# Chapter 8
# Recursion

## Proofs by Mathematical Induction

- Carried out in two steps:
    1. Show the theorem is true for the smallest case.
    2. Assume the theorem is true for the $k^{th}$ case. Show the theorem is true for $(k+1)^{st}$ case.
- Look at proof of $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Look at proof of $\sum_{i=1}^{n} 2i+1 = n(n+2)$

## How Recursion Works

- Information must be kept between function calls.
    - Values of parameters
    - Values of local variables
    - Return address
- The information is kept in an *activation record*.
- Activation records are kept on a stack.
    - When a new function is called, a new activation record for the function is pushed on the stack.
    - When a function returns (terminates), the activation record is popped off the stack.

### Too Much Recursion

- Can you use recursion too often?
    - Fibonacci numbers – $F_i = F_{i-1} + F_{i-2}$
- Loops vs. recursion
- There are many applications where recursion is the way to go.
    - Trees
    - Divide and conquer algorithms
    - Dynamic programming

## Divide and Conquer Algorithms

### Introduction

- The method is called a divide and conquer because you:
  - Divide the problem into pieces.
  - Solve the pieces recursively.
  - Put the pieces back together.
- In analyzing the divide and conquer technique, we must consider:
  - The time to divide the problem into pieces.
  - The time to solve each of the pieces.
  - The time to put the results back together.

## Counterfeit Coins

- The problem:
  - You have only a balance scale.
  - You are given 16 coins, one of which is counterfeit. The counterfeit coins are lighter.
  - Identify the counterfeit coin in the minimum number of weighings.
- Technique 1:
- Technique 2 (divide and conquer):
- Technique 3:
- In this case:

## Gold Nuggets

- Example – Suppose you are trying to reduce the running time for finding the largest and smallest element in an array.
- Method 1:

- Method 2:

- In divide and conquer we must consider:
  - The time to divide the problem into pieces – no work.
  - The time to solve the pieces – consider the base case separately.
  - The time to put the results back together – takes two compares, right?
- When does this breaking in half end? I.e., what is our boundary condition?
- We could use our formula to figure complexity, but we want to know more that just big Oh – we want to know precisely if it is better.

- We use the substitution technique. (You may have learned something you like better in your discrete math class.) We set up a series of equations. Since we would never stop for infinite *n*, let *n* = 32 to see the pattern, and generalize for any *n*. Let us also write *n* as a power of two (as things work out best if we can keep dividing by two at each stage). Assume, $n = 2^b$.
  $T(n) = 2 \times T(2^{b-1}) + 2$
  $T(n) = 2 \times (2 \times T(2^{b-2}) + 2) + 2$
  $T(n) = 2 \times (2 \times (2 \times T(2^{b-3}) + 2) + 2) + 2$
  $T(n) = 2 \times (2 \times (2 \times (2 \times T(2^{b-4}) + 2) + 2) + 2) + 2$
  Now by substituting each equation into the previous equation we have
  $T_{32} = 2 + 2^2 + 2^3 + 2^4 + 2^4$
  The first part looks like a geometric sequence, but the last term does not. Notice, since $n = 32 = 2^5$ that $2^4 = n/2$.
  $T_{32} = 2 + 2^2 + 2^3 + 2^4 + 2^4 = \sum_{i=1}^{4} 2^i + n/2$
  Using our geometric sum technique
  $\sum_{i=1}^{b-1} 2^i = 2^b - 2$
  Thus,
  $T_n = 2^b - 2 + n/2 = 2 \times 2^{b-1} - 2 + n/2 = 2(n/2) - 2 + n/2 = 3n/2 - 2$
  Note this is an improvement over method 1 by about 25%.

## Selection (Order Statistics)

- Finding the $k^{th}$ smallest element in an array.
- Could do complete sort and look at $k^{th}$ location. Overkill.
- Like a quicksort – but only do one half.
- $O(n)$.

## Priority Queues

- Want to delete from queue according to priority.
  - *Max priority queue* – delete the greatest.
  - *Min priority queue* – delete the least.
- Insert normally, but delete based on priority.
- We can implement priority queues using binary search trees, ordered or unordered lists, ordered or unordered arrays, etc.
- Let assume a linked list implementation.
  - Unordered
    - Insert –.
    - Delete –.
  - Ordered
    - Insert –.
    - Delete –.
- We may get some gains if we just partially sort the data. This is true especially if we only delete a small fraction of the information. Then we don't incur the overhead of sorting the entire set of data.
- This partially ordered data is how a heap helps us.

## Heaps and Heapsort

- Heap – complete binary tree in which each node is smaller than its parent.
  - Is this the same thing as a binary search tree (BST)?
- The binary tree for the heap is implemented as an array. This allows us easy access to children and parents as seen in the previous chapter.
- Used as a priority queue – regular insertion, priority deletion.
- *Insertion* – put the new node at the first empty position, and sift the element up (if needed). See Figure 1.
  - 

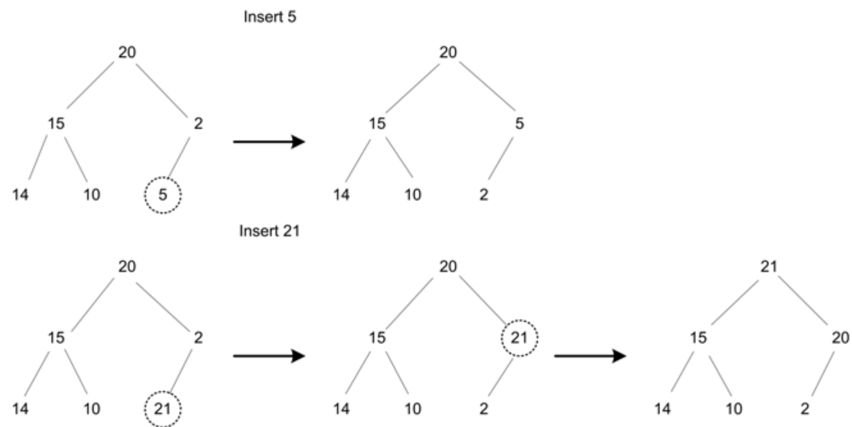


**Figure 1 Insertion into a Heap**

- *Deletion* – select root. Swap with last position (which will no longer be part of queue), and sift-down. See Figure 2.
    - 

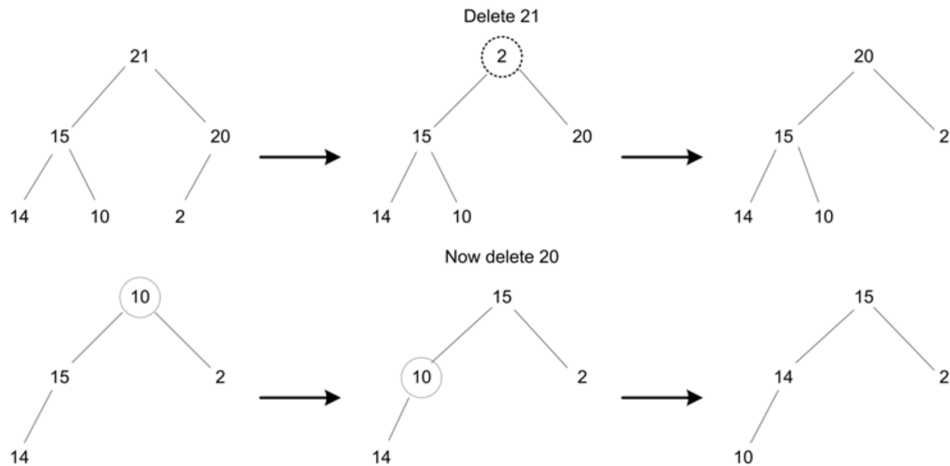


**Figure 2 Deletion from Heap**

- *Initialization* –
- Can use to sort:
    - Analysis: $O(n \log n)$ but about twice as slow as quicksort.
    - $O(n)$ to initialize, $O(\log n)$ to delete, and need to delete $n$ items. Thus $O(n \log n)$.
    - However, exhibits same complexity in worst case.
- This is *an implicit data structure* – no special structure need for heap except the binary tree. Thus no space overhead.

# Dynamic Programming

## Introduction

- The name dynamic programming was coined in 1957 by Richard Bellman to describe a type of optimal control problem.

- The term programming is used to mean a “series of choices” like programming your VCR and has nothing to do with programming a computer.
- The term dynamic conveys the idea that choices may depend on the current state, rather than being decided ahead of time.
- In dynamic programming we do all the possible subproblems early and store them somewhere. Thus, when we have a need for a subproblem, we don’t run the risk of having to recompute it.

## Example

- Consider the Fibonacci sequence: $fib(i) = fib(i-1) + fib(i-2),\ where\ \ fib(1) = fib(2) = 1$.
- There is much duplication of effort.
- Usually, when we break the problem into smaller problems, they are half as big.
- In this case the problems are only one smaller! This makes the problem exponential as it almost doubles the work when problem size decreases by only one.
- In computing *fib(7)*, count the number of cases evaluated more than once. This is seen in Figure 3.

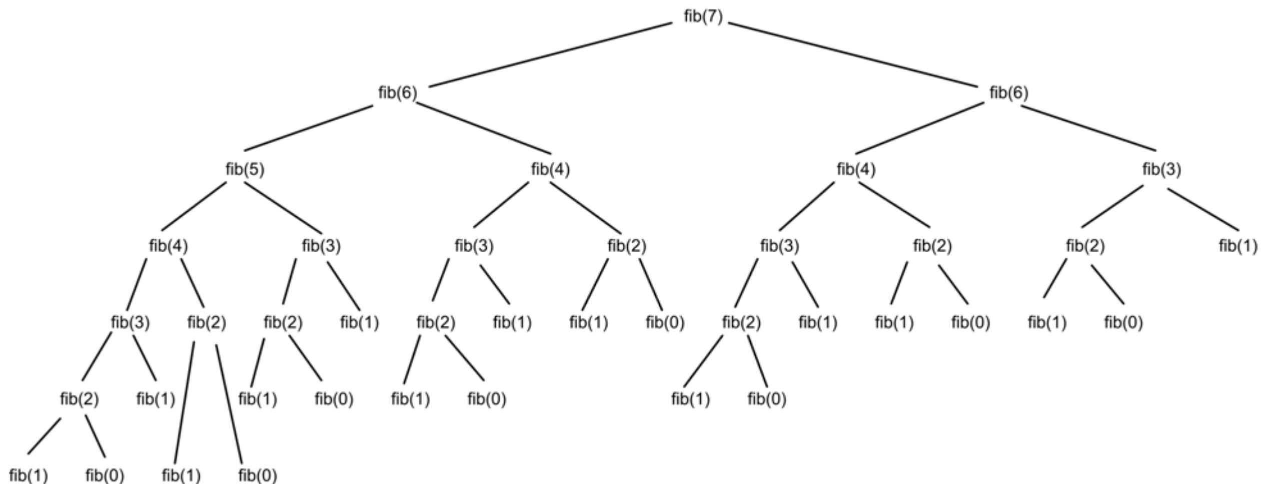


**Figure 3 Fibonacci Repetition**

- The number of times we compute each subproblem is shown in Figure 4.

| Problem | Times |
|---|---|
| *fib(2)* | 8 |
| *fib(3)* | 5 |
| *fib(4)* | 3 |
| *fib(5)* | 2 |
| *fib(6)* | 1 |
| *fib(7)* | 1 |

**Figure 4 Number of Times Subproblems are Computed**

- Number of times subproblems are recomputed for fib(8) is shown in Figure 5.

| Problem | Times |
|---|---|
| *fib(2)* | 13 |
| *fib(3)* | 8 |
| *fib(4)* | 5 |
| *fib(5)* | 3 |
| *fib(6)* | 2 |
| *fib(7)* | 1 |
| *fib(8)* | 1 |

**Figure 5 Number of Subproblems for fib(8)**

- How could we do this more efficiently?
  - Save the results of smaller problems so we avoid recomputation.

## Example

- Consider change making: if we try to make change for 63 cents using a greedy algorithm, it takes 6 coins (2 quarters, 1 dime, 3 pennies).
- If have a 21 cent coin, the greedy method fails to find optimum solution. (3 21-cent coins.)
- If we try to do divide and conquer, we end up solving similar problems repeatedly.
- What if we try to find the least number of coins for each amount?
- We've seen this before – can do greedy or exhaustive. Greedy is exponential as for each coin consider both "use it" and "don't use it."
- The problem is known to be NP complete.
- We save the results of smaller problems in a table so we can reuse the result of the smaller problem.
- This table is shown in Figure 6.
  - Columns – amount to make change for.
  - Rows – use only that coin and smaller.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 5 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 | 4 | 5 | 6 |
| 7 | 1 | 2 | 3 | 4 | 1 | 2 | 1 | 2 | 3 | 2 | 3 | 4 | 5 | 2 |
| 10 | 1 | 2 | 3 | 4 | 1 | 2 | 1 | 2 | 3 | 1 | 2 | 3 | 5 | 2 |
| 25 | 1 | 2 | 3 | 4 | 1 | 2 | 1 | 2 | 3 | 1 | 2 | 3 | 5 | 2 |

**Figure 6 Reuse Table for Change**