

Recitation 2 (HW1)

Online: Xinyi Zhao xz2833@nyu.edu

GCASL 475: Yifan Jin yj2063@nyu.edu

New York University

Basic Algorithms (CSCI-UA.0310-005)

Correctness of the merge function used in merge sort using Loop Invariant

Recall MergeSort($A[1\dots n]$):

1. MergeSort($A[1\dots n/2]$)
2. MergeSort($A[n/2+1\dots n]$)
3. Merge($A[1\dots n/2]$, $A[n/2+1\dots n]$)

Correctness of the merge function used in merge sort using Loop Invariant

B and C are sorted.

```
Merge(B[1...k], C[1...m]):  
    define D[1...k+m]  
    i = 1  
    j = 1  
    for l = 1 to k+m:  
        if j > m or (i <= k and B[i] <= C[j]):  
            D[l] = B[i]  
            i = i + 1  
        else:  
            D[l] = c[j]  
            j = j + 1  
    return D
```

Correctness of the merge function used in merge sort using Loop Invariant

The **Loop Invariant** should be as follows:

At the **end** of the for-loop iteration for index l , $D[1...l]$ consists of **the smallest l elements** of B and C **in sorted order**, and these elements are **$B[1...i-1]$ and $C[1...j-1]$** .

Correctness of the merge function

(I) Initialization: check that the loop invariant holds for $l=1$:

At the **end** of the for-loop iteration for index **1**, $D[1]$ is the smaller element of $B[1]$ and $C[1]$, $D[1] = \min(B[1], C[1])$ (**also the smallest among B and C**).

If $B[1] \leq C[1]$, $D[1] = B[1]$, $i = 2$, $j = 1$;

else $D[1] = C[1]$, $i = 1$, $j = 2$.

So the element are **$B[1...i-1]$ and $C[1...j-1]$** .

Also, one element **is sorted already**.

```
Merge(B[1...k], C[1...m]):  
    define D[1...k+m]  
    i = 1  
    j = 1  
    for l = 1 to k+m:  
        if j > m or (i <= k and B[i] <= C[j]):  
            D[l] = B[i]  
            i = i + 1  
        else:  
            D[l] = C[j]  
            j = j + 1  
    return D
```

Correctness of the merge function

(II) Maintenance:

Assumption: assume that the loop invariant holds for $l = p$: At the **end** of the for-loop iteration for index **p**, $D[1...p]$ consists of the **smallest p elements** of B and C **in sorted order**, and these elements **are B[1...i-1] and C[1...j-1]**.

Correctness of the merge function

(II) Maintenance:

Conclusion: prove that the loop invariant holds for $l = p+1$.

Since $D[1..p]$ consists of the smallest p elements of B and C and $D[1..p]$ are elements from $B[1..i-1]$ and $C[1..j-1]$, and $D[p+1] = \min(B[i], C[j])$, we can conclude that $D[p+1] \geq D[p]$, **so $D[1..p+1]$ is also sorted.**

Also, since B and C are sorted and $D[p+1] = \min(B[i], C[j])$, we can conclude that $D[1..p+1]$ consists of **the $p+1$ smallest elements in B and C .**

Also, If $B[i] \leq C[j]$, $D[p+1] = B[i]$, $i = i+1$; else $D[p+1] = C[j]$, $j = j+1$.
So we know the elements in $D[1..p+1]$ **are $B[1..i-1]$ and $C[1..j-1]$.**

Hence proved

Correctness of the merge function

(III)**Termination**: check that the loop invariant holds for $l=k+m$:

At the **end** of the for-loop iteration for index $l = k+m$, $D[1...l]$ consists of **all the elements** of A and B **in sorted order**, $i = k+1$, $j = m+1$, and these elements are **$A[1...k]$ and $B[1...m]$** .

```
Merge(B[1...k], C[1...m]):  
    define D[1...k+m]  
    i = 1  
    j = 1  
    for l = 1 to k+m:  
        if j > m or (i <= k and B[i] <= C[j]):  
            D[l] = B[i]  
            i = i + 1  
        else:  
            D[l] = C[j]  
            j = j + 1  
    return D
```


Problem 1

Use induction to prove the following for every positive integer n :

$$1 \times 1! + 2 \times 2! + \cdots + n \times n! = (n + 1)! - 1.$$

Note that your answer should follow the following format:

1. Base case: Check that the statement holds for the base case $n = 1$.
2. Induction step:
 - Assumption: Assume that the statement holds for $n = k$.
 - Conclusion: Use the assumption to show that the statement holds for $n = k + 1$.

Recall: We have $n! = 1 \times 2 \times \cdots \times n$. Example: $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$.

Problem 1

Prove: $1 \cdot 1! + 2 \cdot 2! + \dots + n \cdot n! = (n+1)! - 1$

1. Base Case: ?

Problem 1

Prove: $1 \cdot 1! + 2 \cdot 2! + \dots + n \cdot n! = (n+1)! - 1$

1. Base Case: When $n = 1$, $1 \cdot 1 = (1+1)! - 1 = 1$, proved
2. Induction Step:
 - a. Assumption: ?

Problem 1

Prove: $1 \cdot 1! + 2 \cdot 2! + \dots + n \cdot n! = (n+1)! - 1$

1. Base Case: When $n = 1$, $1 \cdot 1! = (1+1)! - 1 = 1$, proved
2. Induction Step:
 - a. Assumption: Assume it holds for k , $1 \cdot 1! + 2 \cdot 2! + \dots + k \cdot k! = (k+1)! - 1$
 - b. Conclusion: ?

Problem 1

Prove: $1*1! + 2*2! + \dots + n*n! = (n+1)! - 1$

1. Base Case: When $n = 1$, $1*1 = (1+1)! - 1 = 1$, proved
2. Induction Step:
 - a. Assumption: Assume it holds for k , $1*1! + 2*2! + \dots + k*k! = (k+1)! - 1$
 - b. Conclusion: for $n = k+1$,

$$\begin{aligned} &1*1! + 2*2! + \dots + k*k! + (k+1)*(k+1)! = (k+1)! - 1 + (k+1)*(k+1)! \\ &= (k+1)!*(1+k+1) - 1 = (k+2)*(k+1)! - 1 = (k+2)! - 1. \end{aligned}$$

Hence this statement holds for $n = k+1$.

Problem 2

Problem 2 (13 points)

Consider the following recursion with $a_1 = 3$, $a_2 = 5$, and for every integer $n \geq 2$ we have:

$$a_{n+1} = 3a_n - 2a_{n-1}.$$

Use strong induction to prove that $a_n = 2^n + 1$.

Note that your answer should follow the following format:

1. Base cases: Check that the statement holds for the base cases $n = 1, 2$.
2. Induction step:
 - Assumption: Assume that the statement holds for $n = 1, \dots, k$.
 - Conclusion: Use the assumption to show that the statement holds for $n = k + 1$.

Problem 2

Base Case:

Problem 2

Base Case:

$$n = 1, a_1 = 2^1 + 1 = 3; \quad n = 2, a_2 = 2^2 + 1 = 5$$

Problem 2

Base Case:

$$n = 1, a_1 = 2^1 + 1 = 3; \quad n = 2, a_2 = 2^2 + 1 = 5$$

Induction Step:

Assume it holds for $n=1\dots k$,

thus it holds for $k, k-1 \rightarrow$ $a_{k-1} = 2^{k-1} + 1, \quad a_k = 2^k + 1$

Problem 2

Base Case:

$$n = 1, a_1 = 2^1 + 1 = 3; \quad n = 2, a_2 = 2^2 + 1 = 5$$

Induction Step:

Assume it holds for $n=1\dots k$,

thus it holds for $k, k-1 \rightarrow$
$$a_{k-1} = 2^{k-1} + 1, \quad a_k = 2^k + 1$$

Then we need to prove it also holds for $k+1$:

$$a^{k+1} = 3 * a_k - 2 * a_{k-1} = 3 * (2^k + 1) - 2 * (2^{k-1} + 1) = 2 * 2^k + 1 = 2^{k+1} + 1$$

Hence proved

Problem 3

Rank the following functions in order of their asymptotic growth. That is, find an order $f_1, f_2, f_3, \dots, f_6$, such that $f_1 = \mathcal{O}(f_2)$, $f_2 = \mathcal{O}(f_3)$, and so on. You do not need to provide an explanation of your ranking.

(a) 2^n

(b) $n^{1.5} \log_2 n$



(c) $n^2 - 1$

(d) $n!$

(e) $2^{\log_2 n}$

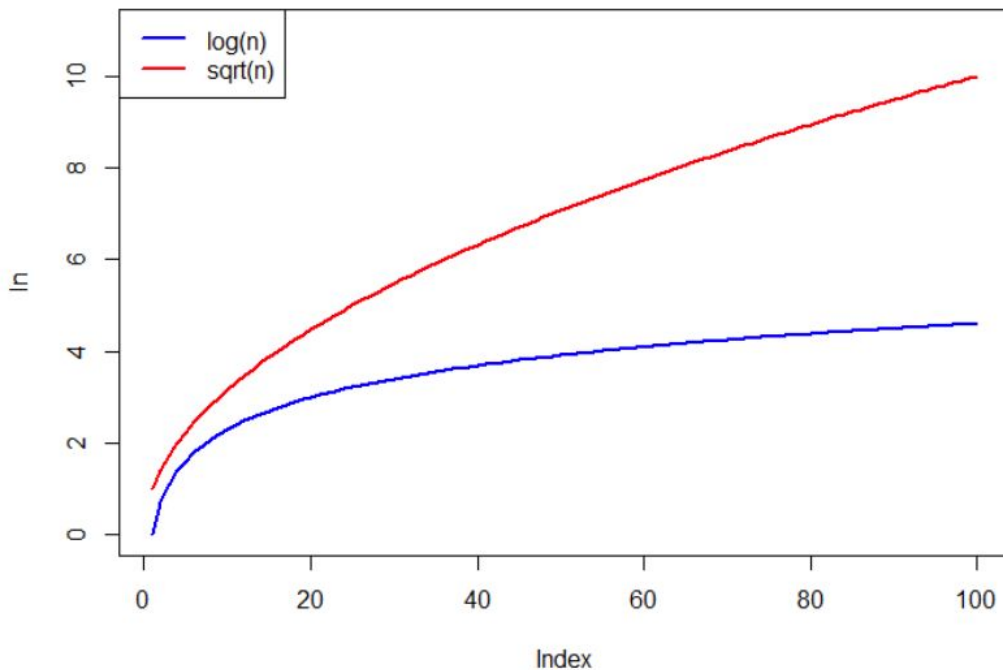
(f) 3^n

Problem 3

O	Complexity	Running Speed of Program with such time complexity	Rate of growth
$O(1)$	constant	fast	slow
$O(\log n)$	logarithmic		
$O(n)$	linear		
$O(n * \log n)$	log linear		
$O(n^2)$	quadratic		
$O(n^3)$	cubic		
$O(2^n)$	exponential	slow	fast
$O(n!)$	factorial		

Problem 3

- a. $2^n = O(2^n)$
- b. $n^{1.5} \log n = O(n^{1.5} \log n)$
- c. $n^2 - 1 = O(n^2)$
- d. $n! = O(n!)$
- e. $2^{(\log n)} = n = O(n)$
- f. $3^n = O(3^n)$



Problem 3

$$f_1 = n = O(n^{1.5 \log n}) \quad e$$

$$f_2 = n^{1.5 \log n} = O(n^2) \quad b$$

$$f_3 = n^2 = O(2^n) \quad c$$

$$f_4 = 2^n = O(3^n) \quad a$$

$$f_5 = 3^n = O(n!) \quad f$$

$$f_6 = n! \quad d$$

Problem 4

Problem 4 (6+6+6+6 points)

Prove or disprove: For each of the following, if it is true, then provide a proof, otherwise provide a counterexample and justify why the counterexample works.

- (a) If $f = \mathcal{O}(g)$ then $g = \mathcal{O}(f)$.
- (b) If $f = \Omega(h)$ and $g = \Omega(h)$, then $f + g = \Omega(h)$.
- (c) If $f = \mathcal{O}(g)$ and $g = \mathcal{O}(h)$ then $f = \mathcal{O}(h)$.
- (d) If $f = \Omega(g)$ and $h = \Omega(g)$ then $f = \Omega(h)$.

Problem 4

$$O \leq$$

$$o <$$

$$\Omega \geq$$

$$\omega >$$

$$\theta =$$

$$f = O(g) \quad f \leq g$$

$$g = \Omega(h) \quad g \geq f$$

Problem 4

(a) If $f = \mathcal{O}(g)$ then $g = \mathcal{O}(f)$.

Problem 4

(a) If $f = \mathcal{O}(g)$ then $g = \mathcal{O}(f)$.

False.

$$\begin{aligned} f &= n, \quad g = n^2, \\ f &= O(g), \quad g \neq O(f) \end{aligned}$$

Question:

$$g = \text{---}(f)?$$

Problem 4

(b) If $f = \Omega(h)$ and $g = \Omega(h)$, then $f + g = \Omega(h)$.

Problem 4

(b) If $f = \Omega(h)$ and $g = \Omega(h)$, then $f + g = \Omega(h)$.

True

h is the lower bound of both f and g . $f \geq h, g \geq h$

So h is also the lower bound of $f+g$. $f+g \geq h$

Problem 4

(c) If $f = \mathcal{O}(g)$ and $g = \mathcal{O}(h)$ then $f = \mathcal{O}(h)$.

Problem 4

(c) If $f = \mathcal{O}(g)$ and $g = \mathcal{O}(h)$ then $f = \mathcal{O}(h)$.

True

g is the upper bound of f , and h is the upper bound of g ,
so h is the upper bound of f .

$f \leq g$, $g \leq h$, so $f \leq h$

Problem 4

(d) If $f = \Omega(g)$ and $h = \Omega(g)$ then $f = \Omega(h)$.

Problem 4

(d) If $f = \Omega(g)$ and $h = \Omega(g)$ then $f = \Omega(h)$.

False

$$g = n, \quad f = n, \quad h = n^2$$
$$f = \Omega(g), \quad h = \Omega(g), \quad f \neq \Omega(h)$$

In other words, $f \geq g$, $h \geq g$, but we cannot compare f and g

Problem 5

Let's learn another sorting algorithm, SELECTION_SORT! In each iteration, SELECTION_SORT finds the largest element among the remaining elements and place it at the end of them.

Example:

$$\begin{aligned} A &= [5, 3, 7, 10, 1] \\ &\Rightarrow [5, 3, 7, 1, 10] \\ &\Rightarrow [5, 3, 1, 7, 10] \\ &\Rightarrow [3, 1, 5, 7, 10] \\ &\Rightarrow [1, 3, 5, 7, 10] \end{aligned}$$

- (a) Write the pseudo-code for SELECTION_SORT.
- (b) Find the time complexity of SELECTION_SORT in the best and worst case scenarios.
- (c) Now consider this variant of SELECTION_SORT: In each iteration, the algorithm finds the two largest elements among the remaining elements and place them at the end.
Is this variant more time-efficient compared to the original SELECTION_SORT? Justify your answer.

Problem 5

(a)

```
pseudo-code for selection sort (A[1...n]):  
  A: array of items  
  n: size of A  
  
  for i = n to 2  
    /* set current element as maximum */  
    max = i  
  
    /* check the element to be maximum */  
    for j = i-1 to 1  
      if A[j] > A[max] then  
        max = j;  
      end if  
    end for  
  
    /* swap the maximum element with the current element */  
    swap A[max] and A[i]  
  end for  
  
end procedure
```

Problem 5

(b) To sort an array with Selection Sort, we must iterate through the array once for each i loop, no matter what the array is.

Therefore the best and worst case time complexity of Selection Sort are the same, $O(n^2)$.

Unlike Insertion Sort, the running time of Selection Sort is independent of what the original array is.

Problem 5

(c) Variant: In each iteration, the algorithm finds the two largest elements among the remaining elements and place them at the end.

Consider the normal time complexity ($T(n)$) and the asymptotic time complexity (Big O notation).

Problem 5

(c) Variant: In each iteration, the algorithm finds the two largest elements among the remaining elements and place them at the end.

Two parts:

- the number of iterations.
- the time needed for each iteration.
 - In other words, the number of operations (comparison, swap) in each iteration

Assume n is even for simplicity.

Problem 5

(c) Variant:

Comparisons: at most $(k-1 + k-2)=2k-3$ comparisons to find the maximum and the second among k elements.

$$\text{Total comparisons} = \sum_{k=2}^n (2k - 3), k = 2, 4 \dots n$$

$$\sum_{k=2}^n (2k - 3) = 2 * 2 - 3 + 2 * 4 - 3 + \dots + 2 * n - 3$$

$$= 2 * 2(1 + 2 + \dots + n/2) - 3 * (n/2)$$

$$= \frac{n^2}{2} - \frac{n}{2}$$

Problem 5

(c) Variant:

Swap operations: in each iteration, we need to make at most 2 swaps to place the maximum and the second maximum at the end of the remaining elements. This gives us $2 \cdot (n/2) = n$ total swappings.

Problem 5

(c) Variant:

Define C_1 and C_2 as the real running time coefficients of comparison and swap.

Assume C_1 and C_2 are constant (based on computer architecture)

The total running time becomes:

$$T(n) = C_1 * \sum_{k=2}^n (2k - 3) + C_2 * n = C_1 * \left(\frac{n^2}{2} - \frac{n}{2}\right) + C_2 * n,$$

for some constants $C_1, C_2 > 0$.

$$T(n) = O(n^2)$$

Problem 5

(c) Original Selection Sort:

Comparisons: at most $k-1$ comparisons to find the maximum and the second among k elements

$$\text{Total comparisons} = \sum_{k=2}^n (k-1), k = 2, 3 \dots n$$

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

$$= \frac{n^2}{2} - \frac{n}{2}$$

Problem 5

(c) Original Selection Sort:

Swap operations: in each iteration, we need to make at most **1 swaps** to place the maximum at the end of the remaining elements. This gives us **n** total swappings.

Problem 5

(c) Original Selection Sort:

Define C_1' and C_2' as the real running time coefficients of comparison and swap.

The total running time becomes:

$$T'(n) = C_1' * \sum_{k=2}^n (k-1) + C_2' * n = C_1' * \left(\frac{n^2}{2} - \frac{n}{2}\right) + C_2' * n,$$

for some constants $C_1', C_2' > 0$.

$$T(n) = O(n^2)$$

Problem 5

(c)

By comparing $T(n)$ with $T'(n)$,

If $C1 = C1'$ and $C2 = C2'$, then $T(n) = T'(n)$.

Thus, the variant is **no** more time-efficient compared to the original Selection sort.

Problem 5

Follow-up questions:

1. Can you find the maximum and the second maximum among k elements using less than $2k-3$ comparisons?
2. What is the least number of comparisons among k elements you can get?
 - a. **Hint:** $k + \log k$
3. How does that improve the time efficiency of the variant algorithm?

Problem 6

Problem 6 (7+10 points)

Consider the following functions which both take as arguments three n -element arrays A , B , and C :

COMPARE-1(A, B, C)

For $i = 1$ **to** n

For $j = 1$ **to** n

If $A[i] + C[i] \geq B[j]$ **Return** FALSE

Return TRUE

COMPARE-2(A, B, C)

$aux := A[1] + C[1]$

For $i = 2$ **to** n

If $A[i] + C[i] > aux$ **Then** $aux := A[i] + C[i]$

For $j = 1$ **to** n

If $aux \geq B[j]$ **Return** FALSE

Return TRUE

Problem 6

(1) When do these two functions return TRUE?

```
COMPARE-1( $A, B, C$ )  
  For  $i = 1$  to  $n$   
    For  $j = 1$  to  $n$   
      If  $A[i] + C[i] \geq B[j]$  Return FALSE  
  Return TRUE
```

```
COMPARE-2( $A, B, C$ )  
   $aux := A[1] + C[1]$   
  For  $i = 2$  to  $n$   
    If  $A[i] + C[i] > aux$  Then  $aux := A[i] + C[i]$   
  For  $j = 1$  to  $n$   
    If  $aux \geq B[j]$  Return FALSE  
  Return TRUE
```

Problem 6

(1) When do these two functions return TRUE?

For all i, j in range $[1, n]$, if $\mathbf{A[i] + C[i] < B[j]}$ always holds, they return TRUE

In other words, $\mathbf{\max(A[i] + C[i]) < \min(B[j])}$

Problem 6

(2) What is the worst-case running time for each function?

```
COMPARE-1( $A, B, C$ )  
  For  $i = 1$  to  $n$   
    For  $j = 1$  to  $n$   
      If  $A[i] + C[i] \geq B[j]$  Return FALSE  
  Return TRUE
```

```
COMPARE-2( $A, B, C$ )  
   $aux := A[1] + C[1]$   
  For  $i = 2$  to  $n$   
    If  $A[i] + C[i] > aux$  Then  $aux := A[i] + C[i]$   
  For  $j = 1$  to  $n$   
    If  $aux \geq B[j]$  Return FALSE  
  Return TRUE
```

Problem 6

(2) What is the worst-case running time for each function?

In the worst case, both function need to execute all for-loops.

Thus, function 1 has the time complexity $O(n^2)$

function 2 has the time complexity $O(n)$

Problem 6

What can we learn from this problem?

Problem 6

What can we learn from this problem?

The two functions have the very same result but with different time complexity.

Problem 6

What can we learn from this problem?

The two functions have the very same result but with different time complexity.

Some mathematics property could optimize the algorithm, running time and space.

$$\mathbf{\max(A[i] + C[i]) < \min(B[j])}$$

Bonus Problem 1

We know that each of the first n positive integers, except one of them, appears in the array A **exactly once**. Thus, A has $n - 1$ elements in total. Our goal is to develop an algorithm to find the **missing number**.

For example:

Input: `nums = [9,6,4,2,3,5,7,1]`

Output: 8

Bonus Problem 1

(a) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

What kind of data structure can we use?

Bonus Problem 1

(a) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

What kind of data structure can we use?

- HashSet (add 1 to n to the set)
- Array ($A[n]$, use the index 0 to $n-1$)

Bonus Problem 1

(a) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

- HashSet (add 1 to n to the set)

(268. Missing Number)

```
1  class Solution {  
2      public int missingNumber(int[] nums) {  
3          Set<Integer> set = new HashSet<>();  
4          int n = nums.length;  
5          set.add(n);  
6          for (int i = 0; i < n; ++i) {  
7              set.add(i);  
8          }  
9          for (int i = 0; i < n; ++i) {  
10             set.remove(nums[i]);  
11         }  
12         return set.iterator().next();  
13     }  
14 }
```

Success Details >

Runtime: 11 ms, faster than 8.78% of Java online submissions for Missing Number.

Memory Usage: 54.4 MB, less than 5.95% of Java online submissions for Missing Number.

Bonus Problem 1

(a) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

- Array ($A[n]$, use the index 0 to $n-1$)

(268. Missing Number)

```
1 ▼ class Solution {  
2 ▼     public int missingNumber(int[] nums) {  
3         int n = nums.length;  
4         int[] arr = new int[n + 1];  
5  
6 ▼         for (int i = 0; i < n; ++i) {  
7             arr[nums[i]] = 1;  
8         }  
9 ▼         for (int i = 0; i <= n; ++i) {  
10             if (arr[i] == 0) return i;  
11         }  
12         return -1;  
13     }  
14 }
```

Success Details >

Runtime: 1 ms, faster than 73.24% of Java online submissions for Missing Number.

Memory Usage: 52 MB, less than 5.95% of Java online submissions for Missing Number.

Bonus Problem 1

(b) Improving previous part, now develop an algorithm which runs in linear time (in n) and requires constant amount of additional space.

Do we really need to store the integers?

Bonus Problem 1

(b) Improving previous part, now develop an algorithm which runs in linear time (in n) and requires constant amount of additional space.

Do we really need to store the integers? - No

Why? - They are continuous integers. We know the range.

How?

Bonus Problem 1

(b) Improving previous part, now develop an algorithm which runs in linear time (in n) and requires constant amount of additional space.

How?

- Sum of 1 to n (if n is very large, may cause overflow)
- Bitwise operation (XOR: exclusive or)

Bonus Problem 1

(b) Improving previous part, now develop an algorithm which runs in linear time (in n) and requires constant amount of additional space.

Sum of 1 to n :

```
1 class Solution {  
2     public int missingNumber(int[] nums) {  
3         int n = nums.length, sum = n;  
4         for (int i = 0; i < n; ++i) {  
5             sum += i;  
6             sum -= nums[i];  
7         }  
8         return sum;  
9     }  
10 }
```

Success [Details >](#)

Runtime: 0 ms, faster than 100.00% of Java online submissions for Missing Number.

Memory Usage: 42.7 MB, less than 65.08% of Java online submissions for Missing Number.

Bonus Problem 1

(b) Improving previous part, now develop an algorithm which runs in linear time (in n) and requires constant amount of additional space.

XOR:

Input: `nums = [9,6,4,2,3,5,7,1]`

Output: 8

(268. Missing Number)

Time complexity: $O(n)$

Space complexity: $O(1)$

```
1 class Solution {  
2     public int missingNumber(int[] nums) {  
3         int n = nums.length, bitmask = n;  
4         for (int i = 0; i < n; ++i) {  
5             bitmask ^= nums[i];  
6             bitmask ^= i;  
7         }  
8         return bitmask;  
9     }  
10 }
```

Success Details >

Runtime: 0 ms, faster than 100.00% of Java online submissions for Missing Number.

Memory Usage: 43.3 MB, less than 61.96% of Java online submissions for Missing Number.

Bonus Problem 2

Consider the array B of size n which consists of n arbitrary positive integers (not necessarily the first n positive integers). Our goal is to find the least positive integer which is missing from B .

A very interview-type problem.

Same problem description with different time and space complexity requirement.

Bonus Problem 2

Bonus Problem 2

Consider the array B of size n which consists of n arbitrary positive integers (not necessarily the first n positive integers). Our goal is to find the least positive integer which is missing from B .

- (a) Develop an algorithm which runs in quadratic time (in n) and requires constant amount of additional space.

Hint: Sort the array!

- (b) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

- (c) Improving previous part, now develop an algorithm which runs in linear time (in n) and requires constant amount of additional space.

Hint: Use the array itself as the additional space!

- (d) Can you do part (c) if the array B may also have negative elements?

Bonus Problem 2

- (a) Develop an algorithm which runs in quadratic time (in n) and requires constant amount of additional space.

Hint: Sort the array!

Bonus Problem 2

- (a) Develop an algorithm which runs in quadratic time (in n) and requires constant amount of additional space.

Hint: Sort the array!

'Constant amount of additional space' means you can define any finite number of variables but cannot define another array of size n (or $2n$, $3n$.. etc) where n is the size of array B.

In other words, the number of additional variables (or the size of array) you define should be independent of n .

Bonus Problem 2

- (a) Develop an algorithm which runs in quadratic time (in n) and requires constant amount of additional space.

Hint: Sort the array!

Assume we have already sorted the array, how can we make use of the sorted array to get the final answer?

1 2 4

2 5 10

1 2 3

Bonus Problem 2

- (a) Develop an algorithm which runs in quadratic time (in n) and requires constant amount of additional space.

Hint: Sort the array!

Algorithm 1

```
function ALGORITHM 1(B)
   $n = \text{size of } B$ 
  Sort the array B

  for  $i$  in  $[1, n]$  do
    if  $B[i] \neq i$  then return  $i$ 
  end if

end for

return  $n + 1$ 
end function
```

Bonus Problem 2

- (a) Develop an algorithm which runs in quadratic time (in n) and requires constant amount of additional space.

Hint: Sort the array!

Time complexity: the sorting algorithm is $O(n^2)$ (e.g Insertion Sort) and the iteration is $O(n)$, so the total time complexity is $O(n^2)$

Space complexity: only define two variables n , i . So only use constant amount of additional space.

Bonus Problem 2

- (a) Develop an algorithm which runs in quadratic time (in n) and requires constant amount of additional space.

Hint: Sort the array!

Is there any other solution with $O(n^2)$ time complexity?

Bonus Problem 2

- (a) Develop an algorithm which runs in quadratic time (in n) and requires constant amount of additional space.

Hint: Sort the array!

Is there any other solution with $O(n^2)$ time complexity?

Algorithm 2

```
function ALGORITHM 2(B)
   $n = \text{size of } B$ 

  for  $i$  in  $[1, n]$  do
    if  $i \notin B$  then return  $i$ 
  end if

  end for

  return  $n + 1$ 
end function
```

Bonus Problem 2

- (b) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

Bonus Problem 2

(b) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

‘Linear amount of additional space (in n)’ means you can define another array of size proportional to n (n , $2n$, $3n$.. etc) where n is the size of array B, but not asymptotically larger than n (e.g n^2).

Bonus Problem 2

- (b) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

How should we make use of the additional space?

Bonus Problem 2

- (b) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

How should we make use of the additional space?

From part (a), we know that any number larger than n in array B will not affect the result. So we can ignore them.

Bonus Problem 2

- (b) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

How should we make use of the additional space?

Thus, we define array C of size n . Use it as buckets (or simpler hashtable) to record whether the corresponding number exists in array B .

$C[i] = 1$ means number i exists in array B

$C[i] = 0$ means number i doesn't exist in array B .

Bonus Problem 2

- (b) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

Algorithm 3

function ALGORITHM 3(B)

n = size of B

 Define an array C of size n with all elements being 0

for i in $[1, n]$ **do**

if $1 \leq B[i] \leq n$ **then** $C[B[i]] = 1$

end if

end for

for i in $[1, n]$ **do**

if $C[i] = 0$ **then return** i

end if

end for

return $n + 1$

end function

Bonus Problem 2

- (b) Develop an algorithm which runs in linear time (in n) and requires linear amount of additional space (in n).

Time complexity: Two loops, each is $O(n)$. So total running time is $O(n)$

Space complexity: Define another array C of size n and two variables n, i . In total, we use linear amount of additional space (in n).

Bonus Problem 2

- (c) Improving previous part, now develop an algorithm which runs in linear time (in n) and requires constant amount of additional space.

Hint: Use the array itself as the additional space!

Bonus Problem 2

- (c) Improving previous part, now develop an algorithm which runs in linear time (in n) and requires constant amount of additional space.

Hint: Use the array itself as the additional space!

Quite difficult.

Requires both time efficient and space saving.

Leetcode Problem 41

Bonus Problem 2

- (c) Improving previous part, now develop an algorithm which runs in linear time (in n) and requires constant amount of additional space.

Hint: Use the array itself as the additional space!

Combine the above two algorithms:

Each time we swap an element x to its place $B[x]$

Unless: $x > n$ or $B[x] = x$

Bonus Problem 2

- (c) Improving previous part, now develop an algorithm which runs in linear time (in n) and requires constant amount of additional space.

Hint: Use the array itself as the additional space!

Algorithm 4

function ALGORITHM 4(B)

$n = \text{size of } B$

for i in $[1, n]$ **do**

while $1 \leq B[i] \leq n$ and $B[B[i]] \neq B[i]$ **do**

$\text{swap}(B[B[i]], B[i])$

end while

end for

for i in $[1, n]$ **do**

if $B[i] \neq i$ **then return** i

end if

end for

return $n + 1$

end function

Bonus Problem 2

- (c) Improving previous part, now develop an algorithm which runs in linear time (in n) and requires constant amount of additional space.

Hint: Use the array itself as the additional space!

Time complexity: Although it seems we have a two-nested loops, since each time when the inner while loop runs, one element will be swapped to its place, and there are at most n elements that need to be swapped to its place.

Thus, the inner while loop only runs for n times in the whole program, the total time complexity is still $O(n)$

Space complexity: only finite number of variables, constant amount of space.

Bonus Problem 2

(d) Can you do part (c) if the array B may also have negative elements?

Bonus Problem 2

(d) Can you do part (c) if the array B may also have negative elements?

Yes. We could just ignore them. (Handle them the same way as integers larger than n .)

Q & A

Thank you