1. (12 %) Using the definition of asymptotic notations defined in the textbook or class materials to PROVE (if it's correct) or DISPROVE (if it's not correct) each of the following problems and JUSTIFY your answers by identifying the corresponding constants C_1, C_2, n_0 as appropriate.

a) $f(n) = 2^{2+n}$ then $f(n) = \Theta(2^n)$

b) $\lg(n) = \Theta(n)$

c) $f(n) = 2^n + 3n$, $g(n) = 3^n + 2n + 1$, then $f(n) = O(g(n))$ and $f(n) = o(g(n))$

d) $f(n) = 2n + 1$, $g(n) = 3*\log^2 n + 2$, then $f(n) = \Omega(g(n))$ and $f(n) = \omega(g(n))$

a) $f(n) = 2^{2+n}$ then $f(n) = \Theta(2^n)$

Answer:

For $f(n) = \Theta(g(n))$, $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for $0 \leq c_1, c_2$ and $n_0 \leq n$

$f(n) = \Theta(2^n)$, $c_1\, 2^n \leq 2^{2+n} \leq c_2\, 2^n$

When $c_1 = 1$, $c_2 = 8$, $n_0 \leq 1$, it valid. Therefore, $f(n) = \Theta(2^n)$

b) $\lg(n) = \Theta(n)$

Answer:

For $f(n) = \Theta(g(n))$, $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for $0 \leq c_1, c_2$ and $n_0 \leq n$

$c_1\, n \leq \lg(n) \leq c_2\, n$

Even we can find c_2 and n_0 to satisfy $\lg(n) \leq c_2\, n$ for O(n), however , since n grows faster than lg(n), we cannot find any c_1 and n_0 to satisfy $c_1\, n \leq \lg(n)$ **$\Omega(n)$,** therefore $\lg(n) \neq \Theta(n)$.

c) $f(n) = 2^n + 3n$, $g(n) = 3^n + 2n + 1$, then $f(n) = O(g(n))$ and $f(n) = o(g(n))$

Answer:

$f(n) = O(g(n))$, $f(n) \leq c_1\, g(n)$ for $0 \leq c_1, n_0 \leq n$

$2^n + 3n \leq c_1 (3^n + 2n + 1)$

When $c_1 = 1$, $n_0 = 1 \leq n$, it is valid, Therefore, $f(n) = O(g(n))$

$$f(n) = o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$\lim_{n\to\infty} \frac{2^n + 5n}{3^n + 2n + 1} = \lim_{n\to\infty} \frac{n(2^{n-1} + 5)}{n(3^{n-1} + 2)} = \lim_{n\to\infty} \frac{2^{n-1})}{3^{n-1}} = 0$$

H1.4

d) f(n) = 2n + 1, $g(n) = 3*log^2 n + 2$, then f(n) = Ω(g(n)) and f(n) = ω (g(n))

Answer:

f(n) =Ω (g(n)) , c_1 g(n) $\leq$ f(n) , for $0 \leq c_1$, $n_0 \leq$ n

$c_1(3*log^2 n + 2) \geq 2n + 1$, pick $0 \leq c_1 = 1$, $n_0 = 1 \leq$ n to satisfy f(n) = Ω(g(n))

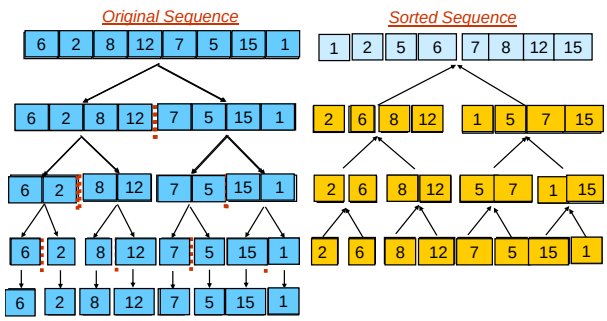
$$f(n) = \omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

$$\lim_{n\to\infty} \frac{2n+1}{3\log^2 n + 2} = \lim_{n\to\infty} \frac{2n}{3\log^2 n} = \lim_{n\to\infty} \frac{2n}{3(\log\ n)^2} = \infty$$

2 (12 %) Using Merge Sort to sort the array A[6, 2, 8, 12, 7, 5, 15, 1], show each step of intermediate sorting results.
Answer:

```
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)
```


Merge Sort



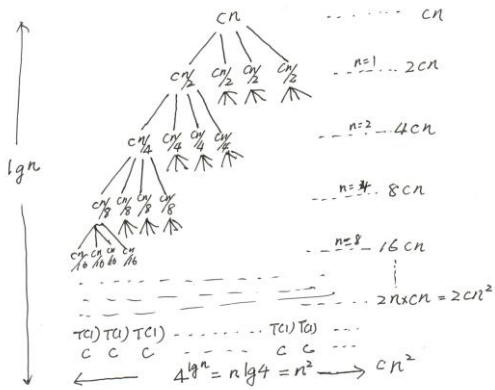
3. (15 %) Assume that algorithm A_1 takes time roughly $T_1(n) = 6n^2 + n + 1$ and algorithm A_2 takes time roughly $T_2(n) = 4T_2(n/2) + 100n$, and suppose that computer A's CPU runs 10^6 instructions/sec. When the input size equals to 10^8, **COMPARE** the running time of the two algorithms and **SUMMARIZE** which algorithm is more efficient.
Answer:

For $T_1(n)$ = O(g(n)) (upper bound (worst case of $T_1(n)$))
g(n) = n^2 $6n^2 + n + 1 \leq c_1 n^2$ $n + 1 \leq (c_1 - 6)n^2$
pick $c_1 = 8$, $n_0 = 1 \leq n$ c_1 g(n) = $8n^2$
$8*(10^8)^2 / 10^6 = 8 \times 10^{10}$ sec

$T_2(n) = 4T_2(n/2) + 100n$ to find Big O(g(n))

Using Master Theorem:
Since a =4 , b = 2 , f(n) = 100n → $n^{log_b a} = n^{log_2 4} = \Theta(n^2)$
$f(n)=O(n^{log_2 4 - \varepsilon})$ for ε=1, By case 1, $T_2(n) = \Theta(n^2)$ → $T_2(n) = O(n^2)$

Recursion Tree for $T_2(n) = 4T_2(n/2) + 100n$



- c=100
- The height is lg *n*
- #leaf nodes = $4^{\lg n} = n^{\log 4} = n^2$. Leaf node cost: $T(1)$.
- Total cost $T(n)=cn+2cn+4cn+8cn+16cn \cdots + 2^{\lg n-1} * cn + cn^2$
  $= (2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + \cdots + 2^{\lg n-1}) cn + cn^2$
  $= 100n [(2^{\lg n} - 1)/(2-1)] + 100n^2$
  $= 100n [2^{\lg n} - 1] + 100n^2$
  $= 100n [n - 1] + 100n^2$
  $= 100n^2 - 100n + 100n^2$
  $= 200n^2 - 100n$

$T_2(n) = 200n^2 - 100n \rightarrow T_2(n) = O(n^2)$

$\rightarrow 200n^2 - 100n \leq c*n^2$ and find the value c and n_0

$\rightarrow$ $200 - 100/n \leq c \rightarrow$ when c =200, and $n_0 = 1 \leq n$,

- The run time of Algorithm T_2 in computer A:
- $n = 10^8 : 200* (10^8)^2/10^6$ ins/sec = $2*10^{12}$ sec

Summary: $T_1(n)$ with $O(n^2)$ and $T_2(n)$ with $O(n^2)$, the two algorithm have roughly the same running time complexity.

4. (18%) Using the Recursion Tree approach to find the T(n) for the following Binary Search Algorithm. And then, find its worst-case performance notation? Suppose that computer A's CPU runs 10^6 instructions/sec. When the input size equals to 10^{10}, Calculate the Binary Search Algorithm's running time?

```
BinarySearch(list[], min, max, key)
if max <min then
   return false
else
   mid = (max+min) / 2
   if list[mid] >key then
      return BinarySearch(list[], min, mid-1, key)
   else if list[mid] <key then
      return BinarySearch(list[], mid+1, max, key)
   else
      return mid
   end if
end if
```

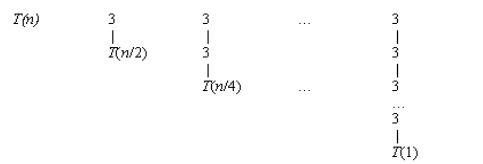
Answer: (Many of you: T(n) = 2T(n/2) +n) which is ☹☹☹☹☹

Binary Search is called on a subarray of length approximately 2/n and there are 3 comparisons in the worst case before a recursive call is needed again. So the recurrence relation is T(n) = T(n/2) + 3, or more specifically, T(k) = T(k/2) + 3 for input size $k \geq 2$.

Also T(1) = 3

Recursion Tree: At each level, let the parent be the constant term 3 and the child be the term T(k/2).

T(n) = T(n/2) + 3



The depth of the tree is approximately $\log_2 n$. Adding the values in the levels of the tree we get $T(n) = 3 \log_2 n$. Therefore $T(n) = \Theta(\log_2 n)$.

Worst case:

$T(n) = O(\log_2 n)$. $\rightarrow g(n) = \log_2 n \rightarrow 3 \log_2 n \leq c * \log_2 n \rightarrow c = 3$, $n_0 = 1 \leq n$

- The run time of Algorithm T_2 in computer A:
- $n = 10^{10} : 3* \log_2 10^{10}/10^6 = 33.21/10^6 = 3.321 \times 10^{-5}$ sec

5. (8 %) Consider the following function:

```
int myexam(int n) {
int answer;
if (n > 0) {
answer =(myexam(n-2)+3*myexam(n/2) + 5);
return answer;
}
else
return 1;
}
```

Write down the complete recurrence relation, T(n), for the running time of myexam(n). Be sure you include a base case T(0). **You do not have to actually solve this relation, just write it down.**
Answer:

T(n) = T(n-2) + T(n/2) + C , for n > 0
T(n) = 1, for n = 0

Or
T(n) = T(n-2) + T(n/2) + Θ(1) , for n > 0
T(n) = Θ(1), for n = 0

6. (15 pts) Given tight asymptotic bounds for the following recurrence relations. Justify your answer by working out details of steps using Master Theorem.

Master Theorem :
for T(n) = aT(n/b)+f(n), n/b may be ⌈n/b⌉ or ⌊n/b⌋
where a ≥ 1, b>1 are positive integers, f(n) be a non-negative function.
Case 1: If f(n)=O($n^{\log_b a-\varepsilon}$) for some ε>0, then T(n)= Θ($n^{\log_b a}$).
Case 2: If f(n)= Θ($n^{\log_b a}$), then T(n)= Θ($n^{\log_b a}$ lg n).
Case 3: If f(n)=Ω($n^{\log_b a+\varepsilon}$) for some ε>0, and if af(n/b) ≤cf(n) for some c<1 and all sufficiently large n, then T(n)= Θ(f(n)).

6. (15 pts) Given tight asymptotic bounds for the following recurrence relations. Justify your answer by working out details of steps using Master Theorem.
a) T(n) = 9T(n/4) + n^2
Answer:

Since a =9 , b = 4 , f(n) = n^2
Compare f(n) = n^2 *to* $n^{\log_b a} = n^{\log_4 9} = n^{1.585}$
Since *f(n)*= Ω($n^{\log_4 9+\varepsilon}$), it might be in Case 3.
Check if a*f(n/b) ≤ c*f(n) for some ε >0, and all sufficiently large n, then T(n) = Θ(f(n)), *i.e.* 9*f(n/4) ≤ c*f(n), or 9*(n/4)^2 ≤ c*n^2 , or (9/16) n^2 ≤ c*n^2
pick c = 9/16 < 1, *So… T(n) = Θ(f(n)) = Θ(n^2)*

b) T(n) = 3T(n/2) + n
Answer:

Since a =3 , b = 2 , f(n) = *n*
Compare f(n) = *n to* $n^{\log_b a} = n^{\log_2 3} = n^{1.585}$
Since $n = O(n^{\log_2 3-\varepsilon})$, it should be in Case 1.
So… T(n) = Θ($n^{\log_b a}$) = Θ($n^{\log_2 3}$)

c) T(n) = 3T(n/3) + log n
Answer:

Since a =3 , b = 3 , f(n) = *log n*
Compare f(n) = *log n to* $n^{\log_b a} = n^{\log_3 3} = n$
Since *log n* = $O(n^{\log_3 3-\varepsilon})$, it should be in Case 1.
So… $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_3 3}) = \Theta(n)$

7 Using array [11, 9, 12, 14, 3, 15, 7, 8, 1] in that order to:
(5 pts) Draw into an **initially binary heap tree structure**.
(10 pts) Draw a **Max-Heap tree** using the binary heap tree from (6.a). You must show the intermediate trees, **please circle the change of nodes for each step or you will lost points.**

Running Time for Max-Heapify

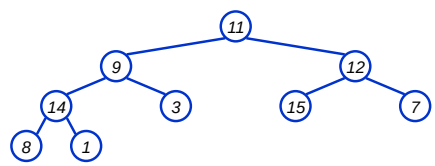
```
Max-Heapify(A,i)
 1 l = Left(i)
 2 r = Right(i)
 3 if l ≤ A.heap-size and A[l] > A[i]
 4        largest  = l
 5 else  largest = i
 6 if r ≤ A.heap-size and A[r] > A[largest]
 7        largest = r
 8 if largest ≠ i
 9       exchange A[i] ↔ A[largest]
10       Max-Heapify(A, largest)
```

Time to fix node i and its children = ***Θ(1)***

PLUS

Time to fix the subtree rooted at one of i's children = T(size of subree at largest)

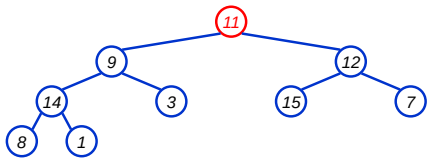
a) (5 %) Draw into an **initially binary heap tree structure**.



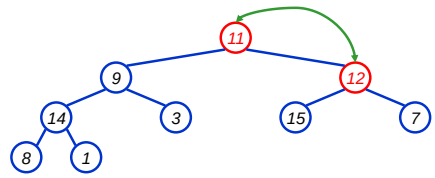
A =

11	9	12	14	3	15	7	8	1

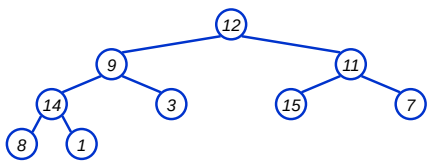
b) (5 %) Draw the final **Max-Heap tree** using the binary heap tree from (a).



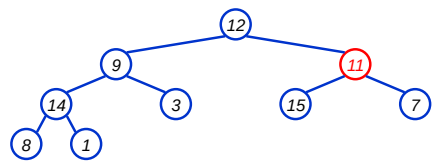
A = | 11 | 9 | 12 | 14 | 3 | 15 | 7 | 8 | 1 |



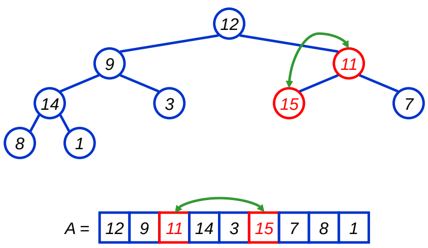
A = | 11 | 9 | 12 | 14 | 3 | 15 | 7 | 8 | 1 |

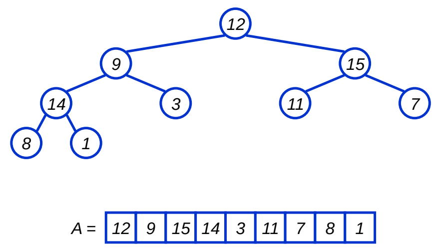


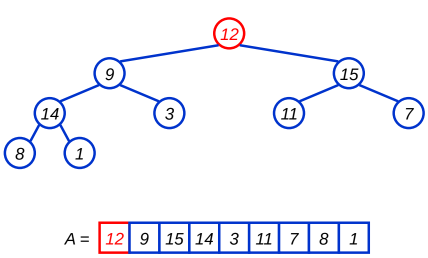
A = | 12 | 9 | 11 | 14 | 3 | 15 | 7 | 8 | 1 |

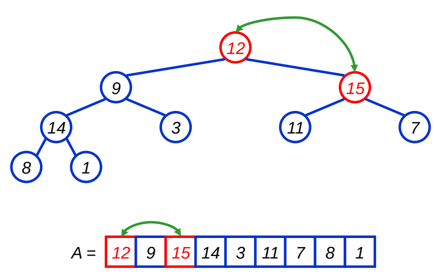


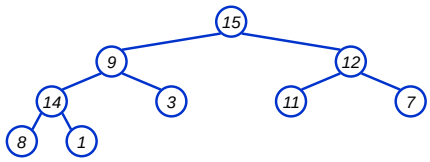
A = | 12 | 9 | 11 | 14 | 3 | 15 | 7 | 8 | 1 |


12
9
11
14
3
15
7
8
1
A = 12 9 11 14 3 15 7 8 1

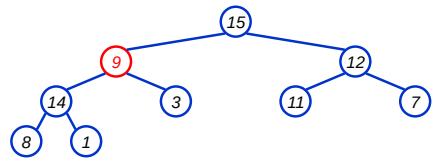

12
9
15
14
3
11
7
8
1
A = 12 9 15 14 3 11 7 8 1


12
9
15
14
3
11
7
8
1
A = 12 9 15 14 3 11 7 8 1


12
9
15
14
3
11
7
8
1
A = 12 9 15 14 3 11 7 8 1

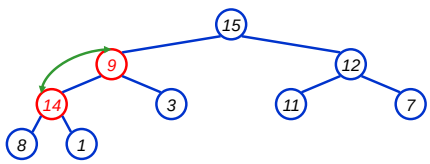


A = | 15 | 9 | 12 | 14 | 3 | 11 | 7 | 8 | 1 |

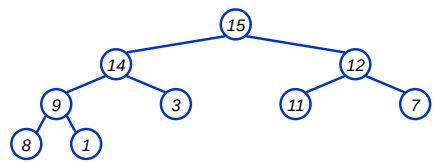


A = | 15 | 9 | 12 | 14 | 3 | 11 | 7 | 8 | 1 |

Done



A = | 15 | 9 | 12 | 14 | 3 | 11 | 7 | 8 | 1 |



A = | 15 | 14 | 12 | 9 | 3 | 11 | 7 | 8 | 1 |

8. (10 %) The running time for Max-Heapify (A, n) recursive subroutine of Heap Sort algorithm is T(n) = T(**largest**) + (1) and **largest** ≤ **2n/3** — i.e., T(n) = T(2n/3) + (1). EXPLAIN clearly why the worst case is **2n/3** and prove it STEP BY STEP by using the approach from the lecture in the class.
Answer:

Why T(2n/3) ?

- The worst case: bottom row 1/2 full
- Number of nodes at -
- level 0 i.e. root is 2^0
- level 1 is 2^1
- level 2 is 2^2
- ...
- level h is 2^h
- Summation of all nodes from level 0 up to level h,
- $S = 2^0 + 2^1 + 2^2 + ... + 2^h$
- From geometric series summation rule we know that
- $x^0 + x^1 + x^2 + ... + x^n = (x^{n+1} - 1)/(x-1)$
- Substituting x = 2, we get
- $S = 2^{h+1} - 1$. i.e. $2^{h+1} = S + 1$
- As 2^{h+1} is the total nodes at level h+1, which means that the total number of leaf nodes (2^{h+1}) is one node more than the total number of non-leaf nodes (S).

Why T(2n/3) ?

Now let's calculate the number of nodes in left subtree, right subtree and total ..

- Assume that number of non-leaf nodes in the left subtree of root = k.
- By the above reasoning, number of leaf nodes in the left subtree of root = k + 1. Number of non-leaf nodes in the right subtree of root = k as the tree is said to be exactly half full.
- Total number of nodes in the left subtree of root = k + k + 1 = 2k +1
- Total number of nodes in the tree, n = (2k + 1) + k + 1(root) = 3k + 2.
- Ratio of nodes in the left subtree and total nodes = (2k + 1) / (3k + 2) which is bounded above by 2/3.
- That's the reason of saying that the children's subtrees each have size at most 2n/3.