3 Algorithm Design Techniques and Strategies

For COMSC 132

Sam Bowne

Sep 16, 2024

Topics

- Brute force
- Recursion
- Divide and conquer
- Dynamic programming
- Greedy algorithms
- Dijkstra’s shortest path algorithm

Brute force

- Solving a problem by trying all possibilities
- Can be VERY slow
- Consider a chess game
- Trying to explore all possible moves several moves ahead becomes a huge problem quickly

Brute force

- Trying to decrypt data without knowing the key
- Simply try all possible keys
- Slow but effective if you have enough time

Brute force search

- Haystack is not sorted
- Must search through list one item at a time
- Complexity O(***n***)

```
import time, random

for n in [100000, 1000000, 10000000]:
  haystack = []
  for i in range(n):
    haystack.append(random.random())
  needles = []
  for rep in range(10):
    needles.append(haystack[random.randint(0, n-1)])

  start = time.time()
  for rep in range(10):
    for i in range(n):
      if needles[rep] == haystack[i]:
        break
  end = time.time()
  elapsed = end - start
  print("Time: ", f'{elapsed:9.4f}', f'{n:,}')
```

```
Time:      0.0743 100,000
Time:      0.7502 1,000,000
Time:      9.3014 10,000,000
```

Brute force search

```
import time, random

for n in [100000, 1000000, 10000000]:
  haystack = []
  for i in range(n):
    haystack.append(random.random())
  needles = []
  for rep in range(10):
    needles.append(haystack[random.randint(0, n-1)])

  start = time.time()
  for rep in range(10):
    for i in range(n):
      if needles[rep] == haystack[i]:
        break
  end = time.time()
  elapsed = end - start
  print("Time: ", f'{elapsed:9.4f}', f'{n:,}')
```

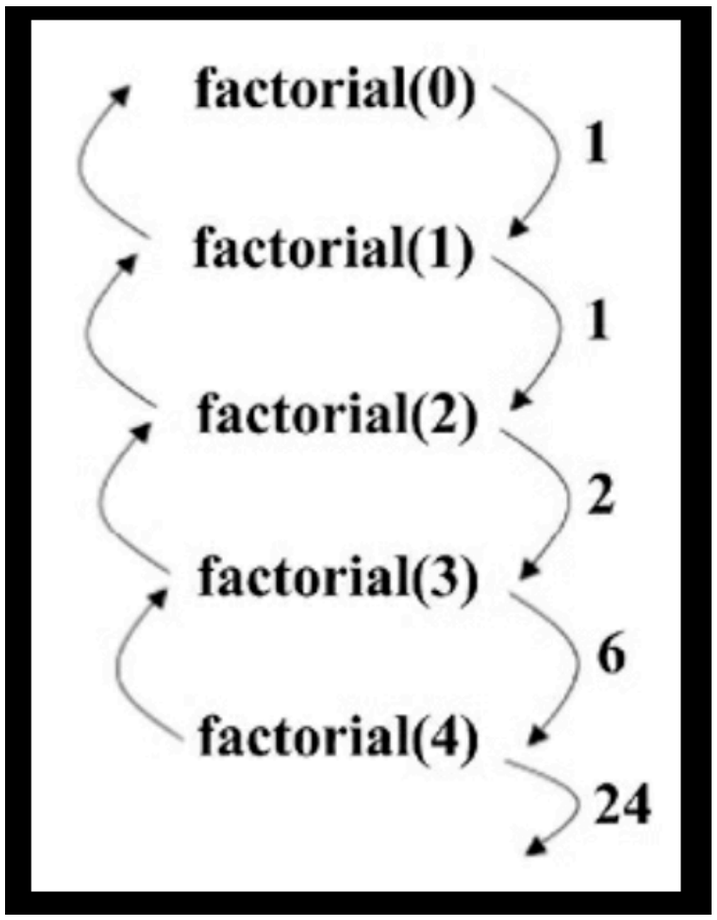
Recursion

- Algorithm calls itself repeatedly
- Until a condition is fulfilled

Factorial by Recursion

```python
def factorial(n):
    # test for a base case
    if n == 0:
        return 1
    else:
    # make a calculation and a recursive call
        return n*factorial(n-1)
print(factorial(4))
```

```
24
```




```python
def factorial(n):
    # test for a base case
    if n == 0:
        return 1
    else:
    # make a calculation and a recursive call
        return n*factorial(n-1)
print(factorial(4))
```

Ackerman's Function

```python
import time

def ackermann(m, n):
    global depth
    depth += 1
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        return ackermann(m - 1, ackermann(m, n - 1))

for i in range(4):
  depth = 0
  start = time.time()
  a =ackermann(i,  i)
  end = time.time()
  elapsed = end - start
  print("a(", i, ",", i, ") ", f'{elapsed:9.4f}',  depth)
```

```
a( 0 , 0 )      0.0000 1
a( 1 , 1 )      0.0000 4
a( 2 , 2 )      0.0000 27
a( 3 , 3 )      0.0009 2432
```

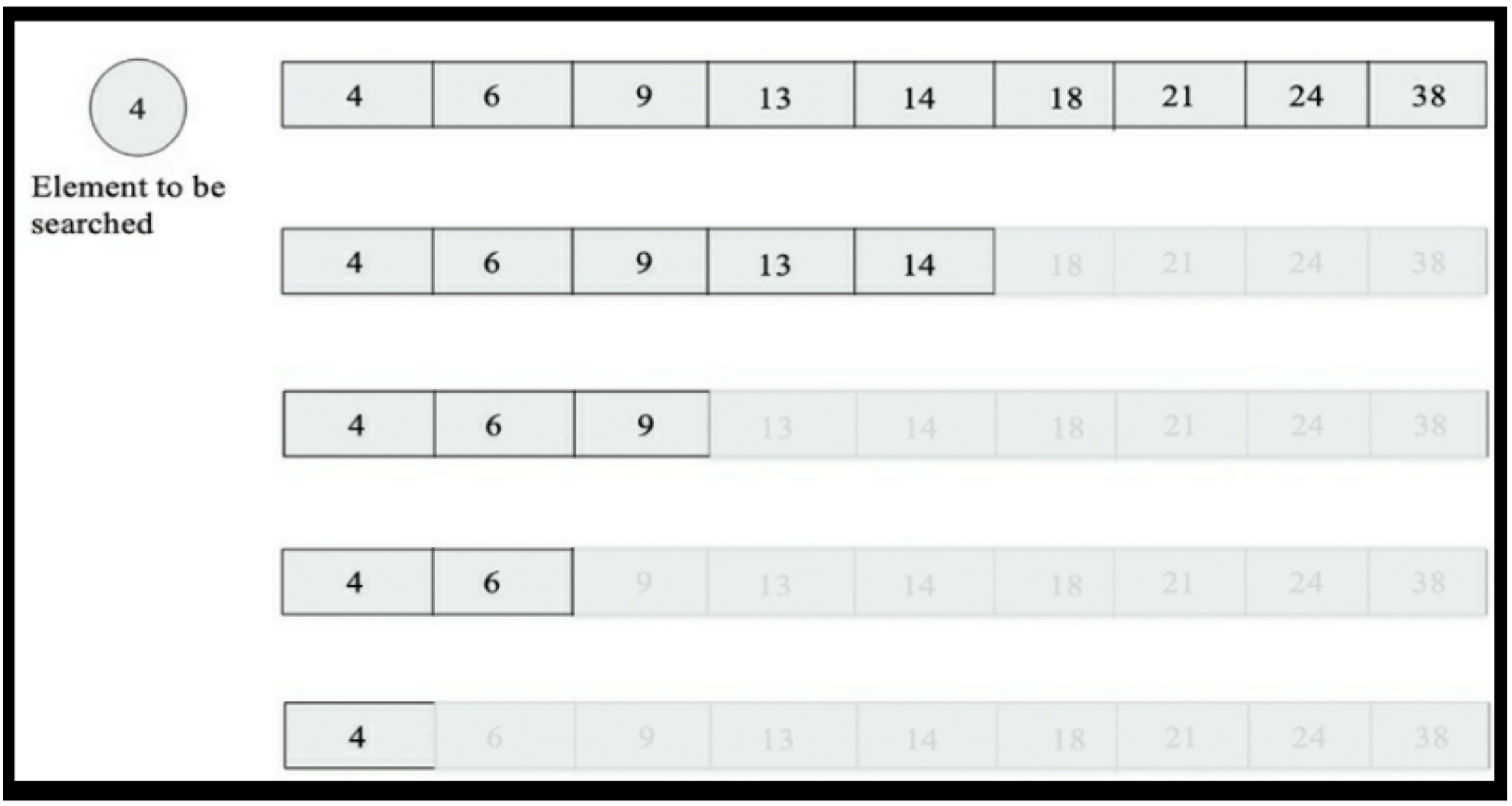
Ackerman's Function

```
import time

def ackermann(m, n):
    global depth
    depth += 1
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        return ackermann(m - 1, ackermann(m, n - 1))

for i in range(4):
  depth = 0
  start = time.time()
  a =ackermann(i,  i)
  end = time.time()
  elapsed = end - start
  print("a(", i, ",", i, ") ", f'{elapsed:9.4f}',  depth)
```

Divide and conquer

- Split a complex problem into two parts
- Solve them separately

Binary search



Binary search

- Haystack "arr" is sorted
- Test one value in the middle of the list
- Needle "key" is either below or above that
- The remaining list is 1/2 the size

```
def binary_search(arr, start, end, key):
    global steps
    while start <= end:
        steps += 1
        mid = start + (end - start)//2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            start = mid + 1
        else:
            end = mid - 1
    return -1
```

Binary search

- Test various values of n
- Perform each search ten times
- Print out total number of steps required

```
for n in [1000, 10000, 100000, 1000000, 10000000]:
  haystack = []
  for i in range(n):
    haystack.append(random.random())
  haystack.sort()

  needles = []
  for rep in range(10):
    needles.append(haystack[random.randint(0, n-1)])

  steps = 0
  for rep in range(10):
    result = binary_search(haystack, 0, n-1, needles[rep])
  print("Steps: ", steps, f'{n:,}')
```

Binary search

- Time complexity O(log ***n***)
- Making list 10 times longer simply adds 30 more steps

```
Steps:   83 1,000
Steps:   124 10,000
Steps:   156 100,000
Steps:   188 1,000,000
Steps:   227 10,000,000
```

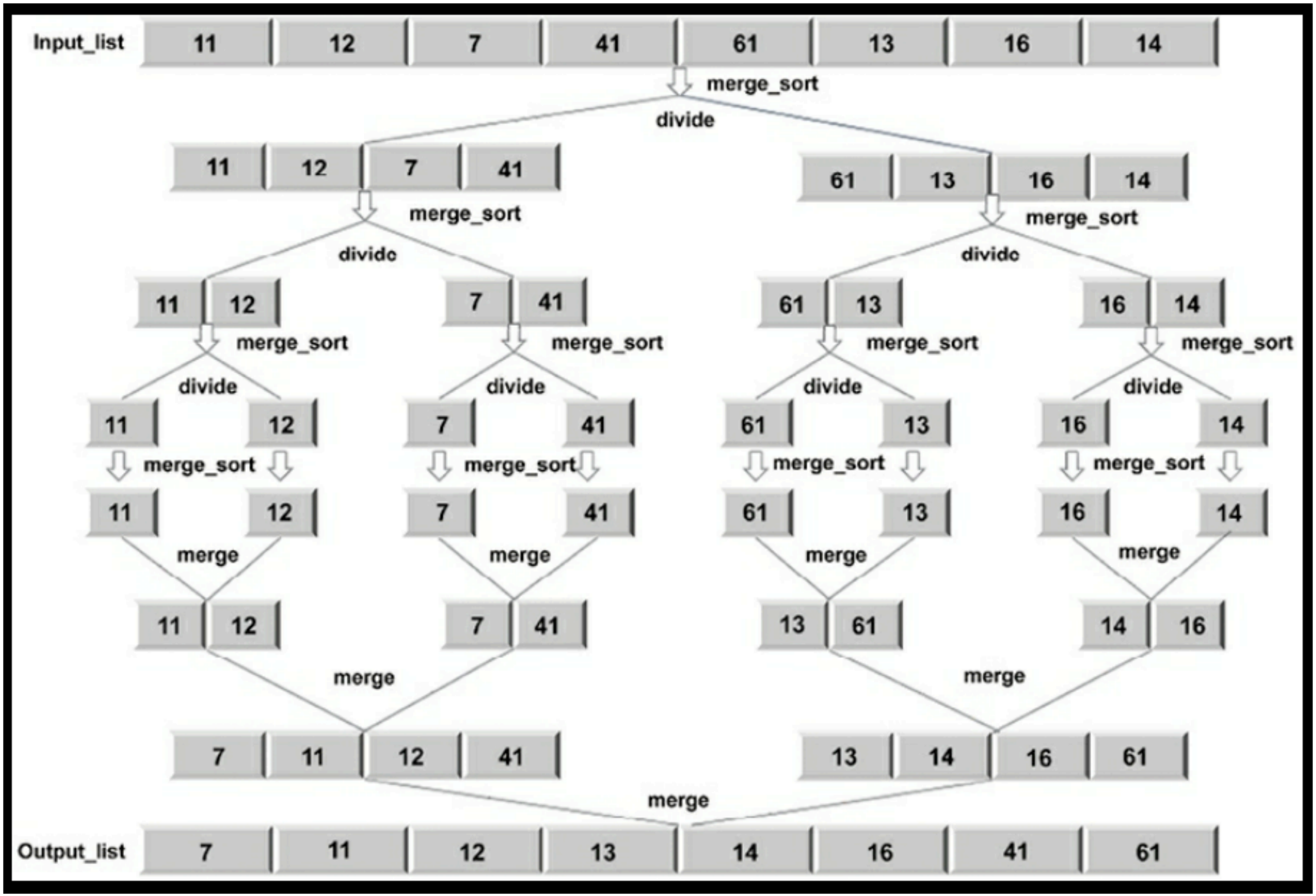
Binary search

```
import random

def binary_search(arr, start, end, key):
    global steps
    while start <= end:
        steps += 1
        mid = start + (end - start)//2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            start = mid + 1
        else:
            end = mid - 1
    return -1

for n in [1000, 10000, 100000, 1000000, 10000000]:
  haystack = []
  for i in range(n):
    haystack.append(random.random())
  haystack.sort()

  needles = []
  for rep in range(10):
    needles.append(haystack[random.randint(0, n-1)])

  steps = 0
  for rep in range(10):
    result = binary_search(haystack, 0, n-1, needles[rep])
  print("Steps: ", steps, f'{n:,}')
```

Merge sort

- Divide list in half
- Repeat until each sub-list has only one element
- Merge the sublists with elements in order
- Repeat until there's one sorted list

Merge sort


Input_list
11 12 7 41 61 13 16 14
merge_sort
divide
11 12 7 41
merge_sort
divide
61 13 16 14
merge_sort
divide
11 12
merge_sort
divide
7 41
merge_sort
divide
61 13
merge_sort
divide
16 14
merge_sort
divide
11 12
merge_sort
divide
7 41
merge_sort
divide
61 13
merge_sort
divide
16 14
merge_sort
11 12
merge
7 41
merge
13 61
merge
14 16
merge
7 11 12 41
merge
13 14 16 61
merge
Output_list
7 11 12 13 14 16 41 61

merge_sort

- Splits list in half recursively
  - Complexity O(log ***n***)
- Calls **merge()** to merge the short sublists together

```
def merge_sort(unsorted_list):
    if len(unsorted_list) == 1:
        return unsorted_list
    mid_point = int(len(unsorted_list)/2)
    first_half = unsorted_list[:mid_point]
    second_half = unsorted_list[mid_point:]
    half_a = merge_sort(first_half)
    half_b = merge_sort(second_half)
    return merge(half_a, half_b)
```

```
import random, time

def merge_sort(unsorted_list):
    if len(unsorted_list) == 1:
        return unsorted_list
    mid_point = int(len(unsorted_list)/2)
    first_half = unsorted_list[:mid_point]
    second_half = unsorted_list[mid_point:]
    half_a = merge_sort(first_half)
    half_b = merge_sort(second_half)
    return merge(half_a, half_b)
```

merge

- Combines two lists into one larger, sorted list
  - Complexity O(***n***)

```
def merge(first_sublist, second_sublist):
    i = j = 0
    merged_list = []
    while i < len(first_sublist) and j < len(second_sublist):
        if first_sublist[i] < second_sublist[j]:
            merged_list.append(first_sublist[i])
            i += 1
        else:
            merged_list.append(second_sublist[j])
            j += 1
    while i < len(first_sublist):
        merged_list.append(first_sublist[i])
        i += 1
    while j < len(second_sublist):
        merged_list.append(second_sublist[j])
        j += 1
    return merged_list
```

```
def merge(first_sublist, second_sublist):
    i = j = 0
    merged_list = []
    while i < len(first_sublist) and j < len(second_sublist):
        if first_sublist[i] < second_sublist[j]:
            merged_list.append(first_sublist[i])
            i += 1
        else:
            merged_list.append(second_sublist[j])
            j += 1
    while i < len(first_sublist):
        merged_list.append(first_sublist[i])
        i += 1
    while j < len(second_sublist):
        merged_list.append(second_sublist[j])
        j += 1
    return merged_list
```

Merge sort

- Complexity O(***n*** log ***n***)
- Making ***n*** ten times larger
- Makes time 30 times longer

```
unsorted_list = [64, 34, 25, 12, 22, 11, 90]
print(merge_sort(unsorted_list))

for n in [1000, 10000, 100000, 1000000, 10000000]:
  haystack = []
  for i in range(n):
    haystack.append(random.random())
  start = time.time()
  merge_sort(haystack)
  end = time.time()
  elapsed = end - start
  print(f'{elapsed:9.4f}',  f'{n:,}')
```

```
[11, 12, 22, 25, 34, 64, 90]
   0.0049 1,000
   0.0576 10,000
   2.4416 100,000
   7.6330 1,000,000
 106.5946 10,000,000
```


```
unsorted_list = [64, 34, 25, 12, 22, 11, 90]
print(merge_sort(unsorted_list))

for n in [1000, 10000, 100000, 1000000, 10000000]:
  haystack = []
  for i in range(n):
    haystack.append(random.random())
  start = time.time()
  merge_sort(haystack)
  end = time.time()
  elapsed = end - start
  print(f'{elapsed:9.4f}',  f'{n:,}')
```

Dynamic programming

- Break the problem into a series of sub-problems
- If the same sub-problem is being solved many times, avoid that:
  - Save the results of the sub-problems and only calculate each one only once
- This is an example of **time-memory trade-off**
  - Use more memory to save time

Fibonacci series by recursion

- fib(1) is calculated many times

```
[4]  def fib(n):
         global count1
         if n == 1:
             count1 += 1
         if n <= 1:
            return 1
         else:
            return fib(n-1) + fib(n-2)
     for i in range(5):
         count1 = 0
         print(fib(i), count1)
```

```
1 0
1 1
2 1
3 2
5 3
```

```
def fib(n):
     global count1
     if n == 1:
         count1 += 1
     if n <= 1:
        return 1
     else:
        return fib(n-1) + fib(n-2)
for i in range(5):
    count1 = 0
    print(fib(i), count1)
```

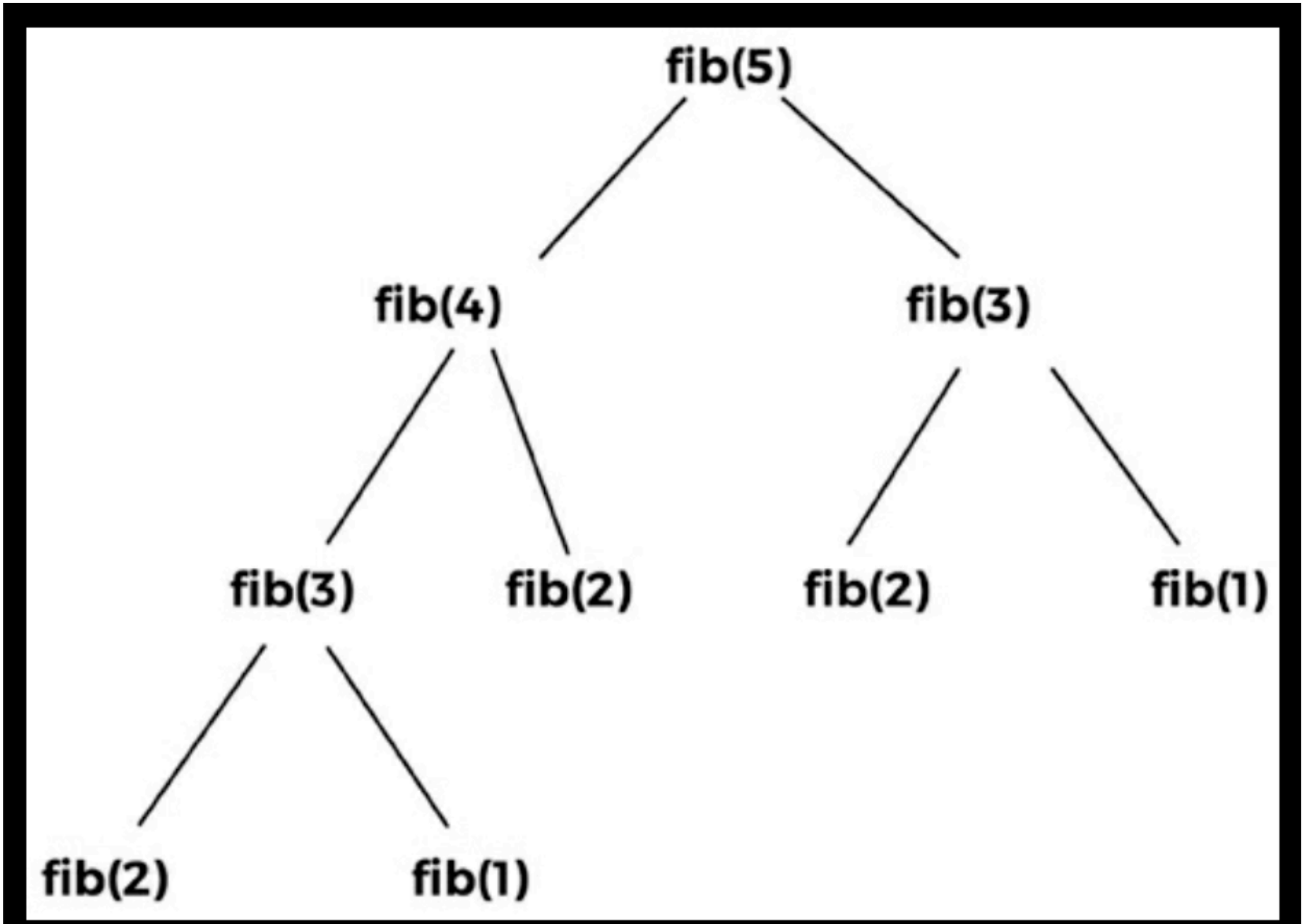
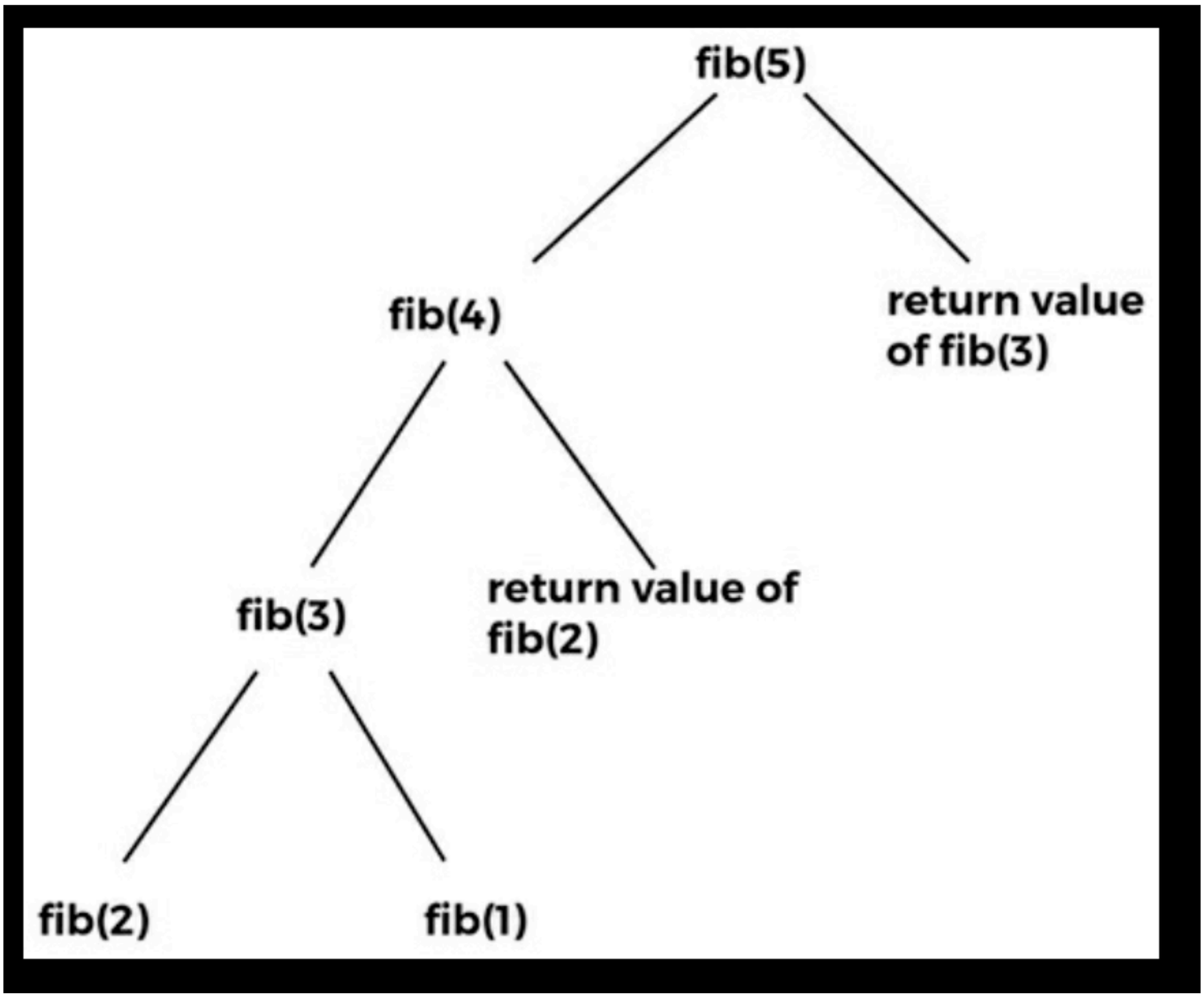


Figure 3.5: Recursion tree for fib(5)

Dynamic programming



Dynamic programming of Fibonacci numbers

- Each fib(n) is calculated only once

```
[5]  def dyna_fib(n):
         if n == 0:
             return 0
         if n == 1:
             return 1
         if lookup[n] is not None:
             return lookup[n]

         lookup[n] = dyna_fib(n-1) + dyna_fib(n-2)
         return lookup[n]
     lookup = [None]*(1000)

     for i in range(6):
         print(dyna_fib(i))

     0
     1
     1
     2
     3
     5
```

```
def dyna_fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    if lookup[n] is not None:
        return lookup[n]

    lookup[n] = dyna_fib(n-1) + dyna_fib(n-2)
    return lookup[n]
lookup = [None]*(1000)

for i in range(6):
    print(dyna_fib(i))
```

Ackerman's Function using recursion

```python
def ackermann(m, n):
    global count11
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        if (m == 1) and (n == 1):
            count11 += 1
        return ackermann(m - 1, ackermann(m, n - 1))

for i in range(4):
  count11 = 0
  a =ackermann(i,  i)
  print("a(", i, ",", i, ") ", "Calculated a(1,1)", count11, "times")
```

```
a( 0 , 0 )  Calculated a(1,1) 0 times
a( 1 , 1 )  Calculated a(1,1) 1 times
a( 2 , 2 )  Calculated a(1,1) 3 times
a( 3 , 3 )  Calculated a(1,1) 52 times
```

Ackerman's Function

```
def ackermann(m, n):
    global count11
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        if (m == 1) and (n == 1):
            count11 += 1
        return ackermann(m - 1, ackermann(m, n - 1))

for i in range(4):
  count11 = 0
  a =ackermann(i,  i)
  print("a(", i, ",", i, ") ", "Calculated a(1,1)", count11, "times")
```

Ackerman's Function with dynamic programming

- Still fails
- See next slide

```
def dynamic_ackermann(m, n):
    global lookup, depth
    depth += 1
    if lookup[m][n] is not None:
        return lookup[m][n]
    if m == 0:
        result = n + 1
    elif n == 0:
        if lookup[m-1][n] is not None:
            result = lookup[m-1][n]
        else:
            result = ackermann(m - 1, 1)
    else:
        if lookup[m][n-1] is not None:
            term2 = lookup[m][n-1]
        else:
            term2 = ackermann(m, n - 1)

        if lookup[m-1][term2] is not None:
            result = lookup[m-1][term2]
        else:
            result = ackermann(m - 1, term2)
    lookup[m][n] = result
    return result
```

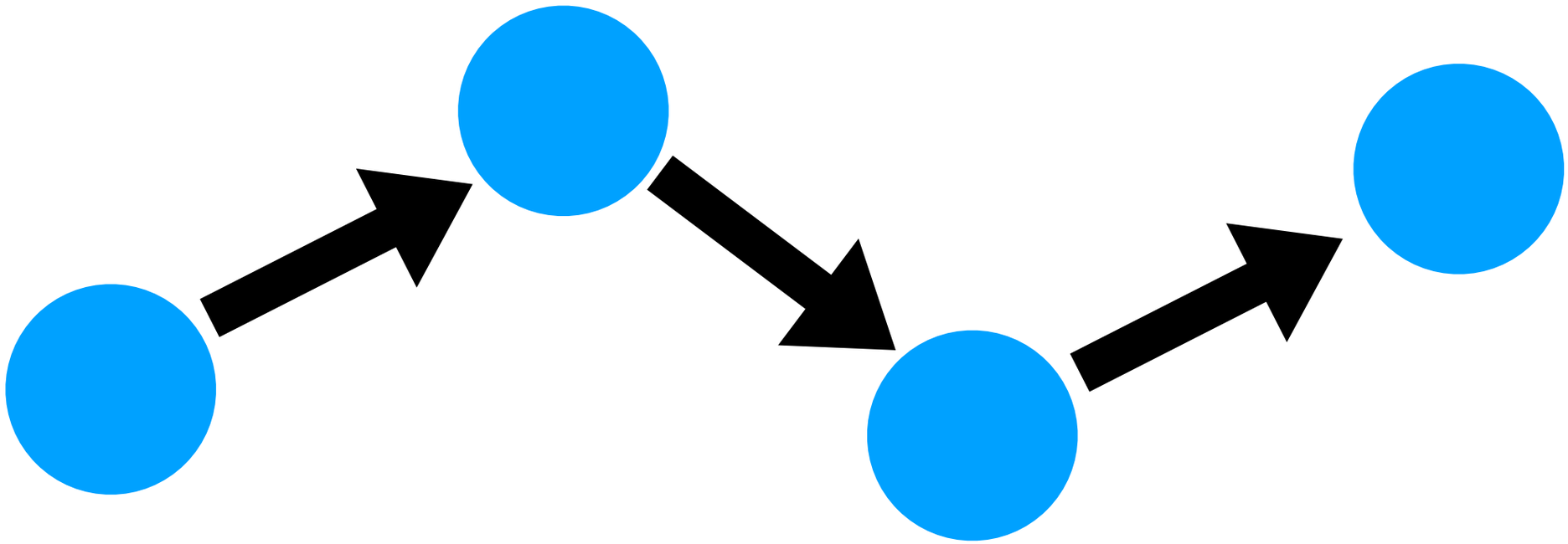
```python
# Define 2D list
lookup = [[None]*100]*100

for i in range(6):
  depth = 0
  a =dynamic_ackermann(i,  i)
  print("a(", i, ",", i, "):", a, "depth:", depth)
```

```
a( 0 , 0 ): 1 depth: 1
a( 1 , 1 ): 2 depth: 1
a( 2 , 2 ): 4 depth: 1
a( 3 , 3 ): 11 depth: 1
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-11-65399764d326> in <cell line: 29>()
     29 for i in range(6):
     30   depth = 0
---> 31   a =dynamic_ackermann(i,  i)
     32   print("a(", i, ",", i, "):", a, "depth:", depth)

                                 2 frames
... last 1 frames repeated, from the frame below ...

<ipython-input-2-3c31c9656599> in ackermann(m, n)
      8             if (m == 1) and (n == 1):
      9                 count11 += 1
---> 10             return ackermann(m - 1, ackermann(m, n - 1))
     11
     12 for i in range(4):

RecursionError: maximum recursion depth exceeded in comparison
```

Greedy algorithms

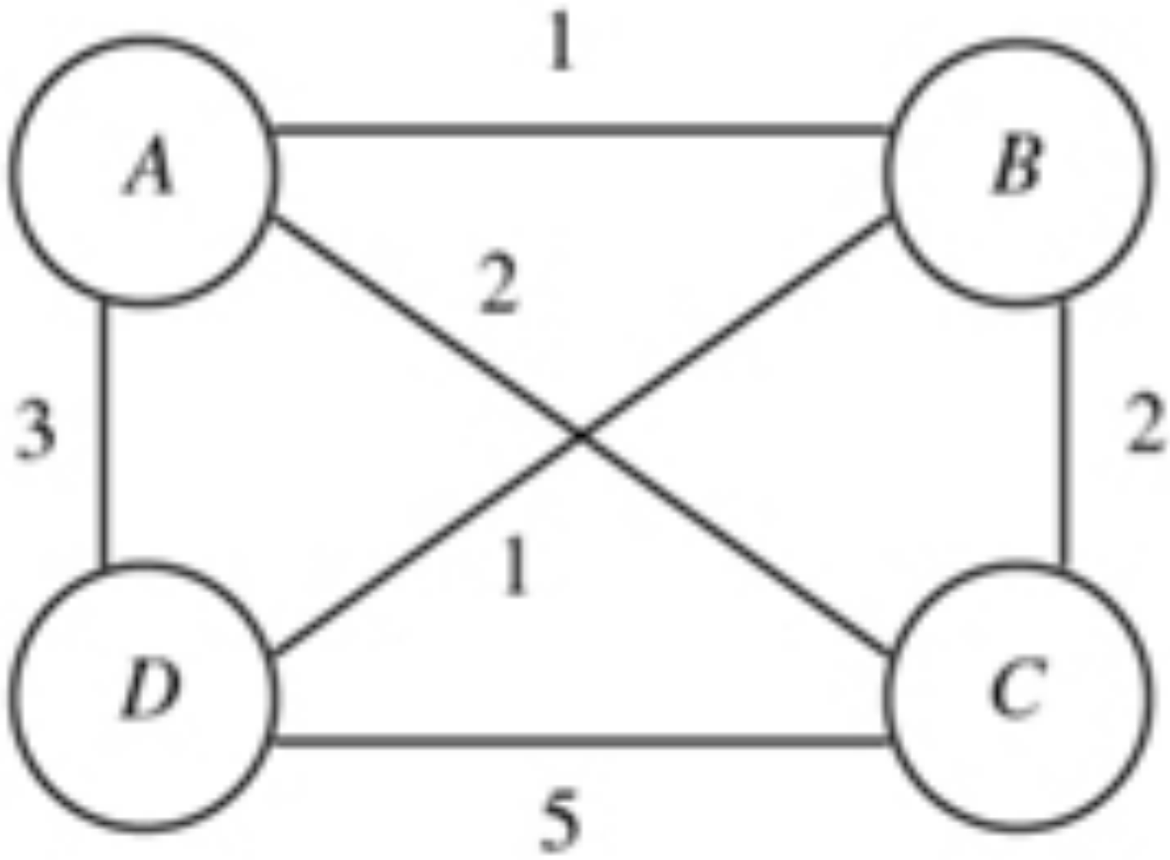
- Consider an optimization problem with many steps
  - Such as the traveling salesman problem
- Greedy algorithm: choose the lowest cost option at each step
  - Travel always to the nearest remaining city

Greedy algorithm



Greedy algorithm

Here is a little example from S.K.Basu's book, *Design Methods and Analysis of Algorithms*:



From A, the greedy cycle is $ABDCA$ of length 9, while $ACBDA$ has length 8.

Making Change

- You have bills in these denominations:
  - $1, $2, $5, $10, $20, $50
- You want to pay $29
- Greedy algorithm: pay with largest bill at each step
  - $20 + $5 + $2 + $2
- Greedy algorithm finds best solution
  - Fewest number of bills

Making Change

- You have bills in these denominations:
  - $1, $14, $25
- You want to pay $29
- Greedy algorithm: pay with largest bill at each step
  - $25 + $1 + $1 + $1 + $1
- Not the best solution
  - $14 + $14 + $1

Shortest path problem

- Possible paths from A to D
  - AD: 9
  - ABCD: 10
  - AEFD: 7
- Shortest is AEFD

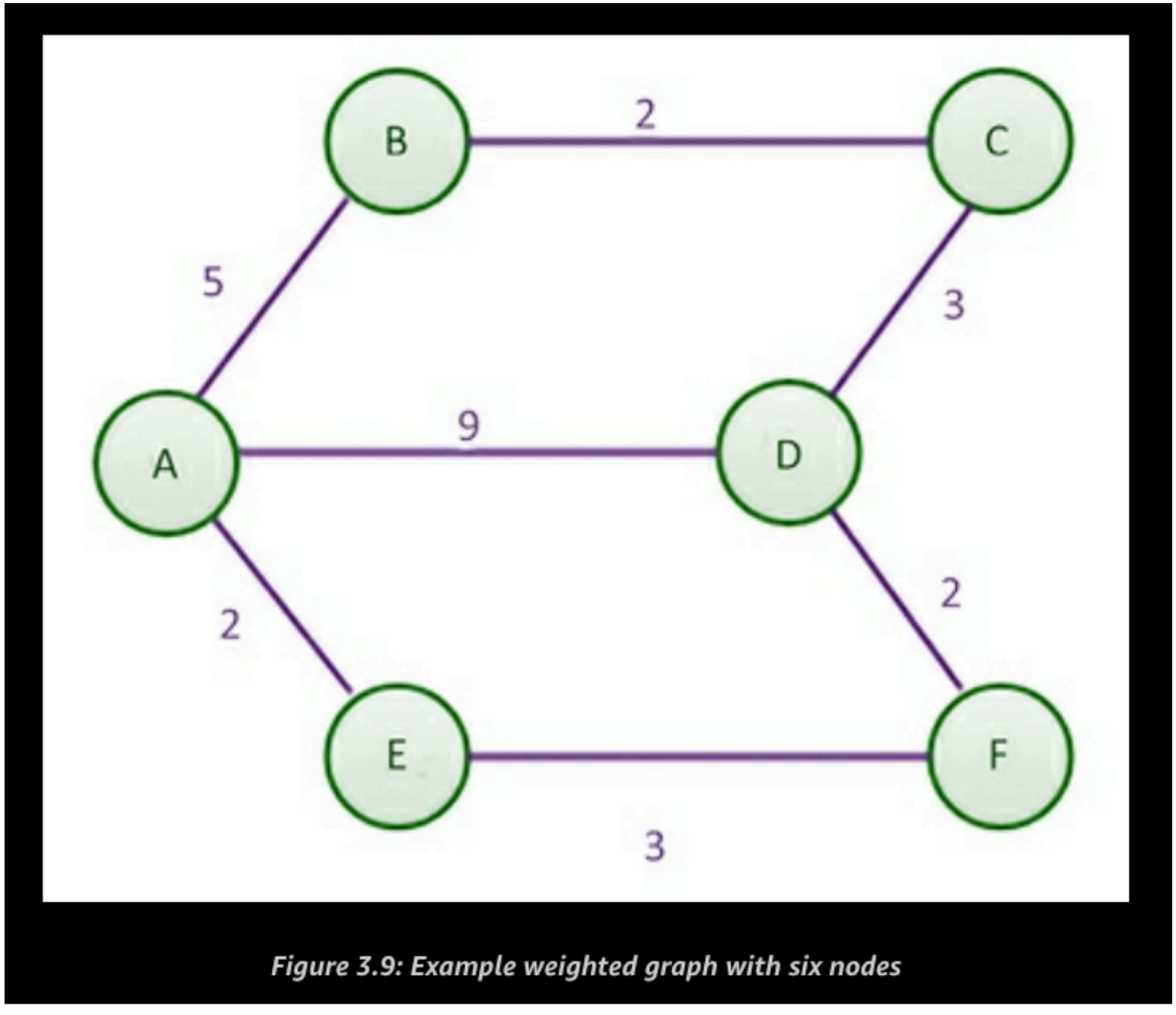


Figure 3.9: Example weighted graph with six nodes

Representation of graph

- Nested dictionaries
- Contains adjacent nodes and distances

```
[15] graph = dict()
     graph['A'] = {'B': 5, 'D': 9, 'E': 2}
     graph['B'] = {'A': 5, 'C': 2}
     graph['C'] = {'B': 2, 'D': 3}
     graph['D'] = {'A': 9, 'F': 2, 'C': 3}
     graph['E'] = {'A': 2, 'F': 3}
     graph['F'] = {'E': 3, 'D': 2}
```

```
graph = dict()
graph['A'] = {'B': 5, 'D': 9, 'E': 2}
graph['B'] = {'A': 5, 'C': 2}
graph['C'] = {'B': 2, 'D': 3}
graph['D'] = {'A': 9, 'F': 2, 'C': 3}
graph['E'] = {'A': 2, 'F': 3}
graph['F'] = {'E': 3, 'D': 2}
```

Table of shortest distances

- Initial table

Node	Shortest distance from source	Previous node
A	0	None
B	∞	None
C	∞	None
D	∞	None
E	∞	None
F	∞	None

Table 3.2: Initial table showing the shortest distance from the source

After visiting node A

Node	Shortest distance from source	Previous node
A*	0	None
B	5	A
C	∞	None
D	9	A
E	2	A
F	∞	None

Table 3.3: Shortest distance table after visiting node A

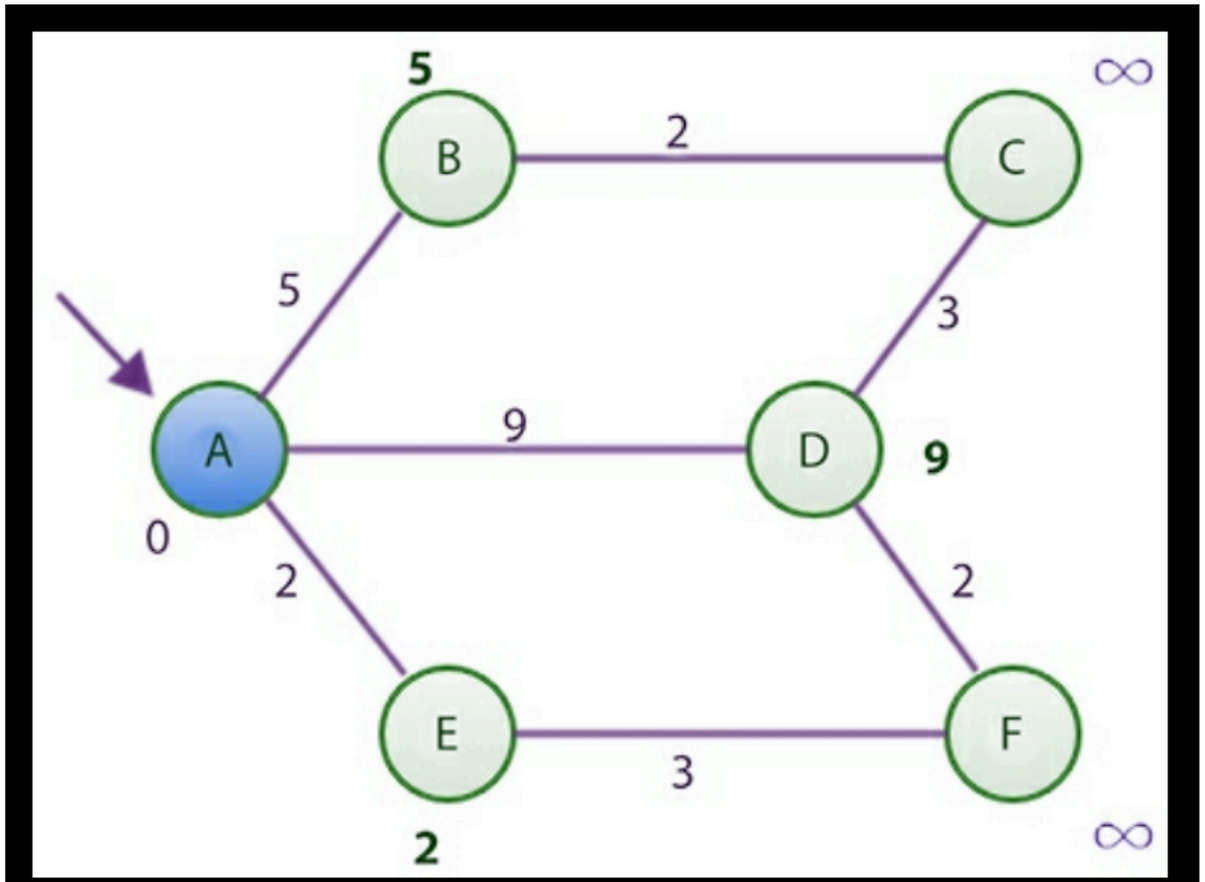


Figure 3.11: Shortest distance graph after visiting node A using Dijkstra's algorithm

After visiting node E

Node	Shortest distance from source	Previous node
A*	0	None
B	5	A
C	∞	None
D	9	A
E*	2	A
F	5	E

Table 3.4: Shortest distance table after visiting node E

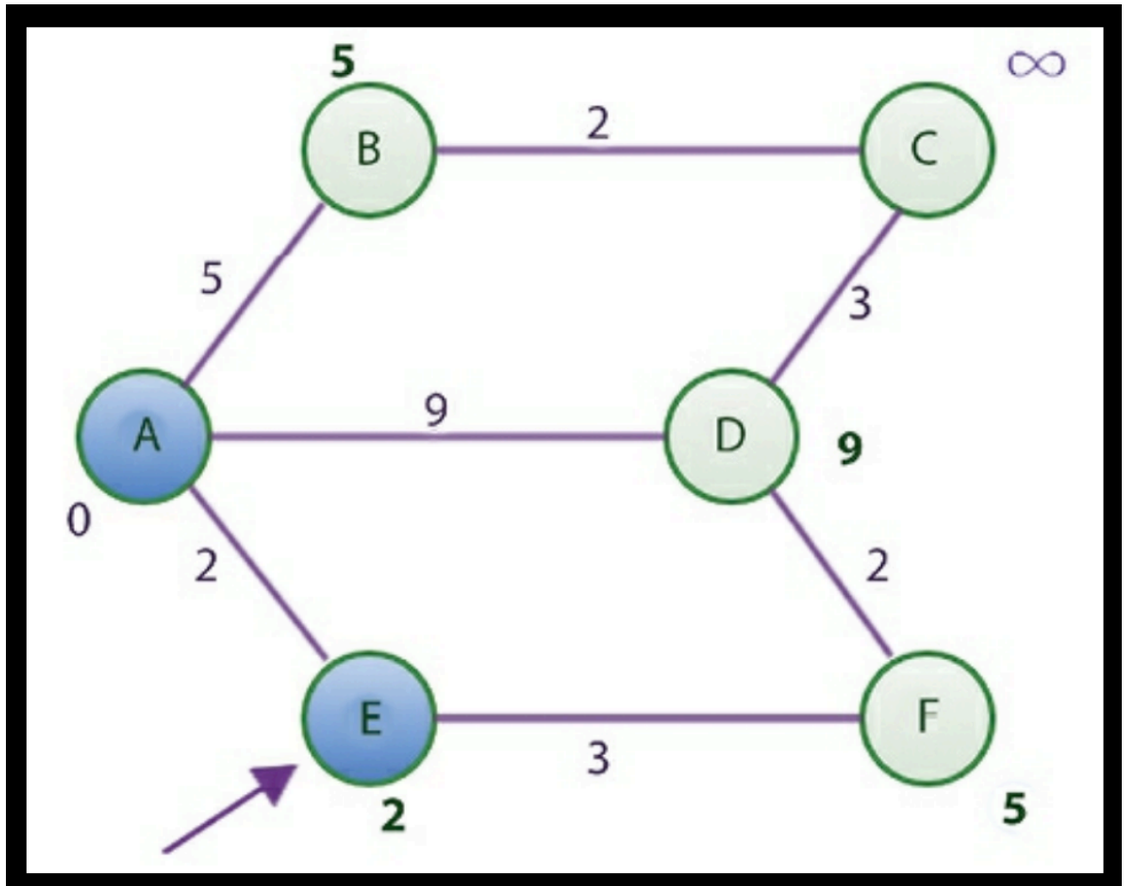


Figure 3.12: Shortest distance graph after visiting node E using Dijkstra's algorithm

After visiting node B

Node	Shortest distance from source	Previous node
A*	0	None
B*	5	A
C	7	B
D	9	A
E*	2	A
F	5	E

Table 3.5: Shortest distance table after visiting node B

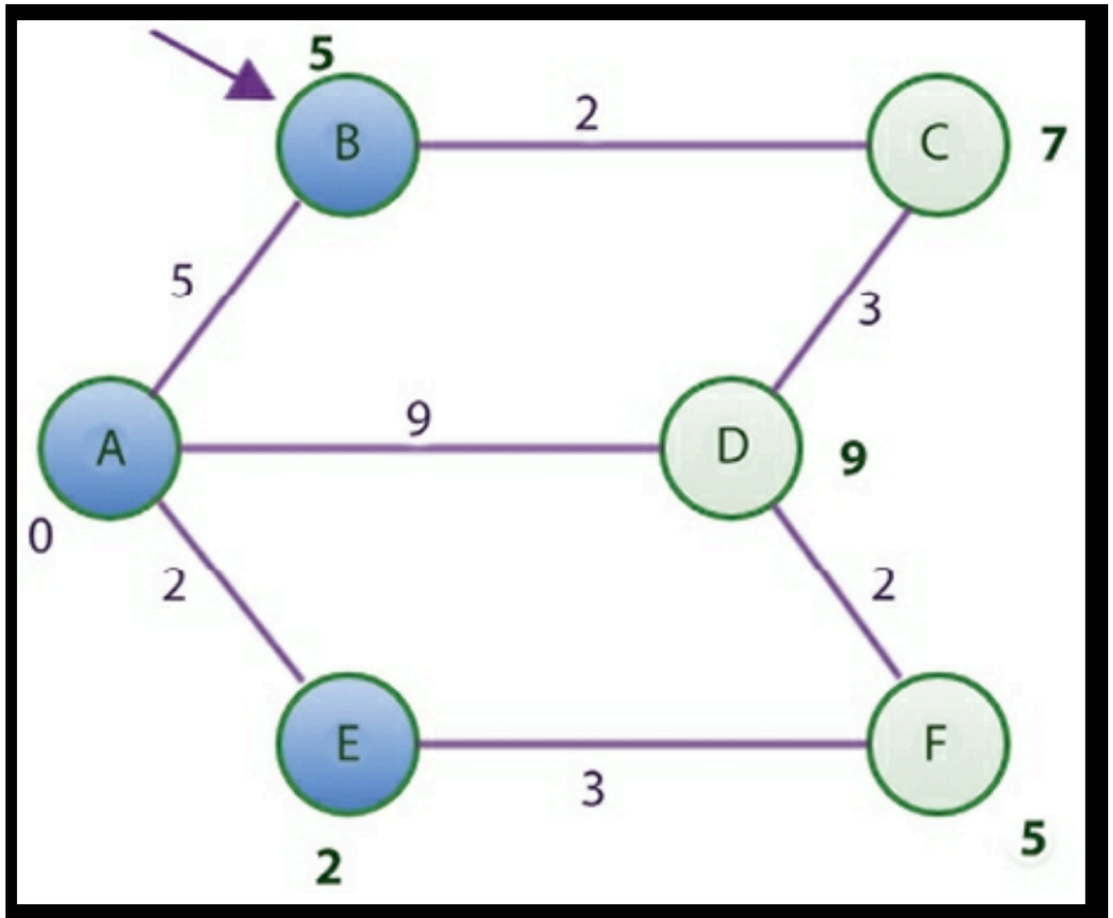


Figure 3.13: Shortest distance graph after visiting node B using Dijkstra's algorithm

After visiting node F

Node	Shortest distance from source	Previous node
A*	0	None
B*	5	A
C	7	B
D	7	F
E*	2	A
F*	5	E

Table 3.6: Shortest distance table after visiting node F

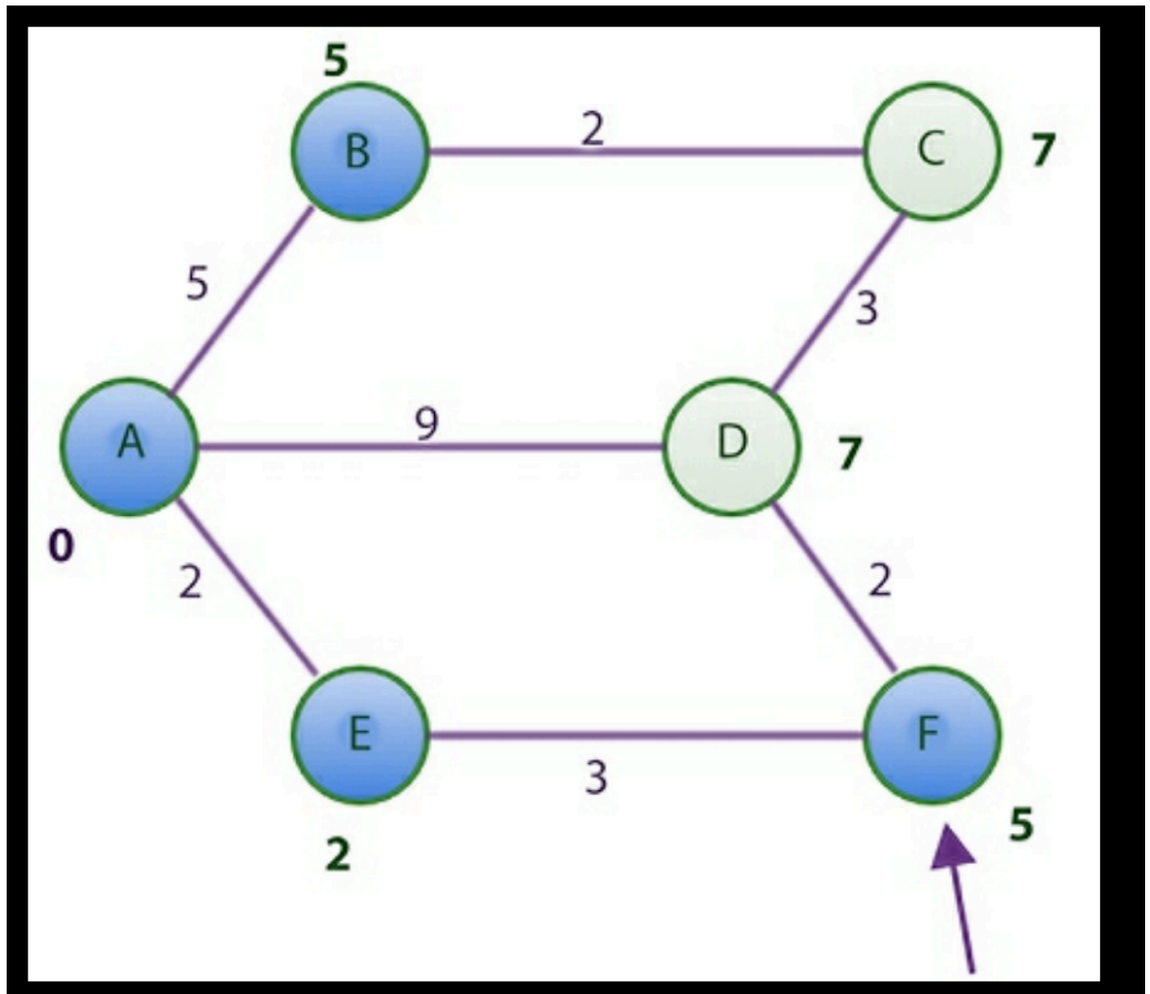


Figure 3.14: Shortest distance graph after visiting node F using Dijkstra's algorithm

After visiting node C

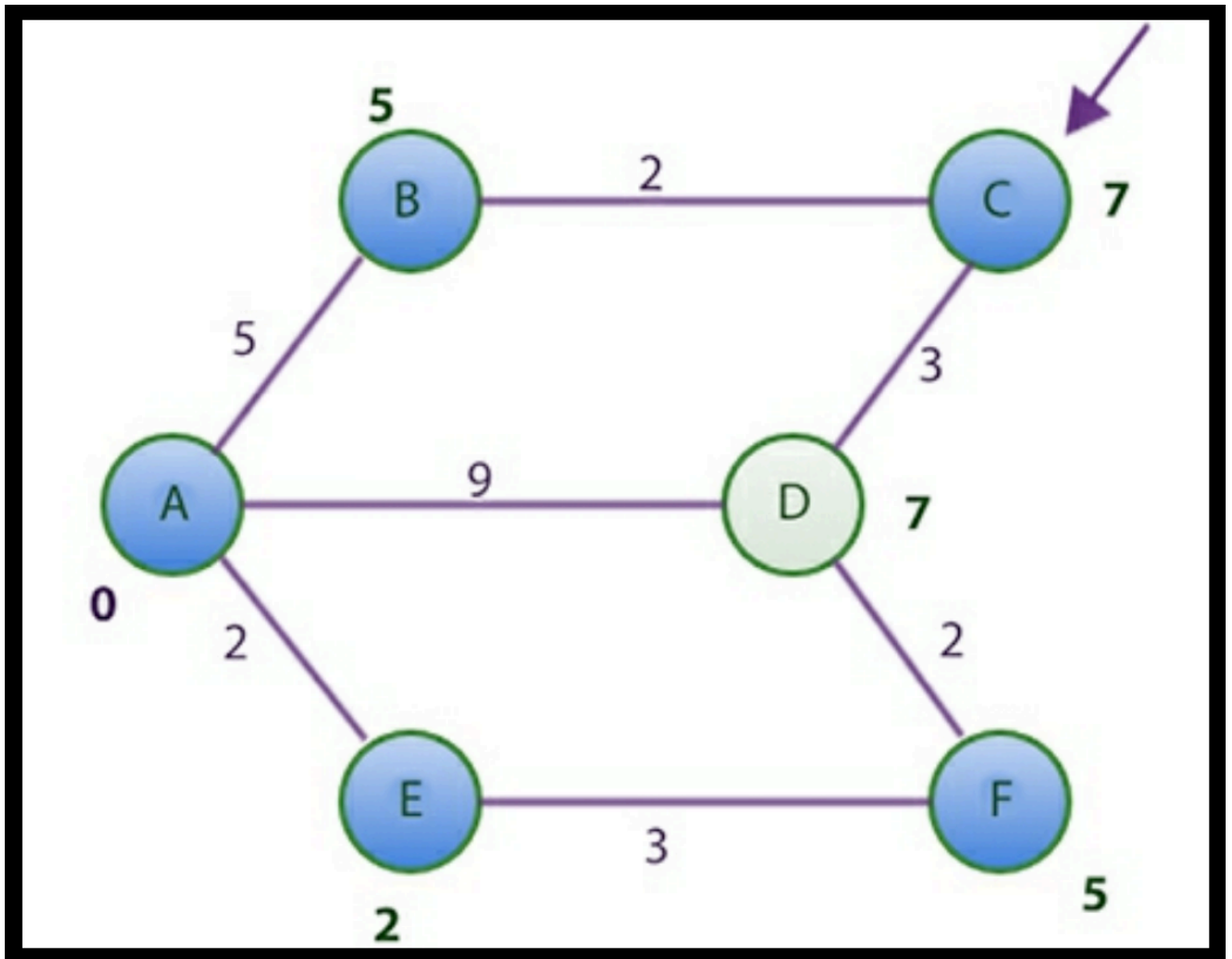


Figure 3.15: Shortest distance graph after visiting node C using Dijkstra's algorithm

After visiting node D

Node	Shortest distance from source	Previous node
A*	0	None
B*	5	A
C*	7	B
D*	7	F
E*	2	A
F*	5	E

Table 3.7: Shortest distance table after visiting node F

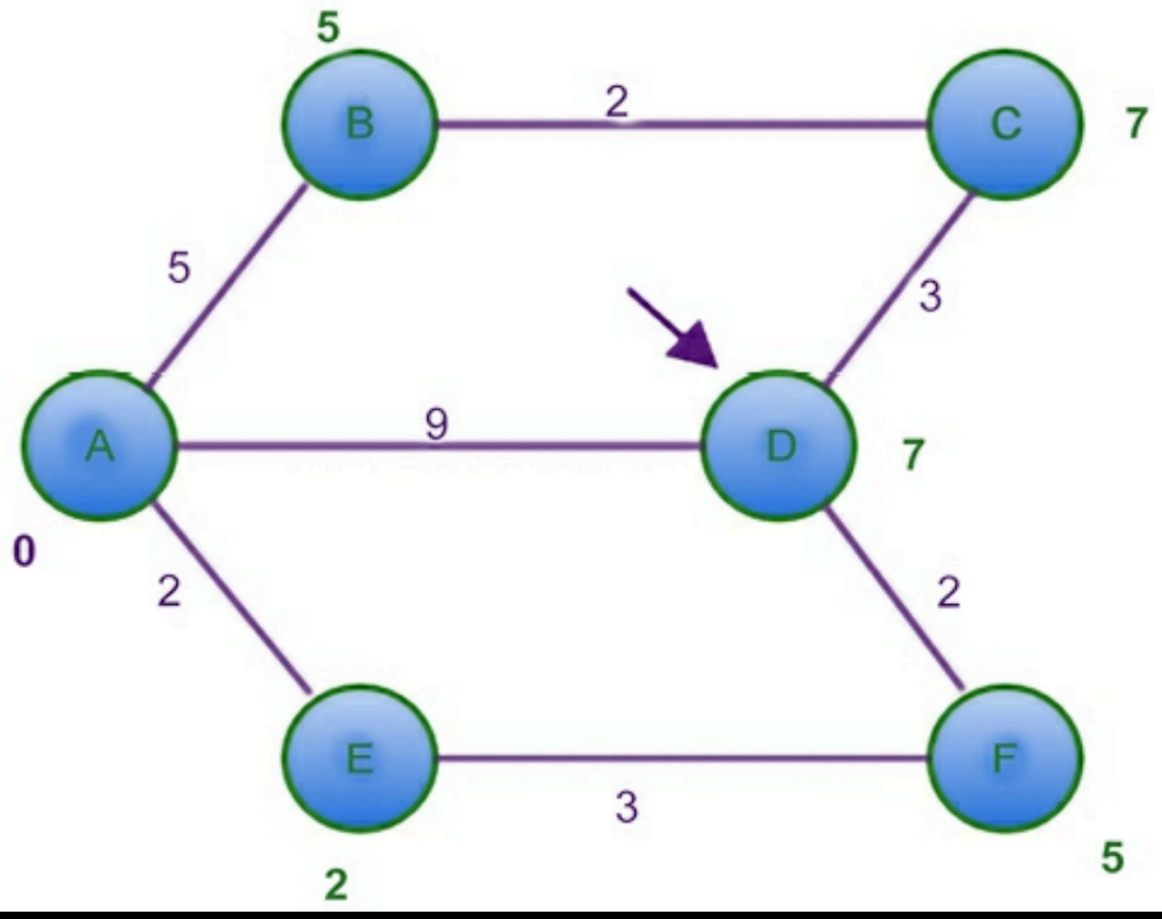


Figure 3.16: Shortest distance graph after visiting node D using Dijkstra's algorithm

Ch 3