Homework Assignment #1
Due: Feb 5
Student: Shuo Yang

---

1. (a) *Proof.* We prove Θ relation using limits. Let $g(n) = n^k$, and taking a limit between $f(n)$ and $g(n)$ gives:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\sum_{i=0}^{k} a_i n^i}{n^k} = \lim_{n\to\infty} \frac{a_0 + a_1 n + \cdots + a_k n^k}{n^k} \tag{1}$$

$$= \lim_{n\to\infty} (\frac{a_0}{n^k} + \frac{a_1}{n^{k-1}} + \cdots + \frac{a_{k-1}}{n} + a_k) = a_k \tag{2}$$

Since a_k is a positive constant, we know that $f(n) = \Theta(n^k)$. □

(b) *Proof.* Let $f(n) = n^c$ and $g(n) = b^n$. Taking a limit between $f(n)$ and $g(n)$ gives:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^c}{b^n} \tag{3}$$

If $c \leq 0$, then $f(n)$ is monotonically decreasing and $g(n)$ is strictly increasing because $b > 1$. This implies that:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^c}{b^n} = 0 \tag{4}$$

Thus, $n^c = o(b^n)$ when $c \leq 0$.
When $c > 0$, let $\lfloor c \rfloor = k$. By repeatedly using L'Hospitals Rule for $k+1$ times, we get:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^c}{b^n} = \lim_{n\to\infty} \frac{c(c-1)\cdots(c-k)n^{k-c}}{b^n \ln^{k+1} b} \tag{5}$$

$$= \frac{c(c-1)\cdots(c-k)}{\ln^{k+1} b} \lim_{n\to\infty} \frac{n^{k-c}}{b^n} \tag{6}$$

Since $k - c < 0$, we have $\lim_{n\to\infty} \frac{n^{k-c}}{b^n} = 0$. Thus:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{c(c-1)\cdots(c-k)}{\ln^{k+1} b} \lim_{n\to\infty} \frac{n^{k-c}}{b^n} \tag{7}$$

$$= \frac{c(c-1)\cdots(c-k)}{\ln^{k+1} b} \times 0 = 0 \tag{8}$$

Thus, $n^c = o(b^n)$ when $c > 0$.
Therefore, for all constants b and c with $b > 1$, we know that $n^c = o(b^n)$. □

(c) *Proof.* We prove this by using the rule on composition of functions from the class handouts. Let $f = n^k$ for any constant k, $g = b^n$ for any constant $b > 1$ and $h = \lg n$. We have already proved that $f = o(g)$ and $h = \omega(1)$ since h is strictly lower bounded by constant. This implies that $f \circ h = o(g \circ h)$, where $f \circ h = \lg^k n$ and $g \circ h = b^{\lg n}$. Since $b > 1$, there must exist some constant ϵ where $\epsilon > 0$ such that $b = 2^\epsilon$. Replacing b with 2^ϵ, we have $g \circ h = (2^\epsilon)^{\lg n} = 2^{\lg n^\epsilon} = n^\epsilon$. Therefore, we have: $\lg^k n = o(n^\epsilon)$ for all constants $\epsilon > 0$ and k. □

2. The conjecture is false.

Proof. We prove by giving a counter example, that is, a function that is both super-polynomial and sub-exponential. For some $b > 1$, let $f(n) = n^{\log_b n}$. Because $\log_b n = \omega(1)$, thus $f(n) = \omega(n^c)$ for any constants c. Since $n = b^{\log_b n}$, $f(n) = (b^{\log_b n})^{\log_b n} = b^{\log_b^2 n}$. By taking a limit between $\log_b^2 n$ and n, and repeatedly using L'Hospitals rule, we get:

$$\lim_{n\to\infty} \frac{\log_b^2 n}{n} = \lim_{n\to\infty} \frac{2(\log_b n)\frac{1}{n\ln b}}{1} \tag{9}$$

$$= \frac{2}{\ln b} \lim_{n\to\infty} \frac{\log_b n}{n} \tag{10}$$

$$= \frac{2}{\ln b} \lim_{n\to\infty} \frac{\frac{1}{n\ln b}}{1} \tag{11}$$

$$= \frac{2}{\ln^2 b} \lim_{n\to\infty} \frac{1}{n} \tag{12}$$

$$= \frac{2}{\ln^2 b} \times 0 = 0 \tag{13}$$

Therefore $\log_b^2 n = o(n)$. Thus $f(n) = b^{\log_b^2 n} = o(b^n)$. Since $f(n)$ is strictly upper bounded by some exponential function b^n, the conjecture is not true. □

3. (a) Using the Master Theorem and its specializations given in class, we have: $a = 4, b = 2, c = \frac{5}{2}$ and $\log_b a = 2$. Since $c > \log_b a$, therefore $T(n) = \Theta(n^c) = \Theta(n^2\sqrt{n})$.
   (b) Using the Master Theorem and its specializations given in class, we have: $a = 3, b = 2, c = 1, d = 1$ and $\log_b a \approx 1.58$. Since $c < \log_b a$, therefore $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$.
   (c) Using the Master Theorem and its specializations given in class, we have: $a = 2, b = 2, c = 1, d = 1$ and $\log_b a = 1$. Since $c = \log_b a$, therefore $T(n) = \Theta(n^c \log^{d+1} n) = \Theta(n \log^2 n)$.

4. **Algorithm design using Divide and Conquer**

Problem: Find the missing integer.

Input: Array $A[1{:}n, 1{:}k]$ of bits, where $k = \Theta(\log n)$. Each row of A represents an integer in the range [0,n], in binary format. All integers are represented by rows of A, except one is missing.

Output: Find the missing integer.

Idea: To represent the integers in the range [0,n], we need at least $\lceil \lg(n+1) \rceil$ bits. We define LSB(least significant bit) as 1_{th} bit, and MSB(most significant bit) as $\lceil \lg(n+1) \rceil_{th}$ bit such that they correspond to the column index of A.

Claim: For integers in the range [0, n], there are exactly $2^{\lceil \lg(n+1) \rceil - 1}$ integers that are less than $2^{\lceil \lg(n+1) \rceil - 1}$.

We can start by examining the MSB for each integer represented in A. During the examination, we use two lists, *LowerList* and *UpperList*, to keep track of the row indexes of the integers that are $< 2^{\lceil \lg(n+1) \rceil - 1}$ or $\geq 2^{\lceil \lg(n+1) \rceil - 1}$ respectively.

After the examination, we compare the size of *LowerList* with $2^{\lceil \lg(n+1) \rceil - 1}$, if it is $<$, we know that the missing integer must have a 0 at the MSB_{th} bit, then we can make a recursive call on integers in A with row indexes stored in *LowerList*. Otherwise, the missing integer must have a 1 at the MSB_{th} bit, accordingly we can make a recursive call on integers in A with row indexes stored in *UpperList*.

Pseudocode:

```
1. Function FindMissingInteger(RowIndexRef, n)
2.       LowerList := [ ]
3.       UpperList := [ ]
4.       MSBIndex := ⌈lg(n + 1)⌉
5.       if (n = 1) // base case
6.            RowIndex := RowIndexRef[ 1 ]
7.            print "The 1_th missing bit is (1 - A[ RowIndex, MSBIndex ])
8.            return
9.       for (i := 1 to n) // build two lists
10.           RowIndex := RowIndexRef[ i ]
11.           if (A[ RowIndex, MSBIndex ] = 0)
12.                LowerList.insert(RowIndex)
13.           else
14.                UpperList.insert(RowIndex)
15.      if (UpperList.size = 0) // corner case
16.           print "The MSBIndex_th missing bit is 1."
17.           for j := MSBIndex-1 to 1
18.                print "The j_th missing bit is 0."
19.           return
20.      if (LowerList.size < 2^⌈lg(n+1)⌉−1)
21.           print "The MSBIndex_th missing bit is 0."
22.           FindMissingInteger(LowerList, LowerList.size)
23.      else
24.           print "The MSBIndex_th missing bit is 1."
25.           FindMissingInteger(UpperList, UpperList.size)
```

Note that though not shown in the pseudocode, we do need to build the initial row index reference (which is [1...n]) before making the first call to the function. And line-15 to line-19 handles a corner case where $2^{\lceil \lg(n+1) \rceil - 1} = n$, which implies that the missing number is n because UpperList is empty and UpperList can possibly refer to only one element which is n.

Proof of correctness

First, the algorithm will always terminate with every input of size $n \geq 1$ because for each recursive call to the function, the input size shrinks to at most $2^{\lceil \lg(n+1) \rceil - 1} - 1$. So eventually it will hit the base case and return back.

Second, we use strong induction on n to prove the correctness of the algorithm.

Proof. Base case: $n = 1$. The algorithm handles the base case and returns the missing integer correctly.

Inductive step: Suppose that when $n = 1, 2, \cdots, k$, where k is some integer > 1, the algorithm works correctly. We prove that algorithm also is correct for $n = k + 1$. We show that the size of the sub-problem we could possibly get is $\leq k$. If the algorithm recurses on LowerList, its size is $2^{\lceil \lg(k+2) \rceil - 1} - 1$. Because,

$$2^{\lceil \lg(k+2) \rceil - 1} \leq k + 1 \tag{14}$$

Therefore,

$$2^{\lceil \lg(k+2) \rceil - 1} - 1 \leq k \tag{15}$$

If the algorithm recurses on UpperList, its size is $(k+1) - 2^{\lceil \lg(k+2) \rceil - 1}$. Because,

$$2^{\lceil \lg(k+2) \rceil - 1} \geq 1 \tag{16}$$

Therefore,

$$(k+1) - 2^{\lceil \lg(k+2) \rceil - 1} \leq k \tag{17}$$

By the induction hypothesis, we know that the algorithm can solve the sub-problem correctly when $n = k + 1$, thus it can solve the problem correctly. □

Run time analysis

Assume that list initialization, insertion all take constant time. Line-2 to line-8 takes $\Theta(1)$ time. Line-9 to line-14 takes $\Theta(n)$ time. Line-15 to line-19 takes $\Theta(\log n)$ time. For line-20 to line-25, we are solving only one sub-problem for at most $2^{\lceil \lg(n+1) \rceil - 1} - 1$ times. Let $T(n)$ be the run time function for the algorithm, we have:

$$\begin{aligned} T(n) &\leq T(2^{\lceil \lg(n+1) \rceil - 1} - 1) + \Theta(n) + \Theta(\log n) + \Theta(1) \\ &= T(2^{\lceil \lg(n+1) \rceil - 1} - 1) + \Theta(n) \end{aligned} \tag{18}$$

Let $k = \lceil \lg(n+1) \rceil$.

$$\begin{aligned} T(n) &\leq T(2^{k-1} - 1) + \Theta(2^k - 1) \\ &= T(2^{k-2} - 1) + \Theta(2^k - 1) + \Theta(2^{k-1} - 1) \\ &= T(2^1 - 1) + \sum_{i=2}^{k} \Theta(2^i - 1) \\ &= T(1) + \Theta(\sum_{i=2}^{k} (2^i - 1)) \\ &= \Theta(1) + \Theta(2^{k+1} - 4) \\ &= \Theta(2^{k+1} - 4) \\ &= 2\Theta(2^k - 2) \\ &\leq 2\Theta(n) \end{aligned} \tag{19}$$

5. **Algorithm design using Divide and Conquer**

To find a minimum positive-sum subarray for array $A[low, high]$, we divide the subarray into two subarrays at the midpoint mid, then the minimum positive-sum subarray $A[i, j]$ must be in one of the following places:

- entirely in the subarray $A[low, mid]$, such that $low \leq i \leq j \leq mid$
- entirely in the subarray $A[mid + 1, high]$, such that $mid + 1 \leq i \leq j \leq high$
- crossing the midpoint, such that $low \leq i \leq mid < j \leq high$.

We can find the minimum positive-sum subarrays for $A[low, mid]$ and $A[mid + 1, high]$ recursively because they are just smaller instances of the same problem. Thus, all that is left to do is find a minimum positive-sum subarray that crosses the midpoint, and take the minimum of the three cases.

The pseudocode is given here:

1. **Function** FindMinPosSumSubarray$(A, low, high)$
2. **if** $low = high$
3. **return** $(low, high, A[low])$
4. $mid := \lfloor (low + high)/2 \rfloor$
5. $(left\_low, left\_high, left\_sum)$:= FindMinPosSumSubarray(A, low, mid)
6. $(right\_low, right\_high, right\_sum)$:= FindMinPosSumSubarray$(A, mid + 1, high)$
7. $B := [\,]$, $C := [\,]$
8. ComputePrefixSum(B, low, mid, A)
9. ComputePrefixSum$(C, mid + 1, high, A)$
10. MergeSort(B)
11. MergeSort(C)
12. $(left\_low, left\_high, left\_sum)$:= ScanPerfixSum(B)
13. $(right\_low, right\_high, right\_sum)$:= ScanPerfixSum(C)
14. $(cross\_low, cross\_high, cross\_sum)$:= FindMinCrossPosSumSubarray$(A, B, C, low, mid, high)$
15. **if** ($left\_sum$ is the minimum positive sum)
16. **return** $(left\_low, left\_high, left\_sum)$
17. **elseif** ($right\_sum$ is the minimum positive sum)
18. **return** $(right\_low, right\_high, right\_sum)$
19. **elseif** ($cross\_sum$ is the minimum positive sum)
20. **return** $(cross\_low, cross\_high, cross\_sum)$
21. **elseif** (no positive sum exists)
22. **return** (the one with absolute value that is closest to zero)

At line-8 and line-9, we use two additional arrays to store prefix sum for each subarray computed by the subroutine *ComputePrefixSum*. Then we merge sort B and C. The subroutine *ScanPerfixSum* just does a linear scan on B and C to find out the minimum positive-sum of two subarrays. The subroutine *FindMinCrossPosSumSubarray* is used to find out the minimum positive-sum that cross the midpoint. It does the following:

- For each positive element k in B, use $1 - k$ as a key and do a binary search against C to find the closest element to the key. Also keep track of the minimum positive sum and its index interval.
- Do the same for C on B.

Run time analysis

ComputePrefixSum requires $\Theta(n)$ time. *MergeSort* requires $\Theta(n \log n)$ time. *ScanPerfixSum* requires $\Theta(n)$ time since both B and C are sorted. *FindMinCrossPosSumSubarray* requires $\Theta(n \log n)$ times since both B and C are sorted and binary search needs $\Theta(\log n)$ time. Putting together, we have:

$$T(n) = 2T(n/2) + \Theta(n \log n) \tag{20}$$

By the extended form of the Master Theorem given in the class, we have $a = 2, b = 2, c = 1, d = 1$ and $\log_b a = c$, thus $T(n) = \Theta(n^c \log^{d+1} n) = \Theta(n \log^2 n)$.

6. (a) Suppose n chips are $c_1, c_2, \cdots, c_n$ among which $c_1, c_2, \cdots, c_k$ are good chips where $k \leq n/2$. These good chips identify each other as good and all the others as bad. It is possible that the same number of bad chips can behave exactly like those good ones, that is, they also identify

each other as good and all the others as bad, say these bad chips are $c_{k+1}, c_{k+2}, \cdots, c_{2k}$. Then there are three possible test outcomes:

i. all chips are bad;

ii. $c_1, c_2, \cdots, c_k$ are good;

iii. $c_{k+1}, c_{k+2}, \cdots, c_{2k}$ are good;

No strategy can distinguish these three possibilities.

(b) The algorithm for finding one good chip from among n chips is:

i. If there is only 1 chip left, then it must be good.

ii. Split all chips into pairs. If n is odd, there will be a stand alone chip.

iii. Do pairwise test. For each pair, if the result is both good, arbitrarily throw away one chip. If the result is one good, one bad or both bad, then throw away the pair.

iv. Go back to step i.

The algorithm takes $\Theta(\frac{n}{2} + \frac{n}{2^2} + \cdots + 1) = \Theta(n)$ pairwise tests.

To reduce the problem to half of size, we show that after step i, step ii and step iii, at least half of the remaining chips are good. Suppose after step ii and step iii, there are x pairs show good/good result, y pairs show one-good/one-bad result and z pairs show bad/bad result. If the result is good/good, then both chips are either good or bad, otherwise at least one of the chips is bad. Since we throw away $x + 2(y + z)$ chips, so only x chips remain. and Among $2(y + z)$ chips, at least half of them are bad.

If n is even, then $n = 2(x + y + z)$. Suppose more than half of the remaining x chips are bad, then the chips they were paired with are also bad. This means

$$\begin{aligned} number\_of\_bad\_chips &> \frac{x}{2} \times 2 + y + z \\ &= x + y + z = \frac{n}{2} \end{aligned} \tag{21}$$

This contradicts the fact that more than half of n chips are good.

When n is odd, then $n = 2(x + y + z) + 1$, and $x + 1$ chips remain. If the stand alone chip is bad, the following relation must hold:

$$\begin{aligned} number\_of\_bad\_chips &\leq \frac{x-1}{2} \times 2 + y + z + 1 \\ &= x + y + z \end{aligned} \tag{22}$$

This means among the remaining x chips, there must be at least $(x - 1)/2 + 1 = (x + 1)/2$ good chips.
If the stand alone chip is good, the following relation must hold:

$$\begin{aligned} number\_of\_bad\_chips &\leq \frac{x}{2} \times 2 + y + z \\ &= x + y + z \end{aligned} \tag{23}$$

This means among the remaining $x + 1$ chips, there must be at least $x/2 + 1$ good chips.
Based on the above analysis, we can conclude that the remaining chips contain at least half good chips.

(c) First, we use the algorithm given in part-b to find a good chip. This takes $\Theta(n)$ pairwise tests. Then, we use this good chip to test all the other $n-1$ chips to find out other good ones. This also takes $\Theta(n)$ pairwise tests. Thus the total number of pairwise tests is still $\Theta(n)$.