Recursive Algorithms, Recurrence Equations, and Divide-and-Conquer Technique

Introduction

In this module, we study recursive algorithms and related concepts. We show how recursion ties in with induction. That is, the correctness of a recursive algorithm is proved by induction. We show how recurrence equations are used to analyze the time complexity of algorithms. Finally, we study a special form of recursive algorithms based on the divide-and-conquer technique.

Contents

Simple Examples of Recursive Algorithms

Factorial: Consider the factorial definition

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

This formula may be expressed recursively as

$$n! = \begin{cases} n \times (n-1)!, & n > 1 \\ 1, & n = 1 \end{cases}$$

Below is a recursive program to compute $n!$.

```
int  Factorial (int n) {
if (n == 1) return 1;
else {
  Temp = Factorial (n − 1);
  return (n * Temp);
  }
}
```

The input parameter of this function is n, and the return value is type integer. When $n = 1$, the function returns 1. When $n > 1$, the function calls itself (called a recursive call) to compute $(n-1)!$. It then multiplies the result by n, which becomes $n!$.

The above program is written in a very basic way for clarity, separating the recursive call from the subsequent multiplication. These two steps may be combined, as follows.

```
int  Factorial (int n) {
if (n == 1) return 1;
else return (n* Factorial (n − 1));
 }
```

Correctness Proof: The correctness of this recursive program may be proved by induction.

- Induction Base: From line 1, we see that the function works correctly for $n = 1$.
- Hypothesis: Suppose the function works correctly when it is called with $n = m$, for some $m \geq 1$.
- Induction step: Then, let us prove that it also works when it is called with $n = m + 1$. By the hypothesis, we know the recursive call works correctly for $n = m$ and computes $m!$. Subsequently, it is multiplied by $n = m + 1$, thus computes $(m+1)!$. And this is the value correctly returned by the program.

Finding Maximum Element of an Array:

As another simple example, let us write a recursive program to compute the maximum element in an array of *n* elements, $A[0{:}\, n-1]$. The problem is broken down as follows.

To compute the Max of n elements for $n > 1$,

- Compute the Max of the first $n-1$ elements.
- Compare with the last element to find the Max of the entire array.

Below is the recursive program (pseudocode). It is assumed that the array type is dtype, declared earlier.

```
dtype  Max (dtype $A[\,]$, int $n$) {
if ($n == 1$) return $A[0]$;
else{
  $T = \mathrm{Max}(A, n-1)$;     //Recursive call to find max of the first $n-1$ elements
  If ($T < A[n-1]$)       //Compare with the last element
      return $A[n-1]$;
  else return $T$;
  }
}
```

Computing Sum of Elements in an Array

Below is a recursive program for computing the sum of elements in an array $A[0{:}\, n-1]$.

$$S = \sum_{i=0}^{n-1} A[i]$$

```
dtype Sum (dtype $A[\,]$, int $n$) {
if ($n == 1$) return $A[0]$;
else{
  $S$ = Sum($A, n-1$);   //Recursive call to compute the sum of the first $n-1$ elements
  $S = S + A[n-1]$;    //Add the last element
  return ($S$)
  }
}
```

The above simple problems could be easily solved without recursion. They were presented recursively only for pedagogical purposes. The next example problem, however, truly needs the power of recursion. It would be very difficult to solve the problem without recursion.

Towers of Hanoi Problem

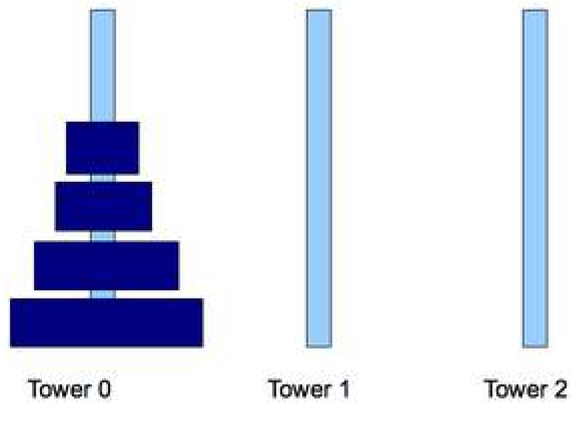
This is a toy problem that is easily solved recursively. There are three towers (posts) $A, B,$ and C. Initially, there are *n* disks of varying sizes stacked on tower *A,* ordered by their size, with the largest disk in the bottom and the smallest one on top. The object of the game is to have all *n* discs stacked on tower *B* in the same order, with the largest one in the bottom. The third tower is used for temporary storage. There are two rules:

- Only one disk may be moved at a time in a restricted manner, from the top of one tower to the top of another tower. If we think of each tower as a stack, this means the moves are restricted to a *pop* from one stack and *push* onto another stack.
- A larger disk must never be placed on top of a smaller disk.

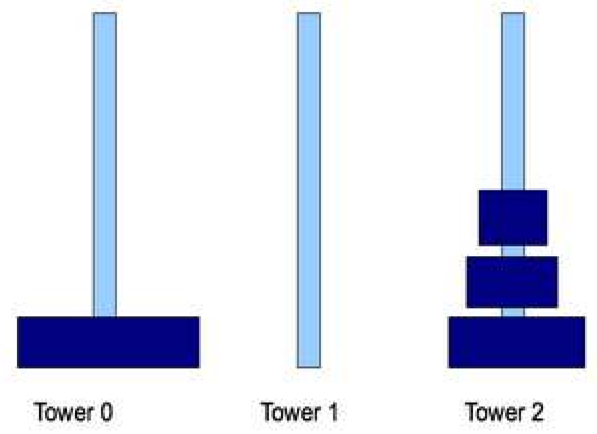
The recursive algorithm for moving *n* disks from tower *A* to tower *B* works as follows. If $n = 1$, one disk is moved from tower *A* to tower *B.* If $n > 1$,

1 Recursively move the top $n - 1$ disks from $A\ to\ C$. The largest disk remains on tower A by itself.
2 Move a single disk from $A\ to\ B$.
3 Recursively move back $n - 1$ disks from $C\ to\ B$.

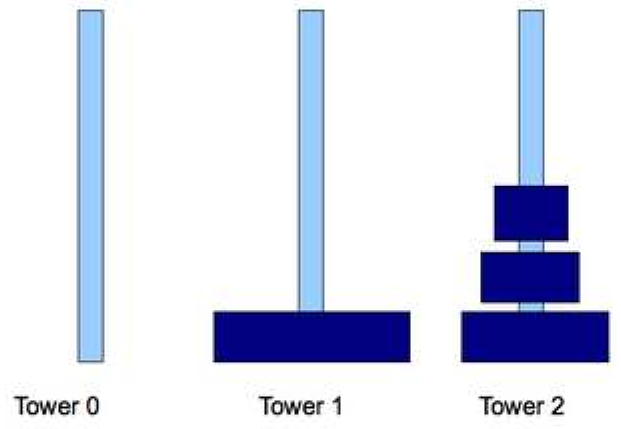
An illustration is shown below for $n = 4$.



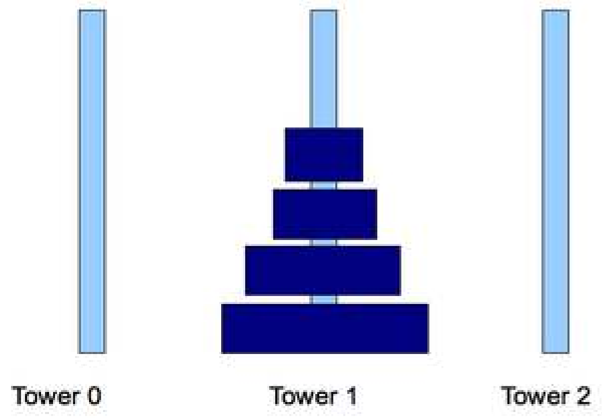
(a) Initial configuration with 4 disks on Tower 0



(b) After recursively moving the top 3 disks from Tower 0 to Tower 2



(c) After moving the bottom disk from Tower 0 to Tower 1



(d) After recursively moving back 3 disks from Tower 2 to Tower 1.

Below is the recursive algorithm. The call Towers (A,B,C,n) moves *n* disks from tower A to B, using C as temporary storage.

```
Towers $(A,B,C,n)$ {
1   if $(n == 1)$ {
2      MoveOne $(A,B)$;
3      return};
4   Towers $(A,C,B,n-1)$;
5   MoveOne $(A,B)$;
6   Towers $(C,B,A,n-1)$;
  }
```

Proof of Correctness

The correctness of the algorithm is proved by induction. For $n = 1$, a single move is made from *A* to *B*. So the algorithm works correctly for $n = 1$. To prove the correctness for any $n \geq 2$, suppose the algorithm works correctly for $n-1$. Then, by the hypothesis, the recursive call of line 4 works correctly and moves the top $n-1$ disks to *C*, leaving the bottom disk on tower *A.* The next step, line 5, moves the bottom disk to *B.* Finally, the recursive call of line 6 works correctly by the hypothesis and moves back $n-1$ disks from *C* to *B.* Thus, the entire algorithm works correctly for n.

An Improvement in Algorithm Style

The above algorithm has a single move appearing twice in the code, once for $n = 1$ and once for $n > 1$. This repetition may be avoided by making $n = 0$ as the termination criteria for recursion.

```
Towers $(A,B,C,n)$ {
if $(n == 0)$ return;
Towers $(A,C,B,n-1)$;
MoveOne $(A,B)$;
Towers $(C,B,A,n-1)$;
}
```

Recurrence Equation to Analyze Time Complexity

Let us analyze the time complexity of the algorithm. Let

$$f(n) = \text{number of single moves to solve the problem for } n \text{ disks.}$$

Then, the number of moves for each of the recursive calls is $f(n-1)$. So, we set up a recurrence equation for $f(n)$.

$$f(n) = \begin{cases} 1, & n = 1 \\ 2f(n-1) + 1, & n \geq 2 \end{cases}$$

We need to solve this recurrence equation to find $f(n)$ directly in terms of n.

Method 1: Repeated Substitution

$$f(n) = 1 + 2 \cdot \underbrace{f(n-1)}_{1+2f(n-2)}$$
$$f(n) = 1 + 2 + 4 \cdot \underbrace{f(n-2)}_{1+2f(n-3)}$$
$$f(n) = 1 + 2 + 4 + 8 \cdot f(n-3)$$
$$f(n) = 1 + 2 + 2^2 + 2^3 \cdot f(n-3)$$

After a few substitutions, we observe the general pattern, and see what is needed to get to the point where the last term becomes $f(1)$.

$$\vdots$$
$$f(n) = 1 + 2 + 2^2 + 2^3 + \cdots + 2^{n-1} \cdot f(1)$$

Then we can use the base case of the recurrence equation, $f(1) = 1$.

$$f(n) = 1 + 2 + 2^2 + 2^3 + \cdots + 2^{n-1}$$

We use geometric sum formula for this summation (where each term equals the previous term times a constant).

$$f(n) = \frac{2^n - 1}{2 - 1}$$

$$f(n) = 2^n - 1.$$

The repeated substitution method may not always be successful. Below is an alternative method.

Method 2: Guess the solution and prove it correct by induction

Suppose we guess the solution to be exponential, but with some constants to be determined.

***Guess**:* $f(n) = A\,2^n + B$

We try to prove the solution form is correct by induction. If the induction is successful, then we find the values of the constant *A* and *B* in the process.

Induction Proof:

Induction Base, $n = 1$:

$$\begin{aligned} &f(1) = 1 && \text{(from the recurrence)} \\ &f(1) = 2A + B && \text{(from the solution form)} \end{aligned}$$

So we need

$$\boxed{2A + B = 1}$$

Induction Step: Suppose the solution is correct for some $n \geq 1$:

$$f(n) = A\,2^n + B \qquad (hypothesis)$$

Then we must prove the solution is also correct for $n + 1$:

$$f(n+1) = A\,2^{n+1} + B \qquad (Conclusion)$$

To prove the conclusion, we start with the recurrence equation for $n + 1$, and apply the hypothesis:

$$\begin{aligned} &f(n+1) = 2f(n) + 1 && (from\ the\ recurrence\ equation) \\ &= 2[A\,2^n + B] + 1 && (Substitute\ hypothesis) \\ &= A\,2^{n+1} + (2B + 1) && \\ &= A\,2^{n+1} + B && (equate\ to\ conclusion) \end{aligned}$$

To make the latter equality, we equate term-by-term.

$$\boxed{2B + 1 = B}$$

So we have two equations for *A* and *B.*

$$\begin{cases} 2A + B = 1 \\ 2B + 1 = B \end{cases}$$

We get $B = -1$ and $A = 1$. So we proved that

$$f(n) = A\,2^n + B$$
$$f(n) = 2^n - 1.$$

Computing Power by Repeated Multiplications

Given a real number X and an integer n, let us see how to compute X^n by repeated multiplications. The following simple loop computes the power, but is very inefficient because the power goes up by 1 by each multiplication.

```
$T = X$
$for\ i = 2\ to\ n$
    $T = T * X$
```

The algorithm is made much more efficient by repeated squaring, thus doubling the power by each multiplication. Suppose $n = 2^k$ for some integer $k \geq 1$.

```
$T = X$
$for\ i = 1\ to\ k$
    $T = T * T$
```

Now, let us see how to generalize this algorithm for any integer *n*. First consider a numerical example. Suppose we want to compute X^{13}, where the power in binary is 1101. Informally, the computation may be done as follows.

1. Compute $X2 = X * X$
2. Compute $X3 = X2 * X$
3. Compute $X6 = X3 * X3$
4. Compute $X12 = X6 * X6$
5. Compute $X13 = X12 * X$.

This algorithm may be formalized non-recursively, with some effort. But a recursive implementation makes the algorithm much simpler and more eloquent.

```
real Power (real $X$, int $n$) { // It is assumed that $n > 0$.
if ($n == 1$)   return $X$;
$T = Power\ (X, \lfloor n/2 \rfloor)$;
$T = T * T$;
If ($n\ mod\ 2 == 1$)
   $T = T * X$;
return $T$ }
```

Let $n = 2m + r$, where $r \in \{0,1\}$. The algorithm first makes a recursive call to compute $T = X^m$. Then it squares T to get $T = X^{2m}$. If $r = 0$, this is returned. Otherwise, when $r = 1$, the algorithm multiplies T by X, to result in $T = X^{2m+1}$.

Analysis

Let $f(n)$ be the worst-case number of multiplication steps to compute X^n. The number of multiplications made by the recursive call is $f(\lfloor n/2 \rfloor)$. The recursive call is followed by one more multiplication. And in the worst-case, if n is odd, one additional multiplication is performed at the end. Therefore,

$$f(n) = \begin{cases} 0, & n = 1 \\ f(\lfloor n/2 \rfloor) + 2, & n \geq 2 \end{cases}$$

Let us prove by induction that the solution is as follows, where the log is in base 2.

$$f(n) = 2\lfloor \log n \rfloor$$

Induction Base, $n = 1$: From the recurrence, $f(1) = 0$. And the claimed solution is $f(1) = 2\lfloor \log 1 \rfloor = 0$. So the base is correct.

Induction step: Integer n may be expressed as follows, for some integer k.

$$2^k \leq n < 2^{k+1}$$

This means $\lfloor \log n \rfloor = k$. And,

$$2^{k-1} \leq \lfloor n/2 \rfloor < 2^k$$

Thus, $\lfloor \log \lfloor n/2 \rfloor \rfloor = k - 1$. To prove the claimed solution for any $n \geq 2$, suppose the solution is correct for all smaller values. That is,

$$f(m) = 2\lfloor \log m \rfloor, \quad \forall\, m < n$$

In particular, for $m = \lfloor n/2 \rfloor$,

$$f(\lfloor n/2 \rfloor) = 2\lfloor \log \lfloor n/2 \rfloor \rfloor = 2(k-1) = 2k - 2$$

Then,

$$f(n) = f(\lfloor n/2 \rfloor) + 2 = 2k - 2 + 2 = 2k = 2\lfloor \log n \rfloor$$

This completes the induction proof.

Misuse of Recursion

Consider again the problem of computing the power X^n by repeated multiplication. We saw an efficient recursive algorithm to compute the power with $(2\log n)$ multiplications. Now, suppose a naive student writes the following recursive algorithm.

```
real Power (real $X$, int $n$) { // It is assumed that $n > 0$.
if ($n == 1$)   return $X$;
return ($Power(X, \lfloor n/2 \rfloor) * Power(X, \lceil n/2 \rceil)$);
}
```

Although this program correctly computes the power, and it appears eloquent and clever, it is very inefficient. The reason for the inefficiency is that it performs a lot of repeated computations. The first recursive call computes $X^{\lfloor n/2 \rfloor}$ and the second recursive call computes $X^{\lceil n/2 \rceil}$, but there is a lot of overlap computations between the two recursive calls. Let us analyze the number of multiplications, $f(n)$, for this algorithm. Below is the recurrence.

$$f(n) = \begin{cases} 0, & n = 1 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + 1, & n \geq 2 \end{cases}$$

The solution is $f(n) = n - 1$. (This may be easily proved by induction.) This shows the terrible inefficiency introduced by the overlapping recursive calls.

As another example, consider the Fibonacci sequence, defined as

$$F_n = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ F_{n-1} + F_{n-2}, & n \geq 3 \end{cases}$$

It is easy to compute F_n with a simple loop in $O(n)$ time. But suppose a naïve student, overexcited about recursion, implements the following recursive program to do the job.

```
int Fib (int $n$) {
if ($n \leq 2$) return (1);
return ($\text{Fib}(n-1) + \text{Fib}(n-2)$)
```

This program makes recursive calls with a great deal of overlapping computations, causing a huge inefficiency. Let us verify that time complexity becomes exponential! Let $T(n)$ be the total number of addition steps by this algorithm for computing F_n.

$$T(n) = \begin{cases} 0, & n = 1 \\ 0, & n = 2 \\ T(n-1) + T(n-2) + 1, & n \geq 3 \end{cases}$$

We leave it as an exercise for the student to prove by induction that the solution is

$$T(n) \geq (1.618)^{n-2}$$

Note that an exponential function has an extremely large growth rate. For example, for $n = 50,\ T(n) > 1 * 10^{10}$.

Recursive Insertion-Sort

An informal recursive description of insertion sort is as follows.

To sort an array of *n* elements, where $n \geq 2,$ do:

1. Sort the first $n-1$ elements recursively.
2. Insert the last element into the sorted part. (We do this with a simple loop, without recursion. This loop is basically the same as what we had for the non-recursive version of insertion-sort earlier.)

Below is a formal pseudocode.

```
ISort (dtype $A[\,]$, int $n$) {
if ($n == 1$) return;
ISort ($A, n-1$);
$j = n - 1$;
while ($j > 0$ and $A[j] < A[j-1]$) {
   SWAP($A[j], A[j-1]$);
   $j = j - 1$;
   }
}
```

Note: Our purpose in presenting recursive insertion-sort is to promote **recursive thinking**, as it simplifies the formulation of algorithms. However, for actual implementation, recursive insertion sort is not recommended. The algorithm makes a long chain of recursive calls before any return is made. (This long chain is called **depth of recursion.)** And the long chain of recursive calls may easily cause stack-overflow at run-time when *n* is large.

Time Complexity Analysis

Let $f(n)$ be the worst-case number of key-comparisons to sort *n* elements. As we discussed for the non-recursive implementation, we know the while loop in the worst-case makes $(n-1)$ key-comparisons. So,

$$f(n) = \begin{cases} 0, & n = 1 \\ f(n-1) + n - 1, & n \geq 2 \end{cases}$$

Method 1: Solution by repeated substitution

$$\begin{aligned} f(n) &= (n-1) + f(n-1) \\ &= (n-1) + (n-2) + f(n-2) \\ &= (n-1) + (n-2) + (n-3) + f(n-3) \\ &\vdots \\ &= (n-1) + (n-2) + (n-3) + \cdots + 1 + \underbrace{f(1)}_{=0} \\ &= (n-1) + (n-2) + (n-3) + \cdots + 1 \qquad \text{(Apply arithmetic sum formula)} \\ &= \frac{(n-1)n}{2} \\ &= \frac{n^2 - n}{2} \end{aligned}$$

Method 2: Guess the solution and prove correctness by induction

Suppose we guess the solution as $O(n^2)$ and express the solution as below, in terms of some constants A, B, C to be determined.

$$f(n) = An^2 + Bn + C$$

Proof by induction:

Base, $n = 1$:

$$\begin{aligned} f(1) &= 0 \qquad &\text{(from the recurrence)} \\ &= A + B + C \qquad &\text{(from the solution form)} \end{aligned}$$

So we need $A + B + C = 0$.

Next, to prove the solution is correct for any $n \geq 2$, suppose the solution is correct for $n - 1$. That is, suppose

$$\begin{aligned} f(n-1) &= A(n-1)^2 + B(n-1) + C \\ &= A(n^2 - 2n + 1) + B(n-1) + C \end{aligned}$$

Then,

$f(n) = f(n-1) + (n-1)$ from the recurrence equation
$= A(n^2 - 2n + 1) + B(n-1) + C + (n-1)$ Use hypothesis to replace for f(n-1)
$= A\,n^2 + (-2A + B + 1)\,n + (A - B + C - 1)$
$= An^2 + Bn + C$

To make the latter equality, we equate term-by-term. That is, equate the n^2 terms, the linear terms, and the constants. So,

$$-2A + B + 1 = B$$
$$A - B + C - 1 = C$$

We have three equations to solve for A, B, C.

$-2A + B + 1 = B \quad \rightarrow \quad A = 1/2$
$A - B + C - 1 = C \quad \rightarrow \quad B = A - 1 = -1/2$
$A + B + C = 0 \quad \rightarrow \quad C = 0$

Therefore, $f(n) = \frac{n^2}{2} - \frac{n}{2}$.

Alternative Guess:

Suppose we guess the solution as $O(n^2)$ and use the definition of O( ) to express the solution with an upper bound:

$$f(n) \le A\,n^2$$

We need to prove by induction that this solution works, and in the process determine the value of the constant *A*.

Induction base, $n = 1$:

$$f(1) = 0 \le A \cdot 1^2$$

Therefore,

$$\boxed{A \ge 0}$$

Next, to prove the solution is correct for any $n \ge 2$, suppose the solution is correct for $n - 1$. That is, suppose

$$f(n-1) \le A\,(n-1)^2$$

Then,

$f(n) = f(n-1) + n - 1$
$\leq A(n-1)^2 + n - 1$
$\leq A(n^2 - 2n + 1) + n - 1$
$\leq An^2 + (-2A+1)n + (A-1)$
$\leq An^2$

To satisfy the latter inequality, we need to make the linear term ≤ 0, and the constant term ≤ 0.

$-2A + 1 \leq 0 \quad \rightarrow \quad \boxed{A \geq 1/2}$
$A - 1 \leq 0 \quad \rightarrow \quad \boxed{A \leq 1}$

The three (boxed) inequalities on *A* are all satisfied by $\frac{1}{2} \leq A \leq 1$. Any value of *A* in this range satisfies the induction proof. We may pick the smallest value, $A = \frac{1}{2}$. Therefore, we have proved $f(n) \leq n^2/2$.

Divide-and-Conquer Algorithms

The *divide-and-conquer* strategy divides a problem of a given size into one or more subproblems of the same type but smaller size. Then, supposing that the smaller size subproblems are solved recursively, the strategy is to try to obtain the solution to the original problem. We start by a simple example.

Finding MAX by Divide-and-Conquer

The algorithm divides an array of n elements, $A[0{:}n-1]$, into two halves, finds the max of each half, then makes one comparison between the two maxes to find the max of the entire array.

```
dtype FindMax (dtype $A[\,]$, int $S$, int $n$)
{ // $S$ is the starting index in the array, and $n$ is the number of elements
if ($n == 1$) return $A[S]$;
$T_1 = \text{FindMax}(A, S, \lfloor n/2 \rfloor)$;                //Find max of the first half
$T_2 = \text{FindMax}(A, S + \lfloor n/2 \rfloor, n - \lfloor n/2 \rfloor)$;   //Find max of the second half
if ($T_1 \geq T_2$)                                  // Comparison between the two maxes
    return $T_1$
else return $T_2$; }
```

Analysis (Special case when $n = 2^k$)

Let $f(n)$ be the number of key-comparisons to find the max of an array of n elements. Initially, to simplify the analysis, we assume that $n = 2^k$ for some integer k. In this case, the size of each half is exactly $n/2$, and the number of comparisons to find the max of each half is $f(n/2)$.

$$f(n) = \begin{cases} 0, & n = 1 \\ 2f(n/2) + 1, & n \geq 2 \end{cases}$$

Solution by Repeated Substitution

$$\begin{aligned}
f(n) &= 1 + 2\,f(n/2) \\
&= 1 + 2[1 + 2f(n/4)] = 1 + 2 + 4f(n/4) \\
&= 1 + 2 + 4 + 8\,f(n/8) \\
&\vdots \\
&= 1 + 2 + 4 + \cdots + 2^{k-1} + 2^k \underbrace{f\left(n/2^k\right)}_{f(1)=0} \\
&= 1 + 2 + 4 + \cdots + 2^{k-1} \qquad \text{(Use Geometric Sum formula)} \\
&= \frac{2^k - 1}{2 - 1} = 2^k - 1 \\
&= n - 1.
\end{aligned}$$

So the number of key-comparisons is the same as when we did this problem by a simple loop.

Analysis for general *n*

The recurrence equation for the general case becomes:

$$f(n) = \begin{cases} 0, & n = 1 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + 1, & n \geq 2 \end{cases}$$

It is easy to prove by induction that the solution is still $f(n) = n - 1$.
(We leave the induction proof to the student.)

Binary Search Algorithm

The sequential search algorithm works on an unsorted array and runs in $O(n)$ time. But if the array is sorted, the search may be done more efficiently, in $O(\log n)$ time, by a divide-and-conquer algorithm known as binary-search.

Given a **sorted** array $A[0{:}n-1]$ and a search key, the algorithm starts by comparing the search key against the **middle** element of the array, $A[m]$.

- If $KEY = A[m]$, then return m
- If $KEY < A[m]$, then recursively search the left half of the array.
- If $KEY > A[m]$, then recursively search the right half of the array.

So after one comparison, if the key is not found, then the size of the search is reduced to about $n/2$. After two comparisons, the size is reduced to about $n/4$, and so on. So in the worst-case, the algorithm makes about $\log n$ comparisons. Now, let us write the pseudocode and analyze it more carefully.

```
int BS (dtype $A[\,]$, int $Left$, int $Right$, dtype $KEY$) {
  // $Left$ is the starting index, and $Right$ is the ending index of the part to search.
  // If not found, the algorithm returns -1.
1  if $(Left > Right)$ return $(-1)$;      // not found
2  $m = \left\lfloor (Left + Right)/2 \right\rfloor$    // Index of the middle element
3  if $(KEY == A[m])$ return $(m)$;
4  else if $(KEY < A[m])$
5        return $(BS(A, Left, m-1, KEY))$;
6  else return $(BS(A, m+1, Right, KEY))$;
```

Let $n = Right - Left + 1 =$ Number of elements remaining in the search.
Let $f(n) =$ Worst-case number of key-comparisons to search an array of n elements.

Analysis (Special case when $n = 2^k$)

In lines 3 and 4 of the algorithm, it appears that there are 2 key comparisons before the recursive call. However, it is reasonable to count these as a single comparison, for the following reasoning:

- The comparisons are between the same pair of elements $(KEY, A[m])$.
- Computers normally have machine-level instructions where a single comparison is made, followed by conditional actions.

- It is also reasonable to imagine a high-level-language construct similar to a "case statement", where a single comparison is made, followed by several conditional cases.

We are now ready to formulate a recurrence equation for $f(n)$. Note that for the special case when $n = 2^k$, the maximum size of the recursive call is exactly $n/2$.

$$f(n) = \begin{cases} 1, & n = 1 \\ 1 + f\left(\frac{n}{2}\right), & n \geq 2 \end{cases}$$

Solution by repeated substitution:

$f(n) = 1 + f(n/2)$
$= 1 + 1 + f(n/4)$
$= 1 + 1 + 1 + f(n/8)$
$= 4 + f\left(n/2^4\right)$
$\vdots$
$= k + f\left(\frac{n}{2^k}\right)$
$= k + f(1)$
$= k + 1$
$= \log n + 1.$

Analysis of Binary Search for general *n*

For the general case, the size of the recursive call is at most $\lfloor n/2 \rfloor$. So,

$$f(n) = \begin{cases} 1, & n = 1 \\ 1 + f(\lfloor n/2 \rfloor), & n \geq 2 \end{cases}$$

We will prove by induction that the solution is

$$f(n) = \lfloor \log n \rfloor + 1$$

(The induction proof is almost identical to our earlier proof for Power.)

Induction Base, $n = 1$: From the recurrence, $f(1) = 1$. And the claimed solution is $f(1) = \lfloor \log 1 \rfloor + 1 = 1$. So the base is correct.

Induction step: Any integer n may be expressed as follows, for some integer k.

$$2^k \leq n < 2^{k+1}$$

This means $\lfloor \log n \rfloor = k$. And,

$$2^{k-1} \le \lfloor n/2 \rfloor < 2^k$$

Thus, $\lfloor \log \lfloor n/2 \rfloor \rfloor = k - 1$. To prove the claimed solution for any $n \ge 2$, suppose the solution is correct for all smaller values. That is,

$$f(m) = \lfloor \log m \rfloor + 1, \quad \forall\, m < n$$

In particular, for $m = \lfloor n/2 \rfloor$,

$$f(\lfloor n/2 \rfloor) = \lfloor \log \lfloor n/2 \rfloor \rfloor + 1 = (k-1) + 1 = k = \lfloor \log n \rfloor$$

Then,

$$f(n) = f(\lfloor n/2 \rfloor) + 1 = k + 1 = \lfloor \log n \rfloor + 1.$$

This completes the induction proof.

Mergesort

The insertion-sort algorithm discussed earlier has a basic *incremental* approach. Each iteration of the algorithm inserts one more element into the sorted part. This algorithm has time complexity $O(n^2)$. The Mergesort algorithm uses a divide-and-conquer strategy and runs in $O(n \log n)$ time.

Let us first review the easier problem of merging two sorted sequences. We consider a numerical example. Suppose we have two sorted sequences A and B, and we want to merge them into a sorted sequence C.

$$\begin{aligned} &A: \quad 4,5,8,10,12,15 \\ &B: \quad 2,3,9,10,11 \\ &C: \end{aligned}$$

We first compare the smallest (first) element of A, with the smallest (first) element of B. The smaller of the two is obviously the smallest element and becomes the first element in the sorted result, C.

$$\begin{aligned} &A: \quad 4,5,8,10,12,15 \\ &B: \quad 3,9,10,11 \\ &C: \quad 2 \end{aligned}$$

Now, one of the two sorted sequences (in this case, B) has one less element. And the merge process is continued the same way.

$$\begin{array}{ll} A: & 4,5,8,10,12,15 \\ B: & 9,10,11 \\ C: & 2,3 \end{array}$$

The merge process is continued until one of the two sequences has no more elements in it, and the other sequence has one or more elements remaining.

$$\begin{array}{ll} A: & 12,15 \\ B: & \\ C: & 2,3,4,5,8,9,10,10,11 \end{array}$$

At this point, the remaining elements are appended at the end of the sorted result without any further comparisons.

$$\begin{array}{ll} A: & \\ B: & \\ C: & 2,3,4,5,8,9,10,10,11,12,15 \end{array}$$

Let $M(m,n)$ be the worst-case number of key comparisons to merge two sorted sequences of length m and n. Then,

$$\boxed{M(m,n) = m + n - 1}$$

The reasoning is simple. With each comparison, one element is copied into the sorted result. So, after at most $m + n - 1$ comparisons, only one element will remain in one of the sorted sequences, which requires no further comparison.

What is the best-case number of key-comparisons? It is $\min(m,n)$. The best-case happens if all elements of the shorter sequence are smaller than all elements of the longer sequence.

A special case of the merge problem is when the two sorted sequences are of equal length. In this case, the worst-case number of comparisons is

$$M\left(\frac{n}{2},\frac{n}{2}\right) = n - 1.$$

The total time of the merge is $O(n)$, which mean $\leq Cn$ for some constant C.

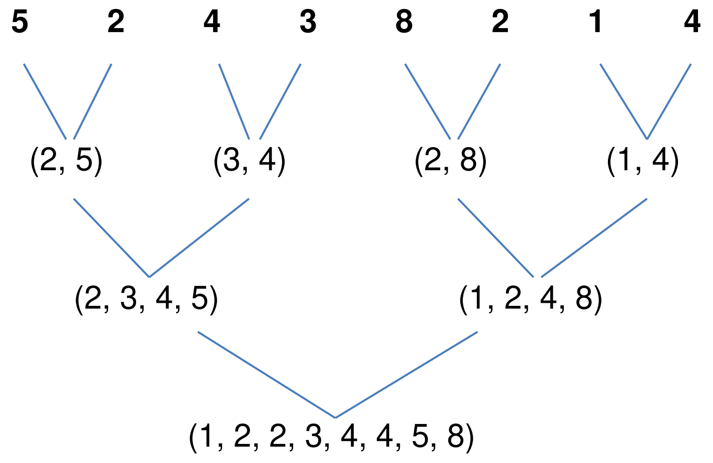
Below is the pseudocode for merging two sorted sequences $A[1{:}m]$ and $B[1{:}n]$ into the sorted result $C[1{:}m+n]$.

```
Merge (dtype $A[\,]$, int $m$, dtype $B[\,]$, int $n$, dtype $C[\,]$) {
// Inputs are sorted arrays $A[1{:}m]$ and $B[1{:}n]$. Output is sorted result $C[1{:}m+n]$.
$i = 1$;  //Index into array A
$j = 1$;  //Index into array B
$k = 1$;  //Index into array C
while $(m \geq i \; and \; n \geq j)${
   if $(A[i] \leq B[j])$
      { $C[k] = A[i]$;   $i = i + 1$};
   else
      {$C[k] = B[j]$;   $j = j + 1$};
   $k = k + 1$;
   }
while $(m \geq i)$     //Empty remaining of array A
      { $C[k] = A[i]$;   $i = i + 1$} ;
while $(n \geq j)$     //Empty remaining of array B
      {$C[k] = B[j]$;   $j = j + 1$};
}
```

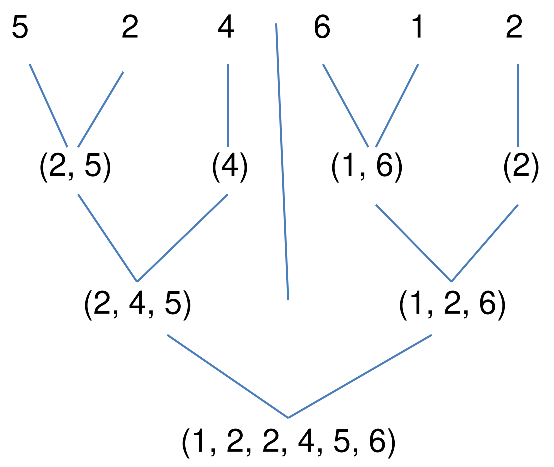
We are now ready to discuss the Mergesort algorithm, which uses a divide-and-conquer technique, and sorts a random array of n elements from scratch.

Mergesort: To sort n elements, when $n \geq 2$, do:
- Divide the array into two halves;
- Sort each half recursively;
- Merge the two sorted parts.

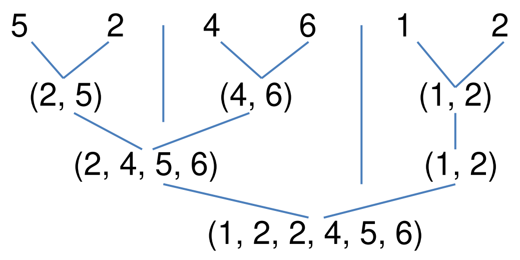
Let us consider a numerical example of Mergesort for $n = 2^3 = 8$. To sort 8 elements, they are divided into two halves, each of size 4. Then each 4 elements are divided into two halves, each of size 2. So at the bottom level, each pair of 2 is merged. At the next level, two sorted sequences of length 2 are merged into sorted sequences of length 4. At the next level, two sorted sequences of length 4 are merged into a sorted 8.



Next, consider an example of Mergesort for $n = 6$. The recursive Mergesort divides the array into two halves, each of size 3. To sort each 3, they are divided into 2 and 1.



The Mergesort algorithm may also be implemented non-recursively. At the bottom level, each pairs of 2 are sorted. Then, pairs of length 2 are merged to get sorted sequences of length 4, and so on. Below is the non-recursive implementation for the last example.



Analysis of Mergesort (Special case when $n = 2^k$)

Let $T(n)$ be the total worst-case time to sort n elements (by recursive Mergesort). The worst-case time to recursively sort each half is $T(n/2)$. And the time to merge the two sorted halves is $O(n)$, which means $\le cn$ for some constant c. Therefore,

$$T(n) \le \begin{cases} 2\,T\left(\frac{n}{2}\right) + c\,n, & n \ge 2 \\ d, & n = 1 \end{cases}$$

Solution by repeated substitution:

$T(n) = cn + 2\,T\left(\frac{n}{2}\right)$

$\le cn + 2(c\frac{n}{2} + 2\,T\left(\frac{n}{4}\right))$

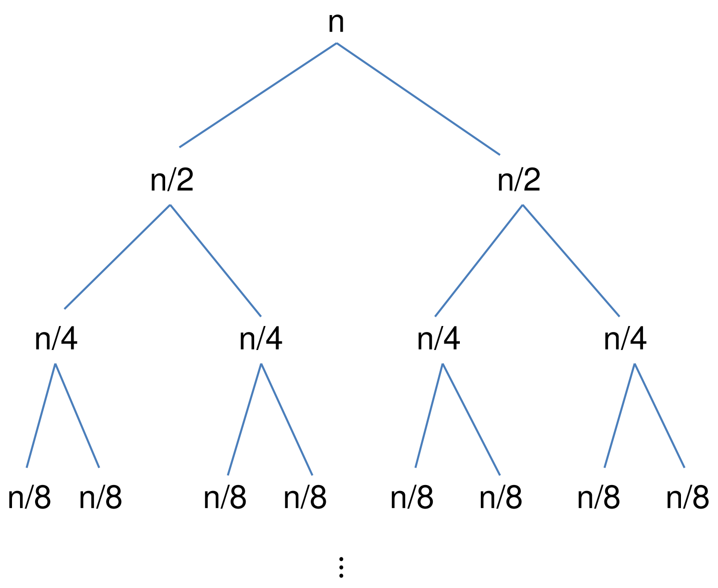
$$\le cn + cn + 4\,T\left(\frac{n}{4}\right)$$
$$\le cn + cn + cn + 8T\left(\frac{n}{8}\right)$$
$$\le 3cn + 2^3 T\left(\frac{n}{2^3}\right)$$
$$\vdots$$
$$\le kcn + 2^k T\left(\frac{n}{2^k}\right)$$
$$\le kcn + 2^k T(1)$$
$$\le kcn + d\,2^k$$
$$\le cn\log n + d\,n$$

Therefore, $T(n)$ is $O(n\log n)$.

Note: When the recurrence is $T(n) \le \cdots$, rather than strict equality, the solution simply becomes $T(n) \le \cdots$. For this reason, we often express the recurrence simply with equality, having in mind that the right side is an upper bound for $T(n)$.

Guess the solution and prove correctness by induction

To arrive at an initial guess, let us consider the merge tree, shown below. At the top level, the algorithm merges $(n/2)$ and $(n/2)$, which costs at most cn time. At the next level, to merge each $(n/4, n/4)$ pair costs $cn/2$. Since there are 2 such pairs, the total cost at this level is $2 \cdot c\,n/2$, thus a total of cn. In summary, the cost of merging at each level of tree is cn time. And there are about $\log n$ levels. (The exact number is not needed.) Therefore, the total costs is $\mathrm{O}(n\log n)$.



$$M\left(\frac{n}{2},\frac{n}{2}\right) = Cn$$

$$2 \cdot M\left(\frac{n}{4},\frac{n}{4}\right) = 2 \cdot \frac{cn}{2} = cn$$

$$4 \cdot M\left(\frac{n}{8},\frac{n}{8}\right) = 4 \cdot \frac{cn}{4} = cn$$

We concluded in our estimate that the total time is $O(n \log n)$. Based on this, there are several possibilities for guessing the solution form.

- $T(n) \leq A\, n \log n + Bn$
- $T(n) \leq A\, n \log n$

We leave the induction proof to the student.

Generalization of time analysis for any integer size n

The recurrence equation for the general case may be expressed as follows.

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + c\, n, & n \geq 2 \\ d, & n = 1 \end{cases}$$

It is easy to prove by induction that the solution is

$$T(n) \leq cn\lceil \log n \rceil$$

The proof is left to the student as exercise.