ADS Tutorial 1 - Solutions[*]

Instructor: Aris Filos-Ratsikas TA: Kat Molinet

October 7, 2024

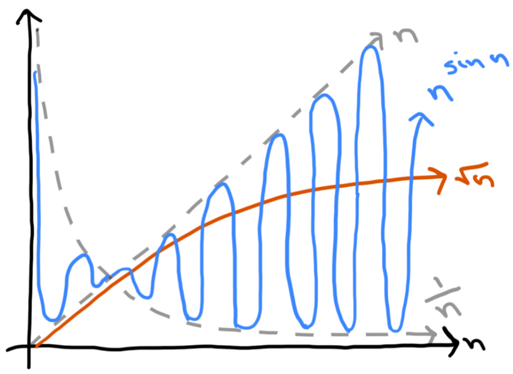
Problem 1

Work out (but don't bother proving) for each pair of expressions below, whether A is $O(B)$, $\Omega(B)$, $\Theta(B)$ (it could be none of these). Assume $k \geq 1, \epsilon > 0, c > 1$ are all constants.

A	B	O	Ω	Θ
$\lg^k n$	n^ϵ	X		
n^k	c^n	X		
$\sqrt{n}$	$n^{\sin n}$			
2^n	$2^{n/2}$		X	
$n^{\ln m}$	$m^{\ln n}$	X	X	X
$\lg(n!)$	$\lg(n^n)$	X	X	X

Solution

- **The 1st and 2nd:** The first two are fairly straightforward to see. Exponential growth dominates polynomial growth, which in turn dominates logarithmic growth. Thus, $\lg^k n = O(n^\epsilon)$ and $n^k = O(c^n)$.
- **The 3rd:** The answer is that $\sqrt{n}$ is neither $O(n^{\sin n})$ nor $\Omega(n^{\sin n})$. This is because the function $n^{\sin n}$ oscillates around the function $\sqrt{n}$ as n tends to infinity. In other words, no matter how big n_0 is, there are always infinitely many $n > n_0$ such that $n^{\sin n} > \sqrt{n}$, just as there are infinitely many $n > n_0$ such that $n^{\sin n} < \sqrt{n}$. To see this explicitly, we can try graphing the two functions. The square root function is simple to graph. For $n^{\sin n}$, we observe that, since $\sin n$ continuously oscillates between -1 and 1 for all n, the function $n^{\sin n}$ continuously oscillates between $n^{-1} = 1/n$ and $n^1 = n$ for all n. We show this below:



[*]The solutions contain additional explanations that are not necessary, if you were to answer such a question in an exam.

- **The 4th:** For this question, $2^n = \Omega(2^{n/2})$. While in asymptotic notation, constants don't matter as scalar multiples of the original function, they very much matter as exponents. For instance, the exponents of n and n^2 only differ by a constant factor of 2; but as we know, there is a big difference between between linear and quadratic runtimes. (By contrast, algorithms with respective runtimes n and $2n$ can be treated equivalently in asymptotic notation.) The same principle is true in this example. The exponents of 2^n and $2^{n/2}$ differ by a factor of two, and so the two functions are not asymptotically equivalent – 2^n dominates $2^{n/2}$. Another way of seeing this:

$$2^{n/2} = (2^n)^{1/2} = \sqrt{2^n},$$

which is clearly asymptotically dominated by 2^n.

- **The 5th:** This question is actually simpler than it seems at first. The way to generalise the single-variable definition of O-notation to the multiple-variable case is quite straightforward; but we don't even need to go that far in this example. Instead, we note that $n^{\ln m} = m^{\ln n}$ for $m, n \geq 1$. To see this, we simply note that $\ln n \cdot \ln m = \ln m \cdot \ln n$ which, by the logarithmic power rule, implies $\ln(m^{\ln n}) = \ln(n^{\ln m})$. Exponentiating both sides gives the desired result, $m^{\ln n} = n^{\ln m}$. Since the two functions are in fact equal, they are, of course, also asymptotically equivalent.

- **The last:** This question is an excellent illustration of some of the nuances involved in scalars in exponents versus as multiples of the original function. Our first observation is that $n^{n/2} < n! < n^n$ for $n \geq 2$. [This can be seen as a result of *Stirling's approximation*. You can also derive this inequality yourself by expanding the expression for $n!$. That $n! \leq n^n$ is obvious. To see that $n^{n/2} \leq n!$, observe that in the expansion of $n!$, there are at least $n/2$ terms with size at least $n/2$.] As a result, n^n *strictly dominates* $n!$; i.e., $n! = o(n^n)$. But consider what happens when we take the log: we now have

$$\log(n^{n/2}) < \log(n!) < \log(n^n),$$

which, by our log rules, can be simplified as

$$\iff \tfrac{n}{2}\log(n) < \log(n!) < n\log(n)$$

The constant factor of $1/2$ that was previously in the exponent now becomes a scalar multiple of the entire function. The inequality above now implies that $\log(n!)$ is, by definition, $\Theta(n\log(n)) = \Theta(\log(n^n))$.

Problem 2

Solve the following recursive formulas without using the Master Theorem.

A. $T(n) \leq T(n/2) + 4$

B. $T(n) \leq T(n/2) + 5n$

C. $T(n) \leq T(n/2) + 3n^2$

D. $T(n) \leq \frac{3}{2}T(n/2) + 1$

Sort the obtained formulas in terms of their asymptotic order.

Solution

We can first solve the recurrence by expanding the inequalities, in order to figure out what the solution would be, and then we can argue using induction. Note that in these solutions, we assume that the runtime of the algorithm on an input of size 1 is 0; i.e., $T(1) = 0$. This makes sense in the context of sorting algorithms – after all, a list of size 1 doesn't need to be sorted – but in general, $T(1)$ might be any constant. Assuming $T(1) > 0$ might change the calculations below slightly, but the general approach is still exactly the same.

A. $T(n) \leq T(n/2) + 4$
The first step is to try and expand the formula by repeatedly substituting into the inequality we're given. The point of this step is to be able to "detect the pattern" and write an expression for $T(n)$ as a function only of n rather than as a recurrence relation.

$$\begin{aligned}
T(n) &\leq T(^n/_2) + 4 \\
&\leq [T((^n/_2)/2) + 4] + 4 \\
&= T(^n/_{2^2}) + 4 \cdot 2 \\
&\vdots \\
&\leq [T((^n/_{2^{i-1}})/2) + 4] + 4(i-1) \\
&= T(^n/_{2^i}) + 4i \\
&\vdots \\
&\leq [T((^n/_{2^{(\log n)-1}})/2) + 4] + 4((\log n) - 1) \\
&= T(^n/_{2^{\log n}}) + 4\log n \\
&= T(^n/_n) + 4\log n \\
&= T(1) + 4\log n \\
&= 4\log n
\end{aligned}$$

We will therefore use induction to prove that $T(n) \leq 4\log n$.

Base Case: From our original recurrence relation (and the assumption that $T(1) = 0$), we know that $T(2) \leq T(1) + 4 = 4$. Our general formula implies that $T(2) \leq 4\log 2 = 4$. Thus, the base case holds.

Induction Step: Assume that our formula holds for case $^n/_2$; i.e., $T(n/2) \leq 4\log(n/2)$. Then we have that

$$\begin{aligned}
T(n) &\leq T(n/2) + 4 && \text{(By the original recurrence relation)} \\
&\leq 4\log(n/2) + 4 && \text{(Applying the induction hypothesis)} \\
&= 4\log n && \text{(Algebraic simplification using the fact that } 4 = 4\log(2))
\end{aligned}$$

Thus, we have shown that $T(n) \leq 4\log n$ for all $n \geq 2$.

B. $T(n) \leq T(n/2) + 5n$
The general format here is exactly the same as in part A. First, we expand the recurrence relation by repeated substitution to try to find the "general pattern" and express it mathematically. Once we've done this, we can prove that our general formula is correct via mathematical induction.

$$
\begin{aligned}
T(n) &\le T(n/2) + 5n \\
&\le [T((n/2)/2) + 5(n/2)] + 5n \\
&= T(n/2^2) + 5n(1 + 1/2) \\
&\le [T((n/2^2)/2) + 5(n/2^2)] + 5n(1 + 1/2) \\
&= T(n/2^3) + 5n(1 + 1/2 + (1/2)^2) \\
&\vdots \\
&\le [T((n/2^{i-1})/2) + 5(n/2^{i-1})] + 5n(1 + 1/2 + (1/2)^2 + \cdots + (1/2)^{i-2}) \\
&= T(n/2^i) + 5n \sum_{j=0}^{i-1} \frac{1}{2^j} \\
&\vdots \\
&\le [T((n/2^{(\log n)-1})/2) + 5(n/2^{(\log n)-1})] + 5n \sum_{j=0}^{(\log n)-2} \frac{1}{2^j} \\
&= T(n/2^{\log n}) + 5n \sum_{j=0}^{(\log n)-1} \frac{1}{2^j} \\
&= T(1) + 5n \sum_{j=0}^{(\log n)-1} \frac{1}{2^j} \\
&= 5n \sum_{j=0}^{(\log n)-1} \frac{1}{2^j} \\
&\le 5n \lim_{k\to\infty} \sum_{j=0}^{k} \frac{1}{2^j} \\
&= 5n \cdot \frac{1}{1 - 1/2} \\
&= 10n
\end{aligned}
$$

Note: The third-to-last step, in which we take the limit of the geometric sequence, is optinal. While it does make the upper bound on $T(n)$ slightly* less tight, it makes the maths easier to work with. (*Although for large n, the tail of the geometric series should be small enough that we're really not losing much here.)

We will now show by induction that $T(n) \le 10n$.

Base Case: From our original recurrence relation (and the assumption that $T(1) = 0$), we know that $T(2) \le T(1) + 5(2) = 10$. Our general formula implies that $T(2) \le 10(n) = 20$. If the first statement is true, then the second statement is certainly true, as well. Thus, the base case holds.

Induction Step: Assume that our formula holds for case $n/2$; i.e., $T(n/2) \le 10(n/2) = 5n$. Then, we have that

$$
\begin{aligned}
T(n) &\le T(n/2) + 5n && \text{(By the original recurrence relation)} \\
&\le 5n + 5n && \text{(Applying the induction hypothesis)} \\
&= 10n && \text{(Algebraic simplification)}
\end{aligned}
$$

Thus, we have shown that $T(n) \le 10n$ for all $n \ge 2$.

C. $T(n) \leq T(n/2) + 3n^2$
The "expansion" step looks almost identical to parts A and B, so we omit the full explanation. As a spot check, the "general formula" in terms of i should look something like the following:

$$T(n) \leq T({}^{n}\!/\!{}_{2^i}) + 3n^2 \sum_{j=0}^{i-1} \frac{1}{4^j}$$

When $i = \log n$, this simplifies to

$$\begin{aligned} T(n) &\leq T({}^{n}\!/\!{}_{2^{\log n}}) + 3n^2 \sum_{j=0}^{(\log n)-1} \frac{1}{4^j} \\ &= T(1) + 3n^2 \sum_{j=0}^{(\log n)-1} \frac{1}{4^j} \\ &= 3n^2 \sum_{j=0}^{(\log n)-1} \frac{1}{4^j}, \end{aligned}$$

assuming $T(1) = 0$. Upper-bounding this sum by its geometric series as we do in part B gives $T(n) \leq 3n^2 \cdot \frac{4}{3} = 4n^2$.

Base Case: From our original recurrence relation (and the assumption that $T(1) = 0$), we know that $T(2) \leq T(1) + 3(2)^2 = 12$. Our general formula implies that $T(2) \leq 4(2)^2 = 16$. If the first statement is true, then the second statement is certainly true, as well. Thus, the base case holds.

Induction Step: Assume that our formula holds for case ${}^{n}\!/\!{}_{2}$; i.e., $T(n/2) \leq 4(n/2)^2 = n^2$. Then we have that

$$\begin{aligned} T(n) &\leq T(n/2) + 3n^2 && \text{(By the original recurrence relation)} \\ &\leq n^2 + 3n^2 && \text{(Applying the induction hypothesis)} \\ &= 4n^2 && \text{(Algebraic simplification)} \end{aligned}$$

Thus, we have shown that $T(n) \leq 4n^2$ for all $n \geq 2$.

D. $T(n) \leq \frac{3}{2}T(n/2) + 1$
As in part C, we omit the full expansion; but the process is exactly the same as in parts A and B. As a spot check, the "general formula" in terms of i should look something like the following:

$$\begin{aligned} T(n) &\leq ({}^{3}\!/\!{}_{2})^{i-1}[{}^{3}\!/\!{}_{2}T({}^{n}\!/\!{}_{2^{i-1}}/2) + 1] + (1 + {}^{3}\!/\!{}_{2} + \cdots + ({}^{3}\!/\!{}_{2})^{i-2}) \\ &= ({}^{3}\!/\!{}_{2})^{i}T({}^{n}\!/\!{}_{2^i}) + \sum_{j=0}^{i-1} ({}^{3}\!/\!{}_{2})^{j} \end{aligned}$$

When $i = \log n$, this becomes

$$\begin{aligned}
T(n) &\le (3/2)^{\log n} T(n/2^{\log n}) + \sum_{j=0}^{(\log n)-1} (3/2)^j \\
&= (3/2)^{\log n} T(1) + \sum_{j=0}^{(\log n)-1} (3/2)^j \\
&= \sum_{j=0}^{(\log n)-1} (3/2)^j && \text{(Assuming } T(1) = 0\text{)} \\
&= \frac{(3/2)^{\log n} - 1}{3/2 - 1} && \text{(Applying the geometric sum formula)} \\
&= 2\big((3/2)^{\log n} - 1\big) \\
&= 2n^{\log 3/2} - 2 && \text{(Using the equality } n^{\log m} = m^{\log n}\text{; see Problem 1.5)}
\end{aligned}$$

Note that unlike in parts B and C, we can't upper bound the geometric sum by its geometric series, since $|3/2| > 1$ and hence the series diverges. Instead, we use the general geometric sum formula.

Base Case: From our original recurrence relation (and the assumption that $T(1) = 0$), we know that $T(2) \le 3/2\, T(1) + 1 = 1$. Our general formula implies that $T(2) \le 2(2)^{\log 3/2} - 2 = 1$. Thus, the base case holds.

Induction Step: Assume that our formula holds for case $n/2$; i.e., $T(n/2) \le 2(n/2)^{\log 3/2} - 2 = 2/3\, n^{\log 3/2} - 2$. Then we have that

$$\begin{aligned}
T(n) &\le 3/2\, T(n/2) + 1 && \text{(By the original recurrence relation)} \\
&\le 3/2\, (2/3\, n^{\log 3/2} - 2) + 1 && \text{(Applying the induction hypothesis)} \\
&= 2n^{\log 3/2} - 2 && \text{(Algebraic simplification)}
\end{aligned}$$

Thus, we have shown that $T(n) \le 2n^{\log 3/2} - 2$ for all $n \ge 2$.

The asymptotic order of these formulas is $O(\log n), O(n^{\log(3/2)}), O(n), O(n^2)$.

Problem 3

Provide a (as-tight-as possible) bound for the asymptotic memory requirements of the following algorithms.

A. The MergeSort algorithm.

B. The QuickSort algorithm where we take the element $A[n]$ as the pivot.

Solution

A. While there are many ways one could implement MergeSort, each with slightly different patters of memory usage, we assume here that the "divide" step of the sorting occurs *in-place*; i.e., elements are manipulated within our original array, rather than copy/pasted into a new array at each step of the recursion. What would the pattern of memory allocation look like, exactly?

Well first of all, we know that we start with an array of size n. At each step, we split the array into two segments of size $\sim n/2$, and then recurse on each of these two segments. Since we assume that all array operations occur in-place, the only additional memory requirement needed to split the array into two subsegments is two additional pointers: one pointer for each of the two subsegments to mark their respective centre points. So each step of the "divide" stage simply adds +2 variables to memory.

To merge the sublists after the divide stage into a sorted list, we allow ourselves an additional empty array of size n for intermediate copy/paste operations. (While there are technically ways of implementing MergeSort fully in place, these methods can be quite complex.) The exact implementation isn't very important here; what matters is that we can easily perform the "merge" step using only an additional $O(n)$ memory.

All together, we have, for the "divide" stage, the memory requirement recurrence $M(n) \leq 2M(n/2)+2$. By the Master Theorem with $\alpha = 2, b = 2$ and $d = 0$, this implies $M(n) = O(n^{\log_2 2}) = O(n)$. The reassembling "merge" stage might take another $O(n)$ memory; but since the same additional $O(n)$ array can be reused throughout the entire execution of the algorithm, we simply sum the memory requirements of the two stages of the algorithm to get an overall requirement of $O(n) + O(n) \sim O(n)$ memory.

B. For the QuickSort algorithm, the partitioning requires $O(1)$ memory and for each iteration, as we only need constant memory to store variables, counters etc. The number of iterations depends on which pivot element we use, but in the worst case the array will be split into subarrays of length $n-1$ and 0. The recurrence then becomes $M(n) \leq M(n-1)+c$. This can be easily seen to evaluate to $M(n) = O(n)$.

Note that unlike MergeSort, implementaions of QuickSort are often fully in-place. Thus, we have no additional memory requirements on top of those needed for the recursion, making the overall memory requirement of QuickSort simply $O(n)$.

Problem 4

A *majority element* in an array of n numbers is one that appears more than $\lceil n/2 \rceil$ times. Design an algorithm that receives as input a *sorted* array A of integers and outputs YES if a majority element exists and NO otherwise. Present the algorithm in terms of pseudocode. The algorithm should run in (worst-case) time $O(\log n)$ and you should formally prove its asymptotic running time. For simplicity, you may ignore issues regarding whether numbers are divisible by 2 (the algorithms can be adjusted to account for that via the appropriate use of the $\lceil \cdot \rceil$ function).

Solution

The key observation here is that any majority element will always be the median (let's call it M) of the list, since any continuous segment of lengh $> \lceil n/2 \rceil$ must cross the centre point of the list. Now that we know what of the value of any majority element must be, the task is simply to find the start and endpoints of the sequence of M's to compute its length. For this, we turn to a technique called "binary search."

Binary search is a method for finding the index of an element within a sorted list about which you have no additional information (besides its length). The idea goes roughly as follows: Since you don't know anything about the contents of the list, you have no idea where your element in question would fall within it. A naïve approach to finding the desired element would be to start at the beginning of the list, and iterate through each element sequentially one-by-one. But this isn't a very good apporach – for instance, what if your element happens to be the last one in the list? By starting at the beginning of the list and iterating step-by-step, we're blinding ourselves to the contents of the remainder of the list. By contrast, if we had more information about the overall structure and distribution of elements within the list, we could use this information to quickly narrow down our search space.

With this intuition in mind, BinarySearch works as follows: For a list A of length n, we start at the centre element $n/2$ and compare this value to the element in question (say, m). Since the list is sorted by assumption, we know that if $m < A[n/2]$, then the element m must lie to the left of index $n/2$. Similarly, $m > A[n/2]$ means that we can effectively ignore the left-hand side of the list and limit our search for m to the right-hand side of the list. Once we know which half of the list contains m, we simply recurse on that half, and so on, until we eventually narrow down our search space to a single element. [For more on BinarySearch, see the subsection on "Sublinear time" in Kleinberg and Tardos (KT) Chapter 2.4.]

The pseudocode of a BinarySearch-based majority element algorithm might look something like the following. Note that the algorithm uses two binary search procedures: one to find the *first* occurrence of x in the array, and one to find the *last* occurrence.

Algorithm 1 Majority in Sorted Array

```
1: procedure BinarySearchLeft($x, i, j$)
2:     if $i = j$ then return $i$;
3:     else
4:         if $x = A\left[\frac{i+j}{2}\right]$ then BinarySearchLeft$\left(x, i, \frac{i+j}{2}\right)$
5:         else BinarySearchLeft$\left(x, \frac{i+j}{2} + 1, j\right)$
6: procedure BinarySearchRight($x, i, j$)
7:     if $i = j$ then return $i$;
8:     else
9:         if $x = A\left[\frac{i+j}{2}\right]$ then BinarySearchRight$\left(x, \frac{i+j}{2}, j\right)$
10:        else BinarySearchRight$\left(x, i, \frac{i+j}{2} - 1\right)$
11: procedure Majority($A$)
12:     if BinarySearchRight$\left(A\left[\frac{n}{2}\right], \frac{n}{2}, n\right)$-BinarySearchLeft$\left(A\left[\frac{n}{2}\right], 1, \frac{n}{2} - 1\right) + 1 > \frac{n}{2}$ then
13:         Return YES;
14:     else Return NO;
```

It's straightforward to show that BinarySearch runs in time $O(\log n)$. Since the remaining operations of the algorithm above run in constant time, the overall runtime of our majority element algorithm is also $O(\log n)$. We provide the formal analysis below.

BinarySearchLeft and BinarySearchRight have the same worst-case running time, so we will only perform the analysis for one of them. In each recursive call of the algorithm, there is a constant number of operations (e.g., checking if two elements are equal or returning an element), so the asymptotic complexity will be given by the number of times that the procedure is called. Also note that in each recursive call, the size of the input to the procedure is halved. More precisely, if $T(n)$ is the running time of the procedure on input size n, we can write

$$T(n) = T(n/2) + c,$$

where c is a constant number for the remaining operations. This gives us a recursive equation, which we can solve to obtain the value of $T(n)$. We will proceed by induction.

To be proven: $T(n) \leq 2c \log n$

Base Case: $n = 2$: Straightforward, $T(2) \leq 2c \leq 2c \log 2$.
Induction Hypothesis: Suppose $T(n/2) \leq 2c \log(n/2)$.
Inductive Step: We have that

$$\begin{aligned} T(n) &= T(n/2) + c \\ &\leq 2c \log(n/2) + c \qquad (1) \\ &\leq 2c \log n + c - c \qquad (2) \\ &= 2c \log n \end{aligned}$$

where Inequality 1 follows from the Induction Hypothesis and Inequality 2 follows from the fact that $\log(n/2) = \log(n) - \log 2$. Therefore, the two BinarySearch procedures together take time at most $4c \log n$

and the Majority algorithm also makes an additional subtraction and comparison, which only take constant time. Therefore, the running time of the algorithm is $O(\log n)$.