4520/6520 Design and Analysis of Algorithms

Midterm

Instructor: Alina Nemira

10/15/2025

Name: ______________________________

Honor Code Statement: By starting work on this assignment, I agree to abide by the principles of academic integrity. I will not commit any act of academic dishonesty while completing this assignment. I am fully aware that any of my own personal actions during the attempt of this assignment that are interpreted as academic dishonesty will be treated as such. I understand that if I am held responsible for an act of academic dishonesty, I will receive an F grade for the course and the incident will be reported to the Dean of Students Office.

The exam contains 13 pages (including this cover page) and 10 problems.
Total of points is 125. You will receive $\min\{\text{your score}, 100\}$ points.

Good luck and productive work!

Distribution of grades

The following table:

Question	**Points**	**Score**
1	10	
2	5	
3	12	
4	20	
5	20	
6	10	
7	16	
8	10	
9	12	
10	10	
Total:	**125**	

Master theorem for recurrence relations. Assume $T(n)$ is the total run-time of a recursive algorithm satisfying the recurrence:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n).$$

where $f(n) \in \Theta(n^c)$ for some constant $c \geq 0$. Then:

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & if\ \ \log_b a > c, \\ \Theta(n^c \log n) & if\ \ \log_b a = c, \\ \Theta(n^c) & if\ \ \log_b a < c. \end{cases}$$

This theorem is provided for your reference. You may use it to solve relevant problems on this exam.

1. Fill in each gap with a suitable word from the list below.

   (a) (4 points) Greedy algorithm makes decisions based only on the current state without considering future consequences, while Dynamic Programming algorithm solves the problem by building on previously computed results.

   List of words:

   √ Greedy

   √ Dynamic Programming

   (b) (6 points) Dynamic Programming algorithm breaks a problem into subproblems that overlap, ensuring that each subproblem is solved only once.
   Greedy algorithm focuses on local optimality to attempt to achieve a global solution.
   Divide and Conquer algorithm divides the problem into independent subproblems that can be solved separately.

   List of words:

   √ Greedy

   √ Divide and Conquer

   √ Dynamic Programming

2. (5 points) The *asymptotic* runtime of a dynamic programming algorithm is most directly influenced by: (*select all that apply, you do not need to justify your choice here*)

   A. Whether the algorithm uses a top-down (recursive) or bottom-up (iterative) approach.

   B. The total number of unique subproblems that need to be solved.

   C. The amount of work required to solve each subproblem.

   D. Whether the subproblems are sorted before solving them.

   E. The order in which subproblems are stored in memory.

3. (12 points) Suppose we are given an array $arr[0...n-1]$ with the *boundary conditions* that $arr[0] \geq arr[1]$ and $arr[n-2] \leq arr[n-1]$.

   We say that an element $arr[i]$ is a *local minimum* if it is less than or equal to both of its neighbors. More formally, $arr[i]$ is a local minimum if $arr[i-1] \geq arr[i]$ and $arr[i] \leq arr[i+1]$. For example, there are six local minima in the following array:

9	7	7	2	1	3	7	5	4	7	3	3	4	8	6	9

   Clearly, we can find a local minimum in $O(n)$ time by scanning through the array. This problem asks you to find a faster algorithm.

   Describe a divide-and-conquer algorithm that, given an array $arr[0...n-1]$ with the boundary conditions mentioned above, finds and returns one local minimum in $O(\log n)$ time. You can assume that $n \geq 3$.

   Hint: With the given boundary conditions, the array must have at least one local minimum. Why?

   Solution: The problem requires a divide-and-conquer algorithm to find a local minimum arr$[i]$ such that arr$[i] \leq$ arr$[i-1]$ and arr$[i] \leq$ arr$[i+1]$, given boundary conditions arr$[0] \geq$ arr$[1]$ and arr$[n-1] \leq$ arr$[n-2]$.The algorithm is a modified binary search. We compare the middle element with its neighbors. If the middle element is less than or equal to both neighbors, it is a local minimum, so we return it. If the middle element is greater than its left neighbor, a local minimum must exist in the left half of the array. If the middle element is greater than its right neighbor, a local minimum must exist in the right half of the array.

Time Complexity Analysis

The algorithm is a divide-and-conquer approach that reduces the problem size by approximately half in each recursive step.

(a) **Work per step:** In each recursive call, the algorithm performs a constant number of comparisons ($O(1)$) to check the middle element against its neighbors.

(b) **Problem reduction:** The problem is reduced to a subproblem of size at most $\lceil n/2 \rceil$.

The recurrence relation for the time complexity $T(n)$ is:

$$T(n) = T(n/2) + O(1)$$

By the **Master Theorem** (Case 2, where $a = 1, b = 2, f(n) = 1$), or by direct unfolding, this recurrence solves to:

$$T(n) = O(\log n)$$

This achieves the required logarithmic time complexity.

Answer to the question:

The boundary conditions arr$[0] \geq$ arr$[1]$ and arr$[n-1] \leq$ arr$[n-2]$ **guarantee** the existence of at least one local minimum.

Why the boundary conditions guarantee a local minimum:

The boundary conditions guarantee that the array "starts high and ends low":

- At index 0, $arr[0] \geq arr[1]$ means the array begins by descending or flat.
- At index $n-1$, $arr[n-2] \leq arr[n-1]$ means it ends by ascending or flat.

As we move from index 0 to $n-1$, every time the array changes direction from decreasing to increasing, a local minimum must occur.

By the discrete version of the Intermediate Value Theorem (or a simple slope argument), at least one local minimum must exist in any array satisfying these boundary conditions.

4. Decide if the statements below are TRUE or FALSE. Explain your answers. **No credit here without a proper explanation!**

   (a) (4 points) If the running time of an algorithm satisfies the recurrence $T(n) = 4T(n/4) + \Theta(4n)$, and $T(1) = 1$, then $T(n) \in O(n \log n)$.

   True. Applying the *Master Theorem* to the recurrence, we have:
   - $a = 4$ (the number of subproblems),
   - $b = 4$ (the factor by which the problem size is divided),
   - $c = 1$ (the exponent of the non-recursive term, $4n$).

   Since $c = \log_b a$, the time complexity is is $T(n) = O(n \log n)$ according to the Master Theorem.

   (b) (4 points) The asymptotic run-time of a recursive dynamic programming solution is always larger than that of the corresponding iterative solution.

   False. Both recursive DP with memoization and iterative DP typically achieve the same asymptotic run-time; differences may appear in constant factors, but not in Big-O terms.

   (c) (4 points) $n \in O(n^3)$. **True.** Since n grows asymptotically slower than n^3$, we have$n$\in O(n^3)$. I
   notation, n^3 serves as a valid upper bound for n.

   (d) (4 points) Radix sort works correctly as long as a *stable* sorting algorithm is used to sort each digit, starting with the ***most*** *significant* digit.

   False. Radix sort processes digits starting from the least significant digit (rightmost digit) to the most significant digit.

   (e) (4 points) In the merge-sort tree of computation, roughly the same amount of work is done at each level of the tree.

   True. At the top level, $\Theta(n)$ work is done to merge all n elements. At the next level, there are two branches, each doing roughly $n/2$ work to merge $n/2$ elements. In total, $\Theta(n)$ work is done on that level. This pattern continues on through to the leaves, where a constant amount of work is done on n leaves, resulting in $\Theta(n)$ work being done on the leaf level, as well.

5. Imagine that you need to build a wireless network by placing signal towers along a straight highway of length L miles. Each tower provides coverage of d miles in both directions (i.e., each tower covers an an interval of length $2d$ centered at its position).

   You are given the positions of n candidate towers along the highway, sorted in increasing order. Your task is to select a subset of these towers to fully cover the highway segment $[0, L]$ using the fewest number of towers possible.

   Assume that the provided tower positions allow for a complete coverage of the highway.

   (a) (10 points) Describe a greedy algorithm to select tower locations so that the entire highway is covered using the fewest number of towers.

   Starting from position 0, always select the furthest tower to the right that can cover the current uncovered point. Repeat this process until the entire highway is covered.

   (b) (4 points) Justify the run-time.

   The list of tower positions is given sorted, so no additional sorting is needed. We iterate through the tower positions at most once, maintaining the furthest-reaching tower available at each step. Thus, the algorithm runs in $O(n)$ time.

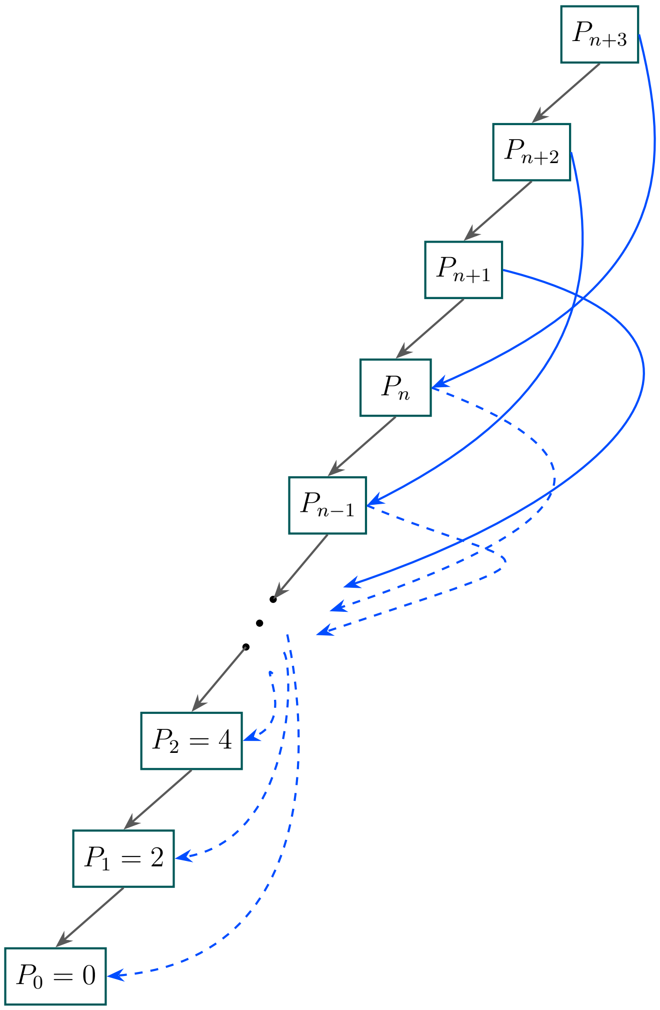
   (c) (6 points) Prove that this greedy algorithm always results in an optimal solution.

   By contradiction, if an optimal solution skips the greedy choice, it would leave a portion uncovered or require more towers. Therefore, the greedy algorithm produces the minimum number of towers needed.

6. The relation between subproblem of some dynamic programming problem is determined by:

$$P_0 = 0, \quad P_1 = 2, \quad P_2 = 4, \quad \text{and} \quad P_{n+3} = nP_n + P_{n+2} \text{ for } n \geq 0.$$

(a) (6 points) Draw a graph of computations for a *dynamic programming* solution of this problem.



(b) (2 points) What is $\Theta(\cdot)$ run-time of the dynamic programming algorithm?

$\Theta(n)$

(c) (2 points) What the $O(\cdot)$ run-time would be without memoization?

$O(2^n)$

7. Consider the following three algorithms:

   - ALGORITHM $\mathcal{A}$ solves problems of size n by recursively dividing them into 2 sub-problems of size $n/2$ and combining the results in constant time c.
   - ALGORITHM $\mathcal{B}$ solves problems of size n by solving one sub-problem of size $n/2$ and performing some processing taking some constant time c.
   - ALGORITHM $\mathcal{C}$ solves problems of size n by solving two sub-problems of size $n/2$ and performing a linear amount of extra work.

   (a) (6 points) For each algorithm, write down a recurrence relation showing how $T(n)$, the running time on an instance of size n, depends on the running time of a smaller instance.

   ALGORITHM $\mathcal{A}$: $T(n) = 2T(n/2) + \Theta(1)$

   ALGORITHM $\mathcal{B}$: $T(n) = T(n/2) + \Theta(1)$

   ALGORITHM $\mathcal{C}$: $T(n) = 2T(n/2) + \Theta(n)$

   (b) (6 points) For each recurrence relation, pick the solution for $T(n)$ from the following list. Just write the letter corresponding to the correct running time.

   Algorithm $\mathcal{A}$: *C*

   Algorithm $\mathcal{B}$: *B*

   Algorithm $\mathcal{C}$: *D*

   A: $T(n) \sim c$
   B: $T(n) \sim c \log n$
   C: $T(n) \sim cn$
   D: $T(n) \sim cn \log n$
   E: $T(n) \sim cn^2$

   (c) (4 points) For each of the following algorithms, pick which of the above classes of algorithms ($\mathcal{A}$, $\mathcal{B}$, or $\mathcal{C}$) applies to that algorithm:

   - Mergesort $\mathcal{C}$

   - Binary search in a sorted array $\mathcal{B}$

8. Consider a dataset of n elements. For each scenario below, specify and justify the most efficient sorting method that can be used.

   (a) (5 points) The elements are associated with integer keys in the range 0 to $n-1$. The key for each element can be obtained in $O(1)$ time.

   Since keys are integers within a known, small range $[0, n-1]$, counting sort is the most efficient. Counting sort runs in $O(n)$ time, which matches the lower bound for reading all n elements. Radix sort could also be applied with similar time complexity, but counting sort is simplest here.

   (b) (5 points) The elements must be sorted based on a comparison function that determines whether one element is less than, greater than, or equal to another. Each comparison takes $O(1)$ time, but no assumptions can be made about the distribution or range of the keys.

   This is a standard comparison-based sorting problem. The optimal time complexity is $O(n \log n)$ in the worst case, achievable using algorithms such as merge sort or heap sort. The $\Omega(n \log n)$ lower bound applies due to the use of comparisons.

9. Each question has only **one** correct answer. *No need for explanations here.*

   (a) (4 points) Which of the following sorting algorithms gives best performance when applied on an array which is sorted or almost sorted (maximum one or two elements are misplaced).

   A. Selection sort

   B. Insertion sort

   C. Merge sort

   D. Counting sort

   E. Radix sort

   (b) (4 points) Strassen's algorithm for matrix multiplication reduces the run-time by introducing which key idea?

   A. It reduces number of matrix additions/subtractions.

   B. It sorts the rows and columns of the input matrices before multiplying.

   C. It expresses the product of two matrices using only 7 recursive multiplications instead of 8, combined with additional additions and subtractions.

   D. It uses dynamic programming to avoid redundant subproblems.

   E. It compresses the matrices into smaller dimensions before multiplying.

   (c) (4 points) Which of the following statements best distinguishes Divide and Conquer from Dynamic Programming?

   A. Divide and Conquer always produces the optimal solution, while Dynamic Programming does not.

   B. Dynamic Programming divides problems into independent subproblems, while Divide and Conquer uses overlapping subproblems.

   C. Divide and Conquer solves overlapping subproblems using recursion, while Dynamic Programming solves independent subproblems iteratively.

   D. Divide and Conquer breaks the problem into independent subproblems, while Dynamic Programming solves overlapping subproblems by storing results of subproblems.

   E. Dynamic Programming and Divide and Conquer both only apply to optimization problems.

10. (10 points) Given three arrays A, B, and C, each containing n integers, provide an algorithm with a time complexity of $O(n^2)$ to determine whether there exist elements $a \in A$, $b \in B$, and $c \in C$ such that:

$$a + b + c = 0.$$

Solution: For each pair $(a, b) \in A \times B$, store $a + b$ in a hash table H. Then return Yes if $-c$ appears in H for any $c \in C$, and return No otherwise.

This page is intentionally left blank to accommodate work that would not fit elsewhere and/or scratch work.