Exercise 1: Master Theorem for Recurrences

Use the Master Theorem for recurrences, to fill the following table. That is, in each cell write \( \Theta(g(n)) \), such that \( T(n) \in \Theta(g(n)) \) for the given parameters \( a, b, f(n) \). Assume \( T(1) \in \Theta(1) \). Additionally, in each cell note the case you used (1st, 2nd or 3rd by the order given in the lecture). We filled out one cell as an example.

\[
T(n) = aT\left(\frac{n}{b}\right) + f(n) \\
\begin{array}{ccc}
T(n) = aT\left(\frac{n}{b}\right) + f(n) & a = 16, b = 2 & a = 1, b = 2 & a = b = 3 \\
\hline
f(n) = 1 & \Theta(n^4), 1st \\
f(n) = n & \Theta(n^4), 1st \\
f(n) = n^4 & \Theta(n^4), 1st \\
\end{array}
\]

Sample Solution

In the lecture we learned to classify \( T(n) = aT\left(\frac{n}{b}\right) + f(n) \) according to the following three cases

\[
T(n) \in \begin{cases} 
\Theta\left(n^{\log_b a}\right), & \text{if } f(n) \in \mathcal{O}\left(n^{\log_b a - \varepsilon}\right) \text{ for some } \varepsilon > 0 \\
\Theta\left(n^{\log_b a \log n}\right), & \text{if } f(n) \in \Theta(n^{\log_b a}) \\
\Theta(f(n)), & \text{if } f(n) \in \Omega\left(n^{\log_b a + \varepsilon}\right) \text{ for some } \varepsilon > 0.
\end{cases}
\]

\[
T(n) = aT\left(\frac{n}{b}\right) + f(n) \\
\begin{array}{ccc}
T(n) = aT\left(\frac{n}{b}\right) + f(n) & a = 16, b = 2 & a = 1, b = 2 & a = b = 3 \\
\hline
f(n) = 1 & \Theta(n^4), 1st & \Theta(\log n), 2nd & \Theta(n), 1st \\
f(n) = n & \Theta(n^4), 1st & \Theta(\log n), 2nd & \Theta(n), 1st \\
f(n) = n^4 & \Theta(n^4 \log n), 2nd & \Theta(n^4), 3rd & \Theta(n^4), 3rd \\
\end{array}
\]

Exercise 2: Peak Element

You are given an array \( A[1 \ldots n] \) of \( n \) integers and the goal is to find a peak element, which is defined as an element in \( A \) that is equal to or bigger than its direct neighbors in the array. Formally, \( A[i] \) is a peak element if \( A[i - 1] \leq A[i] \geq A[i + 1] \). To simplify the definition of peak elements on the rims of \( A \), we introduce sentinel-elements \( A[0] = A[n+1] = -\infty \).

(a) Give an algorithm with runtime \( \mathcal{O}(\log n) \) which returns the position \( i \) of a peak element.

(b) Prove that your algorithm always returns a peak element, give a recurrence relation for the runtime and use it to prove the runtime \( \mathcal{O}(\log n) \).
Sample Solution

(a) Algorithm 1 Peak-Element(A, ℓ, r)

```
if ℓ = r then return A[ℓ]  ▷ base case
m ← ⌊(ℓ + r) / 2⌋
if A[m] ≤ A[m+1] then
    return Peak-Element(A, m + 1, r)
else if A[m] ≤ A[m−1] then
    return Peak-Element(A, ℓ, m − 1)
else return A[m]  ▷ peak element found
```

A call of Peak-Element(A, 1, n) returns a peak element in A.

(b) We show the invariant that during each call of Peak-Element(A, ℓ, r), we have A[ℓ−1] ≤ A[ℓ] and A[r] ≥ A[r+1]. Since A[0], A[n+1] = −∞, this is obviously true for Peak-Element(A, 1, n).

During sub-calls of Peak-Element(A, ℓ, r) this condition is maintained by the If-conditions and the recursive calls and the appropriate sub-array. This implies that we have found a peak element when ℓ = r (at the latest, but we may find one earlier).

During every recursive step, the considered sub-array is at most half the size of the previous one, thus the algorithm terminates eventually. Additionally, in each recurse step we make at most one recursive sub-call. Furthermore, in each recursive step we read at most 5 array entries. Thus we have T(n) ≤ T(n/2) + O(1), which solves to T(n) ∈ O(log n) using the Master Theorem.

Exercise 3: Binary search

(a) Provide the pseudocode of an algorithm BinarySearch implementing the following informal algorithm description. The input is a sorted array A[0..n−1] of keys and a search key k. If there is an index i with A[i] = k, the algorithm returns i, else false.

The algorithm first divides the array at some index m which is in the “middle”. If A[m] > k we start the algorithm recursively on the left sub-array. If A[m] < k we start the algorithm recursively on the right sub-array. Else we have A[m] = k and return m.

(b) Give a recurrence relation for the runtime of BinarySearch and show it has runtime O(log n).

(c) For the data structure “Hierarchy of Arrays” of Exercise Sheet 4, describe an operation Search(k) that takes at most O((log n)^2) time and returns the array number i of an array A_i and an index j such that A_i[j] = k, or false if such a pair i, j can not be found. Explain the runtime.

Sample Solution

(a) Consider the following pseudo code. A call of BinarySearch(A, 0, n−1, k) implements the rough algorithm description given above.

```
Algorithm 2 BinarySearch(A, i, j, k)

if i > j then return false  ▷ k not in A
m ← ⌊i+j⌋ / 2 ▷ determine “middle”
if A[m] > k then return BinarySearch(A, i, m−1, k)  ▷ continue left of “middle”
else if A[m] < k then return BinarySearch(A, m+1, j, k)  ▷ continue right of “middle”
else return m  ▷ found A[m] = k
```

(b) The recurrence relation is

\[ T(n) \leq T\left(\frac{n}{2}\right) + O(1) \text{ with } T(1) = O(1). \]

Due to the Master Theorem (Case 2) the runtime is O(log n).
The operation \textsc{Search}(k) simply loops over the arrays \(A_0, \ldots, A_m\), where \(m \in \mathcal{O}(\log n)\) is the index of the last non-empty array. If the current array \(A_i\) is non-empty, it performs \textsc{Binary-Search}(\(A_i, 0, 2^i - 1, k\)). Using part (b) we know that the total runtime for that is

\[
\mathcal{O}\left( \sum_{i=0}^{m} \log(2^i) \right) = \mathcal{O}\left( \sum_{i=0}^{m} i \right) = \mathcal{O}\left( \frac{m(m+1)}{2} \right) = \mathcal{O}(\log^2 n).
\]