Question 1: Asymptotic Analysis (23 points)

Solve the following recurrence relations using the method specified in each exercise. Your final result should be in the form of an upper bound $\mathcal{O}(f(n))$ for the tightest possible function $f(n)$.

Note: When applying each method, clearly explain intermediate steps, i.e. do not simply state the final result without explanation.

(a) (4 points) Solve $T(n) = 3 \cdot T(n/9) + \sqrt{n}$ for $n \geq 9$ with $T(n) = 1$ for $n < 9$ using the master theorem.

(b) (6 points) Solve $T(n) = 2 \cdot T(n/4) + n^2$ for $n \geq 4$ with $T(n) = 1$ for $n < 4$ using the recursion tree method. You may assume that n is a power of 4.

Note: A description of the tree structure in text form is sufficient, you do not need to draw the tree.

(c) (6 points) Solve $T(n) = T(n/2) + T(n/3) + n^2$ for $n \geq 3$ with $T(n) = 1$ for $n < 3$ using the substitution method.

For the following two exercises, provide a short formal proof.

(d) (3 points) Show that $T(n) = 2n\sqrt{n}$ is not $\mathcal{O}(n)$.

(e) (4 points) Show that $T(n) = 2^{3 \cdot \log_3(n) / \log_3(2)}$ is $\Theta(n^3)$.

SOLUTION:

(a) We write $T(n)$ in standard recurrence form for $a = 3$, $b = 9$ and $d = 0.5$. Since $a \geq 1, b > 1$ and $d \geq 0$, the master theorem applies. Since $a = 3 = 3 = b^d$, the master theorem gives $T(n) = \mathcal{O}(n^d \log_2(n)) = \mathcal{O}(\sqrt{n} \log_2(n))$.

(b) Draw a tree where each node has 2 branches and the node size is $\frac{n}{4^k}$ at level k.

The total work at level $k = 0, 1, \ldots$ is

$$c \cdot 2^k \cdot \left(\frac{n}{4^k}\right)^2 = c \cdot n^2 \cdot \left(\frac{2}{4^2}\right)^k = c \cdot n^2 \cdot \left(\frac{1}{8}\right)^k$$

The tree starts at level $k = 0$ and the last level has subproblem size 1 with $1 = \frac{n}{4^k} \Leftrightarrow k = \log_4(n)$. Hence, the runtime of the algorithm is

$$\begin{aligned} T(n) &= \sum_{k=0}^{\log_4(n)} \left[c \cdot n^2 \left(\frac{1}{8}\right)^k \right] \\ &= c \cdot n^2 \cdot \sum_{k=0}^{\log_4(n)} \left(\frac{1}{8}\right)^k \end{aligned}$$

To bound the runtime expression, we use the bound for a finite geometric series we discussed in lecture 3 with

$$m \cdot \sum_{j=0}^{k} r^j \leq m \cdot \frac{1}{1-r} \qquad 0 < r < 1$$

Applying this with $r = 1/8 < 1$ gives

$$T(n) = c \cdot n^2 \frac{1}{1 - 1/8} = \mathcal{O}(n^2)$$

(c) We guess $T(n) = \mathcal{O}(n^2)$ and use the following inductive proof

- **Inductive hypothesis:** $T(n) \leq c \cdot n^2$ for some c, n_0 and $n \geq n_0$
- **Inductive basis:** $T(1) = 1 \leq c \cdot 1^2$ e.g. $c = 2$ and $n_0 = 1$
- **Inductive step:** Let $n_0 \leq m < k \leq n$ and assume the inductive hypothesis holds for all m. Then

$$\begin{aligned}
T(k) &= T(k/2) + T(k/3) + k^2 \\
&\leq c \cdot (k/2)^2 + c \cdot (k/3)^2 + k^2 && \text{(inductive hypothesis)} \\
&= 2 \cdot k^2/4 + 2 \cdot k^2/9 + k^2 \\
&= k^2/2 + \frac{2k^2}{9} + k^2 \\
&= \left(\frac{3}{2} + \frac{2}{9}\right) \cdot k^2 \\
&\leq 2 \cdot k^2 = c \cdot k^2
\end{aligned}$$

- **Conclusion:** For $k = n$, we have by the inductive hypothesis that $T(n) \leq c \cdot n^2$ for $n \geq n_0$ with $c = 2$, $n_0 = 1$.

(d) **(Proof by contradiction)** Assume that $T(n) = \mathcal{O}(n)$. Then, there exist $c, n_0 > 0$ s.th. $\forall\, n \geq n_0$, it holds that

$$\begin{aligned}
& T(n) \leq c \cdot n \\
\Leftrightarrow \quad & 2n\sqrt{n} \leq c \cdot n && \text{(divide by } n > 0\text{)} \\
\Leftrightarrow \quad & 2\sqrt{n} \leq c
\end{aligned}$$

which is a contradiction since $\sqrt{n}$ cannot be bounded above by a constant.

(e) To prove the asymptotic bound, we need to show that there are constants $n_0, c_1, c_2 > 0$ s.th. for all $n \geq n_0$ it holds that $T(n) \leq c_2 \cdot n^3$ and $T(n) \geq c_1 \cdot n^3$.

$$\begin{aligned}
T(n) &= 2^{3 \cdot \log_3(n)/\log_3(2)} \\
&= 2^{3 \cdot \log_2(n)} && \text{(change of logarithm basis)} \\
&= n^{3 \cdot \log_2(2)} && \text{(logarithm change of exponential basis)} \\
&= n^3
\end{aligned}$$

Therefore, for example for $n_0 = 1$ and $c_1 = c_2 = 1$ it holds that $T(n) \leq c_2 \cdot n^3$ and $T(n) \geq c_1 \cdot n^3$ and hence we have shown that $T(n) = \Theta(n^3)$.

Question 2: Recursive InsertionSort (21 points)

This exercise analyzes a recursive version of the `InsertionSort` algorithm. The algorithm follows the same idea as the iterative version of the algorithm we discussed in Lecture I of the course, but instead of looping through the array, it uses recursive function calls to sequentially sort the array.

Specifically, the algorithm proceeds as follows. Given an array A with n elements, the algorithm first recursively calls itself to sort the first $n-1$ elements of the array. Then, the algorithm moves the last element $A[n]$ to its correct position to yield a completely sorted array. Specifically, the algorithm inspects all elements ordered before $A[n]$ and moves them up one index until it finds the index where $A[n]$ should be inserted.

(a) (7 points) Based on the description of the algorithm above, provide pseudo-code for a routine called `RecInsertionSort`(A, n) that implements the recursive `InsertionSort` algorithm.

Note: When writing the pseudo-code, you should use one-based indexing for the array A i.e. the first element of A is $A[1]$ and the last element is $A[n]$.

(b) (7 points) Analyze the best-case and worst-case runtime of the `RecInsertionSort`(A, n) algorithm. For each of the two cases (best-case and worst-case):

(i) Clearly state which choice of A leads to this case

(ii) Provide a recurrence equation for the runtime in that case based on the pseudocode from (a)

(iii) Provide an upper bound $\mathcal{O}(f(n))$ on the runtime in that case, where the bound is as tight as possible.

(c) (7 points) Prove formally that the `RecInsertionSort`(A, n) algorithm is correct. In your proof, make sure you clearly state the recurrence invariant condition (inductive hypothesis).

Note: If you struggle with providing a formal proof for the inductive step, try first to carefully explain the intuition behind the proof, then try your best to make the proof as formal as possible.

SOLUTION:

(a) The following pseudo-code implements the recursive version of `InsertionSort` described in the exercise.

Algorithm: RECINSERTIONSORT(A, n)

```
1  // Handle the base case
2  if n ≤ 1 then
3      return
4  // Recursively sort first n − 1 elements of the array
5  RecInsertionSort(A, n − 1)
6  // Correctly place the n-th element in the sorted array
7  last = A[n]
8  j = n − 1
9  while j > 0 and A[j] > last do
10     A[j + 1] = A[j]
11     j = j − 1
12 A[j + 1] = last
13 return
```

(b) The runtime of the algorithm consists of the runtime spent on the recursive call in l.5 plus the runtime required to insert the last element and return (ll. 7-13).

Worst-case runtime. In the worst-case, the array is sorted in inverse order. Hence, the while loop traverses through $n-1$ elements with constant time required inside the loop to move elements, giving $\mathcal{O}(n)$ runtime for ll.7-13. The recursive call is on the first $n-1$ elements of the array and takes $T(n-1)$ time. Hence, the runtime recursion is

$$T(n) = T(n-1) + \mathcal{O}(n), \qquad \text{with } T(1) = \mathcal{O}(1)$$

This recursion can be solved directly by repeatedly substituting $T(k)$ for $k = n-1, n-2, \ldots, 1$

$$T(n) = T(n-1) + \mathcal{O}(n) = T(n-2) + 2 \cdot \mathcal{O}(n) = \cdots = n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$$

Best-case runtime. In the best case, the array is already sorted in ascending order and the while loop only checks the condition once, then exits, giving $\mathcal{O}(1)$ runtime, in addition to the runtime of the recursive call, $T(n-1)$.

$$T(n) = T(n-1) + \mathcal{O}(1) \qquad \text{with } T(1) = \mathcal{O}(1)$$

Solving the recurrence gives best-case runtime $n \cdot \mathcal{O}(1) = \mathcal{O}(n)$.

(c) We prove correctness of the algorithm by induction on the size of the array n

- **Recurrence invariant condition:** Assume the recursive call to RecInsertionSort(A, k) correctly sorts the elements of $A[1, \ldots, k]$ for $1 \leq k \leq n$.

- **Inductive basis:** We need to show that the inductive hypothesis holds for $k = 1$ i.e. RecInsertionSort$(A, 1)$ correctly sorts the elements of $A[1, \ldots, 1] = A[1]$. Since $A[1]$ only contains one element, it is trivially sorted by definition and the inductive hypothesis holds.

- **Inductive step:** Assume that the inductive hypothesis holds for recursive calls of size m where $1 \leq m < k \leq n$. We need to show that the inductive hypothesis holds for k i.e. that RecInsertionSort(A, k) correctly sorts the elements of $A[1, \ldots, k]$.

  By the inductive hypothesis, we know that the call to RecInsertionSort$(A, k-1)$ in l.5 correctly sorts the elements in $A[1, \ldots, k-1]$. Let $i^* \in [1, \ldots, k-1]$ be the largest position such that $A[i^*] \leq A[k]$. The while loop transforms the array $\{A[1], \ldots, A[i^*], \ldots, A[k-1], A[k]\}$ to $\{A[1], \ldots, A[i^*], A[k], A[i^*+1], \ldots, A[k-1]\}$. It remains to show that the latter array is sorted which follows from the following three observations.

  1. By definition of i^*, $\{A[i^*], A[k], A[i^*+1]\}$ is sorted.
  2. Further, by the inductive hypothesis, it holds that $A[i^*] \geq A[j]$ for all $j \leq i^*$ so that the array left of $A[k]$ is sorted
  3. Similarly, by the inductive hypothesis, $A[i^*+1] \leq A[j]$ for all $j \geq i^*+1$ so that the array right of $A[k]$ is sorted

  Hence, $\{A[1], \ldots, A[i^*], A[k], A[i^*+1], \ldots, A[k-1]\}$ is sorted.

- **Conclusion:** By induction, the outer call RecInsertionSort(A, n) correctly sorts the elements of $A[1, \ldots, n]$ which proves correctness of RecInsertionSort.

Question 3: Dynamic Programming (21 points)

Setup: A large company that runs a popular video streaming platform hires you as a data science consultant. The company wants to find out how to optimally place advertisements in its content to maximize its ad-revenue using a dynamic programming algorithm. Specifically, given a video of length n minutes, the company wants to decide at which minutes of the video to place an ad. In their initial briefing, they inform you of the following assumptions.

- For technical reasons, ads can only be placed at every full minute i.e. at $i = 1, 2, \ldots, n$.
- While from a technical standpoint it would be possible to place an ad at every minute $i = 1, \ldots, n$ of the video, market research reveals that ads should not be placed within k minutes of each other, as this would lead the viewers to be frustrated and stop using the platform immediately. That is, if you place an ad at minute j of the video, the previous ad can be placed latest at minute $j - k - 1$. You may assume that $0 < k < n - 2$.
- The advertising department of the company has negotiated a list of prices to charge their advertising partners, denoted by $P = [p_1, \ldots, p_n]$ where p_i is the price the advertising partner pays to the company for placing an ad at minute $i = 1, \ldots, n$.

Optimal substructure: To help you get started, the first three exercises guide you through deriving the optimal substructure of the problem. We will focus on the problem where in a video of length i, you have already optimally placed ads throughout minutes $j = 1, \ldots, i - 1$ and are currently deciding whether to place an ad at minute i. In this problem, you face the following two choices:

A. Do not place an ad at minute i

B. Place an ad at minute i and potentially remove any ads in the previous k minutes.

The first three exercises below ask you to find an expression for the revenue r_i from optimally placing ads in a video of length i, taking into account these two choices. The remaining exercises then ask you to write and analyze the algorithm.

(a) (4 points) Assume $i > k$. For both choices A and B, write down an expression for the revenue r_i from optimally placing ads in a video of length i.

Note: Your answers should be in a recursive form i.e. depend on some r_j where $j \leq i$.

(b) (3 points) Assume $0 < i \leq k$. Think about whether and how you would need to modify your answer to (a) in this case. Again, for both decisions A and B, write down an expression for the revenue r_i from optimally placing ads in a video of length i.

(c) (4 points) Based on your answers to (a) and (b), provide a general recursion for the revenue from optimally placing ads in a video of length i.

Note: Make sure that in your answer you clearly distinguish between the different cases for i and do not forget to specify the base case.

(d) (7 points) Based on the optimal-revenue recursion from (c), provide pseudo-code for a bottom-up dynamic programming routine called PlaceAds(n, P, k) that returns the revenue from optimally placing ads in a video of length n, given the price array P and a value for k with $0 < k < n - 2$.

(e) (3 points) Analyze the runtime of your dynamic programming algorithm based on the pseudo-code. Your final result should be in the form of an upper bound $\mathcal{O}(f(n))$ for tightest possible function $f(n)$.

SOLUTION:

(a) In case A, we do not place an ad at minute i and hence our revenue from optimally placing ads is $r_i = r_{i-1}$. In case B, we place an ad at minute i and potentially remove all ads within k minutes, giving $r_i = r_{i-k-1} + P[i]$.

(b) The revenue for decision A has not changed i.e. it still holds that $r_i = r_{i-1}$. For decision B, we now have that the revenue from optimally placing ads is $r_i = P[i]$.

(c) The optimal decision maximizes the revenue from optimally placing ads with choices A and B. Hence, we have the general recursion

$$r_i = \begin{cases} 0 & \text{if } i = 0 \\ \max\{r_{i-1}, P[i]\} & \text{if } 0 < i \leq k \\ \max\{r_{i-1}, r_{i-k-1} + P[i]\} & \text{if } i > k \end{cases}$$

(d) The pseudo-code for the bottom-up dynamic programming algorithm is as follows.

Algorithm: PlaceAds(n, P, k)

```
1 Let r = [0, ..., n] be a new array
2 r[0] = 0
3 for i = 1 to n do
4     if i ≤ k then
5         r[i] = max(r[i − 1], P[i])
6     else
7         r[i] = max(r[i − 1], r[i − k − 1] + P[i])
8 return r[n]
```

(e) To analyze the runtime, we inspect the pseudo-code and derive the runtime of each block of the algorithm. The set-up and return in ll.1-2 and l.8, respectively, each have runtime $\mathcal{O}(1)$. The for-loop has $\Theta(n)$ iterations while performing $\mathcal{O}(1)$ computations in ll. 4-7. Hence, the total runtime of the dynamic programming algorithm is

$$T(n) = \mathcal{O}(1) + \mathcal{O}(n) = \mathcal{O}(n)$$

that is the algorithm has linear runtime.

Question 4: Short statements (15 points)

For each of the following statements, indicate whether the statement is correct or incorrect. If it is correct, give a brief motivation ($\approx$ 2-3 sentences) of why it is correct. If it is incorrect, provide a brief motivation ($\approx$ 2-3 sentences) of why it is incorrect.

(a) (3 points) The master theorem can be applied to the recursion $T(n) = 0.5 \cdot T(n/2) + \sqrt{n}$.

SOLUTION: `Incorrect.` If we try to write the recurrence in standard recurrence form, we get $a = 0.5, b = 2$ and $d = 0.5$. However, the master theorem only applies if $a \geq 1, b > 1$ and $d \geq 0$ (slide 11, lecture III). Hence, the master theorem can not be applied to this recursion.

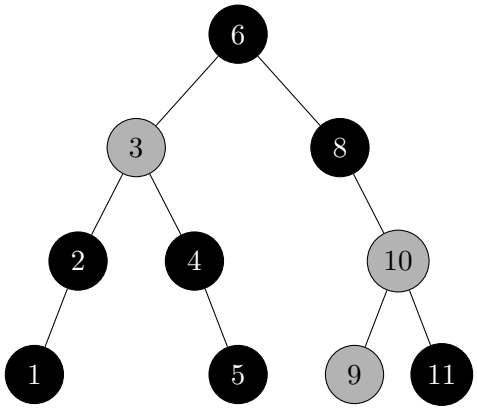
(b) (3 points) The expected runtime of a randomized algorithm is generally not the same as the runtime of the algorithm for an expected input.

SOLUTION: `Correct.` The expected runtime takes expectation over the distribution of choices the algorithm makes, not expectation over the distribution of possible inputs (slide 5, lecture V). Hence, generally the expected runtime is not the runtime for an expected input.

(c) (3 points) With universal hashing, we choose the shape of the hash function completely at random.

SOLUTION: `Incorrect.` With universal hashing, we randomly choose a hash function from a pre-defined set of hash functions called the universal hash family. Hence, the shape of the function is not completely random, but one of the previously defined shapes (slide 25, lecture VI). In fact, in the example we discussed in the lecture (slide 28, lecture VI), the shape of the function was the same for all hash functions in the set, only the coefficients of the hash function were chosen at random.

(d) (3 points) The following binary search tree is a valid red-black tree. [Red nodes are printed in light gray while black nodes are printed in black.]



SOLUTION: `Incorrect.` The tree above violates the red-black properties (slide 23, lecture VII). Specifically, it violates property #4 (if a node is red, then both its children are black) for node 10. In addition, it violates property #5 (for each node, all simple paths from the node to descendant leaves contain the same number of black nodes) since the path $6 \to 9$ has less black nodes than the other paths.

(e) (3 points) The adjacency-matrix and adjacency-list representations of a graph G have the same space requirements when storing a completely dense graph in memory.

SOLUTION: `Correct.` The space requirements to store a graph G are $\mathcal{O}(n^2)$ for an adjacency-matrix representation and $\mathcal{O}(n+m)$ for an adjacency-list representation (slide 11, lecture VIII). For a completely dense graph $m = n^2$ (slide 6, lecture VIII). Hence, both representation have a $\mathcal{O}(n^2)$ space requirement.