Learning Objectives

Learners will be able to...

- **Define and formulate recurrence relations**
- **Differentiate the master theorem and recurrence relations**
- **Identify the significance of space complexity in addition to time complexity**

Analyzing Algorithms

We have previously talked about the importance of things like space and time complexity of algorithms. Using these valuable pieces of information we can compare algorithms. Which ones are faster? Which ones take up less memory? This allows us to select the best one for a given situation.

Analyzing non-recursive are fairly straightforward. We often employ techniques like counting the number of operations, assessing loop iterations, etc. This is relatively easy as the code is easy to follow.

Recursive Algorithms: A Different Beast

Recursion, however, is quite common in many algorithms due to the self-similar of many data structures. These algorithms present unique challenges. Because the function calls itself with different input sizes, understanding the growth of these calls over time is essential to understanding an algorithm's efficiency.

Key aspects to consider when analyzing recursive algorithms include:

- **Base Case(s)**: Every recursive algorithm must have one or more conditions that allow it to terminate. These are known as base cases. Understanding these is crucial, as they determine when the recursion stops.
- **Recurrence Relation**: This defines the relationship between the size of a problem and the size of its sub-problems. The recurrence relation provides a mathematical model that describes the algorithm's behavior.
- **Divide and Conquer**: Many recursive algorithms employ a 'divide and conquer' approach. This means they break the problem down into smaller sub-problems. The manner and efficiency with which an algorithm divides its tasks can greatly affect its overall performance.
- **Stack Space**: Recursive calls utilize stack memory. Thus, it's crucial to consider the space implications of recursion. An algorithm that delves too deep recursively might consume all available stack space, leading to a stack overflow.

Recurrence Relation

Defining Recurrence Relation

A **recurrence relation** is an equation that describes the relationship between the values of a function at different input sizes. It is often used to analyze the time complexity of recursive algorithms.

In a recursive algorithm, a problem is broken down into smaller subproblems, which are then solved recursively. The recurrence relation for a recursive algorithm can be used to determine the number of times the recursive function is called, which is a measure of the algorithm's time complexity.

Fundamentals of Recurrence Relation

Recurrence relations are often used when studying data structures to analyze the time complexity of different operations. A recurrence relation for a function $T(n)$ expresses its value in terms of one or more smaller values of T. Here is a simple example for calculating the recurrence relation for a recursive function:

$$T(n) = T(n-1) + 1$$

The $T(n-1)$ represents the smaller subproblem created by the recursive algorithm. This relation suggest that the work done for problem for a size of n is dependent on the work done for a problem of size ($n-1$), plus some constant work (denoted by the `1`).

Fibonacci Sequence

Let's further explore recurrence relation with a recursive algorithm we have already used, the Fibonacci sequence:

```
public int fibonacci(int n) {
    if (n <= 1) return n;  // Base case
    return fibonacci(n-1) + fibonacci(n-2);
}
```

In this particular algorithm, we are recursing twice — once for `fibonacci(n-1)` and again for `fibonacci(n-2)`. This recurrence relation states that the time complexity of computing the Fibonacci number for a positive integer `n` is equal to the sum of the time complexities of computing the Fibonacci numbers for `n-1` and `n-2`.

$$T(n) = T(n-1) + T(n-2)$$

However, the reality is a bit more nuanced than the equation above. If you look at the `fibonacci` method, you will see that there are other actions besides recursion. The algorithm is making a comparison, adding two integers, returning `1`, etc. All of this work is a constant.

A better expression of the recurrence relation can be described by the equation below. The *c* represents the constant time taken for operations other than the recursive calls.

$$T(n) = T(n-1) + T(n-2) + c$$

Notice something important: The method makes two recursive calls for each non-base case input. This leads to an exponential growth in the number of operations as n grows, making this a very inefficient way to compute Fibonacci numbers. We will cover some ways to increase the efficiency for calculating the nth number of the Fibonacci sequence in another assignment.

Recurrence relations are a powerful tool for analyzing the time complexity of recursive algorithms. They can be used to determine the asymptotic behavior of an algorithm, which can help us to choose the most efficient algorithm for a particular problem.

Master Theorem

Calculating Time Complexity

We have seen how recurrence relations are used to calculate the time complexity of recursive algorithms. We have also talked about how divide and conquer is a specific kind of recursion. In light of this, there is a specific recurrence relation that applies to divide and conquer recursive algorithms.

This specific recurrence relation is called the **master theorem**. It gives us a direct way to determine the time complexity based on:

- How many subproblems there are.
- The size of each subproblem relative to the original problem.
- The cost of dividing the original problem and combining the subproblem solutions.

We can express the master theorem as a recurrence relation with the following formula:

$$T(n) = a * T(n/b) + f(n)$$

Where:

- n is the size of the input. This is the parameter that determines the asymptotic behavior of the algorithm.
- a is a constant greater than 1. This is the growth factor of the recursive calls.
- b is a constant greater than 1. This is the factor used to reduce the size of each subproblem.
- n/b is the size of each new subproblem.
- $f(n)$ is a function that represents the amount of work done outside of the recursive calls. This commonly entails the time it takes to create subproblems and combine their results into the solution.

Recursive Fibonacci Algorithm

Although the recursive Fibonacci algorithm does not fit perfectly into the classic mold for the master theorem, we can still use the theorem's principles to understand the algorithm's nature. Recall that we previously expressed the recurrence relation for finding the nth number of the Fibonacci sequence as:

$$T(n) = T(n-1) + T(n-2) + c$$

The $(n-1)$ and $(n-2)$ represent the creation of smaller subproblems. According to the master theorem, we can divide the size of the input (n) by a factor (b) to get the size of the smaller subproblem. If we make this substitution, the recurrence relation becomes:

$$T(n) = T(n/b) + T(n/b) + c$$

Adding two of the same value is equivalent to multiplying that value by 2. We can make this substitution to our recurrence relation, which becomes:

$$T(n) = 2 \times T(n/b) + c$$

The constant c is defined as the non-recursive actions the algorithm performs. This aligns with $f(n)$ in the master theorem. Our recurrence relation now becomes:

$$T(n) = a \times T(n/b) + f(n)$$

We can see how the original recurrence relation of the recursive Fibonacci algorithm closely maps to the master theorem. While this is not a rigorous application of the master theorem, it mirrors the theorem's emphasis on the branching factor of recursive algorithms to determine their growth.

However, it is worth noting that there are more efficient algorithms, like the iterative approach or using matrix exponentiation, which can compute the Fibonacci sequence in polynomial or even logarithmic time.

Merge Sort Analysis

The master theorem provides three distinct cases which can help ascertain the nature of $T(n)$, the time complexity of the recursive algorithm. The first thing we need to do is calculate the value for d such that $a^d > b^d$. **Note**, the values for a and b come from the master theorem.

The 3 cases are as follows:

- **Case 1:** If $d < log_b a$, then $T(n) = O(n^d)$.
- **Case 2:** If $d = log_b a$, then $T(n) = O(n^d \log n)$.
- **Case 3:** If $d > log_b a$, then $T(n) = O(f(n))$.

Merge Sort Analysis

The master theorem can be used to solve a wide variety of recurrence relations that arise in the analysis of divide-and-conquer algorithms. Lets try it with our typical divide and conquer example, merge sort. We know that a merge sort works by recursively dividing the input array into two halves, sorting each half, and then merging the two sorted halves back together.

```
// code snippet from a merge sort
void sort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        sort(arr, left, mid);      //create the left half
        sort(arr, mid + 1, right); //create the right half
        merge(arr, left, mid, right);
    }
}
```

If we apply the master theorem to a merge sort, we can describe its recurrence as:

$$T(n) = 2 \times T(n/2 + O(n))$$

▼ Original master theorem
Remember, the original master theorem can be expressed as:

$$T(n) = a * T(n/b) + f(n)$$

Where:

- n is the size of the input.
- a is a constant representing the growth factor of the recursive calls.
- b is a constant representing the factor used to reduce the size of each

subproblem.
- n/b is the size of each new subproblem.
- $f(n)$ is a function representing the amount of non-recursive work.

Here is a quick explanation for the substitutions made to the original master theorem:

- We can substitute 2 for a because we divide the array into two subproblems.
- We can substitute 2 for b since each subproblem is half the size the original problem.
- We can substitute $O(n)$ for $f(n)$ because of the linear time merging process.

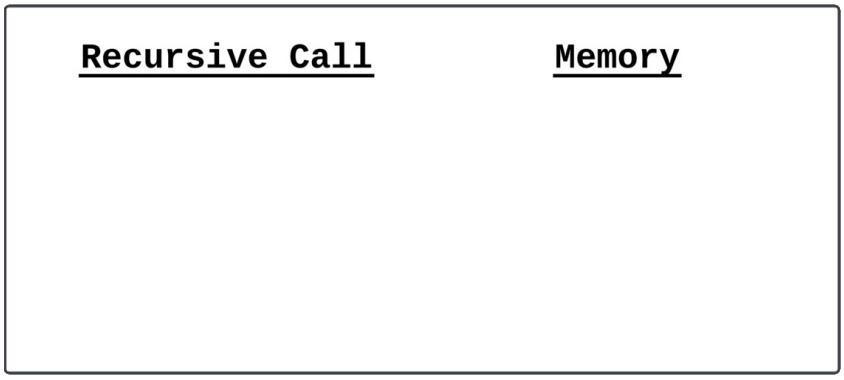
Using the cases at the top of this page, we see that a merge sort falls under **case 2**, $d = log_b a$. The function $F(n) = O(n)^{log_b a})$ which can be simplified to $O(n)$. Therefore, the time complexity of merge sort is $T(n) = O(nlogn)$
.

Space Complexity of Recursive Algorithms

While time complexity often takes center stage in algorithm analysis, understanding the space requirements, especially for recursive algorithms, is equally vital. Every recursive call consumes additional memory, primarily due to the stack frame created during the call. As such, a deep recursive algorithm can quickly lead to a stack overflow if it exhausts the available stack space.

Stack Frames and Recursion

Every time a method (recursive or not) is called, a new stack frame is created to store local variables, parameters, and bookkeeping information for the method. For recursive methods, each recursive call generates a new stack frame. The total stack space consumed by a recursive algorithm is proportional to the maximum depth of its recursive calls.



To determine the space complexity of a recursive algorithm, focus on two main factors:

1. **Non-recursive space**: This includes space for local variables, constants, and temporary storage, which is not part of the recursive call.
2. **Recursive space**: The space required for each recursive call, which is directly linked to the depth of recursion.

The overall space complexity is the sum of non-recursive and recursive space.

Examples

Let's look at some recursive algorithms we already know and calculate the space complexity for each.

1. **Factorial Function**

   ```
   public int factorial(int n) {
       if (n <= 1) return 1;
       return n * factorial(n-1);
   }
   ```

   For this function, each call results in a new recursive call (except the base case). Hence, in the worst case, the depth of recursion is n. Thus, the space complexity is $O(n)$.

2. **Binary Search**

   ```
   public int binarySearch(int[] arr, int l, int r, int x) {
       if (r >= l) {
           int mid = l + (r - l) / 2;
           if (arr[mid] == x) return mid;
           if (arr[mid] > x) return binarySearch(arr, l, mid -
   1, x);
           return binarySearch(arr, mid + 1, r, x);
       }
       return -1;
   }
   ```

   Binary search divides the problem size in half with each recursive call. As a result, the maximum depth of recursion is $\log n$. The space complexity is thus $O(\log n)$.

Optimization with Tail Recursion

Tail recursion occurs when the recursive call is the last operation in the function. Here is the factorial algorithm. This is an example of tail recursion because the recursive call comes in the last line of the method.

```
public static int factorial(int num) {

    if (num == 1)
      return 1;
    else
      //recursive call is the last operation
      return num * factorial(num - 1);
}
```

Many compilers and interpreters can optimize tail recursive calls to reuse the current function's stack frame for the next call, effectively transforming the recursion into iteration and dramatically reducing space requirements.

For example, the factorial function can be rewritten as a tail-recursive function with an accumulator. When optimized, its space complexity reduces to $O(1)$.

```
    public static int factorial(int num, int accumulator) {

        if (num == 1)
          return accumulator;
        else
          return factorial(num - 1, num * accumulator);
    }
```

In the vast realm of recursive algorithms, space often matters as much as time. Being adept at analyzing both provides a holistic understanding of an algorithm's efficiency and constraints, allowing for more informed and optimal solutions.

Formative Assessment 1

Formative Assessment 2