Midterm Solutions

1 True or false (10 marks)

State whether each of the following statement is **True** or **False** and give a short justification for each answer. (Full proofs are not necessary for this question.)

(i) The functions $f(n) = 1.001^n$ and $g(n) = n^{618}$ satisfy $f = \omega(g)$.

Solution. **True.** $\lim_{n->\infty} g(n)/f(n) = 0$ (apply L'Hopital rule sufficiently many times) so $g \in o(f)$, which is equivalent to $f = \omega(g)$.

(ii) If $f = o(g)$ then the functions F, G defined by $F(n) = \log_2(f(n))$ and $G(n) = \log_2(g(n))$ satisfy $F = o(G)$.

Solution. **False.** Consider $f(n) = n^2$ and $g(n) = n^3$. Then $F = \Theta(G)$.

(iii) In the 4SUM problem, the input is an array A of n integers and the valid solution is `True` if there exist 4 indices $i, j, k, \ell \in \{1, 2, \ldots, n\}$ such that $A[i] + A[j] + A[k] + A[\ell] = 0$, and `False` otherwise.

True or false: The 4SUM problem can be solved in time $O(n^2 \log n)$.[1]

Solution. **True.** We can reduce to the CommonSum problem from A1. Create an array B with $B[i] = -A[i], i = 1, 2, \ldots, n$; the 4Sum problem on A is the same as the CommonSum problem on the arrays A and B.

[1] Reminder: We saw some SUM problems in lectures and Assignment 1 that can be solved in time $O(n^2 \log n)$.

(iv) The following `test` procedure has time complexity $\Omega(n^2)$.

```
1       test(n)
2          for j ← 2 to n
3             k ← 1
4             while k < n^2 do
5                k ← 3 * k
6             end while
7          end for
```

Solution. **False.** The inner loop has running time $\Theta(\log_3 n^2) = \Theta(\log n)$. The complexity of the outer loop is $\sum_{j=2}^{n} \Theta(\log n) = \Theta(n \log n)$ which is not in $\Omega(n^2)$.

(v) The *longest exponential subsequence* of a sequence of n positive integers $a_1 \leq a_2 \leq \cdots \leq a_n$ is the longest subsequence such that each integer in the subsequence is at least twice as large as the previous one.

True or false: It is possible to determine the length of the longest exponential subsequence of a sequence of n positive integers in time $O(n^2)$.

Solution. **True.** We can use the same algorithm designed for the Longest Increasing Subsequence in Tutorial 5 and replace comparison $a_j < a_k$ by $2a_j \leq a_k$.

2 Analysis of algorithms (10 marks)

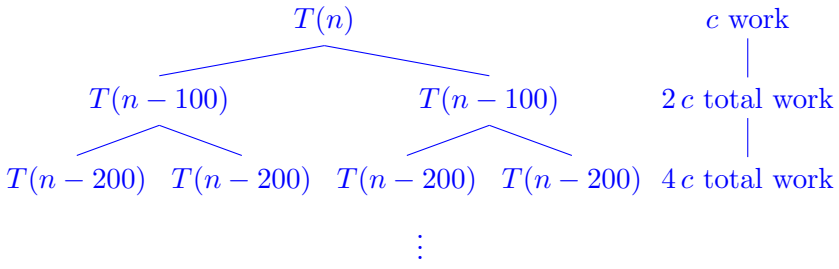
Consider four different algorithms that solve the same problem:

- Algorithm **A** solves a problem of size n by dividing it into two subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in time $\Theta(n^2)$.
- Algorithm **B** solves a problem of size n by recursively solving two subproblems of size $n-100$ and then combining the solutions in constant time.
- Algorithm **C** solves a problem of size n by dividing it into 15 subproblems of size $n/5$, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm **D** solves a problem of size n by dividing it into 12 subproblems of size $n/12$, recursively solving each subproblem, and then combining the solutions in linear time.

For each algorithm, write down the recurrence relation for the worst-case run time $T(n)$ of the algorithm, solve it to obtain the algorithm's time complexity, and rank the algorithms from the fastest to the slowest according to this worst-case time complexity.

Solution.

1. For **A**: $T(n) = 2T(\frac{n}{3}) + c \cdot n^2$, which solves to $\Theta(n^2)$ by the Master Theorem.
2. For **B**: $T(n) = 2T(n-100) + c$. The recursion tree for this recurrence is



   There is $2^d\,c$ total work at level d in the tree, and there are $n/100$ total levels in the tree, so $T(n) = \Theta(\sum_{d=1}^{n/100} 2^d) = \Theta(2^{n/100})$.

3. For **C**: $T(n) = 15T(\frac{n}{5}) + c \cdot n$, which solves to $\Theta(n^{\log_5 15})$ by the Master Theorem ($\log_5 15$ is between 1 nd 2).
4. For **D**: $T(n) = 12T(\frac{n}{12}) + c \cdot n$, which solves to $\Theta(n \log n)$ by the Master Theorem.

The order from smallest worst case run time (asymptotically fastest) to largest (asymptotically slowest) is: D, C, A, B.

3 Divide and Conquer (12 marks)

Definition. An array A of integers is *block-ordered* if all the copies of a given integer occur in adjacent positions. For example, the array $(2, 2, 1, 1, 1, 9, 9, 9, 5)$ is block-ordered and $(1, 9, 1, 4)$ is not.

In the DistinctElements problem, the input is a block-ordered array A of n positive integers in the range $\{1, 2, \ldots, M\}$, where M is some fixed constant, and the valid solution on this input is the number m of distinct integers contained in A. For example, on input $A = (2, 2, 1, 1, 1, 9, 9, 9, 5)$, the valid solution is 4.

(i) Design a Divide & Conquer algorithm that solves the DistinctElements problem. You should aim for an algorithm with time complexity $O(M \log n)$. (You only need to provide the description and pseudocode of the algorithm in this part. You will complete the analysis of the algorithm in the following parts of the question.)

Hint: To obtain the desired time complexity, consider carefully the base case of your algorithm.

Solution. We can

- Divide the array into two subarrays $A[1, \ldots, n/2]$ and $A[n/2, \ldots, n]$,
- Conquer both instances by calling our algorithm recursively on the smaller arrays, and
- Combine the answer by adding the number of distinct elements from each subarray and correcting (by subtracting 1) if there was an element that appeared in both subarrays; this condition can be easily checked by seeing if $A[n/2] = A[n/2 + 1]$ or not.

To obtain the desired time complexity, we want to define the base case to return 1 whenever A contains only 1 distinct element—we can easily check whether this is the case because it occurs if and only if $A[1] = A[n]$.

The pseudocode for this algorithm is as follows.

Algorithm 1: Distinct$(A[1, \ldots, n])$

```
if A[1] = A[n] then
    return 1;
d_1 ← Distinct(A[1, ..., n/2]);
d_2 ← Distinct(A[n/2 + 1, ..., n]);
if A[n/2] ≠ A[n/2 + 1] then
    return d_1 + d_2;
else
    return d_1 + d_2 − 1;
```

(ii) Explain why your algorithm from part (i) solves the DistinctElements problem.

Solution. The base case is correct because if $A[1] = A[n]$, then the block-ordered property of A guarantees that A contains only 1 distinct element. And if A contains only 1 distinct element, then necessarily $A[1] = A[n]$.

The combine step is correct because, since A is block-ordered, the only element that can appear in both $A[1, \ldots, \frac{n}{2}]$ and $A[\frac{n}{2} + 1, \ldots, n]$ is the one that is in a block that crosses the boundary between the two subarrays. So the number of distinct elements in A is $d_1 + d_2$ if $A[n/2] \neq A[n/2 + 1]$ (and the two subarrays have no element in common) and $d_1 + d_2 - 1$ otherwise.

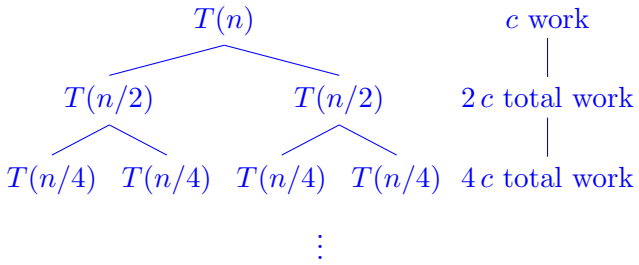
(iii) Write a recurrence relation for the time complexity $T(n)$ of your algorithm for part (i).

Solution. The algorithm calls itself on two subarrays of size $n/2$ and only does a constant amount of extra work, so

$$T(n) = 2\,T(n/2) + O(1).$$

(iv) Use recurrence trees to analyze the time complexity of your algorithm from part (i).

Solution. The recurrence tree for the expression above is



The depth of the tree is $O(\log n)$. Each node at level d does a constant amount of extra work. This means that at depth d, there is a total amount of at most $O(2^d)$ extra work being done at that level. Except that we can do even better: since we stop calling the algorithm recursively whenever we hit an array that contains only 1 distinct element, there can be at most $O(M)$ nodes at any level so that the total amount of work at level d is $O(M)$ for every $d \geq 1$. Therefore, $T(n) = O(\sum_{d=1}^{\log n} M) = O(M \log n)$.

4 Greedy algorithms (10 marks)

In the IntervalCover problem, the input is a set of n points on the line $0 \leq p_1 < p_2 < \cdots < p_n \leq M$, and the valid solution is the minimum number k of unit intervals $[x_1, x_1 + 1], \ldots, [x_k, x_k + 1]$ required to cover all n points. (A point p_i is *covered* by the interval $[x_j, x_j + 1]$ if $x_j \leq p_i \leq x_j + 1$.

(i) Show that the greedy algorithm that at each step selects an interval that covers the largest number of still-uncovered points does not solve the problem.

Solution. Consider the input with 4 points at positions

$$0 \quad 0.8 \quad 0.9 \quad 1.1 \quad 1.2 \quad 2.$$

The algorithm will select an interval containing the four points at positions $0.8, 0.9, 1.1, 1.2$ first, and must then select one more interval for each of the points $0, 2$, resulting in a solution with 3 intervals. But the optimal requires only two intervals, e.g. $[0, 1], [1, 2]$.

(ii) Describe a greedy algorithm that solves the IntervalCover problem.

Until all the points are covered, set $x_i = p_j$ where p_j is the first point not covered by $[x_1, x_1 + 1], \ldots, [x_{i-1}, x_{i-1} + 1]$.

(iii) Prove that the algorithm obtained in part (ii) always returns a valid solution.

Solution. There are multiple possible proofs:

(a) (This is not an exchange or induction proof.) Let $p_{i_1}, \ldots, p_{i_k}$ be the set of points selected by the greedy algorithm. Then we must have $p_{i_j} > p_{i_{j-1}} + 1$ for all $j \in \{2, \ldots, k\}$ by definition of the algorithm. Therefore any unit interval can contain at most 1 of these points, so there is a lower bound of k intervals to cover all points. Since the algorithm returns k, it is optimal.

(b) By induction on the number of points n.

Base case: $n = 1$. Then we must have one interval, which the greedy algorithm provides.

Inductive step: suppose the greedy algorithm produces an optimal solution for all sets of $< n$ points. Let $x_1, \ldots, x_k$ be an optimal solution for $p_1, \ldots, p_n$, sorted so that $x_1 < \cdots < x_k$.

The first interval selected by the greedy algorithm is $[p_1, p_1 + 1]$, and we must have $x_1 \leq p_1$ otherwise p_1 is not covered. Let p_i be the first point not covered by $[p_1, p_1 + 1]$ and p_j be the first point not covered by $[x_1, x_1 + 1]$. Then $p_j \leq p_i$ since $x_1 + 1 < p_1 + 1$, so $x_2, \ldots, x_k$ is a solution of size $k - 1$ for the points $p_j, \ldots, p_n$, so it is also a solution of size $k - 1$ for the points $p_i, \ldots, p_n$. Then the greedy algorithm on $p_i, \ldots, p_n$ produces a solution of size at most $k - 1$ by induction, so the solution for $p_1, \ldots, p_n$ is size at most k.

(c) By induction on the number k of intervals in the optimal solution.

Base Case: $k = 1$. All points are covered by a single interval so $p_n \leq p_1 + 1$; the greedy algorithm will select the interval $[p_1, p_1 + 1]$ which covers all the points.

Inductive Step: Suppose the greedy algorithm produces the optimal solution for any sequence of points with optimal solution $k' < k$, and let $x'_1, \ldots, x'_k$ be the starting point of each interval in the optimal solution. Let $i < n$ be the largest point covered by the intervals $x'_1, \ldots, x'_{k-1}$. By induction, the greedy algorithm produces a sequence of intervals $x_1, \ldots, x_{k-1}$ that cover $p_1, \ldots, p_i$. $p_{i+1}, \ldots, p_n$ are covered by the interval $[x'_k, x'_k + 1]$ so $p_n \leq p_{i+1} + 1$, so there is some interval that covers all the remaining points. The greedy algorithm will choose the first point not yet covered by $x_1, \ldots, x_{k-1}$ which must be later than p_i; the remaining points can be covered with 1 interval so the greedy choice will cover these, for a total of k intervals.

(d) Let $x'_1, \ldots, x'_k$ be an optimal solution sorted such that $x'_i < x'_{i+1}$, and let $x_1, \ldots, x_\ell$ be the solution given by the greedy algorithm (also sorted).

Let j be the first index such that $x'_j < x_j$. Suppose there is a point p_i such that $x'_j \leq p_i < x_j$ (i.e. replacing x'_j with x_j would leave p_i uncovered). Then $p_i \leq x_{j-1} + 1$, otherwise p_i is not covered by the greedy solution, a contradiction. But then $p_i \leq x_{j-1} + 1 \leq x'_{j-1} + 1$, so p_i is covered by some interval $j' \leq j - 1$ in the optimal solution. Then we can set $x'_j \leftarrow x_j$ and all the points remain covered and we reduce the number of differences between the solutions.

(iv) Analyze the time complexity of the algorithm from part (ii).

Solution. If the algorithm assumes that the points are in sorted order: For each point p_i, the algorithm will check $p_i \leq x_j + 1$ where x_j is the most recently selected interval. This takes $O(1)$ time and will be repeated at most $\Theta(n)$ times, for a total of $\Theta(n)$.

If the algorithm does not assume that the points are sorted, it will require $\Theta(n \log n)$ for sorting, for a total of $\Theta(n \log n)$.

5 Dynamic programming (12 marks)

In the SpacedSum problem, the input is an array A of n positive integers and the valid solution for this input is the largest possible value that can be obtained by summing a subset of the entries of A such that the subset contains at most one of any three consecutive entries $A[i], A[i+1], A[i+2]$ for every $1 \leq i \leq n-2$.

In this question, you will design and analyze a dynamic programming algorithm that solves the SpacedSum problem.

(i) Provide an example to show that in some cases, the optimal solution can skip three consecutive entries in the array.

Solution. Consider $n = 5$ and $A = [100, 1, 1, 1, 100]$. An optimal solution that satisfies given constraints is 200. Selecting one of the entries with the value 1 prevents the selection of one of the entries with the value 100.

(ii) Give a precise definition of a subproblem that can be used to solve the SpacedSum problem with a dynamic programming algorithm.

Solution. The subproblem is to compute $S(k)$ = the solution to the SpacedSum problem on the input $A[1, \ldots, k]$.

(iii) Describe and justify the recurrence rule that will be used to compute the solutions to the subproblems defined in part (ii).

Solution. The recurrence is

$$S(k) = \begin{cases} 0 & \text{if } k = 0 \\ A[1] & \text{if } k = 1 \\ \max(A[1], A[2]) & \text{if } k = 2 \\ \max(S(k-3) + A[k], S(k-1)) & \text{if } k > 2. \end{cases}$$

The base cases are justified by the fact that

$S(0) = 0$ since there are no elements that we can add
$S(1) = A[1]$ because the entries of A are positive
$S(2) = \max(A[1], A[2])$ because we can only include one entry of A in this case.

The justification for the recurrence when $k > 2$ follows from the fact that there are two possibilities for $S(k)$:

- if we use the entry $A[k]$ then $S(k) = S(k-3) + A[k]$, as we must skip entries $A[k-1]$ and $A[k-2]$; and
- if we do not use the entry $A[k]$ then $S(k) = S(k-1)$.

The value of $S(k)$ is the maximum of those two possible values.

(iv) Provide the pseudocode for your dynamic programming algorithm that solves the SpacedSum problem using the subproblems and recurrence rule defined above.

Solution.

```
S[0] = 0;
S[1] = A[1];
S[2] = max(A[1],A[2]);
for i from 3 to n do
  S[i] = max(S[i-3]+A[i], S[i-1])
od
RETURN S[n]
```

(v) Analyze the time complexity of the algorithm obtained in part (iv).

Solution. The loop in the algorithm iterates $n-2$ times and each iteration takes a constant amount of time, so the time complexity of the algorithm is $\Theta(n)$.