1 Master Theorem

For solving recurrence relations asymptotically, it often helps to use the Master Theorem:

**Master Theorem.** If \( T(n) = aT(n/b) + O(n^d) \) for \( a > 0, b > 1, \) and \( d \geq 0, \) then

\[
T(n) = \begin{cases} 
O(n^d) & \text{if } d > \log_b a \\
O(n^d \log n) & \text{if } d = \log_b a \\
O(n^{\log_b a}) & \text{if } d < \log_b a
\end{cases}
\]

**Note:** You can replace \( O \) with \( \Theta \) and you get an alternate (but still true) version of the Master theorem that produces \( \Theta \) bounds.

\( d_{\text{crit}} = \log_b a \) is called the **critical exponent**. Notice that whichever of \( d_{\text{crit}} \) and \( d \) is greater determines the growth of \( T(n) \), except in the case where they are perfectly balanced.

**Solution:** As all things should be.

Solve the following recurrence relations and give an \( O \) bound for each of them.

(a)  
(i) \( T(n) = 3T(n/4) + 4n \)

**Solution:** Since \( \log_4 3 < 1 \), by the Master Theorem, \( T(n) = O(n) \).

(ii) \( T(n) = 45T(n/3) + n^3 \)

**Solution:** Since \( \log_3 45 > 3 \), by the Master Theorem, \( T(n) = O(n^{\log_3 45}) \).

(b) \( T(n) = 2T(\sqrt{n}) + 3, \) and \( T(2) = 3 \).

**Hint:** Follow the recursion tree or try to perform a substitution.

**Solution:** *Solution 1:* A priori, this problem does not immediately look like it satisfies the Master theorem because the the recurrence relation is not a map \( n \rightarrow n/b \). However, we notice that we may be able to transform the recurrence relation into one about another variable such that it satisfies the Master Theorem. Let \( k \) be the solution to

\[
n = 2^k.
\]

Then we can rewrite our recurrence relation as

\[
T(2^k) = 2T(2^{k/2}) + 3.
\]

Now, if we let \( S(k) = T(2^k) \), then the recurrence relation becomes something more manageable:

\[
S(k) = 2S(k/2) + 3.
\]

By the Master theorem, this has a solution of \( \Theta(k) \). Since \( n = 2^k \), then \( k = \log n \) and therefore \( \Theta(k) = \Theta(\log n) \).

The intuition between the transformation between \( n \) and \( k \) is that \( n \) could be a number and \( k \), the number of bits required to represent \( n \). The recurrence relation is easier expressed in terms of the number of bits instead of the actual numbers.
Solution 2: The recursion tree is a full binary tree of height \( h \), where \( h \) satisfies \( n^{1/2^h} = 2 \). Solving this for \( h \), we get that \( h = \log \log n \).

The work done at every node of this recursion tree is constant, so the total work done is simply the number of nodes of the tree, which is \( 2^{h+1} - 1 = \mathcal{O}(\log n) \), so \( T(n) = \mathcal{O}(\log n) \).

(c) True/False: Let \( T(n) = aT(n/b) + \Theta(n^d) \) and \( S(n) = aS(n/b) + \Theta(n^c) \). If \( d > c \), then \( S(n) = \mathcal{O}(T(n)) \).

Solution: True. Compare \( d \) and \( c \) to \( \log_b a \).

If they are both the same relative to \( \log_b a \), there are three cases. If both are less, \( S(n) \) and \( T(n) \) are both \( \Theta(n^{\log_b a}) \), so \( S(n) = \mathcal{O}(T(n)) \). If both are equal, something similar happens. If both are less, the resulting times are \( T(n) = \Theta(n^d) \) and \( S(n) = \Theta(n^c) \).

If they are not the same relative to \( \log_b a \), e.g. \( d > \log_b a \) but \( c < \log_b a \), this is still true as \( n^{\log_b a} = \mathcal{O}(n^{\log_b a} \log n) \) and \( n^{\log_b a} \log n = \mathcal{O}(n^d) \) for \( d > \log_b a \).

2 Maximum Subarray Sum

Given an array of \( n \) integers, the maximum subarray is the contiguous subarray (potentially empty) with the largest sum. For example, the maximum subarray of \([-2, 1, -3, 4, -1, 2, 1, -5, 4]\) is \([4, -1, 2, 1]\) with a sum of 6.

Design an algorithm that finds the maximum subarray in \( \mathcal{O}(n \log n) \) time, and analyze its runtime.

Hint: Split the array into two equally-sized pieces. What are the possibilities for the maximum subarray, and how does this apply if we want to use divide and conquer?

Solution: Imagine that we have split the array into two pieces, \( A \) and \( B \). There are two options for the maximum contiguous subarray:

1. It is contained entirely in piece \( A \) or piece \( B \). This can be computed by calling our procedure recursively on the two halves.

2. It crosses the boundary, so part of it is in piece \( A \) and part of it is in piece \( B \).

So our algorithm calls itself recursively on each of the two pieces. We can compute the second case in \( \mathcal{O}(n) \) time by separately computing the maximum contiguous subarray in \( A \) that lies on the boundary and the maximum contiguous subarray in \( B \) that lies on the boundary, then adding those together.

For example, we can compute the maximum subarray in \( A \) that lies on the boundary in \( \mathcal{O}(n) \) time as follows: Start at the boundary, and ‘expand’ the array away from the boundary until we reach the edge of the array. Return the maximum sum reached during this process.

For the base case of an array with one element, we can simply return the element if it is positive, or 0 if the element is negative.

We end up with the recurrence

\[
T(n) = 2T(n/2) + \mathcal{O}(n)
\]

Using the Master theorem, this results in a runtime of \( \mathcal{O}(n \log n) \).
Note: This problem can actually be solved in linear time. Do you have any ideas on how to accomplish this?

The question did not ask for correctness, but for instructive purposes here is how we might do it.

We already argued that the maximum contiguous subarray either crosses the boundary between the two halves or the array or is contained entirely in one piece. We can use induction, so that the inductive hypothesis shows that the maximum contiguous subarray for each piece is correctly computed.

The largest subarray that crosses the boundary must be composed of the largest subarrays that lie on the boundary in either piece, so the procedure for computing it is also correct.

The maximum subarray must be either of these cases, which the algorithm correctly finds.

3 Quantiles

Let $A$ be an array of length $n$. Define the $k$-quantiles to be the set \{a^{(n/k)}, a^{(2n/k)}, \ldots, a^{((k-1)n/k)}\} where $a^{(\ell)}$ is the $\ell$-th smallest element in $A$.

Devise an algorithm to compute the $k$ quantiles in time $O(n \log k)$. For convenience, you may assume that $k$ is a power of 2.

Hint: Recall that QUICKSELECT($A$, $\ell$) gives $a^{(\ell)}$ in $O(n)$ time.

Solution: The idea is to find the median of $A$, partition it into two pieces around this median. Then we recursively find the medians of the two partitions, partition those further, and so on. If we do this $\log k$ times, we will have found all of the $k$-quantiles.

Finding the median and partitioning $A$ takes $O(n)$ time. We also do this for two arrays of size $n/2$, four times for arrays of size $n/4$, etc. So the total time taken is

$$O(n + 2 \cdot n/2 + 4 \cdot n/4 + \cdots + 2^{\log k} n/2^{\log k}) = O(n \log k)$$

4 Complex numbers review

A complex number is a number that can be written in the rectangular form $a + bi$ ($i$ is the imaginary unit, with $i^2 = -1$). The following famous equation (Euler’s formula) relates the polar form of complex numbers to the rectangular form:

$$re^{i\theta} = r \cos \theta + (r \sin \theta)i$$

In polar form, $r \geq 0$ represents the distance of the complex number from 0, and $\theta$ represents its angle. The $n$ roots of unity are the $n$ complex numbers satisfying $\omega^n = 1$. They are given by

$$\omega_k = e^{2\pi ik/n}, \quad k = 0, 1, 2, \ldots, n - 1$$
(a) Let \( \omega_1 = e^{2\pi i 3/10} \), \( \omega_2 = e^{2\pi i 5/10} \) be two 10-th roots of unity. Compute the product \( \omega_1 \cdot \omega_2 \).

Is this a root of unity? Is it an 10-th root of unity?

What happens if \( \omega_1 = e^{2\pi i 6/10}, \omega_2 = e^{2\pi i 7/10} \)?

**Solution:** \( \omega_1 \cdot \omega_2 = e^{2\pi i 8/10} \). This is always an 10-th root of unity (it is in general). But because \( 8/10 = 4/5 \), this is also a 5th root of unity.

If \( \omega_1 = e^{2\pi i 6/10}, \omega_2 = e^{2\pi i 7/10} \), then we ‘wind around’ and the product becomes \( e^{2\pi i 13/10} = e^{2\pi i 3/10} \).

**Note to TAs:** It would probably be very helpful to draw the unit circle with 10 points, show which of these are 5th roots of unity, and describe the behavior.

(b) Show that for any \( n \)-th root of unity \( \omega \), \( \sum_{k=0}^{n-1} \omega^k = 0 \).

**Hint:** Use the formula for the sum of a geometric series \( \sum_{k=0}^{n} \alpha^k = \frac{\alpha^{n+1}-1}{\alpha-1} \). It works for complex numbers too!

**Solution:** Remember that \( \omega^n = 1 \). So
\[
\sum_{k=0}^{n-1} \omega^k = \frac{\omega^n-1}{\omega-1} = \frac{1-1}{\omega-1} = 0
\]

(c) (i) Find all \( \omega \) such that \( \omega^2 = -1 \).

**Solution:** Notice that \( (Re^{2\pi i \theta})^n = R^n e^{2\pi i n \theta} \). Because \(-1\) is on the unit circle, the \( R \) for any such \( \omega \) must be exactly one.

As \(-1 = e^{2\pi i (1/2)} \) (Euler’s identity!), we are looking for all \( \theta \) such that \( 2\theta = \frac{1}{2} + k \) for some integer \( k \).

Setting \( k = 0 \) gives \( \theta = \frac{1}{4} \). Setting \( k = 1 \) gives \( \theta = \frac{3}{4} \). Any other values of \( k \) will just repeat these values of \( \theta \).

Notice that in general, the square roots of \( e^{2\pi i \theta} \) are \( e^{2\pi i \theta/2}, e^{2\pi i (\theta+1)/2} \).

**Note to TAs:** If you visualize the square root formula on the unit circle, you can see the structure of the square roots.

(ii) Find all \( \omega \) such that \( \omega^4 = -1 \).

**Solution:** These \( \omega \) are simply the square roots of the \( \omega \) we found before. The square roots of \( e^{2\pi i 1/4} \) are \( e^{2\pi i 1/8}, e^{2\pi i 5/8} \).

The square roots of \( e^{2\pi i 3/4} \) are \( e^{2\pi i 3/8}, e^{2\pi i 7/8} \), so we are finished.