CS3230 MID-TERM QUIZ
Semester 1, AY2014/2015
SOLUTION SKETCH

Fun Bonus Question: (1 point)

Give the full name of the *time consuming* operation CTR_k introduced in CS3230.

__ CLEAN AND TIDY ROOM on day *k* __________ ***(Free mark, ALWAYS TRY)***

Noted that many students did not read the instructions on top of page 2:

- *Unless otherwise specified, you are expected to* ***prove (justify)*** *your results.*

Q1. (15 points)

(a) (8 Points) SOLUTION: $g_2(n) << g_1(n) \equiv g_3(n) << g_4(n)$

$$g_1(n) = 2n\sum_{k=1}^{n}(\lg k) = 2n\Theta(n\lg n) = \Theta(n^2\lg n) \qquad g_2(n) = \Theta(n^2(\lg\lg n))$$

$$g_3(n) = 4(\lg n)\sum_{k=1}^{n} k = \Theta(n^2\lg n) \qquad g_4(n) = 2^{(3\lg n)} = 2^{\lg(n^3)} = n^3$$

(b) (4 points) Asymptotic Notations (by definition)

Let $f(n) = 8n^3 + 4n^2 - 2n + 1$

By using the definitions of Θ, prove that $f(n) = \Theta(n^3)$

Upper Bound: (showing all steps here)

$$\begin{aligned} f(n) = 8n^3 + 4n^2 - 2n + 1 &\le 8n^3 + 4n^2 + 1 && \text{for all } n\ge1 && \text{(throw away } -2n) \\ &\le 8n^3 + 4n^3 + n^3 && \text{for all } n\ge1 && \text{(upper everything to } n^3) \\ &= 13n^3 && \text{for all } n\ge1 && \text{(simplify)} \end{aligned}$$

Lower Bound: (showing all steps here)

$$\begin{aligned} f(n) = 8n^3 + 4n^2 - 2n + 1 &\ge 8n^3 - 2n && \text{for all } n\ge1 && \text{(throw away } 4n^2 + 1) \\ &= 7n^3 + (n^3 - 2n) && \text{for all } n\ge1 && \text{(simplify)} \\ &= 7n^3 + (n^3 - 2n) && \text{for all } n\ge1 && \text{(simplify)} \\ &\ge 7n^3 && \text{for all } n\ge2 && \text{(since } (n^3 - 2n) > 0 \text{ for } n\ge2) \end{aligned}$$

So, choose $c_1 = 7$, $c_2 = 13$, and choose $n_0 = 2 = \max\{1,2\} = 2$

Then $7n^3 \le f(n) \le 13n^3$ for $n \ge 2$

By definition of Θ, $f(n) = \Theta(n^3)$.

(c) (3 points) (Limit theorem & L'Hopital's rule)

$$\lim_{n\to\infty}\frac{n^2}{2^n} = \lim_{n\to\infty}\frac{2n}{D*2^n} = \lim_{n\to\infty}\frac{2}{D^2*2^n} = 0$$

Hence, by limit theorem $h(n) = o(2^n)$

Q2. (15 points)

(a) (10 points)

(i) (3 points) $R(n) = 4R(n/16) + 25n^{0.5}$

Use Master's Theorem

$f(n) = 25n^{0.5} = \Theta(n^{0.5})$, a=4, b=16. So $n^{\log_b a} = n^{\log_{16} 4} = n^{\log_{16}(16)^{0.5}} = n^{0.5}$.

Since $f(n) = \Theta(n^{0.5}) = \Theta(n^{\log_b a})$,

This is **Case 2**, (with k=0). Hence, $R(n)= n^{0.5}(\lg n)$

(ii) (3 points) $S(n) = 9S(n/3) + 5n^{1.5}$

Use Master's Theorem

$f(n) = 5n^{1.5} = \Theta(n^{1.5})$, a=9, b=3. So $n^{\log_b a} = n^{\log_3 9} = n^2$.

Since $f(n)= \Theta(n^{1.5}) = O(n^{2-e})$, for e=0.2 (for example).

This is **Case 1**. Hence, $S(n)= \Theta(n^2)$

(iii)* (4 points) $$U(n) = \left[\frac{2}{(n-1)}\sum_{k=1}^{n-1} U(k)\right] + 3n$$

(*Hint:* Reuse the *average case analysis of Quicksort* [*Lecture Notes*].)

First, multiple by (n–1) on both sides.

$$(n-1)U(n) = 2\sum_{k=1}^{n-1} U(k) + 3n(n-1)$$

Expand and get rid of full-history.

$$(n-1)U(n) = 2\big[U(1) + U(2) + \ldots + U(n-2) + U(n-1)\big] + 3n(n-1)$$

$$(n-2)U(n-1) = 2\big[U(1) + U(2) + \ldots + U(n-2)\big] + 3(n-1)(n-2)$$

$$(n-1)U(n) = nU(n-1) + 6(n-1)$$

Turn it into a form that *telescopes*. Divide both side by $n(n$–1).

$$(n-1)U(n) = nU(n-1) + 6(n-1)$$

$$\tfrac{U(n)}{n} = \tfrac{U(n-1)}{(n-1)} + \tfrac{6}{n}$$

(Divide by $n(n$+1).)

Now, do the *telescoping process*.

$$\begin{aligned}
\tfrac{U(n)}{n} &= \tfrac{U(n-1)}{(n-1)} + \tfrac{6}{n} = \left[\tfrac{U(n-2)}{(n-2)} + \tfrac{6}{(n-1)}\right] + \tfrac{6}{n} \\
&= \left[\tfrac{U(n-3)}{(n-3)} + \tfrac{6}{(n-2)}\right] + \tfrac{6}{(n-1)} + \tfrac{6}{n} \\
&= \ldots \\
&= \tfrac{U(1)}{1} + 6\left[\tfrac{1}{2} + \tfrac{1}{3} + \ldots + \tfrac{1}{(n-2)} + \tfrac{1}{(n-1)} + \tfrac{1}{n}\right] \\
&= 6H(n) + \Theta(1)
\end{aligned}$$

Hence, $U(n) = 6nH(n) + \Theta(n) = \Theta(n \lg n)$

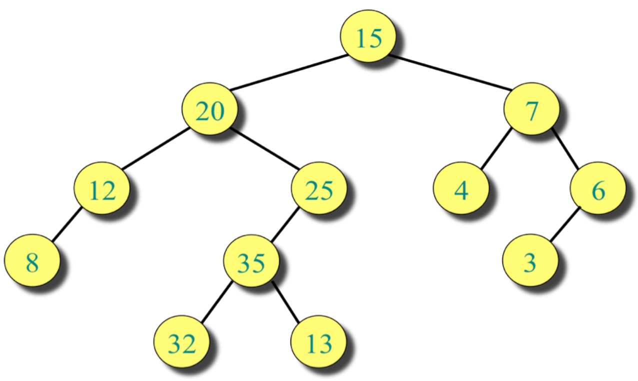
(b) **(5 points) [Radix Sort]**
Run RADIX-SORT on the following list of English words shown below.
Starting with the input in the leftmost column, show the list after each successive sorting step.

BIG	_TEA__	_TAB__	_BIG__
SPY	_SEA__	_TAR__	_HOT__
HOT	_TAB__	_PAR__	_PAR__
TEA	_BIG__	_TEA__	_POT__
RUG	_RUG__	_SEA__	_RUG__
TAR	_TAR__	_BIG__	_SEA__
POT	_PAR__	_HOT__	_SPY__
TAB	_HOT__	_POT__	_TAB__
SEA	_POT__	_SPY__	_TAR__
PAR	_SPY__	_RUG__	_TEA__

Q3. (10 points)

You are given a binary tree T with n nodes and height h. Each node v in the tree has a real-number key $k(v)$. For simplicity, you can assume that these keys are *all distinct*. Then, we say that a node v is a *local maximum* if its key is *greater than* that of *all its adjacent nodes*, i.e. its parent (if it exists) and all its children (if they exist).

(a) **(2 point)** Find a local maximum in the binary tree T given below.



Answer: Local Maximum = Node with key _*35*___

(b) **(8 point)** Describe an $O(h)$ algorithm for finding a *local maximum* in the tree T.

IDEA: Local-Max (v), where v is a vertex in tree.
Start at the root node v. If both children are smaller, then report v as answer
Else find any child w with larger key, recursively search with w.

> **Note:** Special case when v is leaf (no children),
> then only need check against parent node.

```
procedure  Local-Max (v);  (* recursive procedure, v is a vertex in tree *)
(* Maintain Pre-condition:
     {whenever we search at v, then v is larger than parent}  *)
1.  begin
2.      if  (v is a leaf node)
3.         then {report v as local maximum. Done; EXIT};
4.      if  (all children of v have smaller keys)
5.         then {report v as local maximum. Done; EXIT}
6.         else  {Let w = a child of v with larger key;
7.                 Local-Max (w)
8.  end;
```

Call it with Local-Max (*root*);
{Pre-condition: root is *trivially* larger than parent (since root has no parent).}

For the example given in 3(a), the path traced is 15, 20, 25, 35. Report 35.

Correctness proof:

1. We first show pre-condition always true.
At initialization, when we call Local-Max(*root*), the *pre-condition* is trivially true since *root* is larger than parent (*since root has no parent*).
Recursive call is made only at line 7, and we confirm (in line 6) that w is larger than its parent (namely v). Hence, *pre-condition* holds for all recursive calls.

2. The rest of the proof.
In line 3, v is a leaf (no children) and v is larger than parent (*pre-condition*). So, v must be a local maximum.
In line 5, v is larger than parent (*pre-condition*), and also *all its children* (line 4), so v must be a local maximum.

Complexity Analysis: We start at the root. At each level, $\Theta(1)$ time and terminate at (or before) a leaf node. So, worst-case time is $h^*\Theta(1) = \Theta(h)$.

Q4. (10 points)

[Finding *the* Missing Integer]
You are given an array A[1..n] of size n, where $n = (2^k–1)$. You are told that the *unsorted* array A[1..n] contain all the integers from 0 to n (inclusive), except *one*.

For k=3, an example is A[1..7]=[3, 0, 2, 6, 1, 4, 7]. Here the *missing number* is 5.

(a) (5 points) [Finding the Missing Integer]
Give a $\Theta(n)$ algorithm to find the missing integer.
[*Hint:* Use an additional array B[0..n] if necessary.]

SOLUTION: We use array B[0..n] to count the items in A[1..n] (use the first two loops of the Counting Sort). Then, from B, find the element with 0-count.

```
procedure FIND-MISSING; (* recursive procedure, v is a vertex in tree *)
begin
    for i ← 0 to n
        do B[i] ← 0
    for j ← 1 to n
        do B[A[ j]] ← B[A[ j]] + 1            ◁ B[i] = |{key = i}|
    for i ← 0 to n
        do if (B[i] = 0) then report i as missing number.  EXIT
end
```

Complexity analysis: Trivial $\Theta(n)$.

(b) (5 point)**

However, you are *now* told that you cannot directly access the integers in A with a single query operation. (*So, your algorithm from (a) don't work now.*)

The elements in A are represented in binary and stored in k bits, namely,
$(b_{k-1}\, b_{k-2} \ldots b_1\, b_0)$ where each b_j is a binary bit. (*Note:* k=lg n)

You can make the following query operation:

BIT-QUERY(j, i) : "fetch the jth bit of $A[i]$" (constant $\Theta(1)$ time)

which returns a 0 or a 1 in constant $\Theta(1)$ time.

To illustrate this, we show an example using the array A[1..7] above and give some sample queries. Remember, the array A[1..n] is hidden from you.

Hidden Array:	**Sample Queries**
$A[1]$ = 0 1 1	BIT-QUERY(0,2) = 0
$A[2]$ = 0 0 0	
$A[3]$ = 0 1 0	BIT-QUERY(1,3) = 1
$A[4]$ = 1 1 0	
$A[5]$ = 0 0 1	BIT-QUERY(2,6) = 1
$A[6]$ = 1 0 0	
$A[7]$ = 1 1 1	

Using this BIT-QUERY operation, and additional storage if needed, describe an algorithm for finding the *missing integer* with $\Theta(n)$ worst-case running time. Also illustrate how your algorithm works on the above example.

For this, I take a developmental approach (develop solution, bit-by-bit). Enjoy.

Trivial Solution suggested in the Note:

To get the value of $A[i]$, do k BIT-QUERY(j, i) (for $j = k–1, k–2, \ldots, 1, 0$), and use Horner's rule to compute value of $A[i]$. Do this for all $A[i]$.
Total of nk queries, so total time for this step is $\Theta(nk) = \Theta(n\lg n)$ time. [too slow]
Then can call the algorithm from (a) above, which takes only $\Theta(n)$ time.
Algorithm is $\Theta(n\lg n)$ time and is not fast enough. (get *very low marks*)

NEXT KEY IDEAS:

The idea behind this is *binary search*, but we need to figure out – *HOW exactly to do this binary search.* What question do we ask in order to "divide the problem into two"? And then, decide which half of the problem to recursively solve?

We know each integer $A[i]$ is represented as a k-bit binary string.
Suppose the missing integer M, has binary encoding $M = (m_{k-1}m_{k-2} \ldots m_1m_0)$.

Step A: (Determine m_{k-1}) First, we do the following:

```
for i=1 to n do {Count[0] ← 0;  Count[1] ← 0 }          (*from counting sort *)
for i=1 to n do {
  b ← BIT-QUERY(k–1, i)          (* "fetch the (k–1)st bit of A[i])" for all i   *)
  Count[b] ← Count[b] + 1;       (* Count them…                                   *)
}
```

Analysis: Clearly n queries. And also total time $\Theta(n)$.

Then at the end of this computation, if we look at Count[0] and Count[1], what are their values? **STOP READING** and think it one out *first*.

Yes, one of them is $2^{(k-1)}$, while the other is $2^{(k-1)}-1$. Now, why is that so?
[*Hint:* If $m_{k-1} = 0$, which one (of Count[0] or Count[1]) is $2^{(k-1)}$.
And if Count[1] = $2^{(k-1)}$, then what is the value of m_{k-1}?]

In summary, using n queries, we determine value of m_{k-1}, (most significant bit)

Step B: (Determine all the rest.)
What do we do next?
Well, we can do the same thing with the $(k-2)^{th}$ bit. Can we not?
In that case, we can also spend n queries to determine the value of m_{k-2}.

And *then*, we can do the same for all the other bits of M.
This will take a total of $nk = n(\lg n)$ queries. And algorithm takes $\Theta(n\lg n)$ time.

Summary: With **Step A** and **Step B**, we solved the problem.
But, we used too many queries.

The question wants an algorithm that asks $\Theta(n)$ queries.

How can we do better?

To do better, we cannot spend n queries in each round of **Step B**.

How about making a small change to **Step A**?
Incorporate Divide-and-Conquer idea during **Step A**;
Suppose we borrow the *Partitioning* idea (Quicksort), and call this **Step A′**.

Step A′: We partition the array $A[1..n]$ based on the bit #$(k–1)$.

But, we cannot swap the items $A[i]$ and $A[j]$, so just use two lists:
List[0] to store those indices j with $A[j]$=0,
List[1] to store those indices j with $A[j]$=1.
During partitioning, we merely insert (or append) j into List[0] or List[1] depending on value of bit-#$(k–1)$-of-$A[j]$ also increment the size of each list. This takes $\Theta(1)$ per insert/append.

After partitioning, we know Count[0], Count[1] and we know m_{k-1}. ☺

And we also know which "half" the missing number M belongs to.
Then we can recursively find m_{k-2} in the correct half, and so on.

Just a *small change* in the idea. But, we have reduced the problem size to $n/2$.

Let $T(n)$ = number of queries to solve this problem for size n.
Then we have, $T(n) = T(\underline{n}/2) + \Theta(n)$, which gives us $T(n) = \Theta(n)$.

Hidden Array:	**Simulation of the Final algorithm**		
$A[1]$ = 0 1 1	Round 1:	List[0] = {1,2,3,5}	$m_2 = 1$
$A[2]$ = 0 0 0		List[1] = {4,6,7}	Recur on List[1];
$A[3]$ = 0 1 0			
$A[4]$ = 1 1 0	Round 2:	List[0] = {6}	$m_1 = 0$
$A[5]$ = 0 0 1		List[1] = {4,7}	Recur on List[0]
$A[6]$ = 1 0 0			
$A[7]$ = 1 1 1	Round 3:	List[0] = {6}	$m_0 = 1$
		List[1] = {}	$M = (101)_2 = 5$

Now, don't you LOVE the problem?

-- End of Paper --

ADDITIONAL STUFFS (not needed)

An slightly different (*more efficient*) version of LOCAL-MAX (v), and the proof.

```
procedure LOCAL-MAX (v); (* recursive procedure, v is a vertex in tree *)
(* Maintain Pre-condition:
     {whenever we search at v, then v is larger than parent}  *)
1. begin
2.      If (all children of v have smaller keys)
3.         then {report v as local maximum. Done}
4.         else {Let w = a child of v with larger key;
5.                if (w is a leaf)
6.                   then {report w as local maximum. Done}
7.                   else LOCAL-MAX (w).
        8. end;
```

Call it with LOCAL-MAX (*root*);
{Pre-condition: root is *trivially* larger than parent (since root has no parent).}

Correctness proof:

1. We first show pre-condition always true.
At initialization, when we call LOCAL-MAX(*root*), the *pre-condition* is trivially true since *root* is larger than parent (*since root has no parent*).
Recursive call is made only at line 7, and we confirm (in line 4) that w is larger than its parent (namely v). Hence, *pre-condition* holds for all recursive calls.

2. The rest of the proof.
In line 3, v is larger than parent (*pre-condition*), and also all its children, so v must be a local maximum.
In line 6, when w is a leaf, then it is trivially larger than all its children (*none*). And it is also larger than its parent v (line 4). So, w must be a local maximum.

-- End of Paper --