# Sample Solution for Problem Set 1

Data Structures and Algorithms, Fall 2019

September 19, 2019

**2.1-3 (CLRS 3ed).** See Figure 1 for the pseudocode.

```
LinearSearch(A, v)
1: idx = NIL.
2: for (i = 1 to A.length) do
3:     if (A[i] == v) then
4:         idx = i.
5: return idx.
```

**Figure 1**

**The Loop Invariant:** At the beginning of each of the **for** loop, if $v$ does not exist in $A[1 \cdots (i-1)]$ then $idx$ equals $NIL$, otherwise $idx$ contains an index satisfying $A[idx] = v$.

**Initialization:** At the beginning of the first loop, we have $i = 1$, implying $A[1 \cdots (i-1)]$ is an empty sub-array. Thus, $v$ does not exist in $A[1 \cdots (i-1)]$, and indeed we have $idx = NIL$.

**Maintenance:** Assume at the beginning of an iteration we have $i = i'$.

If $A[1 \cdots i']$ does not contain $v$, then by the loop invariant, we know at the beginning of this loop, it must be the case that $idx = NIL$, as $A[1 \cdots (i'-1)]$ does not contain $v$. Moreover, according to the logic of the pseudocode, by the end of this loop, the value of $idx$ will remain to be $NIL$.

On the other hand, if $A[1 \cdots i']$ does contain $v$, then either $A[1 \cdots (i'-1)]$ contains $v$, or $A[i'] = v$, or both. If $A[1 \cdots (i'-1)]$ contains $v$, then by the loop invariant, we know at the beginning of this loop, it must be the case that $idx \neq NIL$ and $A[idx] = v$. Moreover, after this iteration, it will still be the case that $idx \neq NIL$ and $A[idx] = v$. If, however, $A[1 \cdots (i'-1)]$ does not contain $v$ but $A[i'] = v$, then after this iteration, we have $idx = i'$. Either way, in case $A[1 \cdots i']$ does contain $v$, by the end of the current iteration, we have $idx \neq NIL$ and $A[idx] = v$.

**Termination:** The loop will exit when $i = A.length + 1$. By the loop invariant, we know $idx$ will be $NIL$ if $A[1 \cdots (A.length)]$ does not contain $v$; otherwise, $idx$ will contain an index satisfying $A[idx] = v$. This exactly satisfies the requirements posed by the problem. Thus, by returning $idx$, our algorithm correctly solves the problem.

**2.2-2 (CLRS 3ed).** See Figure 2 for the pseudocode.

```
SelectionSort(A)
1: for (i = 1 to n − 1) do
2:     minIdx = i.
3:     for (j = i + 1 to n) do
4:         if (A[j] < A[minIdx]) then
5:             minIdx = j.
6:     Swap(A[i], A[minIdx]).
```

**Figure 2**

One possible loop invariant for the outer **for** loop is: at the beginning of each of the outer **for** loop, $A[1 \cdots (i-1)]$ contains the smallest $i-1$ elements in the original array, in sorted order. As for the inner **for** loop, one possible loop invariant is: at the beginning of each of the inner **for** loop, $A[minIdx]$ is the smallest element in $A[i \cdots (j-1)]$.

The reason that the outer loop only need to run $n-1$ times is: by the above loop invariant, when the outer loop exits, $A[1 \cdots (n-1)]$ contains the $n-1$ smallest elements in the original array, in sorted order. Thus, $A[n]$ has to contain the original largest element, implying the entire array is sorted.

For every input array $A$ of size $n$, the runtime is $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Theta(1) = \Theta(n^2)$. Thus, both the best-case and the worst-case runtime of selection sort is $\Theta(n^2)$.

**2.2-4 (CLRS 3ed).** The basic idea is to pre-compute the result for one particular input $I$ and add the result to the code. Then, when the algorithm executes, it first checks whether the input is $I$. If this is the case, the algorithm outputs the pre-computed result directly. Otherwise, the algorithm runs normally.

More formally, let us consider the class of algorithms that have to look at the entire input at least once. That is, algorithms with best-case runtime $\Omega(n)$, where $n$ is the size of the input. Pick one such algorithm $\mathcal{A}$, assume it achieves best-case runtime $f(n) \in \Omega(n)$ on input $I$. Now, we build another algorithm $\mathcal{A}'$ in the following way. Given an input $I'$, $\mathcal{A}'$ first checks whether $I = I'$, and this takes $\Theta(n)$ time. If indeed $I = I'$, then $\mathcal{A}'$ outputs the pre-computed result directly. Otherwise, $\mathcal{A}'$ executes $\mathcal{A}$. Assume $\mathcal{A}'$ has runtime $g(n)$ on input $I$. Clearly, the best-case runtime of $\mathcal{A}'$ is $O(g(n))$, and $g(n) \in O(f(n))$. In fact, it may well be the case that $g(n) \in o(f(n))$, as $\mathcal{A}'$ does not need to *compute* the result for instance $I$, it simply outputs the result directly.

**2-3 (CLRS 3ed). (a).** The **for** loop executes $\Theta(n)$ times, each of which costs $\Theta(1)$. Thus, the total runtime of the code fragment is $\Theta(n)$.

**(b).** See Figure 3 for the pseudocode.

---

`NaiveEval`$(A, x)$

---

1: $y = 0$
2: **for** ($k = 0$ to $n$) **do**
3: $\quad pow = 1.$
4: $\quad$ **for** ($i = 1$ to $k$) **do**
5: $\quad\quad pow = pow * x.$
6: $\quad y = y + A[k] * pow.$
7: **return** $y$.

---

**Figure 3**

It is easy to see the time complexity is $\Theta(n^2)$, which is much slower than Horner's rule.

**(c). Initialization:** Prior to the first loop, we have $i = n$, thus $y = 0$.

**Maintenance:** Assume at the beginning of an iteration we have $i = i'$. By the loop invariant, we know $y = \sum_{k=0}^{n-(i'+1)} a_{k+i'+1} x^k$. Therefore, by the end of this iteration, we have

$$y = a_{i'} + x \sum_{k=0}^{n-(i'+1)} a_{k+i'+1} x^k = a_{i'} * x^0 + \sum_{k=0}^{n-i'-1} a_{k+i'+1} x^{k+1}$$

$$= a_{i'} * x^0 + \sum_{k=1}^{n-i'} a_{k+i'} x^k = \sum_{k=0}^{n-i'} a_{k+i'} x^k$$

and this is exactly what we need.

**Termination:** The loop exits when $i = -1$. By then, we have $y = \sum_{k=0}^{n} a_k x^k$.

**(d).** We have proved the correctness of the loop invariant in part (c). Moreover, the loop invariant implies $y$ has the correct value when the loop exits. Thus, the code fragment correctly evaluates the polynomial.

**3.1-1 (CLRS 3ed).** Since $f$ and $g$ are asymptotically nonnegative functions, we know:

$$\exists n_1 > 0, \forall n > n_1, f(n) \geq 0,$$
$$\exists n_2 > 0, \forall n > n_2, g(n) \geq 0.$$

Let $N = \max(n_1, n_2)$, we know:

$$\forall n > N, 0 \leq \frac{f(n) + g(n)}{2} \leq \max\left(f(n), g(n)\right) \leq f(n) + g(n)$$

Therefore, by definition, $\max(f(n), g(n)) \in \Theta(f(n) + g(n))$.

**3.1-8 (CLRS 3ed).**

$\Omega(g(n,m)) = \{f(n,m) \mid \exists c, n_0, m_0 > 0, \forall n, m$ such that $n \geq n_0$ or $m \geq m_0, 0 \leq c \cdot g(n,m) \leq f(n,m)\}$,
$\Theta(g(n,m)) = \{f(n,m) \mid \exists c_1, c_2, n_0, m_0 > 0, \forall n, m$ such that $n \geq n_0$ or $m \geq m_0, 0 \leq c_1 \cdot g(n,m) \leq f(n,m) \leq c_2 \cdot g(n,m)\}$.

**3.2-4 (CLRS 3ed). (a).** For sufficiently large $n$, we have:

$$\lceil \lg n \rceil! \geq \left(\frac{\lg n}{2}\right)^{(\lg n)/2} = \left(\frac{2^{\lg\lg n}}{2}\right)^{(\lg n)/2} = \left(2^{(\lg\lg n)-1}\right)^{(\lg n)/2} = n^{((\lg\lg n)-1)/2} \geq n^{(\lg\lg n)/3}$$

On the other hand, for any polynomial $p(n)$ with max degree $c \in \Theta(1)$, when $n$ is sufficiently large, we have $p(n) \leq n^{c+1}$.

Observe that $\lim_{n\to\infty} (n^{(\lg\lg n)/3}/n^{c+1}) = \infty$. Therefore, $\lceil \lg n \rceil!$ is not polynomially bounded.

**(b).** For sufficiently large $n$, we have:

$$\lceil \lg\lg n \rceil! \leq ((\lg\lg n) + 1)^{(\lg\lg n)+1} \leq (\lg n)^{(\lg\lg n)+1} = 2^{\lg\lg n \cdot ((\lg\lg n)+1)} \leq 2^{\lg n} = n$$

Therefore, $\lceil \lg\lg n \rceil!$ is polynomially bounded.

**3.2-5 (CLRS 3ed).** By the definition of the iterative logarithm function, we know $\lg^*(2^n) = 1 + \lg^* n$. As a result:

$$\lim_{n\to\infty} \frac{\lg(\lg^* n)}{\lg^*(\lg n)} = \lim_{n\to\infty} \frac{\lg(\lg^*(2^n))}{\lg^*(\lg 2^n)} = \lim_{n\to\infty} \frac{\lg(1 + \lg^* n)}{\lg^* n} = \lim_{n\to\infty} \frac{\lg(1+n)}{n} = 0$$

Therefore, $\lg^*(\lg n)$ is asymptotically larger.

**3-2 (CLRS 3ed).** See the following table.

| $A$ | $B$ | $O$ | $o$ | $\Omega$ | $\omega$ | $\Theta$ |
|---|---|---|---|---|---|---|
| $\lg^k n$ | $n^\epsilon$ | Y | Y | N | N | N |
| $n^k$ | $c^n$ | Y | Y | N | N | N |
| $\sqrt{n}$ | $n^{\sin n}$ | N | N | N | N | N |
| $2^n$ | $2^{n/2}$ | N | N | Y | Y | N |
| $n^{\lg c}$ | $c^{\lg n}$ | Y | N | Y | N | Y |
| $\lg(n!)$ | $\lg(n^n)$ | Y | N | Y | N | Y |

**3-3 (CLRS 3ed).** **(a).** In the following, "$f \geq g$" means $f \in \Omega(g)$, and "$f = g$" means $f \in \Theta(g)$:

$$\begin{gathered}
2^{2^{n+1}} \geq 2^{2^n} \\
\geq (n+1)! \geq n! \\
\geq e^n \geq n \cdot 2^n \geq 2^n \geq (3/2)^n \\
\geq n^{\lg \lg n} = (\lg n)^{\lg n} \geq (\lg n)! \\
\geq n^3 \geq 4^{\lg n} = n^2 \geq \lg(n!) = n \lg n \geq 2^{\lg n} = n \geq (\sqrt{2})^{\lg n} \\
\geq 2^{\sqrt{2 \lg n}} \\
\geq \lg^2 n \geq \ln n \geq \sqrt{\lg n} \geq \ln \ln n \\
\geq 2^{\lg^* n} \\
\geq \lg^* n = \lg^* (\lg n) \geq \lg(\lg^* n) \\
\geq n^{(1/\lg n)} = 1
\end{gathered}$$

**(b).** One possible $f(n)$ is defined as follows:

$$f(n) = \begin{cases} 5^{2^n} & n \text{ is even} \\ 1/n & n \text{ is odd} \end{cases}$$

Particularly, when $n \geq 0$ is even, $f(n) = \omega(2^{2^{n+1}})$; and when $n \geq 0$ is odd, $f(n) = o(1)$.

**Bonus problem.** **(a).** The basic idea to do bit-wise XOR operation on these $n$ numbers sequentially, and the final result will be the number that appears only once. See Figure 4 for the pseudocode.

```
FindEleAppearingOnce(S)
1: res = S[1]                          ▷ res is a 32-bit positive integer.
2: for (i = 2 to S.length) do
3:     res = res ⊕ S[i].               ▷ ⊕ denotes the bit-wise XOR operation.
4: return res.
```

**Figure 4**

Since computing the bit-wise XOR of two 32-bit numbers takes $O(1)$ time, the time complexity of the above algorithm is $O(n)$. Moreover, since the size of $res$ is 32 bits, the space complexity is $O(1)$.

We still need to argue the correctness of the algorithm. Assume $S = \langle s_1, s_2, s_3, \cdots, s_n \rangle$, and it is a permutation of $\langle a_1, a_1, a_2, a_2, a_3, a_3, \cdots, a_k, a_k, a_{k+1} \rangle$. Our algorithm returns $res = s_1 \oplus s_2 \oplus \cdots \oplus s_n$. Since bit-wise XOR is commutative and associative, we know $res = (a_1 \oplus a_1) \oplus (a_2 \oplus a_2) \oplus \cdots \oplus (a_k \oplus a_k) \oplus a_{k+1} = 0 \oplus 0 \oplus \cdots \oplus 0 \oplus a_{k+1} = a_{k+1}$. That is, eventually, $res$ stores the number that appears only once in $S$.

**(b).** The reason that bit-wise XOR works in part (a) is that bit-wise XOR is "bit-wise modulo two sum". Thus, to solve part (b), we need "bit-wise modulo three sum". See Figure 5 for the pseudocode.

It is easy to verify the time complexity of the algorithm is $O(n)$, and the space complexity of the algorithm is $O(1)$.

To prove the correctness of the algorithm, we follow the same path as in part (a). Assume $S = \langle s_1, s_2, s_3, \cdots, s_n \rangle$, and it is a permutation of $\langle a_1, a_1, a_1, a_2, a_2, a_2, a_3, a_3, a_3, \cdots, a_k, a_k, a_k, a_{k+1} \rangle$. Let $\hat{\oplus}$ denote the "bit-wise modulo three sum" operation, our algorithm returns $res = s_1 \hat{\oplus} s_2 \hat{\oplus} \cdots \hat{\oplus} s_n$. Since $\hat{\oplus}$ is commutative and associative, $res = (a_1 \hat{\oplus} a_1 \hat{\oplus} a_1) \hat{\oplus} (a_2 \hat{\oplus} a_2 \hat{\oplus} a_2) \hat{\oplus} \cdots \hat{\oplus} (a_k \hat{\oplus} a_k \hat{\oplus} a_k) \hat{\oplus} a_{k+1} = 0 \hat{\oplus} 0 \hat{\oplus} \cdots \hat{\oplus} 0 \hat{\oplus} a_{k+1} = a_{k+1}$. That is, eventually, $res$ stores the number that appears only once in $S$.

`FindEleAppearingOnce`($S$)

▷ $arr$ is an integer array of length 32, where $arr[i]$ stores the $i^{\text{th}}$ least significant bit of $S[1]$.

1: $arr =$ `ToBitArray`($S[1]$).
2: **for** ($i = 2$ to $S.length$) **do**
3: **for** ($j = 1$ to 32) **do**
4: $arr[j] = (arr[j] + ((S[i] >> (j-1))\&1)) \mod 3$.

▷ Convert $arr$ to a number and return it.

5: $res = 0$.
6: **for** ($j = 1$ to 32) **do**
7: **if** ($arr[j] \neq 0$) **then**
8: $res = res + (1 << (j-1))$.
9: **return** $res$.

**Figure 5**