Learning Goals

During this lab, you will:

- review the Simplified Master Theorem
- further your understanding of $O(n \log n)$ deterministic quicksort
- understand the primary force behind deterministic quicksort that is quickselect
- solve some more fun problems!

Simplified Master Theorem

The master theorem is a powerful tool in the analysis and classification of recurrences. It may be used to easily classify recurrences that might otherwise be very time-consuming!

**Theorem 1** (Simplified master theorem). Given a recurrence $T(n)$ of the form,

$$T(n) = \begin{cases} 
c_0 & \text{for } n < c \\
 aT\left(\frac{n}{b}\right) + \Theta(n^i) & \text{otherwise}
\end{cases}$$

then...

1. If $a > b^i$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $a = b^i$, then $T(n) = \Theta(n^i \log_b n)$.
3. If $a < b^i$, then $T(n) = \Theta(n^i)$.

This probably seems very magical and hand-wavy. But we’re computer scientists, so let’s delve in and figure this out. Let us assume here that $n$ is some power of $b$.

Understanding the S.M.T.

To understand why the S.M.T. works, let’s draw out the recurrence tree for $T(n)$.

As shown in the diagram on the next page, at the $k$th level of iteration, there are $a^k$ subdivisions of $(n/b^k)^i$ work. Therefore, at the bottom-most level there are $a^{\log_b n}$ subdivisions of $(n/b^{\log_b n})^i = 1$ work. Recall that by the properties of logarithms, $a^{\log_b n} = n^{\log_b a}$—so we can switch the base and the contents of the log.

We can therefore write the total amount of work represented by the recurrence $T(n)$ as

$$\Theta(n^{\log_b a}) + \sum_{k=0}^{\log_b n - 1} a^k \cdot \left(\frac{n}{b^k}\right)^i$$

So what does this mean in the three cases shown in the simplified master theorem?

In the case where $a > b^i$, the work done at the leaves heavily outgrows that done at the root. That is, $n^{\log_b a}$ is the dominating term in the sum and the total work is therefore $\Theta(n^{\log_b a})$. 

Figure 1: A tree representation of the work performed at each level of iteration in the recurrence.

In the case where $a = b^i$, the work done at each level is the same and the total work is just the height of the tree multiplied by the work at each level: $\Theta(n^i \log_b n)$.

In the case where $a < b^i$, then the work done at each subsequent level decreases with respect to the root, and the work done at the root dominates: $\Theta(n^i)$.

Tada! By drawing the recurrence tree and summing the total work performed at each level, we were able to find general expressions for the recurrence solutions for each case of the S.M.T.

Some Problems

Problem 1

We will continue to develop our understanding of the Simplified Master Theorem. For each of these three common recurrences, you should first practice solving the recurrence by expanding (telescoping, or another preferred method), then apply the above method to derive the same recurrence.

Also, you should be able to name some common algorithms that correspond to these recurrence relations!

• $T(n) = T\left(\frac{n}{2}\right) + c$
• $T(n) = 2T\left(\frac{n}{2}\right) + c$
• $T(n) = 2T\left(\frac{n}{2}\right) + n$

Solution

• $T(n) = T\left(\frac{n}{2}\right) + c$ (Binary Search)
  Since $a = 1$ and $b = 2$ and $i = 0$, we have $1 = 2^0 = 1$. This implies the second case of the MT, so we have $T(n) = \Theta(n^0 \log_2 n)$, as expected.
• \( T(n) = 2T\left(\frac{n}{2}\right) + c \) (No common algorithm has this recurrence)
  Since \( a = 2 \) and \( b = 2 \) and \( i = 0 \), we have \( 2 > 1 \). This implies the first case of the MT, so we have
  \( T(n) = \Theta(n\log_2 n) = \Theta(n) \)

• \( T(n) = 2T\left(\frac{n}{2}\right) + n \) (Mergesort)
  Since \( a = 2, b = 2, \) and \( i = 1 \), we have \( 2 = 2^1 = 2 \). This implies the second case of the MT, so we have
  \( T(n) = \Theta(n\log_2 n) \).

**Problem 2**

The Master Theorem is certainly useful as we saw in the recurrences in problem 1. However, there are
many recurrences that cannot be solved using this method. In this problem, examine the pseudocode, and
determine the recurrence relation. State whether or not it can be solved using the simplified master theorem.

```python
def reverse_string(s):
    if len(s) == 0:
        return s
    return reverse_string(s[1:]) + s[0]

def fib(n):
    if n == 1 or n == 0:
        return n
    return fib(n - 1) + fib(n - 2)

def power_of_2(n):
    if n == 0:
        return 1
    else:
        tmp = power_of_2(floor(n / 2))
        if n % 2 == 0:
            return tmp * tmp
        else:
            return 2 * tmp * tmp
```

**Solution**

1. Recurrence: \( T(n) = T(n - 1) + (n - 1) \). No, not solvable with MT.
2. Recurrence: \( T(n) = T(n - 1) + T(n - 2) + c \). No, not solvable with MT.
3. Recurrence: \( T(n) = T(n/2) + c \). Yes, solvable with MT.

**Quicksort and Quickselect Recap**

Recall the algorithm for Quicksort and Quickselect, and pay close attention to the pivot selection step for
Quicksort!

**Quicksort (sorting any array)**

- Within the array pick an arbitrary value and call that the pivot.
- Rearrange the array such that all values less than the pivot come before the pivot, and all values
greater than the pivot come after the pivot.
- Apply the above steps in a recursive manner to the both sub-arrays (elements of smaller values, and
  elements of greater values). For any sub-arrays that contain 1 element, return that sub-array.
• Running time: worst case $O(n^2)$, best case $O(n \log n)$

Quickselect (finding the $k$-th largest element in an array)
• Within the array pick an arbitrary value and call that the pivot.
• Rearrange the array such that all values less than the pivot come before the pivot, and all values greater than the pivot come after the pivot.
• Apply the above steps in a recursive manner to one of the subarrays based on $k$.
• Running time: worst case $O(n^2)$, best case $O(n)$

We see that the given algorithm for Quicksort has an upper bound of $O(n^2)$. Can we do better? The answer is YES! The key lies in how we select the pivot. Notice that currently, our Quickselect algorithm has a worst case runtime of $O(n^2)$. There in fact exists a more complex algorithm for Quickselect known as median-of-medians Quickselect, which has a running time of $\Theta(n)$. Let us first review how the median-of-medians selection algorithm works, and then we will show how this subroutine can be used to improve upon the Quickselect algorithm itself, which in turn can help us improve our Quicksort algorithm!

**Median-of-Medians Selection**

This particular selection algorithm capable of selecting the $i$th largest element in worst-case linear time complexity. The key to this algorithm running in worst-case linear time is that there exists a guarantee that a good split will be done upon partitioning the array (rather than a randomized split). The deterministic partitioning step PARTITION we went over last week will be used as well, but with a slight modification to take in the element to partition around as an input parameter (before it was just chosen at will).

**Median-of-Medians Selection Algorithm**

1. Divide the $n$ elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining $n \mod 5$ elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by first insertion-sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use SELECT (name of this current algorithm) recursively to find the median $x$ of the $\lceil n/5 \rceil$ medians found in step 2. (If there are an even number of medians, then by our convention, $x$ is the lower median.)
4. Partition the input array around the median-of-medians $x$ using the modified version of PARTITION. Let $k$ be one more than the number of elements on the low side of the partition, so that $x$ is the $k$th smallest element and there are $n - k$ elements on the high side of the partition.
5. If $i = k$, then return $x$. Otherwise, use SELECT recursively to find the $i$th smallest element on the low side if $i < k$, or the $(i - k)$th smallest element on the high side if $i > k$.

A more formal proof of the runtime can be found on page 221 of CLRS.

**Some More Problems**

**Problem 1: Checking value and rank**

Given an unsorted array of distinct integers, determine whether there is an element in the array that has a value equal to its rank in the sorted array.

• Should you do this in an iterative or recursive manner and why?
• How might you utilize quicksort and or quickselect?
• What is the running time of the approach? Is it optimal?
Solution

Consider an algorithm with parameters $l$ and $r$ denoting the left and right ends of the array. Now we can choose the median element, $m$ of rank $(r - l)/2$ and partition accordingly. If the value of $m$ is equal to $l + (r - l)/2$ (its rank), then return true, otherwise we can recursively call the algorithm on the left half of the array if $m > l + (r - l)/2$ and on the right half of the array if $m < l + (r - l)/2$. Notice that when we partition the array, we can eliminate exactly half of the remaining elements as candidates if the median is too high or low - if it is $> l + (r - l)/2$ for example, then we know any element above $m$ must have a value greater than its rank because the integers are distinct. Thus our recurrence relation is $T(n) = T(n/2) + O(n)$, which results in a running time of $O(n)$ (see the Quickselect best-case analysis in the problem before).

Problem 2: Finding the popular element

How can you determine whether or not there is an element in a given array $A$ of integers (not necessarily distinct) that occurs at least $\lceil n/2 \rceil$ times?

- Does this particular element have any special properties?
- What is the running time of the optimal approach?

Solution

The key here is to note that if an element were to indeed occur at least $\lceil n/2 \rceil$ times, then it must be a median of $A$. Thus, we can use Quickselect to find the median in $O(n)$ time. However, if an element is a median, it does not need to necessarily occur at least $\lceil n/2 \rceil$ times (why?), so once we have the median we can just linearly scan the array and count how many times it appears. If it appears more than $\lceil n/2 \rceil$ times, we return true, otherwise we return false. Since the linear scan takes $O(n)$ time, the overall algorithm takes $O(n)$ time. Notice we didn’t need any extra space!