Algorithms

1 Divide-and-Conquer Design Strategy

Why are we looking at Divide-and-Conquer? Learning the principles of the Divide-and-Conquer Approach serves as a preamble to how we will be coming up with time complexities for recursive algorithms as everything up until this point has been non-recursive. The Divide-and-Conquer Design Strategy involves the following steps:

1) Divide First, take an instance of the problem we're solving and divide it into 2 or smaller instances of the same problem. Suppose we're looking at a list and trying to solve a problem using the list. We would take the list and divide it into pieces to solve the problem for each individual piece.

2) Conquer Second, conquer or solve each of the smaller instances of pieces of the original instance/list. Unless the divided instances are small enough, we will tend to lean on recursion to continue to divide the instances into sufficiently small problems.

3) Combine Third, we combine the solutions, or conquered smaller instances of the original instance, and put them together to get the answer for the original problem.

3) Summary We can solve a problem by breaking it into smaller instances of the same problem. We can describe the running time of a recursive program by using recurrence relations.

1.1 Divide-and-Conquer: Recursive Binary Search

Previously, we studied an iterative version of Binary Search. Divide-and-Conquer is present in the recursive version of Binary Search.

Divide: If our search key is the middle item of the given sorted array, the problem is already solved. Otherwise divide the given sorted array into two, if our search key is smaller than the middle item, choose the left subarray. If x is larger than the middle item, choose the right subarray. Once we have a chosen subarray, we will continue to divide it, hence recursion.

Conquer: Solve the subarrays through recursion, breaking the arrays into smaller and smaller instances until we can determine if x is present in any of the subarrays.

Combine: Combine the solutions of each subarray, if x was not present in any subarray, then x was not found in the original array. If x was found in a subarray, then combine the subarrays back into the original array to find the index of our search key.

Binary Search Example

Suppose our search key x = 5 and we have the following array:

```
2 3 5 7 9 11 13
```

Divide: The middle item is 7. Since x is less than 7, we will recursively divide and solve the subarray to the left of 7.

```
2 3 5
```

Divide: The middle item is now 3. Since 5 is greater than 3, we will recursively divide and solve the subarray to the right of 3.

```
5
```

Conquer: The middle item is now 5. Since x = 5 = 5, x is determined to be in the subarray.

```
2 3 5 7 9 11 13
```

Combine: Combine the subarrays back together to find that our search key x=5 was present in index 2 of the original array (starting from index 0).

1.2 Sequences

Recurrence relations represent sequences. Understanding sequences is crucial to comprehending recurrence relations. Recurrence relations define sequences of data or numbers. There are multiple ways to list a sequence:

1) List sequence terms Sequences can just be listed as a set of terms:

```
Example: 1, 2, 3 , 4....
```

Here $a_1 = 1; a_2 = 2; a_3 = 3; a_4 = 4$...
This is a sequence of positive integers. So we would say $a_n = n$

```
Example: 2, 4, 6 , 8....
```

Here $a_1 = 2; a_2 = 4; a_3 = 6; a_4 = 8$...
This is a sequence of positive even integers. So we would say $a_n = 2n$

```
Example: 3, 5, 7...
```

Here $a_1 = 3; a_2 = 5; a_3 = 7$...
This is can be one of two sequences. It could be a sequence of positive odd integers greater than 1, in which case $a_4 = 9$. Or it could be a sequence of prime numbers greater than 2, in which case $a_4 = 11$. This last example shows that listing the first terms of a sequence is a good start but isn't sufficient to to find a formula for the sequence.

2) Give an explicit formula Sequences can also be found through an explicit formula, where the n'th term of a sequence can be found by a function A(n). Let's let our sequence be a list of terms:

```
2, 4, 6, 8...
```

Our explicit formula for the n'th term, with 2 being the first term could be:

```
A(n)=2n
```

But, we are not always able to easily figure out an explicit formula given a sequence, which brings us the third method.

3) Give a recurrence relation Recurrence relations are equations that define a sequence using previous or initial terms of the sequence. A commonly known sequence that can be defined by a recurrence relation is the Fibonacci Sequence, where:

```
F(0) = 0, F(1) = 1
```

Where F(n), which describes the value of the n'th term, starting at n=0, can be given by:

```
F(n) = F(n-1) + F(n-2)
```

We prefer using recurrence relations or explicit formulas over simply describing or listing the sequence of terms because we can narrow down the smaller details of the sequence and determine the values of later terms in the sequence.

Types of Sequences

1) Arithmetic Sequence An arithmetic sequence is one where our n'th term is given by multiplying n by some constant d and adding a constant to it for each term. The general solution to an arithmetic sequence is:

```
A(n) = c + dn
```

2) Geometric Sequence A geometric sequence is one where our n'th term is given by multiplying a previous term by a constant. The general solution to a geometric sequence is:

```
A(n) = c for n=0

A(n) = r*A(n-1) for n>0
```

1.3 Solving a Recurrence Relation

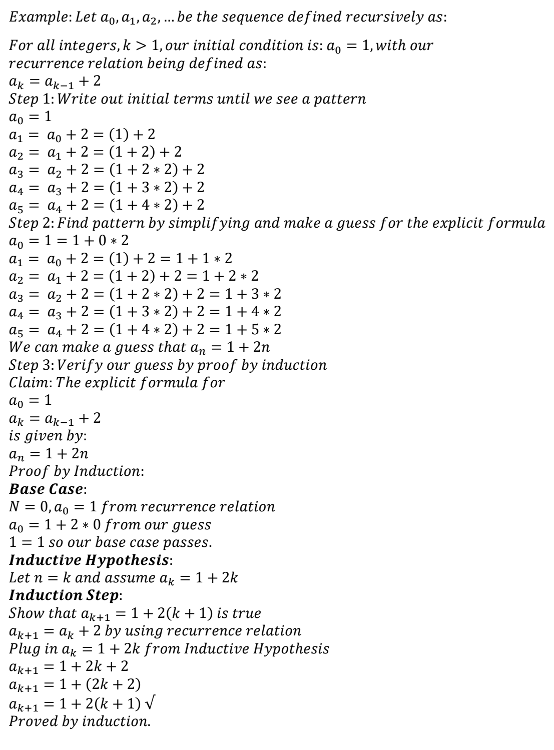
Finding an explicit formula from a recurrence relation. Although recurrence relations can be a good way to represent a sequence, it can make it hard to determine the value of a later term in a series, as it can be very time consuming to find the 1000'th term of a sequence if we have to find all of the terms that come before it, for example. Thankfully, there are 3 solid methods on solving recurrences:

1) Substitution Method We will talk more about this method in later classes, for now, keep the name in your mind.

2) Guess and Test Write out initial terms until you see a pattern, then, make a guess for the explicit formula from said pattern. Finally, verify your guess using proof by induction.

3) Recursive Tree Method Mostly useful to find a big picture of what is going on with an algorithm. But, when using it with actual values, one must be very careful and meticulous with small details. It is especially good for reasoning about Divide-and-Conquer algorithms. First, you take a tree, add up the amount of work on each level, then, if we know how many levels there are in the tree, sum up all of the work done on each level and that is the running time of our algorithm.

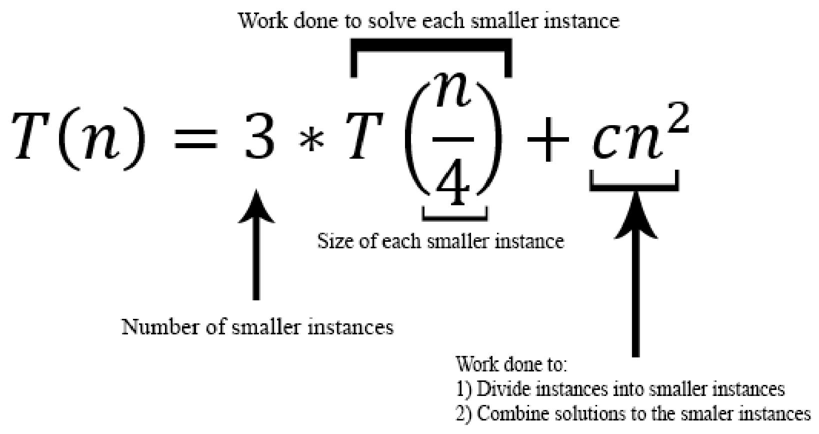
1.4 Guess and Test Example



Example: *Let* $a_0, a_1, a_2, \ldots$ *be the sequence defined recursively as*:

For all integers, $k > 1$, *our initial condition is*: $a_0 = 1$, *with our recurrence relation being defined as*:
$a_k = a_{k-1} + 2$
Step 1: *Write out initial terms until we see a pattern*
$a_0 = 1$
$a_1 = a_0 + 2 = (1) + 2$
$a_2 = a_1 + 2 = (1 + 2) + 2$
$a_3 = a_2 + 2 = (1 + 2 * 2) + 2$
$a_4 = a_3 + 2 = (1 + 3 * 2) + 2$
$a_5 = a_4 + 2 = (1 + 4 * 2) + 2$
Step 2: *Find pattern by simplifying and make a guess for the explicit formula*
$a_0 = 1 = 1 + 0 * 2$
$a_1 = a_0 + 2 = (1) + 2 = 1 + 1 * 2$
$a_2 = a_1 + 2 = (1 + 2) + 2 = 1 + 2 * 2$
$a_3 = a_2 + 2 = (1 + 2 * 2) + 2 = 1 + 3 * 2$
$a_4 = a_3 + 2 = (1 + 3 * 2) + 2 = 1 + 4 * 2$
$a_5 = a_4 + 2 = (1 + 4 * 2) + 2 = 1 + 5 * 2$
We can make a guess that $a_n = 1 + 2n$
Step 3: *Verify our guess by proof by induction*
Claim: *The explicit formula for*
$a_0 = 1$
$a_k = a_{k-1} + 2$
is given by:
$a_n = 1 + 2n$
Proof by Induction:
Base Case:
$N = 0, a_0 = 1$ *from recurrence relation*
$a_0 = 1 + 2 * 0$ *from our guess*
$1 = 1$ *so our base case passes.*
Inductive Hypothesis:
Let $n = k$ *and assume* $a_k = 1 + 2k$
Induction Step:
Show that $a_{k+1} = 1 + 2(k + 1)$ *is true*
$a_{k+1} = a_k + 2$ *by using recurrence relation*
Plug in $a_k = 1 + 2k$ *from Inductive Hypothesis*
$a_{k+1} = 1 + 2k + 2$
$a_{k+1} = 1 + (2k + 2)$
$a_{k+1} = 1 + 2(k + 1)$ √
Proved by induction.

1.5 Recursive Tree Method



Work done to solve each smaller instance
$T(n) = 3 * T\left(\frac{n}{4}\right) + cn^2$
Size of each smaller instance
Number of smaller instances
Work done to:
1) Divide instances into smaller instances
2) Combine solutions to the smaler instances

Figure 1: The meaning of different parts inside a complexity equation

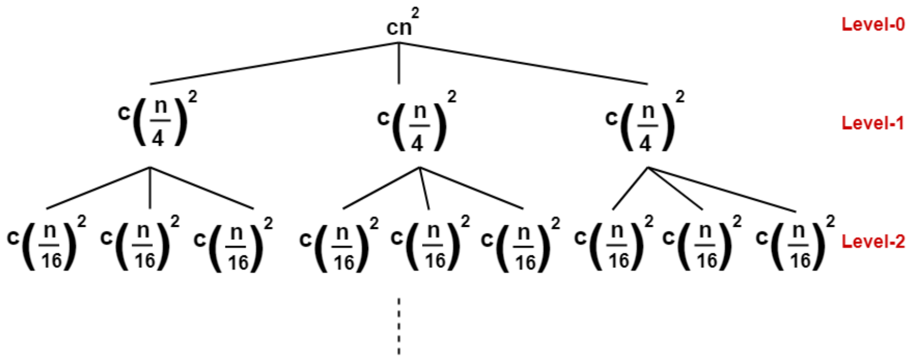


Figure 2: Representation of given T(n) by Akshay Singhal

We want to divide each instance into a small enough instance that we can combine them all to find a solution. For our given T(n), we would divide until we reach a depth i such that the work done in each node is equal to T(1). We would then find the size of each subproblem and then use that to solve for i to find out how many levels is in the tree for the given problem (Please refer to the Thursday, September 23, 2021 notes for this solution).