Section 1: Recurrences and Divide-and-Conquer Algorithms

Amy Zhao

January 26, 2024

1 Recurrence Relations

A **recurrence** is a description of a function in terms of itself which typically consists of one or more base cases and one or more recursive cases. A very common example is the Fibonacci recurrence relation. In the context of algorithms, these relations reflect the runtime of recursive algorithms.

As a motivating example, consider the following function.

```
1 Function MysteryFunction(n):
2     if n > 1 then
3         print("still going")
4         MysteryFunction(n/2)
5         MysteryFunction(n/2)
```

Let $L(n)$ be the number of lines that the function above prints with input $n = 2^k$. For the base case $n = 1$ ($k = 0$), $L(1) = 0$; for $n = 2^k$ where $k \geq 1$, $L(n)$ satisfies:

$$L(n) = 1 + 2L\left(\frac{n}{2}\right). \tag{1}$$

When we write the first few values of $L(n)$, we can guess that $L(n) = n - 1 \equiv L(2^k) = 2^k - 1$ when $n = 2^k$. It's possible to prove this claim through induction.

For the base case, we have $2^0 - 1 = 1 - 1 = 0$, which matches the solution $L(2^0) = L(1) = 0$ specified previously. In the inductive step, assume for any $k \geq 0$, $L(2^k) =$

$2^k - 1$ holds true. By definition of L, we have:

$$L(2^{k+1}) = 1 + 2L(2^{(k+1)-1}) = 1 + 2L(2^k).$$

By the inductive hypothesis, we know $L(2^k) = 2^k - 1$, and we can make a substitution to the equation above:

$$L(2^{k+1}) = 1 + 2(2^k - 1) = 1 + 2^{k+1} - 2 = 2^{k+1} - 1.$$

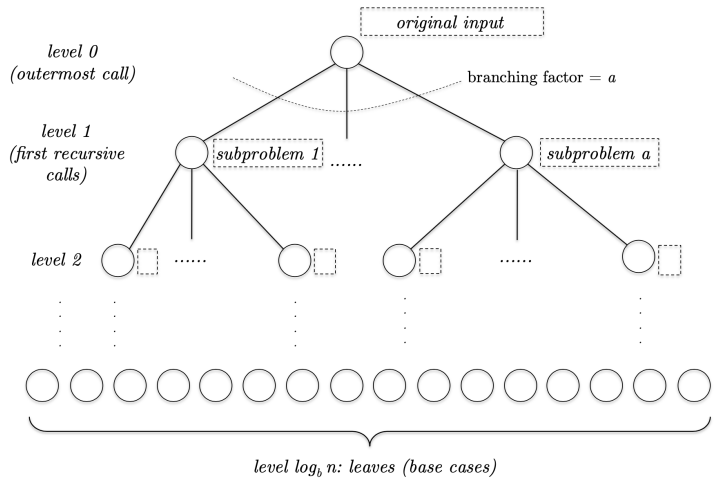
This proves the inductive step, which means the overall claim holds by the principle of induction.

1.1 The Master Theorem

Recall that for divide and conquer algorithms, the runtime of the algorithm $T(n)$ is subject to the following inequality:

$$T(n) \leq \underbrace{a \cdot T\left(\frac{n}{b}\right)}_{\substack{\text{work done by} \\ \text{recursive calls}}} + \underbrace{O(n^d)}_{\substack{\text{work done outside} \\ \text{recursive calls}}}, \tag{2}$$

where a is the number of recursive calls, b is the input size shrinkage factor, and d is the exponent in the runtime of the "combination step".



If $T(n)$ is defined by a standard recurrence, with parameters $a \geq 1$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases} \tag{3}$$

At a high level, the Master Theorem gives us a quick way to determine the runtime of a divide and conquer algorithm, provided we can identify the parameters a, b, and d. Keep in mind that not all recurrences apply to the Master Theorem. Some cases include when a is a function of n; when there are not enough subproblems specified; and when the combination runtime is not positive.

To prove the Master Theorem, let $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$ be the recurrence we solve. Additionally, assume $T(1) = 1$ and that n is a power of b for simplicity. By applying the definition of big O, we have:

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + cn^d. \tag{4}$$

In the recursion tree depicted above, there are $\log_b n + 1$ levels. At level j, there are a^j subproblems, and each subproblem is of size $\frac{n}{b^j}$ which takes at most $c\left(\frac{n}{b^j}\right)^d$ time to solve. This implies that the total time required to complete level j is at most $a^j \cdot c\left(\frac{n}{b^j}\right)^d = cn^d\left(\frac{a}{b^d}\right)^j$. Accounting for all levels, the runtime adheres to the following relationship:

$$T(n) \leq cn^d \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j. \tag{5}$$

We now consider each of the three cases presented in Eq. (3):

$$T(n) \leq \begin{cases} (\log_b n + 1) \cdot cn^d = O(n^d \log n) & \text{if } a = b^d \\ cn^d \sum_{j=0}^{\infty} \left(\frac{a}{b^d}\right)^j = cn^d \left(\frac{1}{1-\frac{a}{b^d}}\right) = cn^d \left(\frac{b^d}{b^d - a}\right) = O(n^d) & \text{if } a < b^d \\ cn^d \left(\frac{\left(\frac{a}{b^d}\right)^{\log_b n + 1} - 1}{\frac{a}{b^d} - 1}\right) = O\left(n^d \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O\left(n^d \cdot \frac{n^{\log_b a}}{n^d}\right) = O(n^{\log_b a}) & \text{if } a > b^d \end{cases} \tag{6}$$

1.1.1 Example

Suppose you are choosing between the following three algorithms.

1. Algorithm A solves problems by dividing them into 5 subproblems of half the size, recursively solving each subproblem, and then combining the solution in linear time.
2. Algorithm B solves problems of size N by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.
3. Algorithm C solves problems of size n by dividing them into nine subproblems of size $\frac{n}{3}$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

The recurrence for each algorithm and the resultant time complexity is defined below:

$$A: \ T(n) = 5T\left(\frac{n}{2}\right) + O(n) \Longrightarrow O(n^{\log_2 5}) = O(n^{2.33})$$
$$B: \ T(n) = 2T(n-1) + O(1) \Longrightarrow O(2^n)$$
$$C: \ T(n) = 9T\left(\frac{n}{3}\right) + O(n^2) \Longrightarrow O(n^2 \log n)$$

Thus, algorithm C has the quickest runtime among the three algorithms.

2 Loop Invariants

In order to prove correctness for divide and conquer algorithms, we introduce the notion of a loop invariant. A **loop invariant** is a condition that is true at the beginning and end of every iteration of a loop. Proofs involving loop invariants are analogous to proofs by induction. In the base case (initialization), we show that the loop invariant is true before beginning the loop. In the inductive step, we show that if the loop holds over some number of iterations, it must hold over the next iteration as well. When the loop exits, the algorithm should output the correct answer because the loop invariant has held.

One key difference between a proof by loop invariant and a proof by induction is that the inductive step is applied finitely (until the loop terminates), as opposed to

the indefinite application of the inductive step in regular mathematical induction. An example of this proof style is presented in the following sections, contrasted with a more traditional induction proof.

3 Divide and Conquer Algorithms

Any problem which requires a divide and conquer algorithm can be split into three primary steps.

1. Divide the input into smaller subproblems.
2. Solve the subproblems recursively.
3. Combine the solutions for the subproblems into a solution for the original problem.

To apply these steps, a common approach is to first start with the entire input data and make a comparison of two values. Depending on the results of this comparison, we determine which portion of the input data to recurse on.

3.1 Binary Search

Binary search is one of the most common examples of divide and conquer you might encounter. The goal of binary search is to find the index of a target element by search the appropriate half of a sorted array, determined by comparing the middle element to the target element. If the target value is less than the middle value, the algorithm continues on the lower half. If the target is greater than the middle element, the algorithm continues on the upper half.

```
Input  : Sorted array A of n elements and a target T
Output: Index of T if it exists; −1 otherwise
1  Function BinarySearch(A, n, T):
2      L ← 0, R ← n − 1
3      while L ≤ R do
4          m ← ⌊(L+R)/2⌋
5          if A[m] < T then
6              L ← m + 1
7          else if A[m] > T then
8              R ← m − 1
9          else
10             return m
11     end
12     return −1
```

To prove correctness, we first consider the initialization of the loop. The array A is given as a sorted array, and the array remains sorted because the loop doesn't shuffle the. When the length of the list is at least $n = 1$, then we have $0 \leq L \leq R \leq n - 1$. We also know that the target index is definitely between L and R if the target exists.

Suppose L' and R' represent the values of L and R at the end of the loop. We want to show that if the invariant is true at the beginning of the loop body, it must hold at the end of the loop as well. m is the average of L and R, so it must be that $L \leq m \leq R$. The execution of the remaining portion of the loop depends the value of $A[m]$, so there are a couple of cases to consider.

- $A[m] < T$: In this case, we shift our search to the "upper" half of the array. $L' = m + 1$ and $R' = R$. Since A is sorted, the target element must exist after the element $A[m]$ in the list, and we can safely discard the "lower half". We maintain the relationship of $L' \leq R' \leq n - 1$ (assuming that the loop continues to another iteration).

- $A[m] > T$: In this case, we shift our search to the "lower" half of the array. $L' = L$ and $R' = m - 1$. Since A is sorted, the target element must exist before the element $A[m]$ in the list, and we can safely discard the "upper half". We maintain the relationship of $L' \leq R' \leq n - 1$ (assuming that the loop continues to another iteration).

- $A[m] = T$: We have found the index of the target element to return (m), and

it is safe to terminate the loop.

If no iteration encounters the third case listed, then the target element doesn't exist in the array, and the loop eventually encounters the case where $L > R$. The algorithm accounts for this correctly by returning -1.

The recurrence for binary search is:

$$T(n) = T\left(\frac{n}{2}\right) + O(1), \tag{7}$$

as each recursive call divides the search space in half, and there is only constant time work done outside of the recursive calls. This means that $a = 1$, $b = 2$, and $d = 0$, corresponding to a runtime of $O(n^0 \log n) = O(\log n)$.

3.1.1 Recursive Binary Search

Unsurprisingly, there is a recursive algorithm for binary search as well.

```
   Input  : Sorted array A of n elements, target T
   Output: Index of T if it exists; −1 otherwise
 1 Function RecursiveBinarySearch(A, n, T):
 2     if n = 1 then
 3         if T = A[1] then
 4             return 1
 5         else
 6             return −1
 7     else
 8         m ← ⌊n/2⌋
 9         if T < A[m] then
10             return RecursiveBinarySearch(A[1 : m], T)
11         else if T > A[m] then
12             return RecursiveBinarySearch(A[m + 1 : n], T)
13         else
14             return m
15     end
```

To prove correctness, let $P(n)$ be the predicate which is true if the algorithm works on any sorted array A of length n and target T. We wish to prove $\forall n \in \mathbb{N} : P(n)$. For the base case, we consider $P(1)$. If T is indeed in the array, then it must be that $T = A[1]$. If T is not in the array, we return -1, which is the desired result. In both cases, the algorithm behaves properly.

In the inductive step, first fix a natural $k \in \mathbb{N}$. Suppose the predicate is true for all ℓ such that $1 \leq \ell \leq k$. In other words, $P(1), P(2), \ldots, P(k)$ are all true, and we wish to prove $P(k+1)$. For notation purposes, set $N = k+1$. We first handle the case where the target T does not exist in array A, which implies $T \neq A[m]$ regardless of the value of m. Furthermore, we have $T \notin A[1:m]$ and $T \notin A[m+1:N]$. Both $A[1:m]$ and $A[m+1:N]$ have lengths $\left\lfloor \frac{N}{2} \right\rfloor$ and $\left\lceil \frac{N}{2} \right\rceil$, so these subarrays are covered by the inductive hypothesis. Thus, the algorithm will return -1.

The other case occurs when $T \in A$; it follows that there is some index $1 \leq j \leq N$ such that $T = A[j]$. If $j = m$, the algorithm correctly returns m. If $j < m$, we have that $T = A[j] < A[m]$ because the array is sorted, and the recursive call on line 10 will run. Here, $m = \left\lfloor \frac{N}{2} \right\rfloor < N$, so the recursive call will return the correct index under the inductive hypothesis. If $j > m$, we have that $T = A[j] > A[m]$ because the array is sorted, and the recursive call on line 12 will run. $A[m+1:N]$ has length $\left\lceil \frac{N}{2} \right\rceil < N$, so the recursive call will return the correct index for T.

3.2 Repeated Squaring

Given two integers a and n, we would like to compute a^n. The direct method that computes a^n by performing n multiplications of a is computationally inefficient for large inputs. It turns out that we can implement a divide and conquer approach.

```
   Input  : Base a and exponent n
   Output: Result of a^n
1  Function Power(a, n):
2      if n = 0 then
3          return 1
4      else if n is even then
5          k ← Power(a, n/2)
6          return k · k
7      else
8          k ← Power(a, (n−1)/2)
9          return k · k · a
```

To analyze the time complexity, we solve the following recurrence for $n \geq 1$:

$$\begin{aligned} T(n) &\leq T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 \\ &\leq T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + 2 + 2 \\ &\leq \cdots \leq T(1) + \underbrace{2 + \cdots + 2}_{\log_2(n) \text{ copies}} \leq 2\log_2(n). \end{aligned} \tag{8}$$

This implies that the algorithm runs in $O(\log n)$ time. We could alternatively apply the Master Theorem; there is one recursive call per iteration, and the "input" size is divided in half. The recurrence mirrors the one presented for binary search, Eq. (7).

3.3 Maximum Sum of Subarray

Suppose we want to design a divide and conquer algorithm to find the value of the maximum sum of any subsequence of array A. As an example, input $A = [2, 1, -1, 3, -1, -2, 3]$ should output 5 because the subsequence which produces the maximum sum is $[2, 1, -1, 3]$. The main idea is to split A into two parts of equal length and find the maximum sum subsequence of each part, as well as the maximum sum that passes through both partitions. We set $A_L = A[1 : \lfloor \frac{n}{2} \rfloor]$ and $A_R = A[\lfloor \frac{n}{2} \rfloor + 1 : n]$; M_L is the maximum sum subsequence of A_L, and M_R is the maximum sum subsequence of A_R.

Let N_L be the maximum subsequence of A_L that ends with the last element of A_L. We can compute this by starting from the last element and iterating backward over the elements of A_L. For each element encountered, we add it to a running sum and determine if we have a "new" maximum sum. Similarly, let N_R be the maximum sum subsequence of A_R that starts with the first element of A_R. We can compute N_R by iterating over the elements of A_R in the forward direction and tracking the running sum and maximum sum. The sum $N = N_L + N_R$ represents the maximum sum subsequence of A that passes through both portions of A. The maximum sum subsequence of A is simply the maximum value of the three sums calculated, $M = \max\{M_L, M_R, N\}$.

```
Input  : Array A with length n
Output: Maximum sum of a subsequence
Function LeftSum(A, n):
    S ← 0                                   // used to track sum of A[i : n]
    N ← −∞                                  // used to track max sum of A[i : n]
    for i = n to i = 1 do
        S ← S + A[i]
        N ← max{N, S}
    end
    return N
Function RightSum(A, n):
    S ← 0                                   // used to track sum of A[1 : i]
    N ← −∞                                  // used to track max sum of A[1 : i]
    for i = 1 to i = n do
        S ← S + A[i]
        N ← max{N, S}
    end
    return N
Function MaxSum(A, n):
    if n = 1 then
        M ← A[1]
    else
        A_L ← A[1 : n/2]
        A_R ← A[n/2 + 1 : n]
        M_L ← MaxSum(A_L, n/2)
        M_R ← MaxSum(A_R, n/2)
        N_L ← LeftSum(A_L, n/2)
        N_R ← RightSum(A_R, n/2)
        N ← N_L + N_R
        M ← max{M_L, M_R, N}
    return M
```

For a given array A of size n, we make two recursive calls on input sizes $\frac{n}{2}$. Finding the maximum subarray that crosses the midpoint takes $O(n)$ time. The time complexity can thus be expressed as the following recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n), \tag{9}$$

which yields a time complexity of $O(n \log n)$ by the Master Theorem.