COMP2119 Introduction to Data Structures and Algorithms
Practice Problem Set 2 - Algorithm Analysis, Data Structures & hashing

Release Date: Sept 23, 2024

Please do NOT submit the answer of this part.

Chapter 2 - Algorithm Design & Analysis

1. Indicate, by writing "T" or "F", for each pair of expressions $(f(n), g(n))$ in the table below, whether $f(n)$ is O, Ω, or Θ of $g(n)$. (For example, if $f(n) = O(g(n))$, you should write "T" in the column of O, write "F" otherwise.)

$f(n)$	$g(n)$	O	Ω	Θ
n^2	n^3			
2^n	3^n			
$\log_2 n$	$\log_{10} n$			
$\log \sqrt{n}$	$\sqrt{\log n}$			
$n^{\cos n}$	$\sqrt[3]{n}$			
$\log_3(n!)$	$e^{\log_5 n}$			

Solution:

$f(n)$	$g(n)$	O	Ω	Θ
n^2	n^3	T	F	F
2^n	3^n	T	F	F
$\log_2 n$	$\log_{10} n$	T	T	T
$\log \sqrt{n}$	$\sqrt{\log n}$	F	T	F
$n^{\cos n}$	$\sqrt[3]{n}$	F	F	F
$\log_3(n!)$	$e^{\log_5 n}$	F	T	F

2. Prove or disprove the following statements by definition.

(a) $f(n) = 5n + 4$ is $O(n^2)$.

(b) $f(n) = (n + 2\sqrt{n} + \log_2 n)^5$ is $\Theta(n^5)$.

(c) $f(n) = 0.00001n$ is $O(1)$.

(d) $f(n) = 3n^5 - 2n^2 + 4n + 2$ is $\Theta(n^5)$.

(e) $f(n) = \log_3(n!)$ is $\Omega(e^{\log_5 n})$.

(f) If $f(n) = O(g(n))$, then $f(n) + g(n) = \Theta(g(n))$.

(g) If $f(n) = \Theta(1)$ and $g(n) = \Omega(1)$, then $f(n) + g(n) = \Omega(1)$.

(h) Given that $\sqrt{g(n)} \geq 1$ and $f(n) \geq 1$. If $f(n) = O(g(n))$, then $\sqrt{f(n)} = O(\sqrt{g(n)})$.

Solution:

(a) True

Proof:
When $n \geq 1$,

$$\begin{aligned} 5n + 4 &\leq 5n^2 + 4n^2 \\ &\leq 9n^2 \end{aligned}$$

We find $c = 9$, $n_0 = 1$, such that $\forall n \geq n_0$, $0 \leq f(n) \leq c \cdot n^5$.
Therefore, $5n + 4 = O(n^2)$.

(b) True

Proof:
When $n \geq 1$,

$$\begin{aligned} (n + 2\sqrt{n} + \log_2 n)^5 &\geq (n + 2(0) + 0)^5 \\ &\geq n^5 \end{aligned}$$

$$\begin{aligned} (n + 2\sqrt{n} + \log_2 n)^5 &\leq (n + 2n + n)^5 \\ &\leq 1024n^5 \end{aligned}$$

We find $c_1 = 1$, $c_2 = 1024$, $n_0 = 1$, such that $\forall n \geq n_0$, $0 \leq c_1 \cdot n^5 \leq f(n) \leq c_2 \cdot n^5$.
Therefore, $(n + 2\sqrt{n} + \log_2 n)^5 = \Theta(n^5)$.

(c) False

Proof:
Assume that $O(0.00001n) = O(1)$ is true. If so, it must be also true that $\exists c_1 > 0$, such that $\forall n \geq n_0$, $n_0 = 1$, $0 \leq 0.00001n \leq c_1$. It is not true when $n > \frac{c_1}{0.00001}$.
There is contradiction and $f(n)$ is not $O(1)$.

(d) True

Proof:
When $n \geq 1$,

$$\begin{aligned} 3n^5 - 2n^2 + 4n + 2 &\geq 3n^5 - 2n^2 \\ &\geq n^5 + 2(n^5 - n^2) \\ &\geq n^5 \end{aligned}$$

$$\begin{aligned} 3n^5 - 2n^2 + 4n + 2 &\leq 3n^5 + 4n + 2 \\ &\leq 4n^5 + (4n + 2 - n^5) \\ &\leq 4n^5 \end{aligned}$$

We find $c_1 = 1$, $c_2 = 4$, $n_0 = 1$, such that $\forall n \geq n_0$, $0 \leq c_1 \cdot n^5 \leq f(n) \leq c_2 \cdot n^5$.
Therefore, $3n^5 - 2n^2 + 4n + 2 = \Theta(n^5)$.

(e) True

Proof:
When $n \geq 10$,

$$\begin{aligned} \log_3(n!) &= \sum_{k=1}^{n} \log_3 k \\ &\geq \sum_{n=1}^{n} \log_3 3 \\ &\geq n \\ &\geq e^{\log_5 n} \end{aligned}$$

We find $c = 1$, $n_0 = 10$, such that $\forall n \geq n_0$, $0 \leq c \cdot e^{\log_5 n} \leq f(n)$.
Therefore, $\log_3(n!) = \Omega(e^{\log_5 n})$.

(f) True

Proof:
Since $f(n) = O(g(n))$, $\exists c > 0, n_0 > 0$ such that $\forall n \geq n_0$, $0 \leq f(n) \leq c \cdot g(n)$.

When $n \geq n_0$,

$$\begin{aligned} f(n) + g(n) &\geq 0 + g(n) \\ &\geq g(n) \end{aligned}$$

$$\begin{aligned} f(n) + g(n) &\leq c \cdot g(n) + g(n) \\ &\leq (c + 1) \cdot g(n) \end{aligned}$$

We find $c_1 = 1$, $c_2 = c + 1$, $n_0 > 0$, such that $\forall n \geq n_0$, $0 \leq c_1 \cdot g(n) \leq f(n) + g(n) \leq c_2 \cdot g(n)$. Therefore, if $f(n) = O(g(n))$, then $f(n) + g(n) = \Theta(g(n))$.

(g) True

Proof:
Since $f(n) = \Theta(1)$, $\exists c_1 > 0, \exists c_2 > 0, n_1 > 0$ such that $\forall n \geq n_1$, $0 \leq c_1 \leq f(n) \leq c_2$.
Since $g(n) = \Omega(1)$, $\exists c_3 > 0, n_2 > 0$ such that $\forall n \geq n_2$, $0 \leq c_3 \leq f(n)$.

When $n \geq \max(n_1, n_2)$,

$$\begin{aligned} f(n) + g(n) &\geq c_1 + c_3 \\ &\geq c \end{aligned}$$

We find $c = c_1 + c_3$, $n_0 = \max(n_1, n_2)$, such that $\forall n \geq n_0$, $0 \leq c \leq f(n)$.
Therefore, $f(n) + g(n) = \Omega(1)$.

(h) True

Proof:
Since $f(n) = O(g(n))$, $\exists c_1 > 0, n_0 > 0$ such that $\forall n \geq n_0$, $0 \leq f(n) \leq c_1 \cdot g(n)$.

When $n \geq n_0$,

$$\begin{aligned} \sqrt{f(n)} &\leq \sqrt{c_1 \cdot g(n)} \\ &\leq \sqrt{(c_1^2 + 2c_1 + 1) \cdot g(n)} \\ &\leq \sqrt{(c_1 + 1)^2 \cdot g(n)} \\ &\leq (c_1 + 1) \cdot \sqrt{g(n)} \end{aligned}$$

We find $c = c_1 + 1$, $n_0 > 0$, such that $\forall n \geq n_0$, $0 \leq \sqrt{f(n)} \leq c \cdot \sqrt{g(n)}$.
Therefore, $\sqrt{f(n)} = O(\sqrt{g(n)})$.

3. Consider the following algorithm:

Algorithm 3.1 A Random Function

```
1: function F(A, B)                                   ▷ A and B are positive integers
2:
3:     P = 0
4:     while (A > 0) do
5:         if (A  mod 2 == 1) then
6:             P = P + B
7:         end if
8:
9:         A = Floor(A/2)
10:        B = 2 * B
11:    end while
12:
13:    return P
14:
15: end function
```

(a) What will be the function output if A = 10, B = 6?

(b) What will be the function output if A = 13, B = 5?

(c) Guess what the algorithm does.

(d) Give an asymptotically (worst case) tight bound for the running time of the algorithm. Explain your answer.

(e) State the space complexity of the program in terms of asymptotic notations. Explain your answer.

Solution:

(a) 60

(b) 65

(c) Multiply A and B to get product P

(d) The tight bound for the running time of the algorithm is $O(\log A)$.

The while loop executes for $\log A$ times and the statements in each iteration finished in $O(1)$ time. The other steps complete in $O(1)$ time. Hence, the overall running times of the algorithm is $O(\log A)$.

(e) The space complexity of the program is $O(1)$.

The program uses only 3 integer variables to store the values. Each integer variable takes $O(1)$ space. Therefore, the overall spcae complexity is $O(1)$.

4. Consider the following recursive algorithm that draws some lines on a lattice of integer coordinates:

Algorithm 4.1 Draw Lines on Coordinate Plane

```
1: function DOODLE(n, m)                    ▷ n and m are non-negative integers
2:
3:     if (n > 0) then
4:         DrawLine((n,n),(m,m))            ▷ draw a line from (n,n) to (m,m)
5:         DrawLine((n,m),(m,n))
6:         DOODLE(n − 1, m)
7:     end if
8:
9: end function
```

Given that each "if" statement takes constant time c_1 to execute. The DrawLine function takes constant time c_2 to execute.

(a) Let $T(n)$ represent the running time of DOODLE(). Find a recurrence equation for $T(n)$ in terms of c_1 and c_2.

(b) Solve $T(n)$ and give an asymptotically (worst case) tight bound for the running time of DOODLE().

Solution:

(a) For $n \leq 0$, the function returns after executing line 3. i.e., $T(0) = c_1$.
For $n > 0$, the running time is the one required in lines 3 to 5,
i.e. $T(n) = c_1 + c_2 + c_2 + T(n–1)$.

The recurrence equation is: $T(n) = T(n-1) + c_1 + 2c_2$; $T(0) = c_1$.

(b) Solving the recurrence equation:

$$\begin{aligned} T(n) &= T(n-1) + c_1 + 2c_2 \\ &= T(n-2) + c_1 + 2c_2 + (c_1 + 2c_2) \\ &= T(n-2) + 2c_1 + 2(2c_2) \\ &= T(n-i) + ic_1 + i(2c_2) \\ &= T(0) + c_1 n + 2c_2 n \\ &= c_1 + c_1 n + 2c_2 n \end{aligned}$$

Therefore, a tight bound for the running time is $O(n)$.

5. Given an array $A[1, \ldots, n]$ that consists of all $n + 1$ integers from p to $p + n$ except one. We need an algorithm that find the missing integer.

   (a) Present a brute-force algorithm that tries all possible integers from p to $p + n$, and return the missing integer. Briefly explain how it works and analyze the time complexity of your algorithm.

   (b) Design an algorithm that consider the sum of $A[1]$ to $A[n]$ verses the sum of p to $p + n$ and deduce the missing integer. Briefly explain how it works and analyze the time complexity of your algorithm.

   (c) The algorithm below iterates through array A, and swap the elements (in $O(1)$ time) in the process, so that the array is split into two halves, with elements in the first half always smaller than that in the second half. The algorithm then determine which half is missing an element and re-apply the algorithm on that half.

   Analyze the time complexity of this algorithm. You may assume $n = 2^k - 1$ for some integer k.

Algorithm 5.1 FindMissingRecursive

```
1: function FindMissingRecursive(A, start, end, p)        ▷ Find missing element in A
2:
3:     if (start > end) then
4:         return p
5:     else if (start == end) then
6:         if (A[start] == p) then
7:             return p + 1
8:         else
9:             return p
10:        end if
11:    end if
12:
13:    bound = start
14:    mid_point = ⌊(start + end)/2⌋
15:    mid_value = p + ⌊(end − start)/2⌋
16:
17:    for (i = start to end) do
18:        if (A[i] ≤ mid_value) then
19:            swap(A[bound], A[i])
20:            bound = bound + 1
21:        end if
22:    end for
23:
24:    if (bound == mid_point) then
25:        return FindMissingRecursive(A, start, bound − 1, p)
26:    else
27:        return FindMissingRecursive(A, bound, end, mid_value + 1)
28:    end if
29:
30: end function
```

Solution:

(a) The function FindMissing(A, n, p) is written as following:

Algorithm 5.2 FindMissing

```
1: function FindMissing(A, n, p)                    ▷ Find missing element in A[1...n]
2:
3:     for (j = p to p + n) do
4:         found = False
5:
6:         for (i = 1 to n) do
7:             if (A[i] == j) then
8:                 found = True
9:             end if
10:        end for
11:
12:        if (found == False) then
13:            return j
14:        end if
15:    end for
16:
17: end function
```

Explanation:
The algorithm uses a two-level nested loop to compare each of the elements in A with all possible integers from p to $p + n$ until the target missing element is found.

Time complexity:
The algorithm uses a two-level nested loop to compare each of the elements in A with all possible integers from p to $p + n$. The statements inside the for loops use $O(1)$ time to run. Therefore, the time complexity is $O((n + 1)(n)) = O(n^2)$.

(b) The function FindMissingBySum(A, n, p) is written as following:

Algorithm 5.3 FindMissingBySum

```
1: function FindMissingBySum(A, n, p)               ▷ Find missing element in A[1...n]
2:     sum = 0
3:     for (i = 1 to n) do
4:         sum = sum + A[i]
5:     end for
6:     full_sum = (n + 1) * (p + p + n)/2
7:
8:     return full_sum − sum
9: end function
```

Explanation:
The algorithm subtract the sum of integers p to $p + n$ by the sum of values values in array A, which equals the missing integer.

Time complexity:
The for loop iterates the array once in $O(n)$ time. The other statements run in $O(1)$ time. Therefore, the overall time complexity is $O(n)$.

(c) The algorithm scan the array once and repeat the process on half of the array. The time complexity is $O(n + n/2 + n/4 + ... + 1) = O(2n - 1) = O(n)$.

Alternatively, let $T(n)$ be the number of elements scanned, then:

$$T(n) = \begin{cases} 1 & \text{for } n = 1 \\ n + T(\lfloor \frac{n}{2} \rfloor) & \text{for } n \geq 2 \end{cases}$$

With $n = 2^k - 1$, $T(2^k - 1) = 2^k - 1 + T(2^{k-1} - 1)$ for $k \geq 2$.

Solving $T(2^k - 1)$:

$$\begin{aligned} T(2^k - 1) &= 2^k - 1 + T(2^{k-1} - 1) \\ &= 2^k - 1 + 2^{k-1} - 1 + T(2^{k-2} - 1) \\ &= 2^k - 1 + 2^{k-1} - 1 + \ldots + 2^2 - 1 + T(1) \\ &= 2^2(2^{k-1} - 1) - (k - 1) + 1 \\ &= 2(2^k - 1) - k \\ &= 2n - \log(n + 1) \\ &= O(n) \end{aligned}$$

6. Jeff visited a casino in Macau and played n rounds of a gambling game. He is unhappy with the total amount of money won. Suppose now a time machine appears, which allows him to go back to the past and play the game again starting at any of the n rounds. He can choose to leave the game at any round afterwards or play until round n. However, after leaving the game, he cannot join the game anymore.

   Assume that the game data is saved in a list S, which consists of n integers each representing the amount you win/lose in a round. The numbers are stored chronologically, starting with the result of the first round and ending with the result of the last round. You may assume that there is at least one positive entry in the list S.

   Now, Jeff wants to find out the maximum amount of money he can win under the above-mentioned constraints.

   Example:
   Suppose Jeff's game result is S_1 = [-2, 4, -3, 2, 3, -1, 2, -4, -3, 5]. He can win the maximum amount of money by joining the game at round 2 and leaving the game after round 7. His net gain would then be $+4 - 3 + 2 + 3 - 1 + 2 = 7$.

   Your Tasks:

   (a) Design a divide-and-conquer recursive algorithm to solve the problem faster than $O(n^2)$ time. Briefly explain how your algorithm works and analyze the time complexity of your algorithm.

   (b) Design another algorithm to solve the problem in $O(n)$ time and $O(1)$ extra space. Briefly explain how your algorithm works and analyze the time complexity of your algorithm.

Solution:

(a) The function MaxGain1(S, $left$, $right$) is written as following:

Algorithm 6.1 MaxGain1

```
1:  function MaxGain1(S, left, right)          ▷ Find max gain in subarray S[left : right]
2:
3:      if (left == right) then
4:          return Max(S[left], 0)
5:      end if
6:
7:      middle = Floor((left + right)/2)
8:      maxGain_L = MaxGain1(S, left, middle)              ▷ Line 8: Compute the 1st case
9:      maxGain_R = MaxGain1(S, middle + 1, right)         ▷ Line 9: Compute the 2nd case
10:
11:     maxBorderGain_L = 0                                ▷ Line 11-24: Compute the 3rd case
12:     borderGain_L = 0
13:     maxBorderGain_R = 0
14:     borderGain_R = 0
15:
16:     for (i = middle to left) do
17:         borderGain_L = borderGain_L + S[i]
18:         maxBorderGain_L = Max(maxBorderGain_L, borderGain_L)
19:     end for
20:
21:     for (i = middle + 1 to right) do
22:         borderGain_R = borderGain_R + S[i]
23:         maxBorderGain_R = Max(maxBorderGain_R, borderGain_R)
24:     end for
25:
26:     return Max(maxGain_L, maxGain_R, maxBorderGain_L + maxBorderGain_R)
27:
28: end function
```

Explanation:
Consider the middle element of S. To achieve maximum gain, there are three possible cases:

(1) The set of integers to sum up locates on the left side of the middle element.
(2) The set of integers to sum up locates on the right side of the middle element.
(3) The set of integers to sum up includes the middle element.

Algorithm MaxGain1 recursively finds the maximum gain only considering the 1^{st} half and 2^{nd} half of S respectively. Afterwards, the algorithm iterates the 1^{st} half and 2^{nd} half of S once to sum up the elements and find out the maximum gain for the 3^{rd} case. Lastly, the algorithm compares the maximum gain generated from the 3 possible cases and return the largest one as the result. For the base case, if there is one element in S only, return that number as the maximum gain.

Time Complexity:
Let $T(n)$ be the running time of the algorithm. A recurrence relation can be set up and solved as following:

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + O(n) \ ; \ T(1) = 1 \\
&= 2T(\frac{n}{2}) + cn \\
&= 2[2T(\frac{n}{4}) + c(\frac{n}{2})] + cn \\
&= 2^{\log n}[T(1)] + (\log n)cn \\
&= n(1) + cn \log n \\
&\rightarrow O(n \log n)
\end{aligned}$$

(b) The function MaxGain2(S, n) is written as following:

Algorithm 6.2 MaxGain2

```
1:  function MaxGain2(S, n)          ▷ Find the maximum gain in S with list length n
2:
3:      tempGain = 0
4:      maxGain = 0
5:
6:      for (i = 0 to n − 1) do
7:          tempGain = tempGain + S[i]
8:          tempGain = Max(0, tempGain)
9:          maxGain = Max(maxGain, tempGain)
10:     end for
11:
12:     return maxGain
13:
14: end function
```

Explanation:
Observe that if the sum of some consecutive integers in S becomes less than 0, it is meaningless to add the list elements afterwards for finding the maximum gain.

Algorithm MaxGain2 loops through S and sum up the digits in the process. In one iteration, update the maximum possible gain if the cumulative sum exceeds it. If the cumulative sum becomes less than 0, reset it as 0. Return the maximum possible gain after reaching the end of S.

Time Complexity:
The algorithm consists of a single for loop which iterates for n times. The statements inside the for loop runs in constant time. Therefore, the overall running time of the algorithm is $O(n)$.

Chapter 3 - Basic Data Structures

1. Answer the following problems related to data structure operations.

   (a) Given an empty linked list L, write down the elements inside the linked list (starting from the first element) after the following operations:

   insert(12), insert(3), insert(5), delete(5), search(3), insert(2), search(12), delete(2), delete(12), insert(85), insert(62)

   (b) Given an empty stack S, write down the popped elements in order after the following operations:

   push(12), push(3), push(5), pop(), push(26), push(2), pop(), pop(), push(85), pop(), pop()

   (c) Given an empty queue Q, write down the popped elements in order after the following operations:

   enqueue(12), enqueue(3), enqueue(5), dequeue(), enqueue(26), enqueue(2), dequeue(), dequeue(), enqueue(85), dequeue(), dequeue()

   (d) Given a List L and an element x. Assume there are n elements in L and we only know the head pointer. What are the time complexities of Search(L, x), Insert(L, k, x), Delete(L, x) operations for the linked list and doubly linked list implementation respectively?

   (e) Given a Stack S and an element x. Assume there are n elements in S now. What are the time complexities of Push(S, x), Pop(S) operations?

Solution:

(a) 3, 85, 62

(b) 5, 2, 26, 85, 3

(c) 12, 3, 5, 26, 2

(d) The time complexities are listed below:

	Search(L, x)	Insert(L, k, x)	Delete(L, x)
Linked List	$O(n)$	$O(n)$	$O(n)$
Doubly Linked List	$O(n)$	$O(n)$	$O(n)$

(e) Both operations costs $O(1)$ time.

2. The *Node* structure below represents a Linked List node:

```
// Data structure to store a linked list node
struct Node {
    int data;
    Node *next;
};
```

(a) Given the head pointer of a singly linked list H, write the pseudocode of an algorithm which reverses the second half of the nodes in the linked list in $O(1)$ space.

Example:
Given the linked list

$$1 \to 2 \to 3 \to 4 \to 5$$

the linked list after reversal should become

$$1 \to 2 \to 3 \leftarrow 4 \leftarrow 5.$$

(b) Briefly explain how the algorithm works. Analyze it's time and space complexity.

Solution:

(a) Pseudocode:

Algorithm 2.1 Reverse the second half of a linked list

```
1:  function reverseList(H)                    ▷ H: Pointer to the head of a linked list
2:
3:      fast = H
4:      slow = H
5:
6:      while ((fast.next != NIL) and (fast.next.next != NIL)) do
7:          fast = fast.next.next
8:          slow = slow.next
9:      end while
10:
11:     pre = NIL                              ▷ Reverse the second half of the nodes
12:     cur = slow
13:     cnext = slow
14:
15:     while (cur != NIL) do
16:         cnext = cur.next
17:         cur.next = pre
18:         pre = cur
19:         cur = cnext
20:     end while
21:
22:     return head
23:
24: end function
```

(b) Explanation:
Two pointers (slow and fast) are used: The slow one moves one step forward while fast moves two steps forward. Then the slow pointer will point to the middle node of the list when (fast.next == null) or (fast.next.next == null). After the middle node has been found, three pointers are used to reverse the second half of the list.

Time Complexity:
The time complexity of the algorithm is $O(n)$. The program traverses the elements in the linked list once to find the middle node in $O(n)$ time. Next, the program iterates the later half of the linked list once to reverse the nodes in $O(n)$ time. The other operations outside the while loop costs $O(1)$ time. Therefore, the overall time complexity is $O(n)$.

Space Complexity:
The space complexity of the algorithm is $O(1)$. Only 5 pointers with $O(1)$ space each are declared in the program. Therefore, the overall space complexity is $O(1)$.

3. Jeff is a factory worker. He is responsible for controlling a machine by issuing control sequences involving two buttons A and B. A *valid control sequence* is some consecutive steps of button pressed that obeys the following two rules:

   (R1) At any particular step of the valid control sequence, Jeff should not have pressed button B more times than button A.

   (R2) Upon the end of a valid control sequence, Jeff should have pressed button A and button B the same number of times.

   One day, Jeff is naughty and presses the buttons randomly. Suppose Jeff's action sequence is recorded in a list S, which consists of n characters of "A" and "B". He wishes to find the length of the longest valid control sequence in his recorded action.

   Example:
   Suppose S = [B, B, A, B, A, A, B, A, B, B, A]. The longest valid control sequence is [A, B, A, A, B, A, B, B], which starts from step 3 and stops at step 10. The length of the longest valid control sequence is therefore 8.

   Explanation:

   - The valid control sequence cannot start from step 1 or step 2, or otherwise the number of times button B is pressed would be more than that for button A after move 1 or move 2 (thus violating rule R1).
   - The valid sequence then starts from step 3 with button A pressed, and remains valid till step 10. If step 11 ("A") is included, Jeff would have pressed button A for 5 times and button B for 4 times only, which violates rule R2.

   Your Tasks:

   (a) Present the pseudo-code of a brute-force algorithm that iterates through all possible consecutive moves in S. Analyze the time complexity of your algorithm.

   (b) Present the pseudo-code of another algorithm to solve the problem in $O(n)$ time by using a stack. Note that you can only use the following stack operations: *isEmpty*, *push*, *pop* and *top*.

   Explain how your algorithm works and analyze the time complexity of your algorithm.

Solution:

(a) Algorithm MaxMove1 iterates through all possible combinations of consecutive moves in S to find the maximum number of valid moves.

Algorithm 3.1 MaxMove1

```
1:  function MaxMove1(S)        ▷ Find the maximum valid moves in list S with length n
2:
3:      maxLength = 0
4:
5:      for (i = 0 to S.length − 1) do
6:          for (j = i to S.length − 1) do
7:
8:              SS = S[i : j]
9:              counter = 0
10:
11:             for k in SS do
12:                 if (k == "A") then
13:                     counter = counter + 1
14:                 else
15:                     counter = counter - 1
16:                 end if
17:
18:                 if (counter == 0) then
19:                     maxLength = Max(maxLength, SS.length)
20:                 else if (counter <= 0) then
21:                     break
22:                 end if
23:             end for
24:
25:         end for
26:     end for
27:
28:     return maxLength
29:
30: end function
```

Time Complexity:
The algorithm runs in $O(n^3)$ time. It consists of a triple for loop which loops $O(n)$ times each. The statements between the middle and the innermost for loop can run within constant time. Inside the innermost for loop, the statements can run within constant time. All in all, the algorithm runs in $O(n^3)$ time.

(b) Algorithm MaxMove2 is written as following:

Algorithm 3.2 MaxMove2

```
1: function MaxMove2(S)        ▷ Find the maximum valid moves in list S with length n
2:
3:     if (S.length == 0) then
4:         return 0
5:     end if
6:
7:     maxLength = 0
8:     tempStack = Stack()
9:     tempStack.push(−1)
10:
11:    for (i = 0 to S.length − 1) do
12:        if (S[i] == "A") then
13:            tempStack.push(i)
14:        else
15:            if (tempStack.top() != None) then
16:                tempStack.pop()
17:                maxLength = Max(maxLength, i - tempStack.top())
18:            else
19:                tempStack.push(i)
20:            end if
21:        end if
22:    end for
23:
24:    return maxLength
25:
26: end function
```

Explanation:
Observe that for a valid action sequence, it should start with A and end with B.

Algorithm MaxMove2 first loops through S. If the current list element is A, the program pushes it's index into the stack. If the current list element is B and the stack is not empty, it pops out the first element inside. It is the index of the farest A character which can keep the action sequence between and including this A and B pair valid. Then, it calculates the length of this action sequence by minus the current index and index of the first element inside the stack, and updates the max length if necessary.

If the current list element is B and the stack is empty, push the index of B into the stack. This means it is impossible to meet a valid sequence before it. After the iterations, return the max length found.

Time Complexity:
The algorithm consists of a single for loop which loops for n times, with the statements inside the for loop finishes in constant time. The statements outside the for loop also finishes in constant time. Therefore, the algorithm can finish running in $O(n)$ time.

4. Suppose you are given two stacks $S1$ and $S2$, we aim to implement a queue using only $S1$ and $S2$.

   (a) Write algorithms for implementing *enqueue* and *dequeue* operations using only the stack operations (*isEmpty*, *push*, *pop* and *top*).

   (b) Starting from two empty stacks, describe the content in them after executing these operations:

   (i) enqueue(1); enqueue(2); enqueue(3);
   (ii) dequeue();
   (iii) enqueue(4); enqueue(5);

   (c) Explain your code step by step in the last operation of (b)(iii), i.e., the enqueue(5) operation.

Solution:

(a) Function Enqueue($S1$, $S2$, x) first use a temp stack $S2$ to store all existing elements popped from $S1$. Then, x is pushed into $S1$. Afterwards, all the elements in $S2$ are popped out and pushed into $S1$. This ensures that element x is being inserted in the last position of the stack.

Function Dequeue($S1$) pops out the first element being pushed into the stack $S1$. It is possible since we implement enqueue operation in the way described above.

Algorithm 4.1 Implement Queue By Stack

```
1:  function ENQUEUE(S1, S2, x)                    ▷ Conduct enqueue operations by stack
2:
3:      if (S1.isEmpty()) then
4:          S1.push(x)
5:      else
6:          while (!S1.isEmpty()) do
7:              S2.push(S1.pop())
8:          end while
9:
10:         S1.push(x)
11:
12:         while (!S2.isEmpty()) do
13:             S1.push(S2.pop())
14:         end while
15:     end if
16:
17: end function
18:
19: function DEQUEUE(S1)                           ▷ Conduct dequeue operations by stack
20:
21:     if (S1.isEmpty()) then
22:         return NULL
23:     else
24:         return S1.pop()
25:     end if
26:
27: end function
```

(b) (i) $S1$: [3, 2, 1], $S2$: [ ]
(ii) $S1$: [3, 2], $S2$: [ ]
(iii) $S1$: [5, 4, 3, 2], $S2$: [ ]

(c) At the start of operation enqueue(5), we have $S1$: [4, 3, 2], $S2$: [ ]. Then we pop all elements from $S1$, and push them into $S2$. Then we push 5 into $S1$, and pop all elements from $S2$, and push them into $S1$.

(Answers may vary depending on implementation.)

5. Define an item to be a pair of (k, t), where k is a key, and t is a tag. Let S be a set of items, all of which have different tags but may have identical keys. Given an interval $[x, y]$, a range query returns the tags of all the items in S whose keys fall in $[x, y]$. For example, assume that $S = (1, f), (5, d), (10, z), (10, w); (20, g)$. Then, a range query with [5, 15] returns d, z, w.

Describe a data structure that consumes $O(n)$ space, and answers any range query in $O(\log m + k)$ time, where n is the number of items in S, m is the number of distinct keys in S, and k is the number of tags reported (e.g. In our earlier example, $n = 5$, $m = 4$, $k = 3$). You can assume that no item insertion and deletion happen during range queries.

Solution:

Build an array containing all the distinct keys in S, sorted in ascending order. Each key is associated with a linked list, containing all the tags associated with this key.

Given a query, first find in $O(\log m + k')$ all the distinct keys that fall into the query range, where k' is the number of retrieved (distinct) keys. For each such key, traverse its linked list to report all of its associated tags. This takes $O(k' + k) = O(k)$ time. The overall time cost is therefore $O(\log m + k)$.

6. Four people (A to D) come to a river at night. There is a narrow bridge that can only hold two people at the same time. They have only one torch. They need to use the torch when crossing the bridge as it is completely dark. Person A can cross the bridge in w minutes, B in x minutes, C in y minutes, and D in z minutes. When two people cross the bridge together, they must move at the slower person's pace. The problem is to figure out a way to let them cross the river using the shortest amount of time.

   Model this problem as a graph problem. State clearly on

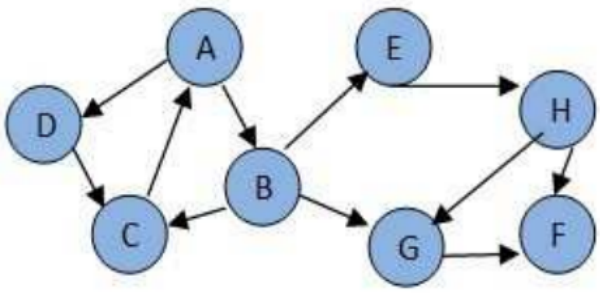
   - What type of graph you will use
   - What a node/vertex represents
   - What an edge represents
   - The corresponding graph problem to be solved.

   Solution:

   We will use a simple weighted graph to model the problem. Each node represents a state of the situation (i.e. where the four people are), with R denotes the river. For example, state "ABCRD" means person A, B, C are on the left side of the river, while D is on the right side of the river. Edges represents the time required to move from one node to the next node.

   The graph problem we need to solve is the minimal weighted path in the weighted directed graph from the node representing the original state (i.e. ABCDR) to the node representing the final state (i.e. RABCD).

7. A directed graph is drawn below. List out the visiting order of vertices by using BFS and DFS. The algorithms should visit the vertex with the smallest alphabet first.



   Solution:

   BFS: A, B, D, C, E, G, H, F
   DFS: A, B, C, E, H, F, G, D

Chapter 4 - Hashing

1. Insert the keys {53, 62, 75, 18, 27, 33, 40, 98} into the following hash tables. You only need to draw the final hash table.

   (a) A hash table of size 13 with hash function $h(k) = k \bmod 13$. Collisions are handled by open addressing with linear probing, using the function $f(i) = 4i$.

   (b) A hash table of size 5 with hash function $h(k) = k \bmod 5$. Collisions are handled by chaining.

Solution:

(a) Calculating hash values:

key	53	62	75	18	27	33	40	98
$h(k)$	1	10	10	5	1	7	1	7

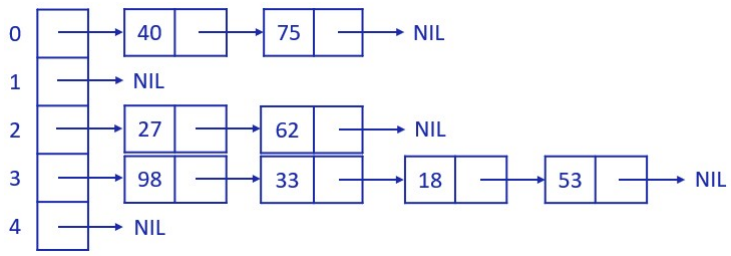
Final hash table:

0	27
1	53
2	
3	
4	40
5	75
6	
7	33
8	
9	18
10	62
11	98
12	

(b) Calculating hash values:

key	53	62	75	18	27	33	40	98
$h(k)$	3	2	0	3	2	3	0	3

Final hash table:



2. Consider the following hash function: $h(k,i) = (h'(k) + \frac{1}{2}(i+i^2)) \bmod m$, where $m = 2^p$ and $h'(k) = k \bmod m$ for some positive integer p. Prove or disprove that for any k, the probe sequence $< h(k,0), h(k,1), ..., h(k,m–1) >$ is a permutation of $< 0,1,2,...,m–1 >$.

Solution:

Assume that $h(k,i) = h(k,j)$, which i and j are integers such that $0 \le i < j < m$. Then,

$$\begin{aligned} h(k,i) - h(k,j) &= \frac{1}{2}(j+j^2) - \frac{1}{2}(i+i^2) \\ &= \frac{(j-i)(j+i+1)}{2} \\ &= ma, \text{ where a is an integer} \end{aligned}$$

Since $(j-i)$ and $(j+i+1)$ have opposite parity, consider two stituations:

(1) When $(j-i)$ is even and $(j+i+1)$ is odd, then $(j-i)$ must be divisible by $2m$, while $(j-i)$ must be a positive integer smaller than or equal to m, which is not divisible by $2m$. This results in contradiction.

(2) When $(j-i)$ is odd and $(j+i+1)$ is even, then $(j+i+1)$ must be divisible by $2m$, since $1+i+j \le 1+m-1+m-2 = 2(m-1) < 2m$. This results in contradiction.

Therefore, if $h(k,i) = h(k,j)$, then $i = j$, probe sequence $< h(k,0), h(k,1),...,h(k,m-1) >$ is a permutation of $< 0,1,2,...,m–1 >$.

3. Let A and B be two sets, in which each set contains n integers. Verbally describe an algorithm to find the distinct integers in the set $A \cup B$ with $O(n)$ average runtime. Analyze the time complexity of your algorithm.

Example:
For input $A = \{2,6,7,10,11\}$ and $B = \{6,8,11,14,16\}$, the output is $\{2,6,7,8,10,11,14,16\}$.

Solution:

Algorithm:
Firstly, output all the elements in A. Next, we insert all the elements of A in a hash table. Next, for each element in B, search the hash table to see if the element exists in A. If not, output the element.

Time Complexity:
The time complexity of the algorithm is $O(n)$.

Output all the elements in S_1 uses $O(n)$ time. Hash table creation uses $O(n)$ time. Searching in the hash table uses $O(1)$ time on average. There are n searches and it costs $O(n)$ time on average. Therefore, the overall average time complexity is $O(n)$.

Further Practice from the Textbook:
To gain more practice, here are a list of supplementary problems you may attempt from the course textbook **Introduction to Algorithms, 3rd edition**:

Chapter 2 - Algorithm Design & Analysis

1. Asymptotic Bounds:
   Exercise 3.1-1, 3.1-4, 3.2-5, 3.2-8, 3-2, 3-3, 3-4
2. Recursive Algorithm Design:
   Exercise 2.3-7, 9.3-8

Chapter 3 - Basic Data Structures

1. Linked Lists:
   Exercise 10.2-2, 10.2-3, 10.2-7, 10-1
2. Stack:
   Exercise 10.1-1, 10.1-2, 10.1-6
3. Queue:
   Exercise 10.1-3, 10.1-7
4. Graph Algorithms:
   Exercise 22.2-9, 22.3-7, 22.3-10

Chapter 4 - Hashing

1. Hash Table Operations:
   Exercise 11.2-2, 11.4-1
2. Hash Table Performance:
   Exercise 11.2-5, 11.4-3