Department of Computer Science
Ashoka University

Discrete Mathematics: CS-1104-1 & CS-1104-2

Assignment 3 Part 2

Collaborators: None **Name: Rushil Gupta**

---

1 Straightforward

1. (a) $T(n) = 4T(n/2) + c$

   Using Master Theorem, $a = 4,\ b = 2,\ d = 0$ (Since $f \in O(1)$)
   So, $a > b^d \to$ Case 1 $\to T(n) \in \Theta(n^{log_b a}) \to T(n) \in \Theta(n^2)$

   (b) $T(n) = T(n/4) + T(n/2) + n \cdot c$
   Note that we cannot directly resolve this recurrence using the Master theroem. So, we will need to simplify it.

   Observe that $T(n) \leq 2T(n/2) + n \cdot c$
   Using Master Theorem, $a = 2,\ b = 2,\ d = 1$ (Since $f = n \cdot c \in O(n)$)
   So, $a = b^d \to$ Case 2 $\to T(n) \in \Theta(n \cdot log_a n) \to T(n) \in \Theta(n \cdot log_2 n)$

   (c) $T(n) = T(n-2) + T(n-4)$, where $T(0) = T(1) = T(2) = T(3) = c_0$
   Note that we cannot directly resolve this recurrence using the Master theroem. So, we will need to simplify it.

   Observe that:
   $T(n) = T(n-2) + T(n-4) \leq 2T(n-2)$
   $\leq 2(2T(n-4)) = 4T(n-4)$
   $\ldots$
   $= 2^k T(n-2k)$
   So, $n - 2k = 0 \to k = n/2$
   So, $T(n) \leq 2^{n/2} T(0) = 2^{n/2} c_0$
   So, $T(n) \in O(2^{n/2})$

2. (a) (i) **Definition:**
   - Input: Integers $m,\ n \geq 0$
   - Output: Integer The product of m and n

   (ii) **Decomposition:**
   $m \cdot n = m + m \cdot (n-1)$
   So, in Python:

```python
def product(m, n):
    if n == 0:
        return 0
    return m + product(m, n-1)
```

   (iii) **Deconstruction:**
   - Cost of binary addition: $O(p)$
   - Cost of binary comparison: $O(1)$
   - Cost of function call: $O(1)$
   - Cost of return: $O(1)$

   So, time compexity is given by the recurrence relation: $T(n) = T(n-1) + O(p) + O(1)$
   $= T(n-2) + 2O(p) + 2O(1) = \cdots = T(0) + nO(p) + nO(1)$
   Since $p = log_2 n$, $T(n) = O(nlogn)$ (Using Master Theorem)

(iv) **Design:**
No, since each value of $m \cdot n$ is computed exactly once, we don't need memoization.

(v) **Iteration:**
Yes, we can convert the recursive solution to an iterative solution.

```
def product(m, n):
    for i in range(n):
        m += m
    return m
```

(vi) **Correctness:**
- Base Case: $m \cdot 0 = 0$
- Inductive Hypothesis: Assume multiply$(m, n) = m \cdot n$ holds true for n
- Inductive Step: multiply$(m, n+1) = m +$ multiply(m, n)

$$= m + m \cdot n \text{ (I.H.)}$$

$$= m \cdot (n+1)$$

So, the recursive solution is correct.

(b) i. **Definition:**
- Input: Integers $m > 0,\ n \geq 0$
- Output: Integer The n^{th} power of m

ii. **Decomposition:**
$m^n = m \cdot m^{n-1}$
So, in Python:

```
def power(m, n):
    if n == 0:
        return 1
    return m * power(m, n-1)
```

iii. **Deconstruction:**
- Cost of binary multiplication: $O(p^2)$
- Cost of binary comparison: $O(1)$
- Cost of function call: $O(1)$
- Cost of return: $O(1)$
- Cost of unary decrement: $O(1)$

So, time compexity is given by the recurrence relation: $T(n) = T(n-1) + O(p^2) + O(1)$
$= T(n-2) + 2O(p^2) + 2O(1) = \cdots = T(0) + nO(p^2) + nO(1)$
Since $p = log_2 n$, $T(n) = O(n(logn)^2)$

iv. **Design:**
No, since each value of m^n is computed exactly once, we don't need memoization.

v. **Iteration:**
Yes, we can convert the recursive solution to an iterative solution.

```
def power(m, n):
    result = 1
    for i in range(n):
        result *= m
    return result
```

vi. **Correctness:**
- Base Case: $m^0 = 1$
- Inductive Hypothesis: Assume power$(m, n) = m^n$ holds true for n
- Inductive Step: $m^{n+1} = m \cdot power(m, n)$

$$= m \cdot m^n \text{ (I.H.)}$$

$$= m^{n+1}$$

So, the recursive solution is correct.

3. (a) **Definition:**
   - Input: String s
   - Output: Integer The length of the string s

   (b) **Decomposition:**
   $|s| = 1 \text{if} s[0] = \backslash 0 \text{ else } 1 + |s[1:]|$

   So, in Python:

```python
def length(s):
    if s[0] == '\0':
        return 1
    return 1 + length(s[1:])
```

   (c) **Deconstruction:**
   - Cost of binary comparison: $O(1)$
   - Cost of function call: $O(1)$
   - Cost of return: $O(1)$
   - Cost of assignment: $O(1)$

   So, time compexity is given by the recurrence relation: $T(n) = T(n-1) + O(1)$
   $= T(n-2) + 2O(1) = \cdots = T(0) + nO(1)$
   So, $T(n) = O(n)$

   (d) **Design:**
   No, since each value of $|s|$ is computed exactly once, we don't need memoization.

   (e) **Iteration:**
   Yes, we can convert the recursive solution to an iterative solution.

```python
def length(s):
    count = 0
    while s[count] != '\0':
        count += 1
    return count
```

   (f) **Correctness:**
   - Base Case: $|s| = 1$ if $s[0] = \backslash 0$
   - Inductive Hypothesis: Assume $\text{length}(s) = |s|$ holds true for s for some $P(k)$
   - Inductive Step: $P(k+1) \equiv \text{length}(s) = |s|$ holds true for s
   $\text{lenght}(s) = 1 + \text{length}(s[1:])$
   $= 1 + |s[1:]| = 1 + k$ (I.H.)
   $= k + 1$
   So, the recursive solution is correct.

4. (a) **Definition:**
   - Input: Integers $a,\ b \geq 0$
   - Output: Integer The value of $a \mod b$

   (b) **Decomposition:**
   $a \mod b = a - b$ if $a < b$ else $(a - b) \mod b$
   So, in Python:

```python
def mod(a, b):
    if a < b:
        return a
    return mod(a - b, b)
```

   (c) **Deconstruction:**
   - Cost of binary subtraction: $O(p)$
   - Cost of binary comparison: $O(1)$
   - Cost of function call: $O(1)$
   - Cost of return: $O(1)$

So, time compexity is given by the recurrence relation:
$T(a) = T(a - b) + O(p) + O(1)$
$= T(a - 2b) + 2O(p) + 2O(1) = \cdots = T(a - kb) + kO(p) + kO(1)$
Since $a - kb < b$, $T(a) = O(p)$
But, since $p = log_2 n$, $T(a) = O(loga)$

(d) **Design:**
No, since each value of $a \mod b$ is computed exactly once, we don't need memoization.

(e) **Iteration:**
Yes, we can convert the recursive solution to an iterative solution.

```
def mod(a, b):
    while a >= b:
        a -= b
    return a
```

(f) **Correctness:**
- Base Case: $a \mod b = a$ if $a < b$
- Inductive Hypothesis: Assume $\text{mod}(a, b) = a \mod b$ holds true for a
- Inductive Step: $\text{mod}(a, b) = (a - b) \mod b$

5. (a) **Definition:**
- Input: Array A of size n
- Output: Array The sorted array A

(b) **Decomposition:**
- Base Case: If $n = 0$, return A
- Inductive Step: Insert $A[n-1]$ into the sorted array $A[0 : n-2]$ so that it is in the correct position

So, in Python:

```
def sort(A):
    if len(A) <= 1:
        return A
    A[0:len(A)-1] = sort(A[0:len(A)-1])
    key = A[len(A)-1]
    i = len(A) - 2
    while i >= 0 and A[i] > key:
        A[i+1] = A[i]
        i -= 1
    A[i+1] = key
    return A
```

(c) **Deconstruction:**
The time compexity is given by the recurrence relation: $T(n) = T(n - 1) + O(n) + O(1)$
$= T(n - 2) + 2O(n) + 2O(1) = \cdots = T(0) + nO(n) + nO(1)$
So, $T(n) = O(n^2)$

(d) **Design:**
No, since each value of A is computed exactly once, we don't need memoization.

(e) **Iteration:**
Yes, we can convert the recursive solution to an iterative solution.

```
def sort(A):
    for i in range(1, len(A)):
        key = A[i]
        j = i - 1
        while j >= 0 and A[j] > key:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = key
    return A
```

(f) **Correctness:**

- Base Case: If $n = 0$, return A
- Inductive Hypothesis: Assume insertion_sort($A[0 : n-1]$) sorts $A[0 : n-1]$ - Inductive Step: Need to show insertion_sort($A[0 : n]$) sorts $A[0 : n]$

Inner Loop:

$P(q) \equiv$ The first q elements of A are sorted
- Base Case: $q = 0$. The function returns A, which is trivially true
- Inductive Hypothesis: Assume the function works for some k, which also means $P(k)$ holds
- Inductive Step: Need to show that $P(k+1)$ holds

- If $A[q+1] \geq A[q]$, then it means $P(q+1)$ is true.
- Otherwise, $A[q+1] < A[q]$. Then, we will need to show that $A[q+1]$ is moved to the correct position. Note that the larger element is moved to the right, and then the next element in checked, and so on.

Therefore, $P(q+1)$ holds.

Outer Loop:

$P(n) \equiv \text{sort}(A, n) = A_{\text{sorted}}$

- Base Case: $n = 0$. The function returns A, which is trivially true
- Inductive Hypothesis: Assume the function works for some k, which also means $P(k)$ holds
- Inductive Step: Need to show that $P(k+1)$ holds

By definition, we have: $\text{sort}(A, k+1) = \text{sort}(A_1, k+1)$, where A_1 is the array after the first iteration of the inner loop.
We also know that that A_i is sorted up until the i'th element.
So, the problem becomes $\text{sort}(A_1, k)$. By the inductive hypothesis, we know that $\text{sort}(A_1, k) = A_{\text{sorted}}$.
So, $\text{sort}(A, k+1) = A_{\text{sorted}}$.

2 ¬Straightforward

1. (a) **Definition:**
   - Input: Integers $n \geq 0$
   - Output: The steps to move n disks from tower A to tower C using tower B as an auxiliary tower

   (b) **Decomposition:**

   i. Base Case: If $n = 0$, return

   ii. Inductive Step: Move $n - 1$ disks from tower A to tower C using tower B as an auxiliary tower. Move disk n from tower A to tower C. Move $n - 1$ disks from tower B to tower C using tower A as an auxiliary tower.

   So, in Python:

```
def hanoi(n, A, B, C):
    if n == 0:
        return
    hanoi(n-1, A, C, B)
    print(f"Move disk {n} from {A} to {C}")
    hanoi(n-1, B, A, C)
```

   (c) **Deconstruction:**
   The time compexity is given by the recurrence relation: $T(n) = 2T(n-1) + O(1)$
   $= 2(2T(n-2) + 2O(1)) + 2O(1) = \cdots = 2^n T(0) + 2^n O(1)$
   So, $T(n) = O(2^n)$

   (d) **Design:**
   No, since each value of n is computed exactly once, we don't need memoization.

   (e) **Iteration:**
   No, since the recursive solution is the most optimal solution. This is because the problem is inherently recursive and the iterative solution would be much more complex.

   (f) **Correctness:**

   i. Base Case: If $n = 0$, return

   ii. Inductive Hypothesis: Assume hanoi(k, A, C, B) moves k disks from tower A to tower C using tower B as an auxiliary tower

   iii. Inductive Step: Need to show hanoi$(k+1, A, C, B)$ moves $k+1$ disks from tower A to tower C using tower B as an auxiliary tower

   By the inductive hypothesis, we know that hanoi(k, A, C, B) moves k disks from tower A to tower C using tower B as an auxiliary tower.
   So, the problem becomes moving the $k + 1^{th}$ disk from tower A to tower C using tower B as an auxiliary tower.
   This is done by moving the top k disks from tower A to tower B, moving the $k + 1^{th}$ disk from tower A to tower C, and then moving the k disks from tower B to tower C using tower A as an auxiliary tower.
   Therefore, hanoi$(k+1, A, C, B)$ moves $k+1$ disks from tower A to tower C using tower B as an auxiliary tower.

2. (a) **Definition:**
   - Input: Array A of size n
   - Output: The sorted array A

   (b) **Decomposition:**

i. Base Case: If $n = 0$, return A

ii. Inductive Step: Split the array A into two halves. Sort the first half and the second half. Merge the two sorted halves.

So, in Python:

```
def merge(A, B):
    result = []
    i = j = 0
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            result.append(A[i])
            i += 1
        else:
            result.append(B[j])
            j += 1
    result += A[i:]
    result += B[j:]
    return result

def sort(A):
    if len(A) <= 1:
        return A
    mid = len(A) // 2
    left = sort(A[:mid])
    right = sort(A[mid:])
    return merge(left, right)
```

(c) **Deconstruction:**
The time compexity is given by the recurrence relation: $T(n) = 2T(n/2) + O(n)$
Using Master Theorem, $a = 2,\ b = 2,\ d = 1$ (Since $f = n \in O(n)$)
So, $a = b^d \to$ Case 2 $\to T(n) \in \Theta(n \cdot log_a n) \to T(n) \in \Theta(n \cdot log_2 n)$

(d) **Design:**
No, since each value of A is computed exactly once, we don't need memoization.

(e) **Iteration:**
Yes, we can convert the recursive solution to an iterative solution.

```
def merge_sort(A):
    stack = []

    # Split the array into individual elements and append them to
        the stack
    for i in range(len(A)):
        stack.append([A[i]])


    # Merge the elements in pairs, then merge the pairs in pairs,
        and so on until only one element is left which is the
        sorted array
    while len(stack) > 1:
        A = stack.pop()
        B = stack.pop()
        stack.append(merge(A, B))

    return stack[0]

def merge(A, B):
    result = []
    i = j = 0
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            result.append(A[i])
            i += 1
        else:
            result.append(B[j])
            j += 1
    result += A[i:]
    result += B[j:]
    return result
```

(f) **Correctness:**

i. Base Case: If $n = 0$, return A

ii. Inductive Hypothesis: Assume sort($A[0 : n/2]$) and sort($A[n/2 : n]$) sorts $A[0 : n/2]$ and $A[n/2 : n]$

iii. Inductive Step: Need to show sort(A) sorts A

By the inductive hypothesis, we know that sort($A[0 : n/2]$) and sort($A[n/2 : n]$) sorts $A[0 : n/2]$ and $A[n/2 : n]$.
So, the problem becomes merging the two sorted halves.
This is done by comparing the first element of each half and appending the smaller element to the result.
Therefore, sort(A) sorts A.

3 Bonus

1. $T(n) = T(\lfloor log_2 n \rfloor) + c$

   Note that: $T(n) \approx T(log_2 n) + O(1)$

   So, $T(n) = T(log_2(log_2 n) + O(1)) + O(1) = \cdots \approx T(1) + O(log_2 n)$

   So, $T(n) \in \Theta(log_2 n)$

2. (a) **Definition:**
   - Input: Array A of size n
   - Output: The pairs of numbers in the array A that add up to 0

   (b) **Decomposition:**

   i. Base Case: If $n = 0$, return

   ii. Inductive Step: Check if $-A[i]$ is in the array. If it is, then add $(A[i], -A[i])$ to the result.

   So, in Python:

```python
def two_sum(A):
    result = []
    for i in range(len(A) // 2):
        if -A[i] in A:
            result.append((A[i], -A[i]))
    return result
```

   (c) **Deconstruction:**
   The time compexity is given by the recurrence relation: $T(n) = nO(n) + O(1)$
   So, $T(n) = O(n^2)$

   (d) **Design:**
   Yes, we can use memoization to store the values of $-A[i]$ in a set. This would reduce the time complexity to $O(n)$.

   (e) **Iteration:**
   Yes, we can convert the recursive solution to an iterative solution.

```python
def two_sum(A):
    result = []
    seen = set()
    for i in range(len(A)):
        if -A[i] in seen:
            result.append((A[i], -A[i]))
        seen.add(A[i])
    return result
```

   Time Complexity:
   We have the recurrence relation: $T(n) = nO(1) + O(1)$
   So, $T(n) = O(n)$

   (f) **Correctness:**

   i. Base Case: If $n = 0$, return

   ii. Inductive Hypothesis: Assume two_sum($A[0 : n-1]$) returns the pairs of numbers in the array $A[0 : n-1]$ that add up to 0

   iii. Inductive Step: Need to show two_sum(A) returns the pairs of numbers in the array A that add up to 0

By the inductive hypothesis, we know that two_sum$(A[0:n-1])$ returns the pairs of numbers in the array $A[0:n-1]$ that add up to 0.
So, the problem becomes checking if $-A[i]$ is in the array.
If it is, then add $(A[i], -A[i])$ to the result.
Therefore, two_sum(A) returns the pairs of numbers in the array A that add up to 0.