3 Recursion

Recursion is a recurring theme in mathematics and computer science. In this chapter we look at *induction* as a way of proving the correctness of some algorithms and *divide-and-conquer* as a design technique.

3.1 Induction

Induction is the most widely-used technique in proving propositions in computer science and software engineering. One reason for this is the vast number of recursive data structures and algorithms.

Suppose $p(n)$ is a logical statement (a claim that could be true or false) involving a natural number $n \in \mathbb{N}$. In mathematical logic parlance, p is called a *predicate*. The principle of induction states that

$$(p(n_0) \wedge (\forall k \geq n_0 : p(k) \Rightarrow p(k+1))) \Rightarrow (\forall n \geq n_0 : p(n)) \ .$$

This means that if

- a property p holds for some number n_0; and
- for all numbers k greater than (or equal to) n_0: if p holds for k then it also holds for $k+1$,

then we can conclude that the property p holds for all numbers greater than (or equal to) n_0.

Example 3.1. Suppose $p(n)$ states that $2n - 5 > 0$. It can be seen that $p(3)$ holds (because $2 \times 3 - 5 = 1 > 0$).

Also if $p(k)$ is true, that is $2k-5 > 0$, then $2k-5+2 > 0$ is true, which means $2(k+1)-5 > 0$ is true. Thus, $p(k+1)$ is true. In other words, $p(k) \Rightarrow p(k+1)$.

From these two, based on the principle of induction, we conclude that $p(n)$ holds for all $n \geq 3$.

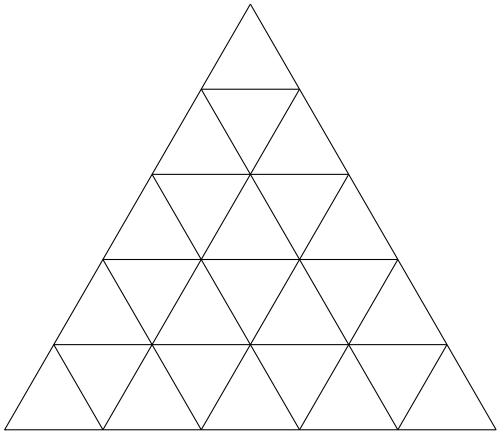
Inductive proofs

In order to prove $\forall n \geq n_0 : p(n)$, we need to prove two things:

1. **basis (base case)**: prove $p(n_0)$ is true; that is, prove that the statement holds for some natural number n_0 (usually 0 or 1, but it could be any other natural number).
2. **inductive step**: prove $[p(k) \Rightarrow p(k+1)]$ is true; that is, prove that if the statement is true for any k, it will also be true for $k+1$.

A common approach to prove the inductive step is to assume that the statement is true for some k (inductive hypothesis) and then use this assumption to show that the statement is also true for $k+1$.

Example 3.2. Consider a large triangle that contains smaller triangles with the following pattern:



Let $p(n)$ be the following statement: "A large triangle of size n contains n^2 small triangles." Prove that $p(n)$ is true for every natural number $n \geq 1$.

Proof. **basis**: $p(1)$ is true.

inductive step: [Show that if $p(k)$ is true, then $p(k+1)$ will be true as well.] Assume a large triangle of size k has k^2 small triangles. Then a large triangle of size $k+1$ has $k^2 + k + k + 1 = k^2 + 2k + 1 = (k+1)^2$ small triangles; that is, $p(k+1)$ is true. □

Before we look at another example, answer the following question.

Question 3.1. Does the expressions $\sum_{i=1}^{n} i$ have a closed form?

Example 3.3. Consider the function described below. The function takes a natural number n as input. Prove that SUM(n) returns $n(n+1)/2$.

procedure SUM(n)
  $s \leftarrow 0$
  for i **from** 1 **to** n
    $s \leftarrow s + i$
  return s

Proof. Let $p(n)$ be the following statement: "the value of s after the n-th iteration of the loop is $\frac{n(n+1)}{2}$". Since SUM(n) returns s after n iterations of the loop, to prove that SUM(n) $= n(n+1)/2$, it suffices to prove that $p(n)$ is true for every natural number n.

basis: $p(0)$ is true because before the first iteration of the loop the value of s is 0 (which equals $0(0+1)/2$).

inductive step: if $p(k)$ is true, that is, the value of s after k iterations is

$$\frac{k(k+1)}{2} \ ,$$

then after one more iteration the value of s is

$$\frac{k(k+1)}{2} + (k+1) = \frac{k(k+1) + 2(k+1)}{2} = \frac{(k+1)(k+2)}{2} \ .$$

That is, after $k+1$ iterations the value of s is $(k+1)((k+1)+1)/2$. This means that $p(k+1)$ is true. Therefore, $p(k) \Rightarrow p(k+1)$.

The truth of the basis, $p(0)$, together with the inductive step imply that $p(n)$ is true for every natural number n. □

Example 3.4. Obtain a closed form for the expression $\sum_{i=0}^{n} 2^i$.

$$\begin{aligned}
\sum_{i=0}^{n} 2^i &= 2^0 + 2^1 + 2^2 + \cdots + 2^n \\
&= 2^0 + 2^1 + 2^2 + \cdots + 2^n + 1 - 1 \\
&= 1 + 2^0 + 2^1 + 2^2 + \cdots + 2^n - 1 \\
&= 2^0 + 2^0 + 2^1 + 2^2 + \cdots + 2^n - 1 \\
&= 2 * 2^0 + 2^1 + 2^2 + \cdots + 2^n - 1 \\
&= 2^1 + 2^1 + 2^2 + \cdots + 2^n - 1 \\
&= 2 * 2^1 + 2^2 + \cdots + 2^n - 1 \\
&= 2^2 + 2^2 + \cdots + 2^n - 1 \\
&\vdots \\
&= 2^n + 2^n - 1 = 2 * 2^n - 1 = 2^{n+1} - 1
\end{aligned}$$

The equality $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ can be proved by induction.

△

Example 3.5. Prove the correctness of the following algorithm.

procedure INSERTION-SORT(A)
  for i **from** 0 **to** $length(A) - 1$
    $a \leftarrow A[i]$
    $j \leftarrow i$
    while $(j \geq 1) \wedge (A[j-1] > a)$
      $A[j] \leftarrow A[j-1]$
      $j \leftarrow j - 1$
    $A[j] \leftarrow a$

Proof. Let $p(n)$ be the following statement: "after one complete iteration of the for-loop, where i is n, the sub-array $A[0..n]$ is in sorted order". We use induction to prove the correctness of $p(n)$, for $n \geq 0$.

basis: $p(0)$ is true because the first iteration of the for-loop (in which $i = 0$), effectively does nothing. Note that the while-loop is not executed. This is because a sub-array of size one is always sorted[1].

inductive step: Let us assume $p(k)$ holds. Then in the next iteration, when $i = k + 1$, the elements of the sub-array $A[0..k + 1]$, starting from the k-th element, are shifted to the right so that the $(k + 1)$-th element is inserted at the correct position[2]. Finally, the last line of the body of the loop inserts the element at the correct position. Therefore $p(k + 1)$ holds. □

[1] In fact, the for-loop of a typical Insertion Sort starts from index 1 (the second element of the array). We modified the algorithm here to make the proof straightforward. The modification does not change the correctness or the asymptotic efficiency of the algorithms.

[2] This is a rather informal proof. A more formal proof would use induction for the inner while-loop as well.

3.2 Divide and Conquer

Divide and Conquer (D&C) is an algorithm design technique. In this technique a (large) problem is solved recursively in three steps:

i) **Divide** the problem into a number of subproblems that are smaller instances of the same problem.

ii) **Conquer** the subproblems by solving them recursively.

iii) **Combine** the solutions to the subproblems into the solution for the original problem.

When the subproblems are large enough, they are solved recursively. This is the **recursive case** of the algorithm.

For small enough subproblems, a direct solution (without subdividing) must exist. This forms the **base case** of the algorithm.

Example 3.6. Merge sort is a D&C algorithm defined as follows. It sorts the sub-array $A[l \,..\, r]$.

```
1  procedure MERGE-SORT(A, l, r)
2    if l < r
3      m ← ⌊(l+r)/2⌋
4      MERGE-SORT(A, l, m)
5      MERGE-SORT(A, m + 1, r)
6      MERGE(A, l, m, r)
```

For each subproblem (sub-instance), the three steps of D&C in this algorithm are:

- Divide: divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
- Conquer: sort the two subsequences recursively using merge sort.
- Combine: merge the two sorted subsequences to produce the sorted sequence.

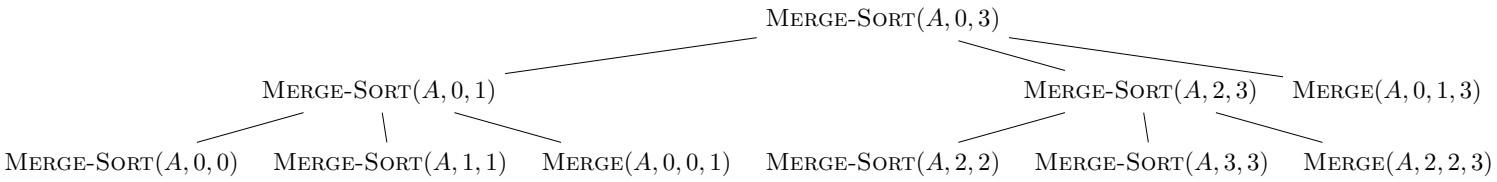
Question 3.2. What is the base case of MERGE-SORT?

Invocation trees

The *invocation tree* of a procedure shows the calls (invocations) that are made by the procedure. It is recursively defined as follows.

1. The root of the tree corresponds to the call to the procedure (with the required arguments).
2. The children of the root (if any) from left to right are invocation trees corresponding to calls made by the root.

Example 3.7. Draw the invocation tree of MERGE-SORT$(A, 0, 3)$.



△

Question 3.3. Consider an invocation tree. In which order are the calls initiated? In which order are they completed?

Analysis of D&C Algorithms

The first step in determining the running time of a recursive algorithm is to write its **recurrence** equation which describes the running time for a given instance size in terms of the running times of smaller instances.

In general, the running time of a recursive algorithm can be expressed as a function with two cases:

1. Time to solve the smallest problem (base case) = $\Theta(1)$.
2. Time to solve a bigger problem (recursive case) = time to divide + time to conquer + time to combine

Example 3.8. What is the running time recurrence equation of merge sort?

Let $T(n)$ be the time to sort an array with n elements (that is, to solve an instance of size n). Time to divide is $\Theta(1)$. Time to conquer is $2T(n/2)$. Time to combine (merge) is $\Theta(n)$. Therefore,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ \Theta(1) + 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1. \end{cases}$$
$$= \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1. \end{cases}$$

△

Note that the coefficients that appear in the recurrence equation usually reflect the number of subproblems a problem is divided to.

The time complexity of a recursive algorithm is not directly visible from its recurrence equation. We need to "solve" the recurrence equation – that is, find a closed-form expression for the running time.

Solving Recurrence Equations using Recurrence Trees

We use recurrence trees (also called recursion trees) to find a closed-form expression (or asymptotic bound) for the complexity of a recursive algorithm.

A recurrence tree is obtained by expanding the corresponding recurrence equation and has the following properties:

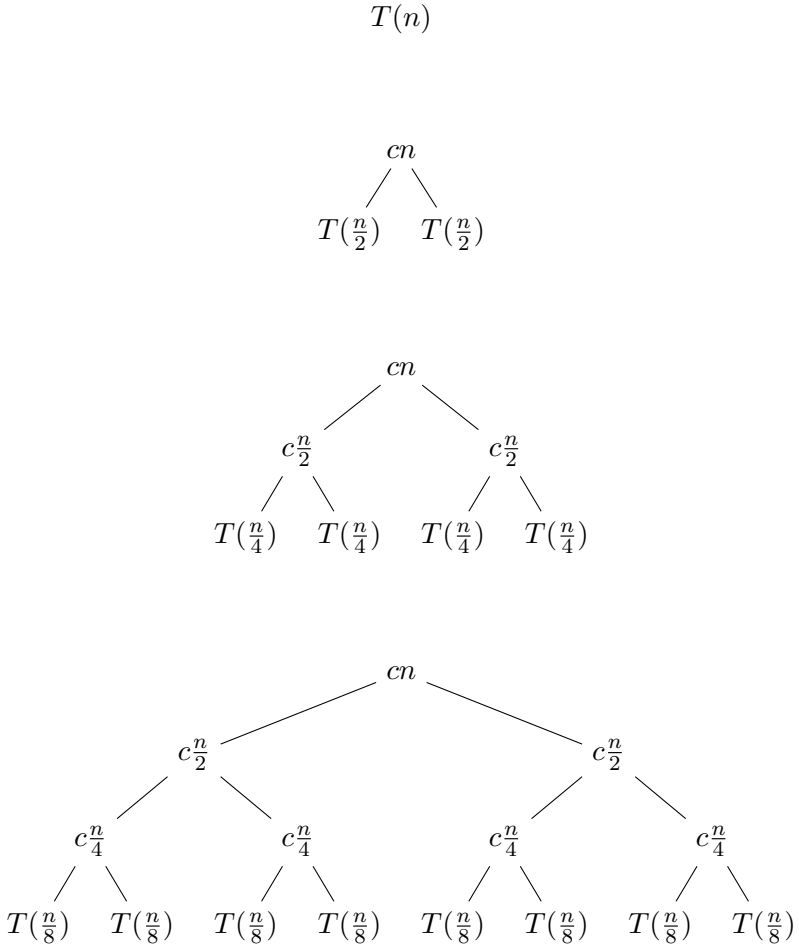
- it is a tree;
- each node is an expression for a cost (running time);
- the number of children of a node is the number of subproblems a problem is divided into.
- each subtree corresponds to a subproblem;
- the size of the subproblem of a subtree is smaller than that of its parent (a consequence of D&C design technique);
- each subtree in the tree is a recurrence tree for a subproblem.
- the cost expression for non-leaf nodes is the sum of divide and combine times;
- the sum of all nodes (including the root) of a subtree is the cost of the corresponding subproblem;

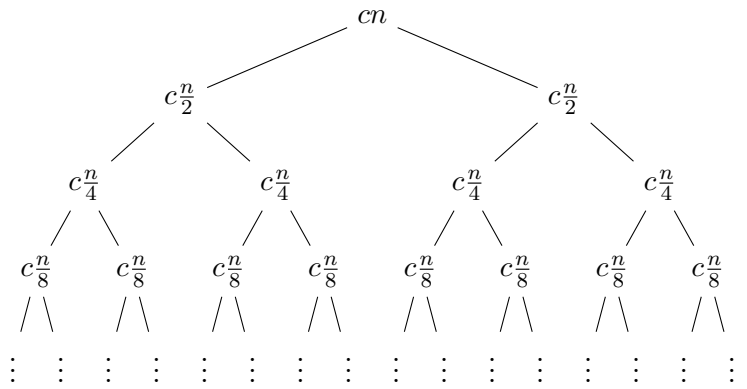
Therefore the cost of a problem of size n at the root of the tree is the sum of all nodes in the tree. An easy method to find the overall sum is to first sum the costs at each *depth* of the tree and then find the sum of these sums.

The tree is expanded downwards until the base case subproblems are reached. The entire tree need not be drawn. The purpose of the tree is to find a closed-form expression for the cost.

Example 3.9. Draw the recurrence tree of the merge sort algorithm. What are the sums at each depth? What is the sum of the entire tree?

The recurrence tree is shown at various stages of expansion.





- The number of nodes at depth d is 2^d.
- The cost expression of each node at depth d is $c\frac{n}{2^d}$.
- The sum at each depth is $2^d \times c\frac{n}{2^d} = cn$.
- We reach the base case when $\frac{n}{2^d} = 1$; that is, when $d = \log n$.
- The sum of all nodes is $T(n) = cn \log n = \Theta(n \log n)$.

△

Example 3.10. Give a D&C algorithm for solving a Tower of Hanoi problem of size (height) n. Express the corresponding recurrence equation and solve it using recurrence trees.

procedure MOVE-TOWER(*height*, *source*, *destination*, *auxiliary*)
  if *height* ≥ 1
    MOVE-TOWER(*height* -1, *source*, *auxiliary*, *destination*)
    MOVE-DISK(*source*, *destination*)
    MOVE-TOWER(*height* -1, *auxiliary*, *destination*, *source*)

The recurrence equation is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n-1) + \Theta(1) & \text{if } n > 1. \end{cases}$$

The recurrence tree is ...

- The number of nodes at depth d is 2^d.
- The cost expression of each node at depth d is c.
- The sum at each depth is $c2^d$.
- We reach the base case when $n - d = 1$, that is, when $d = n - 1$.
- The sum of the tree is $T(n) = \sum_{d=0}^{n-1} c2^d = c(2^n - 1) = \Theta(2^n)$.

Note that similarly we can define a recurrence equation for the number of disks that need to be moved as follows:

$$M(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2M(n-1) + 1 & \text{if } n > 1. \end{cases}$$

The recurrence tree would be identical except that c is replaced with 1. Therefore the number of disks that need to be moved in order to solve a Tower of Hanoi problem of height n is equal to $2^n - 1$.

△

Example 3.11. Binary search is a D&C algorithm. Given an array A of size n that is sorted based on a function key and a value x that we are searching for, it returns an index i such that $key(A[i]) = x$. It returns NULL if there is no such value.

```
1  procedure BINARY-SEARCH(A, key, x, l, r)
2    if l > r
3      return NULL
4    m ← ⌊(l+r)/2⌋
5    if x < key(A[m])
6      return BINARY-SEARCH(A, key, x, l, m − 1)
7    else if x = key(A[m])
8      return m
9    else
10     return BINARY-SEARCH(A, key, x, m + 1, r)
```

Give a tight bound on the worst-case time complexity of this algorithm using recurrence trees.

The recurrence equation is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(\frac{n}{2}) + \Theta(1) & \text{if } n > 1. \end{cases}$$

The recurrence tree is ...

- The number of nodes at depth d is 1.
- The cost expression of each node at depth d is c.
- The sum at each depth is c.
- The base case is reached when $\frac{n}{2^d} = 1$, that is, when $d = \log n$.
- The sum of the tree is $T(n) = c \log n = \Theta(\log n)$.

△

Fast Exponentiation

Logarithms can be used to evaluate exponentiations efficiently. The process is efficient because logarithms reduce exponentiation to multiplication. The drawback is limited numerical precision.

In some problems (such as primality testing and cryptography) we need the exact value of an expression like a^n where n can be quite large. A simple algorithm is to multiply a by itself n times. The time complexity of the simple approach is linear (that is, $\Theta(n)$).

Using D&C techniques, this can be computed more efficiently. Observe that if n is even, then $a^n = a^{\frac{n}{2}} a^{\frac{n}{2}}$. If n is odd, then $a^n = a^{\lfloor \frac{n}{2} \rfloor} a^{\lfloor \frac{n}{2} \rfloor} a$. In either case the problem can be reduced to a subproblem half its size. Therefore the following algorithm evaluates the exponentiation efficiently.

```
1  procedure POWER(a, n)
2    if n = 0
3      return 1
4    else
5      p ← POWER(a, ⌊n/2⌋)
6      if n is even
7        return p × p
8      else
9        return a × p × p
```

Question 3.4. What is the time complexity of the fast exponentiation algorithm?

More examples

Example 3.12. Consider the following recursive implementation of the factorial function:

procedure FACTORIAL(n)
  if $n = 0$
    return 1
  else
    return $n \times$ FACTORIAL($n - 1$)

a) Without a formal analysis, do you think there is any difference between the time complexity of this algorithm and an iterative approach?

b) Is there any difference between worst-case and best-case time complexity of the algorithm?

c) How does the recurrence tree look like?

Example 3.13. Let F_n denote the n-th Fibonacci number; that is, $F_0 = 0$ and $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. You can use induction to show that the following holds:

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n .$$

How do you think this equation can be used to devise an efficient D&C algorithm to compute F_n with time complexity $O(\log n)$?