Unit 3

Plotting Using PyLab

Install pylab into python

cmd->pip install matplotlib

Basic plots

Two basic plot types which you will find are used very often are (x,y) line and scatter plots and histograms. Some code for making these two types of plots is included in this section.

Line and scatter plots

Line plots

A common type of graph to plot is a line relating x-values to particular y-values. The code to draw the graph in below:

```
****************************************************
# lineplot.py
import numpy as np
import pylab as pl
# Make an array of x values
x = [1, 2, 3, 4, 5]
# Make an array of y values for each x value
y = [1, 4, 9, 16, 25]
# use pylab to plot x and y
pl.plot(x, y)
# show the plot on the screen
pl.show()
****************************************************
```

Scatter plots

Alternatively, you may want to plot quantities which have an x and y position. For Example, plotting the location of stars or galaxies in a field for example such as the plot.

```
# scatterplot.py
import numpy as np
import pylab as pl
# Make an array of x values
x = [1, 2, 3, 4, 5]
# Make an array of y values for each x value
y = [1, 4, 9, 16, 25]
# use pylab to plot x and y as red circles
pl.plot(x, y, ’ro’)
# show the plot on the screen
pl.show()
```

Changing the line color

It is very useful to be able to plot more than one set of data on the same axes and to be able to differentiate between them by using different line and marker styles and colours. You can specify the colour by inserting a 3rd parameter into the plot() command.

```
pl.plot(x, y, ’r’)
```

This should give you the same line as before, but it should now be red.
The other colours you can easily use are:

Character	Color
b	Blue
g	Green
r	Red
c	Cyan
m	Magenta
y	Yellow
k	Black
w	white

Changing the line style

You can also change the style of the line e.g. to be dotted, dashed, etc. Try:

```
plot(x, y, ’--’)
```

This should give you a dashed line now.

Changing the marker style

Lastly, you can also vary the marker style you use. Try:

```
plot(x, y, ’b*’)
```

This should give you blue star-shaped markers. The table below gives some more
Options for setting marker types:

’s’ square marker
’p’ pentagon marker
’*’ star marker
’h’ hexagon1 marker
’H’ hexagon2 marker
’+’ plus marker
’x’ x marker
’D’ diamond marker
’d’ thin diamond marker

Plot and axis titles and limits

It is very important to always label the axes of plots to tell the viewer what they are looking at. You can do this in python by using the commands:

```
pl.xlabel('put text here')
pl.ylabel('put text here')
```

You can make a title for your plot by:

```
pl.title('Put plot title here')
```

You can change the x and y ranges displayed on your plot by:

```
pl.xlim(x_low, x_high)
pl.ylim(y_low, y_high)
```

```
#lineplotAxis.py
import numpy as np
import pylab as pl
# Make an array of x values
x = [1, 2, 3, 4, 5]
# Make an array of y values for each x value
y = [1, 4, 9, 16, 25]
# use pylab to plot x and y
pl.plot(x, y)
# give plot a title
pl.title('Plot of y vs. x')
# make axis labels
pl.xlabel('x axis')
pl.ylabel('y axis')
# set axis limits
pl.xlim(0.0, 7.0)
pl.ylim(0.0, 30.)
# show the plot on the screen
pl.show()
```

Plotting more than one plot on the same set of axes

It is very easy to plot more than one plot on the same axes. You just need to define the x and y arrays for each of your plots and then:

```
plot(x1, y1, 'r')
plot(x2, y2, 'g')
```

```
import numpy as np
import pylab as pl
# Make x, y arrays for each graph
x1 = [1, 2, 3, 4, 5]
y1 = [1, 4, 9, 16, 25]
x2 = [1, 2, 4, 6, 8]
y2 = [2, 4, 8, 12, 16]
# use pylab to plot x and y
pl.plot(x1, y1, ’r’)
pl.plot(x2, y2, ’g’)
# give plot a title
pl.title(’Plot of y vs. x’)
# make axis labels
pl.xlabel(’x axis’)
pl.ylabel(’y axis’)
# set axis limits
pl.xlim(0.0, 9.0)
pl.ylim(0.0, 30.)
# show the plot on the screen
pl.show()
```

Histograms

Histograms are very often used in science applications and it is highly likely that you will need to plot them at some point! They are very useful to plot distributions

e.g. what is the distribution of galaxy velocities in my sample? etc. In Matplotlib you use the hist command to make a histogram. Take a look at the short macro below which makes the plot.

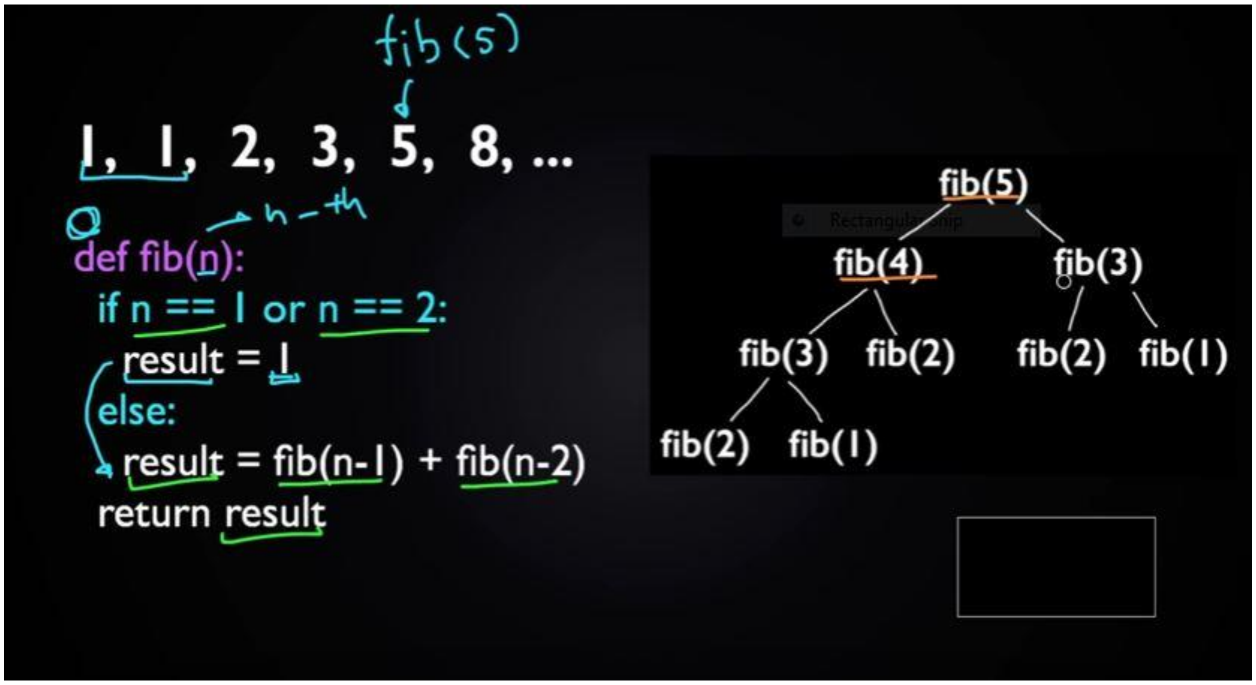
```
# histplot.py
import numpy as np
import pylab as pl
# make an array of random numbers with a gaussian distribution with
# mean = 5.0
# rms = 3.0
# number of points = 1000
data = np.random.normal(5.0, 3.0, 1000)
# make a histogram of the data array
pl.hist(data)
# make plot labels
pl.xlabel(’data’)
pl.show()
```

Dynamic programming

Dynamic programming was invented by Richard Bellman in the 1950.

Dynamic programming is a very powerful technique to solve a particular class of problems. The idea is very simple, If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again.. Shortly **'Remember your Past'**. If the given problem can be broken up in to smaller sub-problems and these smaller sub problems are in turn divided in to still-smaller ones, and in this process, if you observe some over-lapping sub problems, then its a big hint for DP. Also, the optimal solutions to the sub problems contribute to the optimal solution of the given problem.

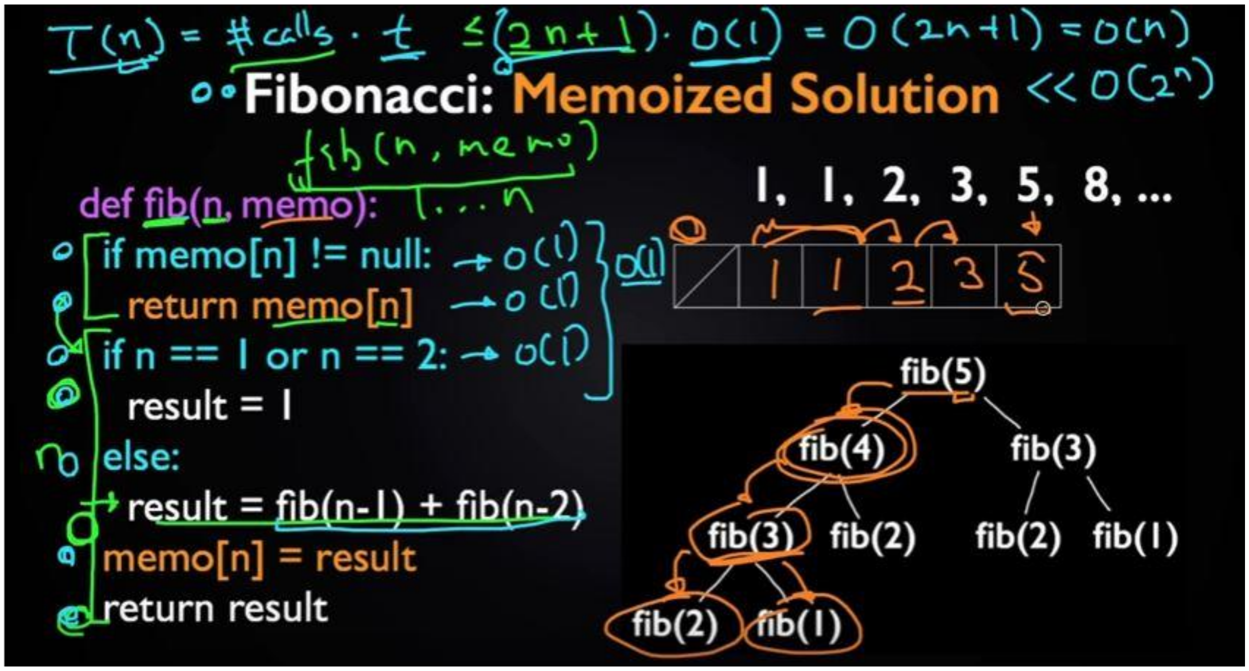
- **Fibonacci sequence revisited**



```
def fib(n):
  if n == 1 or n == 2:
    result = 1
  else:
    result = fib(n-1) + fib(n-2)
  return result
```

There are two ways of doing this.

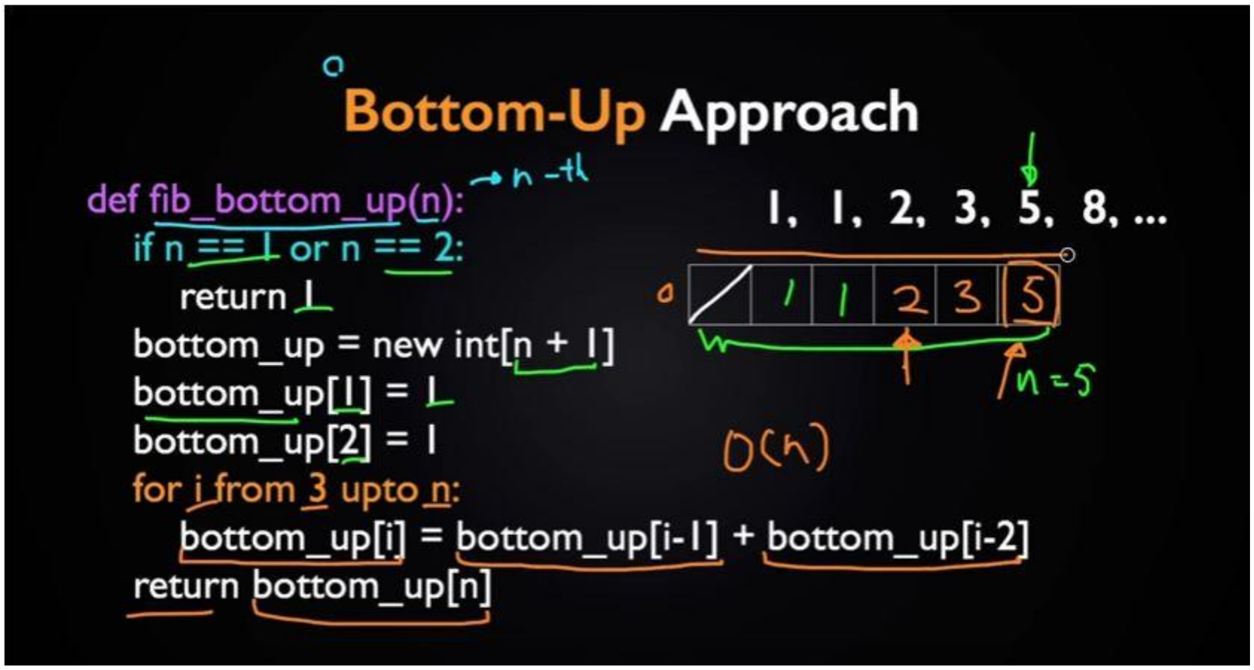
1.) Top-Down : Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as ***Memoization***.


$T(n) = \#calls \cdot t \le (2n+1) \cdot O(1) = O(2n+1) = O(n)$

Fibonacci: Memoized Solution $<< O(2^n)$

```
def fib(n, memo):
    if memo[n] != null:
        return memo[n]
    if n == 1 or n == 2:
        result = 1
    else:
        result = fib(n-1) + fib(n-2)
    memo[n] = result
    return result
```

2.) Bottom-Up : Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as ***Dynamic Programming***.



Bottom-Up Approach

```
def fib_bottom_up(n):
    if n == 1 or n == 2:
        return 1
    bottom_up = new int[n + 1]
    bottom_up[1] = 1
    bottom_up[2] = 1
    for i from 3 upto n:
        bottom_up[i] = bottom_up[i-1] + bottom_up[i-2]
    return bottom_up[n]
```

```
def fib(n):
    """Assumes n is an int >= 0
    Returns Fibonacci of n"""
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
print(fib(25))
```

While this implementation of the recurrence is obviously correct, it is terribly inefficient. Try, for example, running fib(120), but don't wait for it to complete. The complexity of the implementation is a bit hard to derive, but it is roughly O(fib(n)). That is, its growth is proportional to the growth in the value of the result, and the growth rate of the Fibonacci sequence is substantial. For example, fib(120) is 8,670,007,398,507,948,658,051,921. If each recursive call took a nanosecond, fib(120) would take about 250,000 years to finish.

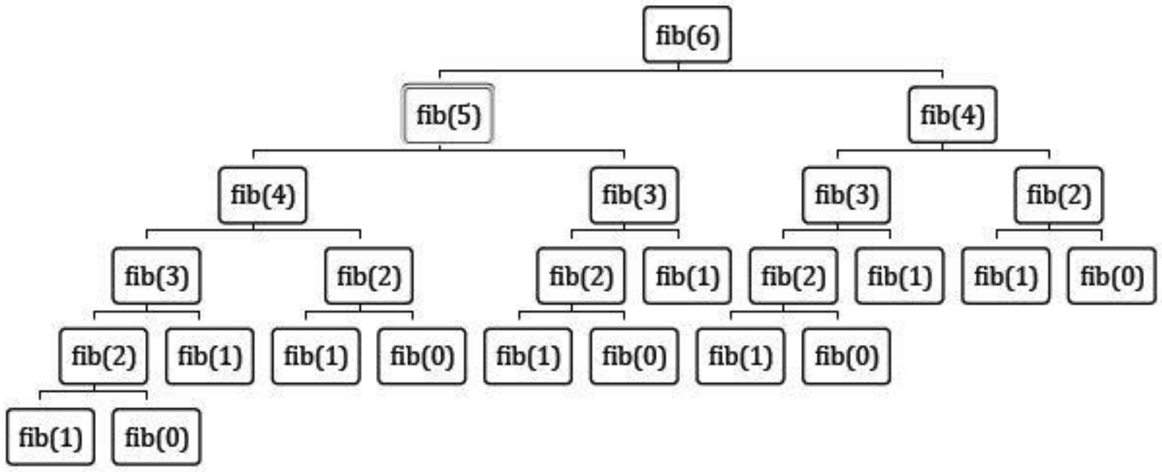
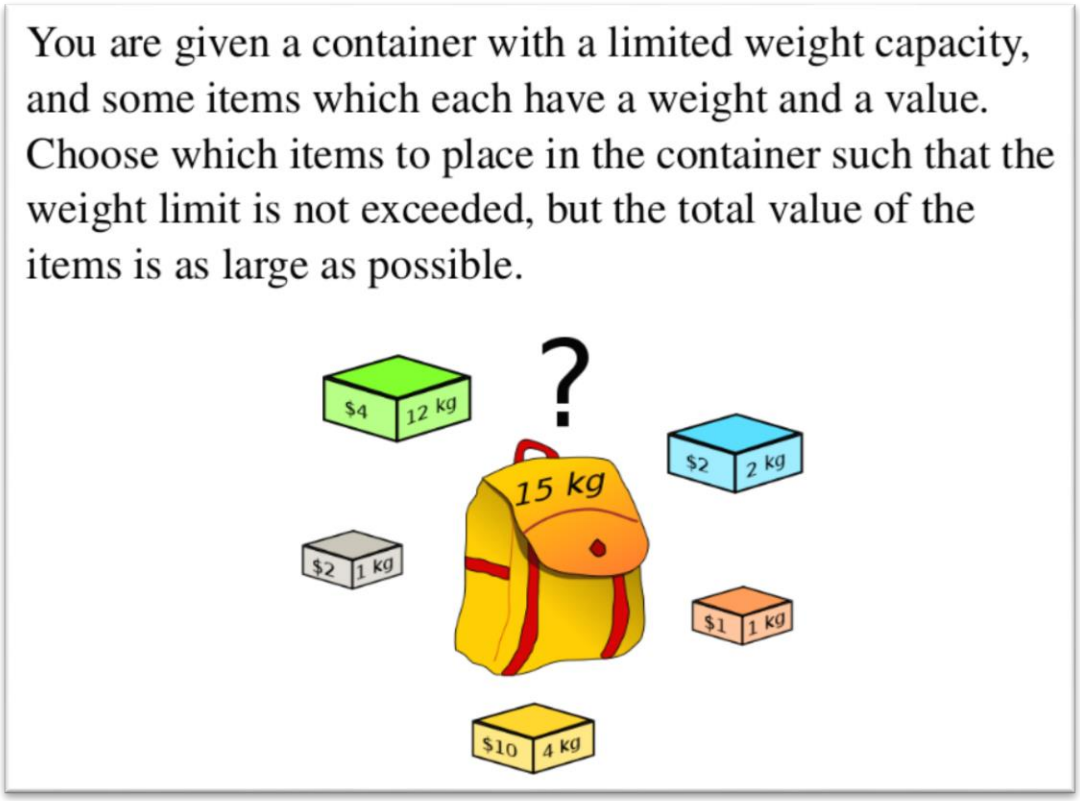


Figure 18.2 Tree of calls for recursive Fibonacci

Notice that we are computing the same values over and over again. For example fib gets called with 3 three times, and each of these calls provokes four additional calls of fib. It doesn't require a genius to think that it might be a good idea to record the value returned by the first call, and then look it up rather than compute it each time it is needed. This is called memorization, and is the key idea behind dynamic programming.

```
def fastFib(n, memo = {}):
    """Assumes n is an int >= 0, memo used only by recursive calls
    Returns Fibonacci of n"""
    if n == 0 or n == 1:
        return 1
    try:
        return memo[n]
    except KeyError:
        result = fastFib(n-1, memo) + fastFib(n-2, memo)
        memo[n] = result
        return result
print(fastFib(120))
```

- **Dynamic Programming and the 0/1 Knapsack Problem**

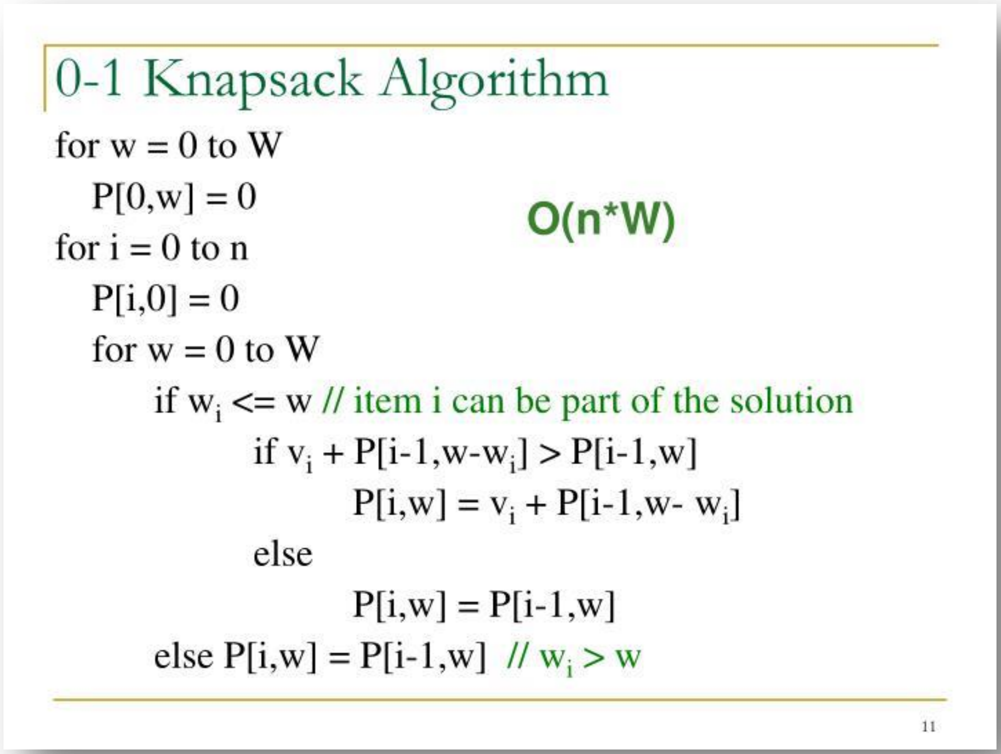


You are given a container with a limited weight capacity, and some items which each have a weight and a value. Choose which items to place in the container such that the weight limit is not exceeded, but the total value of the items is as large as possible.

?
$4
12 kg
$2
2 kg
15 kg
$2
1 kg
$1
1 kg
$10
4 kg

See Video: (On Video)

https://www.youtube.com/watch?v=dZMltkeP7_c



0-1 Knapsack Algorithm

```
for w = 0 to W
   P[0,w] = 0
for i = 0 to n
   P[i,0] = 0
   for w = 0 to W
        if $w_i$ <= w // item i can be part of the solution
              if $v_i$ + P[i-1,w-$w_i$] > P[i-1,w]
                    P[i,w] = $v_i$ + P[i-1,w- $w_i$]
              else
                    P[i,w] = P[i-1,w]
        else P[i,w] = P[i-1,w]  // $w_i$ > w
```

O(n*W)

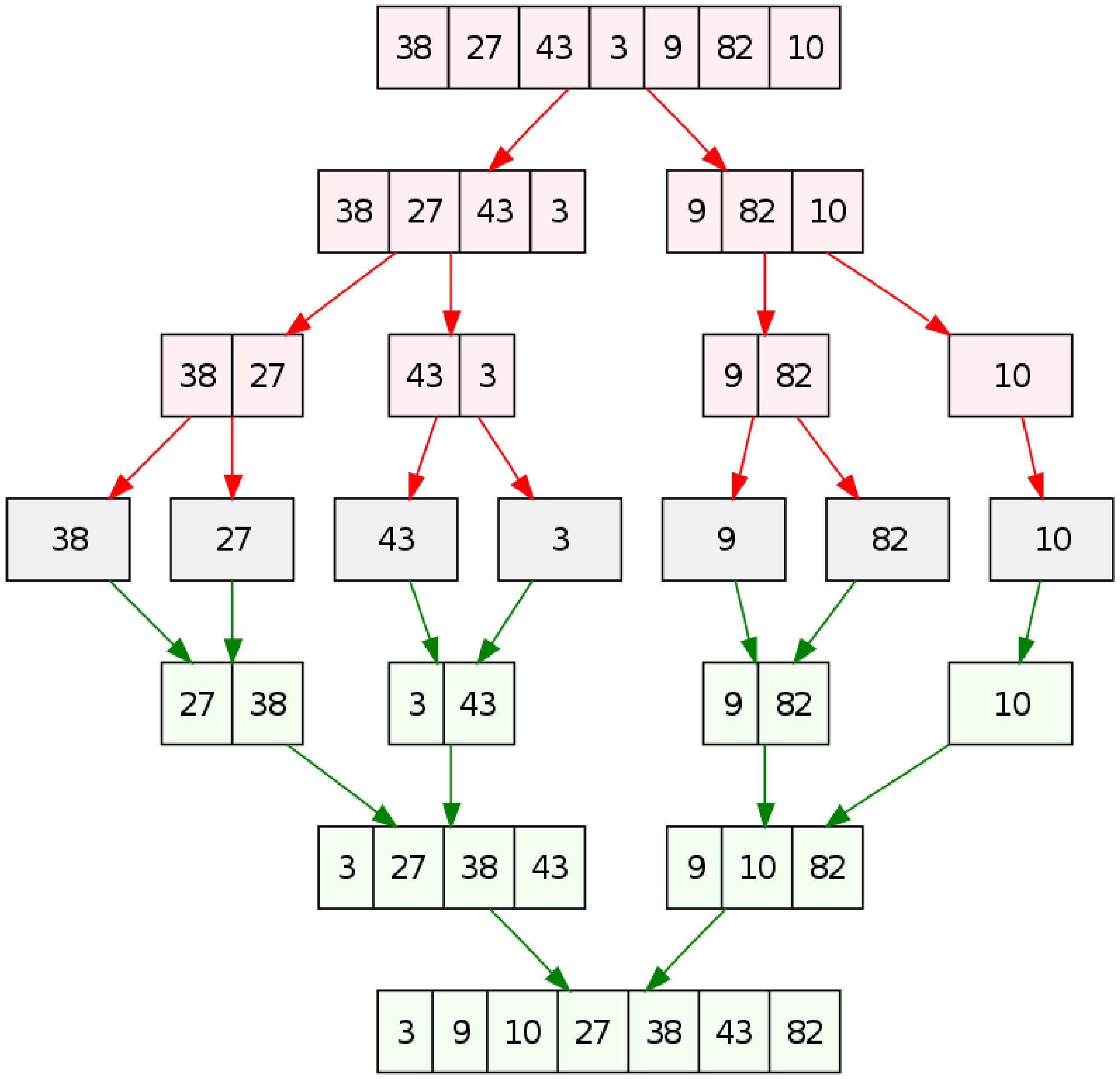
11

<u>Dynamic Programming and Divide-and-Conquer</u>

In computer science, **divide and conquer** is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This divide-and-conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFT).

Understanding and designing divide-and-conquer algorithms is a complex skill that requires a good understanding of the nature of the underlying problem to be solved. As when proving a theorem by induction, it is often necessary to replace the original problem with a more general or complicated problem in order to initialize the recursion, and there is no systematic method for finding the proper generalization. These divide-and-conquer complications are seen when optimizing the calculation of a Fibonacci number with efficient double recursion.



Advantages

1. **Solving difficult problems**
2. **Algorithm efficiency**
3. **Memory access**
4. **Round off control**

Prog: Program of Merge Sort

```
def mergeSort(alist):
   print("Splitting ",alist)
   if len(alist)>1:
      mid = len(alist)//2
      lefthalf = alist[:mid]
      righthalf = alist[mid:]

      mergeSort(lefthalf)
      mergeSort(righthalf)

      i=0
      j=0
      k=0
      while i < len(lefthalf) and j < len(righthalf):
         if lefthalf[i] < righthalf[j]:
            alist[k]=lefthalf[i]
            i=i+1
         else:
            alist[k]=righthalf[j]
            j=j+1
         k=k+1

      while i < len(lefthalf):
         alist[k]=lefthalf[i]
         i=i+1
         k=k+1

      while j < len(righthalf):
         alist[k]=righthalf[j]
         j=j+1
         k=k+1
   print("Merging ",alist)
alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

---THE END-------------------------------------