# 4520/6520 Design and Analysis of Algorithms

## Mockup Midterm

Instructor: Alina Nemira

10/06/2025

Name: ______________________________

Honor Code Statement: I will not commit any act of academic dishonesty while completing this assignment. I am fully aware that any of my own personal actions while attempting this assignment that are interpreted as academic dishonesty, will be treated as such. I understand that if I am held accountable for an act of academic dishonesty that I will receive a grade of "0" (zero) for this assignment and the incident will be reported to the Dean of Students Office.

The mockup midterm contains 12 pages (including this cover page) and 11 problems. Total of points is 125. You will receive min{your score, 100} points.

Good luck and productive work!

## Distribution of grades

| Question | Points | Score |
|---|---|---|
| 1 | 8 | |
| 2 | 10 | |
| 3 | 10 | |
| 4 | 5 | |
| 5 | 20 | |
| 6 | 10 | |
| 7 | 14 | |
| 8 | 12 | |
| 9 | 10 | |
| 10 | 16 | |
| 11 | 10 | |
| **Total:** | **125** | |

1. (8 points) A smart guy Carl thinks it is possible to create a better version of the Merge sorting algorithm by breaking the array into three pieces (instead of two as it was described in class). Of course, this idea came to his mind, after he could come up with an algorithm to merge three sorted arrays in linear time. He implemented his algorithm and called the procedure Carl'sMerge(a[1..n], $l$, r). Supposing a[1..$l$], a[$l$ + 1..r] and a[r + 1..n] are sorted, Carl'sMerge merges them into one sorted array in $\Theta(n)$ time. Based on this procedure, Carl has designed his recursive algorithm that follows.

**Algorithm 1. Carl's version of MergeSort**

```
1. function Carl'sSort($a[1..n]$)
2. if $n > 1$ then
3.        $l \leftarrow \lfloor \frac{n}{3} \rfloor$
4.        $r \leftarrow \lfloor \frac{2n}{3} \rfloor$
5.        Carl'sSort($a[1..l]$)
6.        Carl'sSort($a[l+1..r]$)
7.        Carl'sSort($a[r+1..n]$)
8.        Carl'sMerge($a[1..n], l, r$)
```

Estimate the running time of this algorithm. Is this algorithm faster than regular MergeSort? Explain your answer.

**Solution.** The algorithm has three recursive calls to subproblems of size $n/3$, and spends $\Theta(n)$ for merging. Hence, if $T(n)$ shows the running time we have:

$$T(n) = 3T(n/3) + \Theta(n).$$

Using the master theorem we conclude that Carl's algorithm runs in $\Theta(n \log n)$ time. Carl's algorithm runs in $\Theta(n \log n)$ time, similar to regular MergeSort.

2. (10 points) Describe a divide-and-conquer algorithm that finds the maximum difference between elements of a given array of size $n$ in $O(n)$ time.

For instance, on input $[4, 5, 10, -4, 2, 4, -7]$, your algorithm should return 17.

Note: for full marks, your answer must make use of the divide-and-conquer approach.

**Solution.** A divide-and-conquer approach is to recursively split the array in half. For each half, we find the minimum and maximum elements. The overall minimum is the minimum of the two subarray minimums, and the overall maximum is the maximum of the two subarray maximums. The final difference is the overall maximum minus the overall minimum.

The recurrence relation is $T(n) = 2T(n/2) + \Theta(1)$, because we make two recursive calls on problems of half the size and do a constant number of comparisons to find the overall min and max. By the Master Theorem, this solves to $T(n) = \Theta(n)$.

Base case: If the array has one element, min and max are both that element. If it has two elements, a single comparison finds the min and max.

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

(a) (4 points) **Top-down** dynamic programming solutions make recursive calls according to the recurrence relation while **bottom-up** dynamic programming solutions strategically iterate over each state.

*List of words:*
top-down
bottom-up

(b) (6 points) **Divide and Conquer** algorithm solves a problem by dividing it into smaller non-overlapping subproblems.

**Greedy** algorithm chooses at each step the best at a time option that eventually will form a solution.

**Dynamic Programming** algorithm solves a problem by dividing it into smaller subproblems that overlap; stores their solutions to avoid repeated calculations.

*List of words:*
Greedy
Divide and Conquer
Dynamic Programming

4. (5 points) Which of the following must be done when converting a top-down dynamic programming solution into a bottom-up dynamic programming solution? Select all that applies, no explanation needed here.

A. Change plus to minus in the recurrence relation.
B. Use recursion instead of iterations.
C. Create new base cases.
**D. Use iterations instead of recursion.**
**E. Find the correct order in which to iterate over the states.**

5. Decide if the statements below are True or False. Explain your answers. *No credit here without a proper explanation!*

(a) (4 points) With all equal-weighted intervals, a greedy algorithm based on earliest finish time will always select the maximum number of compatible intervals.
**TRUE.** The algorithm is equivalent to the earliest finish time algorithm for unweighted intervals, which is proven to be optimal.
(b) (4 points) Any dynamic programming algorithm that solves $n$ subproblems must run in $\Omega(n)$ time.
**TRUE.** To solve $n$ distinct subproblems, an algorithm must perform at least one operation for each subproblem, resulting in a total time of at least $\Omega(n)$.
(c) (4 points) $n^2 \in O(2^n)$.
**TRUE.** Exponential functions like $2^n$ grow much faster than polynomial functions like $n^2$. Therefore, $n^2$ is bounded above by $c \cdot 2^n$ for some constant $c$ and large enough $n$.
(d) (4 points) The order of items in the sequence interface is intrinsic, while the set interface has an extrinsic order of items.
**FALSE.** It is the opposite. A sequence has an extrinsic order (imposed by the user), while a set's order is intrinsic (determined by the elements themselves, e.g., sorted order).
(e) (4 points) A divide and conquer algorithm for finding a maximum element in an array problem that divides the array into two half-size subarrays and does a constant amount of extra work to merge the results would yield a linear time algorithm.
**TRUE.** The recurrence would be $T(n) = 2T(n/2) + \Theta(1)$, which solves to $T(n) = \Theta(n)$ by the Master Theorem.

6. In some town, $n$ houses are located on one side of the road, one by one.

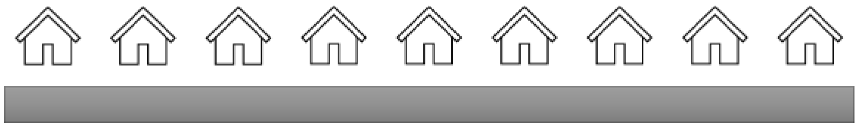
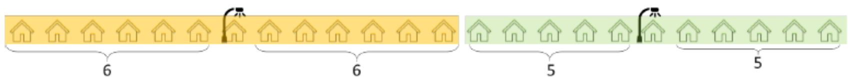


Figure 1: An example of a street illuminated by lamp posts.

You are given an array of $m$ lamp posts with non-negative integer light values, where 0 means that the lamp post can only light up a section outside one house, 1 means lighting a given house plus two immediately neighboring houses from the lamp pole, and a light value of $k$ means that houses a distance $k$ away, in addition, will be illuminated. You can place a lamp post next to any house. A house may be lit by several lamp posts.

(a) (2 points) If we have an array of five lamp posts $[3, 5, 2, 6, 1]$, what is the minimum number of lamp posts needed to light the street with len $= 24$ houses?

**Solution.** Two lamp posts are enough. A lamp post with light value $k$ covers $2k+1$ houses. We have coverage lengths of [7, 11, 5, 13, 3]. To cover 24 houses, we can use the two largest, 6 and 5 (lengths 13 and 11). For instance, place the lamp post with value 6 at house 7 to cover houses [1, 13]. Place the lamp post with value 5 at house 19 to cover houses [14, 24]. Please see picture:



(b) (8 points) Prove the greedy algorithm that sorts the set of lamp posts in decreasing order (and picks lamp posts until all houses on the street are lit) is optimal.

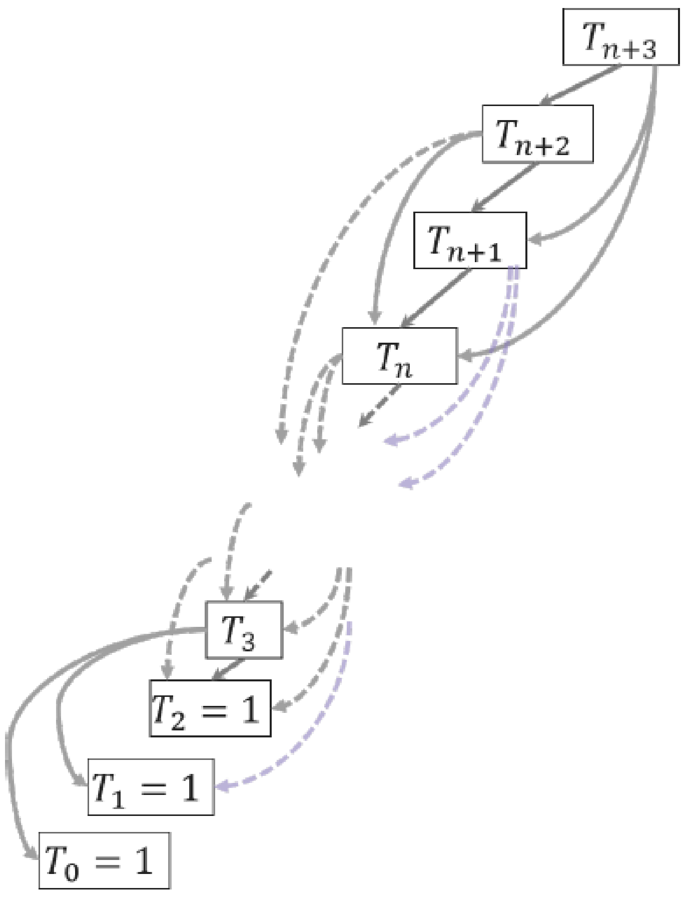
**Solution.** This problem is analogous to the set cover problem, which is NP-hard. The greedy strategy described (picking the lamp post with the largest light value) is not guaranteed to be optimal for all placements. However, if the strategy is to always cover the first uncovered house with the lamp post that reaches farthest to the right, this is a standard greedy approach for the interval covering problem.

Let the greedy solution be $G = \{g_1, g_2, ..., g_k\}$ and an optimal solution be $O = \{o_1, o_2, ..., o_m\}$, where $k \geq m$. The greedy choice is to cover the leftmost uncovered house with the lamp that provides the maximum reach. Let's say our first choice $g_1$ covers up to point $r_1$. Some lamp in the optimal solution, $o_1$, must also cover the start point. The greedy choice ensures that $r_1$ is greater than or equal to the reach of $o_1$. We can thus substitute $o_1$ with $g_1$ in the optimal solution and still have a valid cover. By continuing this exchange argument, we show that the greedy solution uses no more lamps than the optimal one. Therefore, the greedy algorithm is optimal.

7. The Tribonacci sequence $T_n$ is defined as follows:

$T_0 = 0$, $T_1 = 1$, $T_2 = 1$, and $T_{n+3} = T_n + T_{n+1} + T_{n+2}$ for $n \geq 0$. Given $n$, return the value of $T_n$.

(a) (10 points) Draw a graph of computations for a dynamic programming solution of this problem.



(b) (2 points) What is a tight bound on the running time of the dynamic programming algorithm?

**Solution.** The running time is $\Theta(n)$. We need to compute $T_3, T_4, ..., T_n$, each taking constant time using the previous three values. This results in a linear number of computations.

(c) (2 points) What would the run-time be without memoization (saving solutions to the subproblems in memory)?

**Solution.** Without memoization, we would recompute subproblems repeatedly. The recurrence is $T(n) = T(n-1) + T(n-2) + T(n-3)$. This leads to an exponential number of calls, and the run time would be $O(3^n)$.

8. (12 points) Three divide-and-conquer algorithms A, B and C are proposed to solve the same problem. Suppose they are all correct, what are their running time (in $\Theta(\cdot)$ notation) and which one is more efficient? Justify your answer.

**Algorithm A**: divides the problem of size n into 10 subproblems of size $n/10$ and combines the solutions in linear time.

$$T(n) = 10T(n/10) + \Theta(n) \Rightarrow T(n) \in \Theta(n \log n)$$

**Algorithm B**: divides the problem of size n into 4 subproblems of size $n/2$ and combines the solutions in constant time.

$$T(n) = 4T(n/2) + \Theta(1) \Rightarrow T(n) \in \Theta(n^2)$$

**Algorithm C**: divides the problem of size n into 9 subproblems of size $n/3$ and combines the solutions in quadratic time.

$$T(n) = 9T(n/3) + \Theta(n^2) \Rightarrow T(n) \in \Theta(n^2 \log n)$$

**Answer.** Algorithm A, with a runtime of $\Theta(n \log n)$, is the most efficient. $\Theta(n \log n)$ grows slower than both $\Theta(n^2)$ and $\Theta(n^2 \log n)$.

9. (10 points) An integer array contains $n$ distinct integers. It is known that exactly $k = \lceil \sqrt{n} \rceil$ elements in the array are even. The odd integers in the array appear in sorted order. Describe an $O(n)$-time algorithm to sort the array.

**Solution.**

1. Create two new arrays: one for even numbers (`evens`) and one for odd numbers (`odds`).
2. Iterate through the input array once. If an element is even, add it to `evens`. If it is odd, add it to `odds`. This takes $O(n)$ time.
3. The `odds` array is already sorted by the problem definition.
4. Sort the `evens` array. It contains $k = \lceil \sqrt{n} \rceil$ elements. Using an efficient sorting algorithm like Merge Sort, this takes $O(k \log k) = O(\sqrt{n} \log(\sqrt{n})) = O(\sqrt{n} \log n)$ time. Since $\sqrt{n} \log n$ is much smaller than $n$, this step is efficient.
5. Merge the sorted `odds` and sorted `evens` arrays back into the original array. This is a standard merge operation and takes $O(n)$ time.

The total time complexity is dominated by the linear-time steps, so the algorithm runs in $O(n)$.

10. Each question may have one or more correct answers. No need for explanations here.

(a) (4 points) How many comparisons does selection sort make on an input array that is already sorted?
A. Constant
B. Logarithmic
C. Linear
**D. Quadratic**
E. Exponential

(b) (4 points) How many comparisons does insertion sort make on an input array that is already sorted?
A. Constant
B. Logarithmic
**C. Linear**
D. Quadratic
E. Exponential

(c) (4 points) What data structure is appropriate if you need insertion, deletion, and look up to be performed in $O(1)$ time (on average)?
A. Static sorted array
B. Dynamic unsorted array
C. Dynamic sorted array
D. Linked list
**E. Hash set**

(d) (4 points) Which of the following operations has a time complexity of $\Theta(1)$?
**A. Adding an item to the back of a queue**
**B. Removing an item from the front of a queue**
C. Finding an arbitrary element in a queue
D. Iterating through a queue
**E. Peeking at the front of a queue**

11. (10 points) You are given an integer array of size $n$. Two elements in the array are called coupled if they differ by one. Describe an algorithm that in $O(n)$ counts all *distinct* coupled pairs in the array.

**Solution.** An efficient approach uses a hash set to achieve $O(n)$ average time complexity.

1. Create a hash set and insert all elements from the input array into it. This takes $O(n)$ time on average.
2. Initialize a counter for coupled pairs to zero.
3. Iterate through each element 'x' in the hash set.
4. For each 'x', check if 'x + 1' exists in the hash set. The lookup operation is $O(1)$ on average.
5. If 'x + 1' exists, increment the counter.

This process checks each potential pair '(x, x+1)' exactly once, avoiding duplicates. The total time complexity is dominated by the initial insertion into the hash set, making it $O(n)$ on average.