BHU/20/04/05/0067

ATILE IORFA RAPHAEL

CMP 418

COMPUTER SCIENCE

**Part A: Solving Recurrence**

**Problem 1-1: Order of Growth**

**Group 0**

Arrange the functions in increasing order of growth:

1. f5=log⁡(n2)f\_5 = \log(n^2)f5​=log(n2)
2. f3=nlog⁡nf\_3 = n \log nf3​=nlogn
3. f2=n2f\_2 = n^2f2​=n2
4. f4=2nf\_4 = 2^nf4​=2n
5. f1=2345f\_1 = 2^{345}f1​=2345 (This is a constant, not depending on nnn, and should be the smallest in growth since it's fixed regardless of nnn)

**Explanation:**

* log⁡(n2)\log(n^2)log(n2) grows slower than nlog⁡nn\log nnlogn because logarithmic growth is slower than polynomial growth.
* nlog⁡nn\log nnlogn grows slower than n2n^2n2 due to the polynomial order.
* n2n^2n2 grows slower than 2n2^n2n because polynomial growth is slower than exponential growth.
* 23452^{345}2345 is a constant value, so it does not grow with nnn. However, if we interpret the number 23452^{345}2345 as a fixed constant function (which is not dependent on nnn), then this should actually be the smallest as it's a constant independent of nnn.

**Group 1**

Arrange the functions in increasing order of growth:

1. f5=(log⁡log⁡n)3f\_5 = (\log \log n)^3f5​=(loglogn)3
2. f1=log⁡((log⁡n)3)f\_1 = \log ((\log n)^3)f1​=log((logn)3)
3. f3=3log⁡nf\_3 = 3\log nf3​=3logn
4. f2=(log⁡n)3(log⁡(3n))f\_2 = (\log n)^3 (\log (3n))f2​=(logn)3(log(3n))
5. f4=n3log⁡nf\_4 = n^3 \log nf4​=n3logn
6. f6=log⁡3nn3f\_6 = \log^3 n n^3f6​=log3nn3

**Explanation:**

* (log⁡log⁡n)3(\log \log n)^3(loglogn)3 is the slowest since double logarithm grows very slowly.
* log⁡((log⁡n)3)\log((\log n)^3)log((logn)3) is slightly faster but still involves a slow-growing logarithmic function.
* 3log⁡n3\log n3logn grows faster than any combination involving only logs and not nnn.
* (log⁡n)3(log⁡(3n))(\log n)^3 (\log (3n))(logn)3(log(3n)) grows faster than 3log⁡n3\log n3logn since it's a product of cubic and single log terms.
* n3log⁡nn^3 \log nn3logn involves polynomial growth which is faster than the logarithmic terms combined.
* log⁡3nn3\log^3 n n^3log3nn3 is similar in growth rate to n3log⁡nn^3 \log nn3logn.

**Group 3**

Arrange the functions in increasing order of growth:

1. f1=43nf\_1 = 4^{3n}f1​=43n
2. f4=23nf\_4 = 2^{3n}f4​=23n
3. f3=23n+1f\_3 = 2^{3n+1}f3​=23n+1
4. f5=25nf\_5 = 2^{5n}f5​=25n
5. f2=2n4f\_2 = 2^{n^4}f2​=2n4

**Explanation:**

* 43n4^{3n}43n can be rewritten as (22)3n=26n(2^2)^{3n} = 2^{6n}(22)3n=26n which actually grows faster than the other powers of two shown here (except for 25n2^{5n}25n), so let's adjust this.
* 23n2^{3n}23n is smaller than 23n+12^{3n+1}23n+1 because adding a constant exponent shifts it slightly higher.
* 23n+12^{3n+1}23n+1 grows faster than 23n2^{3n}23n.
* 43n4^{3n}43n as 26n2^{6n}26n grows faster than all previous since its effective base is larger.
* 25n2^{5n}25n is larger than any constant factor of exponent like 333 here.
* 2n42^{n^4}2n4 is exponential in a polynomial, which vastly outgrows all previous terms.

**Corrected Order:**

1. f4=23nf\_4 = 2^{3n}f4​=23n
2. f3=23n+1f\_3 = 2^{3n+1}f3​=23n+1
3. f1=43nf\_1 = 4^{3n}f1​=43n
4. f5=25nf\_5 = 2^{5n}f5​=25n
5. f2=2n4f\_2 = 2^{n^4}f2​=2n4

**Problem 1-2: Prove or Disprove Assertions**

**(a) If t(n)∈O(g(n))t(n) \in O(g(n))t(n)∈O(g(n)) then g(n)∈Ω(t(n))g(n) \in \Omega(t(n))g(n)∈Ω(t(n)).**

**Proof:**

* The definition of t(n)∈O(g(n))t(n) \in O(g(n))t(n)∈O(g(n)) means that there exist constants c>0c > 0c>0 and n0≥0n\_0 \geq 0n0​≥0 such that for all n≥n0n \geq n\_0n≥n0​, t(n)≤c⋅g(n)t(n) \leq c \cdot g(n)t(n)≤c⋅g(n).
* Similarly, g(n)∈Ω(t(n))g(n) \in \Omega(t(n))g(n)∈Ω(t(n)) means that there exist constants c′>0c' > 0c′>0 and n1≥0n\_1 \geq 0n1​≥0 such that for all n≥n1n \geq n\_1n≥n1​, g(n)≥c′⋅t(n)g(n) \geq c' \cdot t(n)g(n)≥c′⋅t(n).
* This implies that both are true if and only if t(n)t(n)t(n) and g(n)g(n)g(n) are asymptotically equivalent in terms of their growth rates, which is true by definition.

Thus, the statement is **true**.

**(b) Θ(αg(n))=Θ(g(n))\Theta(\alpha g(n)) = \Theta(g(n))Θ(αg(n))=Θ(g(n)) where α>0\alpha > 0α>0.**

**Proof:**

* The definition of Θ(g(n))\Theta(g(n))Θ(g(n)) involves both O(g(n))O(g(n))O(g(n)) and Ω(g(n))\Omega(g(n))Ω(g(n)), indicating that there are constant bounds on either side.
* Θ(αg(n))\Theta(\alpha g(n))Θ(αg(n)) will just scale both bounds by the same constant α\alphaα, maintaining the same growth rate without changing the asymptotic classification.

Thus, the statement is **true**.

**(c) Θ(g(n))=O(g(n))∩Ω(g(n))\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))Θ(g(n))=O(g(n))∩Ω(g(n)).**

**Proof:**

* Θ(g(n))\Theta(g(n))Θ(g(n)) is precisely the intersection of functions that are simultaneously bounded above by some multiple of g(n)g(n)g(n) and bounded below by some multiple of g(n)g(n)g(n), by definition.

Thus, the statement is **true**.

**(d) For any two nonnegative functions t(n)t(n)t(n) and g(n)g(n)g(n) defined on the set of nonnegative integers either t(n)∈O(g(n))t(n) \in O(g(n))t(n)∈O(g(n)) or t(n)∈Ω(g(n))t(n) \in \Omega(g(n))t(n)∈Ω(g(n)) or both.**

**Counterexample:**

* Consider t(n)=n2sin⁡2(n)t(n) = n^2 \sin^2(n)t(n)=n2sin2(n) and g(n)=n2g(n) = n^2g(n)=n2.
* Neither t(n)∈O(g(n))t(n) \in O(g(n))t(n)∈O(g(n)) nor t(n)∈Ω(g(n))t(n) \in \Omega(g(n))t(n)∈Ω(g(n)) since sin⁡2(n)\sin^2(n)sin2(n) oscillates between 0 and 1.
* There exists no single ccc such that t(n)≤c⋅g(n)t(n) \leq c \cdot g(n)t(n)≤c⋅g(n) or g(n)≤c⋅t(n)g(n) \leq c \cdot t(n)g(n)≤c⋅t(n) for all nnn.

Thus, the statement is **false**.

**Part B: Sorting Problems**

**Problem 2-1: ComparisonCountingSort Algorithm**

Given array A=[50,30,80,10,40,60,20]A = [50, 30, 80, 10, 40, 60, 20]A=[50,30,80,10,40,60,20].

1. **Basic Operation:**
   * Comparison between elements of the array.
   * Specifically, line 5 is the basic operation where comparisons occur.
2. **What is it computing?**
   * It computes the sorted order of the array by determining how many elements each element is greater than (frequency counting).
3. **Is the algorithm stable?**
   * **No**, this algorithm is not stable. It does not maintain the relative order of equal elements. Stability can be achieved by modifying the placement step to keep the order of duplicates.
4. **Is it in place?**
   * **No**, the algorithm is not in place since it uses an auxiliary array SSS.
5. **Derive the complexity class of the algorithm:**
   * The time complexity of the algorithm is O(n2)O(n^2)O(n2), where nnn is the number of elements in the array. This is due to the nested loops used for counting comparisons.

**Steps for Sorting A=[50,30,80,10,40,60,20]A = [50, 30, 80, 10, 40, 60, 20]A=[50,30,80,10,40,60,20]:**

1. **Initialize Count array:**

Count=[0,0,0,0,0,0,0]\text{Count} = [0, 0, 0, 0, 0, 0, 0]Count=[0,0,0,0,0,0,0]

1. **Count comparisons:**
   * For each pair (i,j)(i, j)(i,j) where i<ji < ji<j, increase Count based on comparisons:

(50,30)→Count=[0,1,0,0,0,0,0](50,80)→Count=[1,1,0,0,0,0,0](50,10)→Count=[1,1,0,1,0,0,0](50,40)→Count=[1,1,0,1,1,0,0](50,60)→Count=[2,1,0,1,1,0,0](50,20)→Count=[2,1,0,1,1,0,1]…\begin{align\*} (50, 30) & \rightarrow \text{Count} = [0, 1, 0, 0, 0, 0, 0] \\ (50, 80) & \rightarrow \text{Count} = [1, 1, 0, 0, 0, 0, 0] \\ (50, 10) & \rightarrow \text{Count} = [1, 1, 0, 1, 0, 0, 0] \\ (50, 40) & \rightarrow \text{Count} = [1, 1, 0, 1, 1, 0, 0] \\ (50, 60) & \rightarrow \text{Count} = [2, 1, 0, 1, 1, 0, 0] \\ (50, 20) & \rightarrow \text{Count} = [2, 1, 0, 1, 1, 0, 1] \\ \ldots \end{align\*}(50,30)(50,80)(50,10)(50,40)(50,60)(50,20)…​→Count=[0,1,0,0,0,0,0]→Count=[1,1,0,0,0,0,0]→Count=[1,1,0,1,0,0,0]→Count=[1,1,0,1,1,0,0]→Count=[2,1,0,1,1,0,0]→Count=[2,1,0,1,1,0,1]​

Continuing this process will fill the count array with counts of elements each index should be positioned before.

1. **Place elements in SSS based on Count:**

S[Count[i]]=A[i]S[\text{Count}[i]] = A[i]S[Count[i]]=A[i]

Resulting SSS array after placements will be sorted.

**Problem 2-2: Quicksort Example**

Given the array [E,X,A,M,P,L,E,S][E, X, A, M, P, L, E, S][E,X,A,M,P,L,E,S].

1. **Choose Pivot:**
   * Choose the first element, EEE, as the pivot.
2. **Partition Array:**
   * Rearrange elements such that those less than pivot come before, and those greater come after.
3. **Recursive Sort:**
   * Recursively apply quicksort to partitions.

**Quicksort Steps:**

1. **Initial Array:** [E,X,A,M,P,L,E,S][E, X, A, M, P, L, E, S][E,X,A,M,P,L,E,S]
   * Pivot = E
2. **Partitioning:**
   * Left: [A,E,E][A, E, E][A,E,E]
   * Right: [X,M,P,L,S][X, M, P, L, S][X,M,P,L,S]
3. **Recursive Sort:**
   * Apply quicksort to each partition recursively.

Final sorted array: [A,E,E,L,M,P,S,X][A, E, E, L, M, P, S, X][A,E,E,L,M,P,S,X].

**Part C: Searching Problems**

**Problem 3-1: Brute Force Comparisons**

**(a) Pattern 000010 0 0 0 100001 in 1000 zeros**

* **Comparisons:** Every position will be a failed match, resulting in 555 comparisons per position, yielding 5×1000=50005 \times 1000 = 50005×1000=5000 comparisons.

**(b) Pattern 010100 1 0 1 001010 in 1000 zeros**

* **Comparisons:** Similar analysis to above, but mismatch occurs after first 111, so there will be 222 comparisons at each position, yielding 2×1000=20002 \times 1000 = 20002×1000=2000 comparisons.

**(c) Pattern BRANDING**

Text: THERE\_IS\_MORE\_TO\_LIFE\_THAN\_INCREASING\_ITS\_SPEED

* **Brute Force Algorithm:** Align pattern with each text position, and compare character-by-character.
* Minimum comparisons occur when pattern is at the end: 47−8+1=4047 - 8 + 1 = 4047−8+1=40 positions.

**Problem 3-2: Constructing and Sorting a Heap**

Given list: 1,8,6,5,3,7,4,21, 8, 6, 5, 3, 7, 4, 21,8,6,5,3,7,4,2.

1. **Construct Heap:**
   * Start with empty heap and insert keys one by one, maintaining heap property.
2. **Heapsort:**
   * Remove maximum element and rebuild heap until empty.

**Steps:**

1. Insert 111: [1][1][1]
2. Insert 888: [8,1][8, 1][8,1]
3. Insert 666: [8,1,6][8, 1, 6][8,1,6]
4. Insert 555: [8,5,6,1][8, 5, 6, 1][8,5,6,1]
5. Insert 333: [8,5,6,1,3][8, 5, 6, 1, 3][8,5,6,1,3]
6. Insert 777: [8,7,6,1,3,5][8, 7, 6, 1, 3, 5][8,7,6,1,3,5]
7. Insert 444: [8,7,6,1,3,5,4][8, 7, 6, 1, 3, 5, 4][8,7,6,1,3,5,4]
8. Insert 222: [8,7,6,2,3,5,4,1][8, 7, 6, 2, 3, 5, 4, 1][8,7,6,2,3,5,4,1]

**Heapsort:**

1. Remove 8, heap: [7,5,6,2,3,1,4][7, 5, 6, 2, 3, 1, 4][7,5,6,2,3,1,4]
2. Remove 7, heap: [6,5,4,2,3,1][6, 5, 4, 2, 3, 1][6,5,4,2,3,1]
3. Remove 6, heap: [5,3,4,2,1][5, 3, 4, 2, 1][5,3,4,2,1]
4. Remove 5, heap: [4,3,1,2][4, 3, 1, 2][4,3,1,2]
5. Remove 4, heap: [3,2,1][3, 2, 1][3,2,1]
6. Remove 3, heap: [2,1][2, 1][2,1]
7. Remove 2, heap: [1][1][1]

Sorted list: [1,2,3,4,5,6,7,8][1, 2, 3, 4, 5, 6, 7, 8][1,2,3,4,5,6,7,8].

**Problem 3-3: Algorithm Trace**

**Algorithm:** Given the array [68,55,44,79,19,9][68, 55, 44, 79, 19, 9][68,55,44,79,19,9].

**Trace the algorithm:**

Initial Array: [68,55,44,79,19,9][68, 55, 44, 79, 19, 9][68,55,44,79,19,9]

* **Pass 1:** 797979 swapped with 444444, 686868, and 555555.
* **Pass 2:** [68,55,44,79,19,9][68, 55, 44, 79, 19, 9][68,55,44,79,19,9] to [55,68,44,79,19,9][55, 68, 44, 79, 19, 9][55,68,44,79,19,9].
* **Pass 3:** Continue until sorted.

Final sorted array: [9,19,44,55,68,79][9, 19, 44, 55, 68, 79][9,19,44,55,68,79].

1. **Basic Operations:** Comparisons and swaps.
2. **Complexity Class:** The worst-case time complexity is O(n2)O(n^2)O(n2) due to nested iterations over elements.
3. **In Place:** Yes, since no auxiliary storage is used.
4. **Stable:** No, because equal elements can be reordered by swaps. Stability can be achieved by modifying the swap to check and preserve order.

**Part D: General Problems**

**Problem 4-1: Knapsack Problem**

**Items:**

| **Item No.** | **Weight** | **Value** |
| --- | --- | --- |
| 1 | 2 | $15 |
| 2 | 3 | $20 |
| 3 | 1 | $10 |
| 4 | 2 | $12 |

Knapsack Capacity = 5.

1. **Exhaustive Search:**
   * Check all subsets and calculate value for those within weight limit.
   * Possible solutions: {1,2},{3,2},{3,1,4}\{1, 2\}, \{3, 2\}, \{3, 1, 4\}{1,2},{3,2},{3,1,4} (for maximizing value).
2. **Dynamic Programming:**
   * Use a DP table to store maximum value at each weight capacity from 000 to WWW.
   * Fill DP table using decision whether to include each item or not based on maximizing value.

**Dynamic Programming Solution:**

1. Define dp[i][w]dp[i][w]dp[i][w] as maximum value for first iii items with weight limit www.
2. Recurrence relation:

dp[i][w]=max⁡(dp[i−1][w],dp[i−1][w−wi]+vi)dp[i][w] = \max(dp[i-1][w], dp[i-1][w-w\_i] + v\_i)dp[i][w]=max(dp[i−1][w],dp[i−1][w−wi​]+vi​)

1. Solution: dp[n][W]dp[n][W]dp[n][W].

**Optimal Solution:**

* Items {2,3}\{2, 3\}{2,3} with weight = 4 and value = $30.

**Problem 4-2: Longest Common Subsequence (LCS)**

Given strings "KADUNA" and "KANO".

1. **Recurrence Relation:**
   * Let XXX and YYY be two sequences of lengths mmm and nnn, respectively.
   * Define L[i][j]L[i][j]L[i][j] as the length of the LCS of X[1..i]X[1..i]X[1..i] and Y[1..j]Y[1..j]Y[1..j].

L[i][j]={0if i=0 or j=0L[i−1][j−1]+1if X[i]=Y[j]max⁡(L[i−1][j],L[i][j−1])if X[i]≠Y[j]L[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1][j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(L[i-1][j], L[i][j-1]) & \text{if } X[i] \neq Y[j] \end{cases}L[i][j]=⎩⎨⎧​0L[i−1][j−1]+1max(L[i−1][j],L[i][j−1])​if i=0 or j=0if X[i]=Y[j]if X[i]=Y[j]​

1. **Algorithm for LCS:**
   * Construct LLL table using above recurrence.
   * Backtrack from L[m][n]L[m][n]L[m][n] to extract LCS.
2. **Example:**
   * Strings: "KADUNA" and "KAKNO".
   * LCS=KANOLCS = KANOLCS=KANO with length 333.

**Problem 4-3: Sorting and Removing Duplicates**

1. **Proposed Algorithm:**
   * Sort books by catalogue number using merge sort or quicksort.
   * Traverse sorted list and remove duplicates.
2. **In Place:**
   * Sorting can be in-place using quicksort; duplicate removal involves additional space for tracking.
3. **Performance Evaluation:**
   * Sorting: O(nlog⁡n)O(n \log n)O(nlogn).
   * Duplicate Removal: O(n)O(n)O(n) post-sorting.
4. **Additional Information:**
   * Catalog numbers are unique identifiers; duplicates can be verified by sorted traversal.

**Problem 4-4: Minimizing Maximum Completion Time**

Given tasks with durations [3,4,5,2,4][3, 4, 5, 2, 4][3,4,5,2,4] and two processors.

1. **Minimizing Maximum Completion Time:**
   * Greedily assign tasks to processors based on current load.
   * Optimal schedule involves balancing load between processors.
2. **Optimal Task Assignment:**
   * Processor 1: [5,2][5, 2][5,2] Total time = 777.
   * Processor 2: [4,4,3][4, 4, 3][4,4,3] Total time = 111111.

Thus, the maximum completion time is minimized by scheduling tasks optimally.