

C-2.2

Suppose we have two stacks S1 and S2. Enqueue just push the element to S1. Dequeue is implemented by doing pop on S2 if S2 is not empty. If S2 is empty then pop all the elements from S1 to S2 then pop the first element from S2.

If we set the weight for Enqueue to be 2, then we will have one extra credit when moving each element from S1 to S2. Thus the for n operations the running time is O(n). As a result, each operation runs in O(1) amortized time.

C-2.7

Algorithm shuffleDeck(s)

{Input s: sequence of cards to be shuffles}

	Array	Linked List
shuffledSquence ← Create new sequence		
while ¬ s.isEmpty() do	O(n)	O(n)
randomElement ← s.removeAtRank(randomInt(s.size()))	$O(n^2)$	O(n)
shuffledSquence.addLast(randomElement)	O(n)	O(n)

return shuffledSquence

The running time Array based sequence is $O(n^2)$

The running time Linked List based sequence is O(n)

Assignment 4

R-2.8

	Running Time	
22 15 26 44 10 3 9 13 29 25	10	{insertion of 10 items in unsorted sequence}
3 15 26 44 10 22 9 13 29 25	9	{comparisons are needed}
3 9 26 44 10 22 15 13 29 25	8	
3 9 10 44 26 22 15 13 29 25	7	
3 9 10 13 26 22 15 44 29 25	6	
3 9 10 13 15 22 26 44 29 25	5	
3 9 10 13 15 22 25 44 29 26	4	
3 9 10 13 15 22 25 26 29 44	3	
3 9 10 13 15 22 25 26 29 44	1	

Total running time = 53

R-2.9

22 15 26 44 10 3 9 13 29 25	10	
15 22 26 44 10 3 9 13 29 25	1	
15 22 26 44 10 3 9 13 29 25	1	
15 22 26 44 10 3 9 13 29 25	1	
10 15 22 26 44 3 9 13 29 25	8	{4 swaps + 4 comparisons}
3 10 15 22 26 44 9 13 29 25	10	
3 9 10 15 22 26 44 13 29 25	10	
3 9 10 13 15 22 26 44 29 25	2	
3 9 10 13 15 22 26 29 44 25	6	
3 9 10 13 15 22 25 26 29 44	0	

Total running time = 49

R-2.10

4 3 2 1 4 {4 insertions}

3 4 2 1 2 {1 comparison + 1 swap}

2 3 4 1 4 {2 comparisons + 2 swaps}

1 2 3 4 6

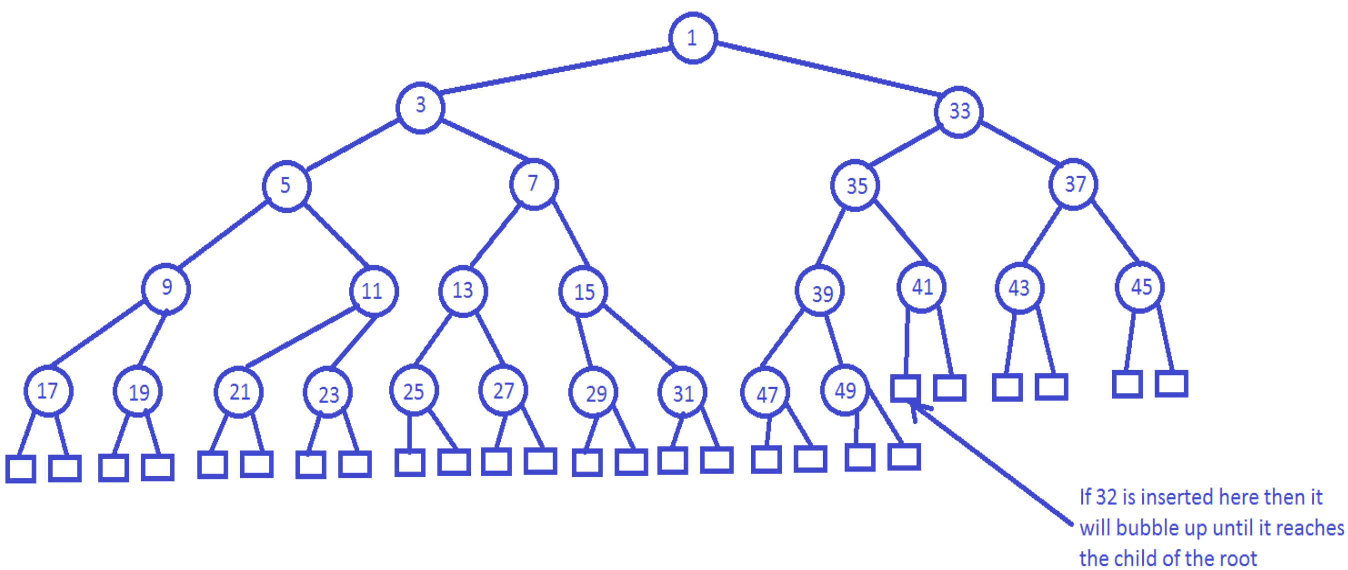
Total running time = 16

Any array sorted in descending order should have $\Omega(n^2)$ running time

R-2.13

Since the items in the vector are sorted, then the key of any item will be greater than or equal to its parent. As a consequence, tree T is a heap.

R-2-18



C-2.32

Algorithm reportSmallerKeys(H, x)

```
    s ← create new sequence
    i ← 1
    while  i <= H.size() ∧ H[i] <= x do
        s.insertLast(H[i])
        i ← i +1
return s
```

Assignment 5

R-4.2

Algorithm mergeSort(S, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if S.size() > 1 then

(S1, S2) $\leftarrow$ partition(S, n/2)

mergeSort(S1, C)

mergeSort(S2, C)

S $\leftarrow$ merge(S1, S2, C)

Algorithm merge(A, B, C)

Input sequences A and B with n/2 elements each, comparator C

Output sorted sequence of A and B

S $\leftarrow$ empty sequence

while ¬A.isEmpty() ∧ ¬B.isEmpty() do

if C.isLessThan(B.first().element(), A.first().element()) then

S.insertLast(B.remove(B.first()))

else

S.insertLast(A.remove(A.first()))

while ¬A.isEmpty() do

S.insertLast(A.remove(A.first()))

While ¬B.isEmpty() do

S.insertLast(B.remove(B.first()))

return S

R-4.5

```
Algorithm specialMerge(A, B, C)
    Input sequences A and B with n/2 elements each, comparator C
    Output sorted sequence of A and B
    S ← empty sequence
    while ¬A.isEmpty() ∧ ¬B.isEmpty() do
        if C.isLessThan( B.first().element(), A.first().element() ) then
            S.insertLast(B.remove(B.first()))
        Else if C.isEqual(B.first().element(), A.first().element()) then
            S.insertLast(B.remove(B.first())
            A.remove(A.first())
        else
            S.insertLast(A.remove(A.first()))
    while ¬A.isEmpty() do
        S.insertLast(A.remove(A.first()))
    While ¬B.isEmpty() do
        S.insertLast(B.remove(B.first()))
    return removeRepeated(s)
Algorithm removeRepeated(s)
    For i←0 to s.size()-2 do
        If s.elemAtRank(i) = s.elemAtRank(i+1) then
            s.removeAtRank(i)
            i ← i-1
    return s
```

R-4.9

Since the list is already sorted then best running time will be the case, which is O(n log(n))

C-4.10

1. Sort the sequence S using Heap-Sort. The running time should be O(n log n)
2. Initialize two variables currentCount and maxCount
3. Iterate on the sorted sequence, and increment the currentCount until the ID changes, then compare currentCount with maxCount. If currentCount is greater than set maxCount as currentCount. The running time for the operation is O(n)

The total running time is O(n log n)