Python for Data Scientists L8: Algorithms, searching and sorting

Iterative and recursion algorithms

Iterative algorithms

- looping constructs (while and for loops) lead to **iterative** algorithms

- can capture computation in a set of **state variables** that update on each iteration through loop

Iterative algorithms

- “multiply x * y” is equivalent to “add x to itself y times”

- capture **state** by
  - result ← 0
  - an **iteration** number starts at y

    y ← y-1 and stop when y = 0
  - a current **value of computation (result)**

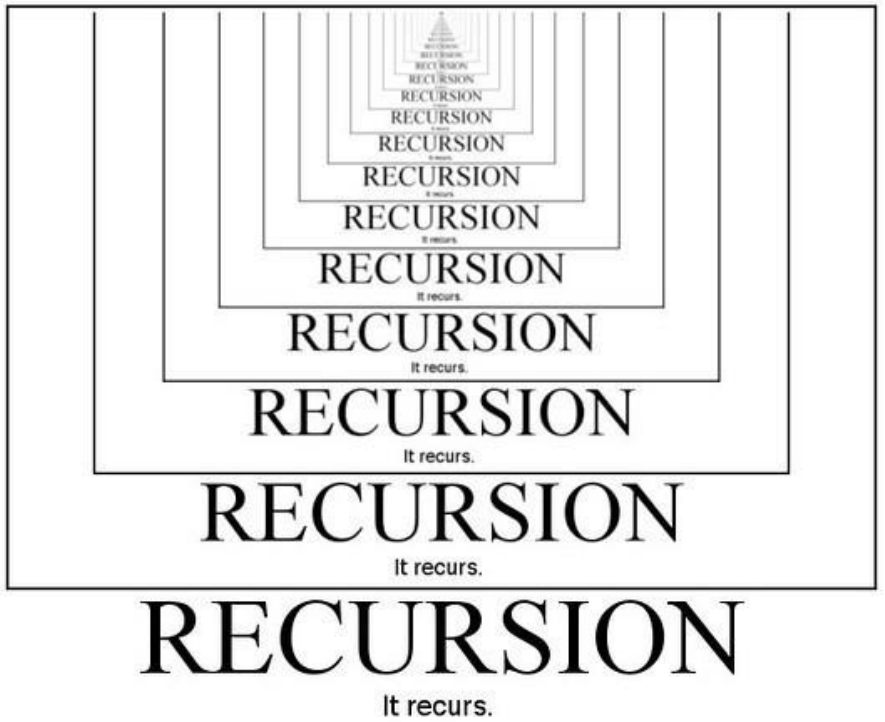
    result ← result + x

Iterative algorithms

```
def multiplication_iter(x, y):
    result = 0
    while y > 0:
        result += x
        y -= 1
    return result
```

Recursion

The process of repeating items in a self-similar way



Recursion

- **recursive step :** think how to reduce problem to a simpler/smaller version of same problem

- **base case** :
  - keep reducing problem until reach a simple case that can be solved directly
  - when y = 1, x*y = x

Recursion : example

$x * y = \boxed{x + x + x + \ldots + x}$ y items

$= x + \boxed{x + x + \ldots + x}$ y - 1 items

$= x + x * (y-1)$ Recursion reduction

```
def multiplication_rec(x, y):
    if y == 1:
        return x
    else:
        return x + multiplication_rec(x, y-1)
```

Recursion : example

Python 3.6

```
1  def multiplication_rec(x, y):
2      if y == 1:
3          return x
4      else:
5          return x + multiplication_rec(x, y-1)
6
7  multiplication_rec(2,3)
```

Recursion : example

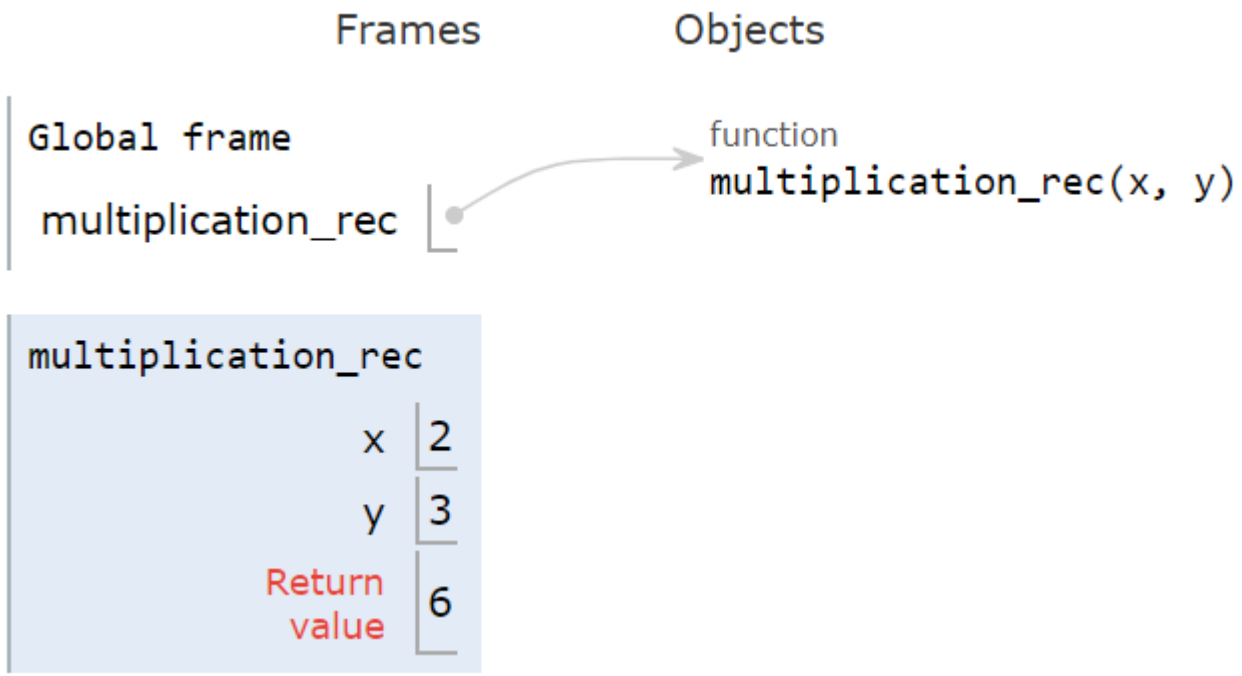
Python 3.6

```
1  def multiplication_rec(x, y):
2      if y == 1:
3          return x
4      else:
5          return x + multiplication_rec(x, y-1)
6
7  multiplication_rec(2,3)
```

Recursion : example

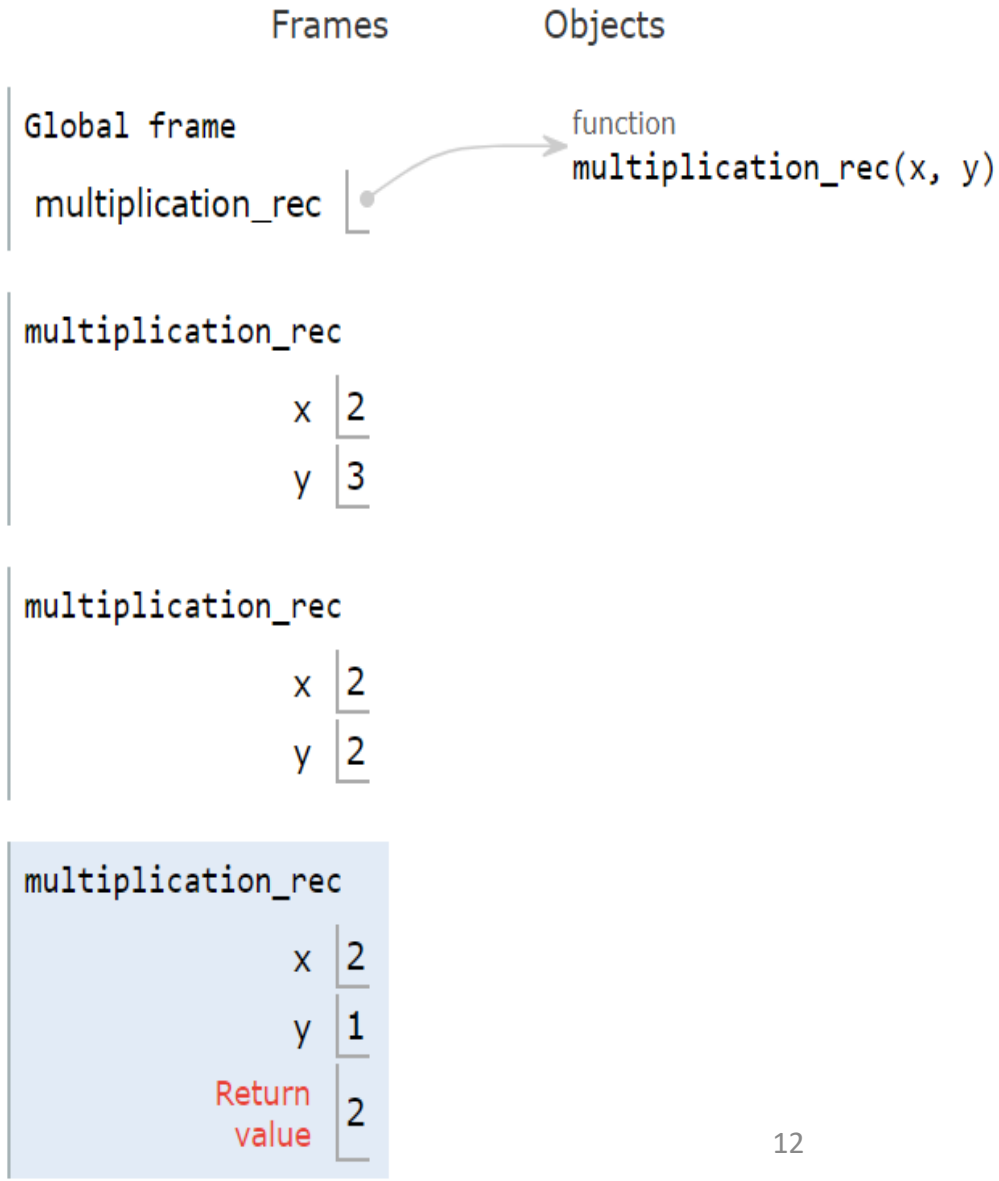
Python 3.6

```
1  def multiplication_rec(x, y):
2      if y == 1:
3          return x
4      else:
5          return x + multiplication_rec(x, y-1)
6
7  multiplication_rec(2,3)
```



Recursion

- each recursive call to a function creates its **own scope/environment**
- flow of control passes back to **previous scope** once function call returns value



Recursion vs Iterative algorithms

- recursion may be simpler, more intuitive
- recursion may be efficient for programmer but not for computers

```
def fib_iter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        a = 0
        b = 1
        for i in range(n-1):
            tmp = a
            a = b
            b = tmp + b
        return b
```

O(n)

```
def fib_recur(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

O(2^n)

Recursion : Proof by induction

How do we know that our recursive code will work ?

→ Mathematical Induction

To prove a statement indexed on integers is true for all values of n:

- Prove it is true when n is smallest value (e.g. n = 0 or n = 1)
- Then prove that if it is true for an arbitrary value of n, one can show that it must be true for n+1

Recursion : Proof by induction

- x * y = x + x * (y-1)

- Proof:
  - If y =1, x * 1 = x + x * (1-1) → True
  - Suppose that it is correct for some n : x * n = x + x * (n-1), lets prove it for n+1

    x * (n+1) = x * n + x

    = x + x * (n-1) + x

    = x + x * n → True

Thus by induction, code correctly returns answer

Recursion with multiple base cases

Example: **Fibonacci**

- **Fibonacci sequence** where each number is the sum of the two preceding ones, starting from 0 and 1.
- The beginning of the sequence is thus:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ….

Base cases: Fib(0) = 0 and Fib(1) = 1
Recursive case : Fib(n) = Fib(n-1) + Fib(n-2), for n > 1

```
def Fib(n):
    if n == 0
        return 0
    elif n == 1:
        return 1
    else:
        return Fib(n-1) + Fib(n-2)

print(Fib(9))
```

Algorithmic Paradigms

Greedy algorithms

Divide and Conquer Algorithms

Dynamic programming

Greedy algorithms

- A greedy algorithm sometimes works well for optimization problems.
- It works in phases, at each phase:
  - You take the best you can get right now, regardless of future consequences.
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum.

Greedy algorithms

Suppose you want to count out a certain amount of money, using the fewest possible bills and coins

→ Greedy solution : At each step, take the largest possible bill or coin that does not overshoot

Example: To make $6.39, you can choose:

- a $5 bill
- a $1 bill, to make $6
- a 25¢ coin, to make $6.25
- A 10¢ coin, to make $6.35
- four 1¢ coins, to make $6.39

→ **Optimum solution**

Greedy algorithms

- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins

→ Greedy solution to count out 15 krons:

- A 10 kron piece
- Five 1 kron pieces, for a total of 15 krons

A better solution would be to use two 7 kron pieces and one 1 kron piece!

→The greedy algorithm results in a solution, but not in an optimal solution

Greedy algorithms

- Locally Optimal (greedy part)
- irreversible
- Simple and appealing, but don’t always give the best solution
- Application scenarios:
  - Playing chess by making best move without lookahead
  - Giving fewest number of coins as change

Dynamic programming

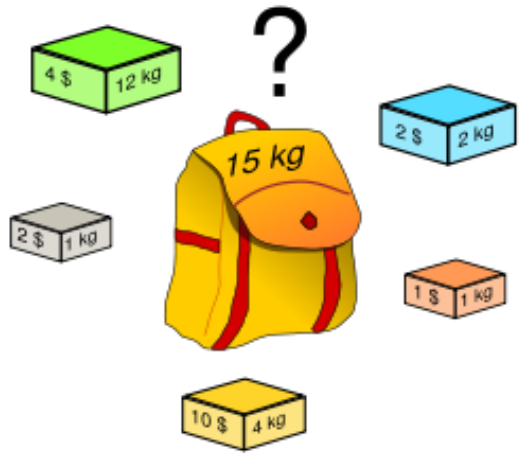
- Dynamic Programming (DP) is an algorithm design technique for *optimization problems:* often minimizing or maximizing.

- DP solves problems by combining solutions to sub-problems.

Dynamic programming: knapsack

Problem: Knapsack

- Given n objects and a “knapsack.”
- Item i has weight $w_i > 0$ and has value v_i
- Knapsack has capacity of W.
- Goal: maximize total value without overfilling knapsack



Dynamic programming: knapsack

Application scenarios:

- Packing goods of high value (or high importance) in a container.
- Allocating bandwidth to messages in a network.
- Placing files in fast memory. The values indicate access frequencies.
- In a simplified model of a consumer, the capacity is a budget, the values are utilities, and the consumer asks himself what he could buy to maximize his happiness.

Dynamic programming: knapsack (Poll)

Which one of these greedy solutions is the optimal one:

1- Greedy by value: Repeatedly add item with maximum v_i.

- Example: S={(3,5),(2,3),(2,3)}, W=4 ?

2- Greedy by weight: Repeatedly add item with minimum w_i.

- Example: S={(1,1),(3,2),(4,8)}, W=4 ?

3- Greedy by ratio: Repeatedly add item with maximum ratio v_i/w_i .

- Example: S={(3,5),(2,3),(2,3)}, W=4 ?

None of greedy algorithms is optimal!

Dynamic programming: knapsack

- OPT(i, w) = max profit subset of items 1, …, i with weight limit w.
  - Case 1: OPT does not select item i.

    OPT selects best of { 1, 2, …, i-1 } using weight limit w
  - Case 2: OPT selects item i.

    new weight limit = w – w_i

    OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Dynamic programming: knapsack

Example:

- n = 4 (# of elements)
- W = 5 (max weight)
- Elements (weight, value):
  (2,3), (3,4), (4,5), (5,6)

Dynamic programming: knapsack

Example:

Initialize the base cases

n = 4 (# of elements)
W = 5 (max weight)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$w = 1$
$w\text{-}w_i = -1$

IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 2}$
$w\text{-}w_i = 0$

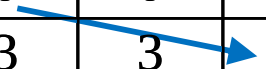
IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					



$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 5}$
$w\text{-}w_i = 3$

IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
w = 1
$w\text{-}w_i = -2$

IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 2}$
$w\text{-}w_i = -1$

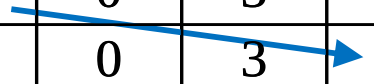
IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], vi + M [i – 1, w – wi] }.

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					



$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 3}$
$w\text{-}w_i = 0$

IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 4}$
$w\text{-}w_i = 1$

IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 5}$

$w\text{-}w_i = 2$

IF (w_i > w)

M [i, w] ← M [i – 1, w]

ELSE

M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

i = 3
$v_i = 5$
$w_i = 4$
w = 1..3
$w\text{-}w_i = -3..-1$

IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i = 3$
$v_i = 5$
$w_i = 4$
w = 4
$w\text{-}w_i = 0$

IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

$i = 3$
$v_i = 5$
$w_i = 4$
$\mathbf{w = 5}$
$w\text{-}w_i = 1$

IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

i = 4
v_i = 6
w_i = 5
w = 1..4
w-w_i = -4..-1

IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 4$
$v_i = 6$
$w_i = 5$
$\mathbf{w = 5}$
$w\text{-}w_i = 0$

IF (w_i > w)
M [i, w] ← M [i – 1, w]
ELSE
M [i, w] ← max { M [i – 1, w], v_i + M [i – 1, w – w_i] }

Dynamic programming: knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	**7**

We're DONE!!

The max possible value that can be carried in this knapsack is *$7*

The optimal knapsack should contain:
Item 1 and Item 2

Dynamic programming

- Reduce time by increasing the amount of space
- Solve the problem by solving sub-problems of increasing size and saving each optimal solution in a table (usually).
- The table is then used for finding the optimal solution to larger problems.
- Time is saved since each sub-problem is solved only once.

Dynamic Programming versus Greedy

Dynamic programming can be viewed as restricted exhaustive search, but also as an extension of the greedy paradigm.

→ Instead of following only one path of currently optimal decisions, we follow all such paths that might bring us to the optimum which is feasible only if there are not too many paths to follow.

Divide and Conquer

Consider how people find information in a phone book or dictionary

1. the goal is to search for a word *w* in region of the book
2. the initial region is the entire book
3. at each step pick a word *x* in the middle of the current region
4. there are now two smaller regions: the part before *x* and the part after *x*
5. if *w* comes before *x*, repeat the search on the region before *x*, otherwise search the region following *x* (go back to step 3)

Divide and Conquer

- Divide-and-conquer.
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.

- Most common usage.
  - Break up problem of size n into two equal parts of size ½n.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.

- Applications:
  - Binary search
  - Merge sort

Divide and Conquer vs DP and Greedy Algorithms

- Both the greedy approach and dynamic programming extend solutions from smaller sub-instances incrementally to larger sub-instances.

- Divide and Conquer approach still follows the pattern of reducing a given problem to smaller instances of itself, but it makes jumps rather than incremental steps.

Searching and sorting algorithms

Search algorithms

Goal: finding an item or group of items with specific properties within a collection of items

Search algorithms

Linear search

- **brute force** search
- list does not have to be sorted

Binary search

- list MUST be sorted to give correct answer

Linear search: unsorted list (Poll)

```
def linear_search(L, element):
    found = False
    for i in range(len(L)):
        if element == L[i]:
            found = True
    return found
```

O(len(L))
O(1)

→ O(n), where n is the length of the list

Worst case: must go through all the list to decide that the element is not here

Linear search: sorted list (Poll)

```
def search(L, element):
    for i in range(len(L)):
        if L[i] == element:
            return True
        if L[i] > element:
            return False
    return False
```

O(len(L))
O(1)

→ O(n), where n is the length of the list

Must only look until reach a number greater than element

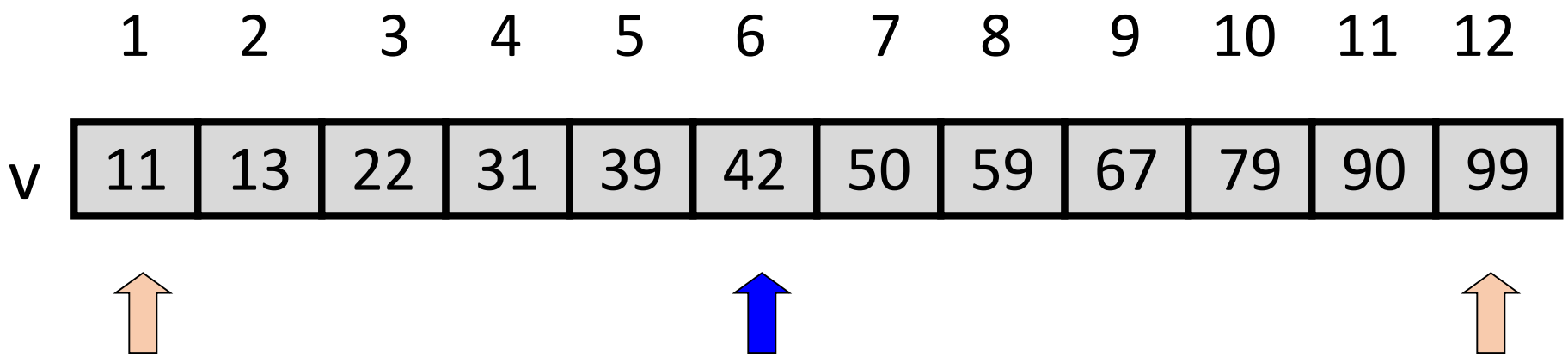
Binary search: Sorted list

1. Pick an index, i, that divides list in half
2. Check if L[i] == element
3. If not, check if L[i] is larger or smaller than element.
4. Depending on answer, search left or right half of L for element

→ A new version of a divide-and-conquer algorithm :
Break into smaller version of problem (smaller list), plus some simple operations
Answer to smaller version is answer to original problem

Binary search: Sorted list

target x = 65



L: 1

Mid: 6

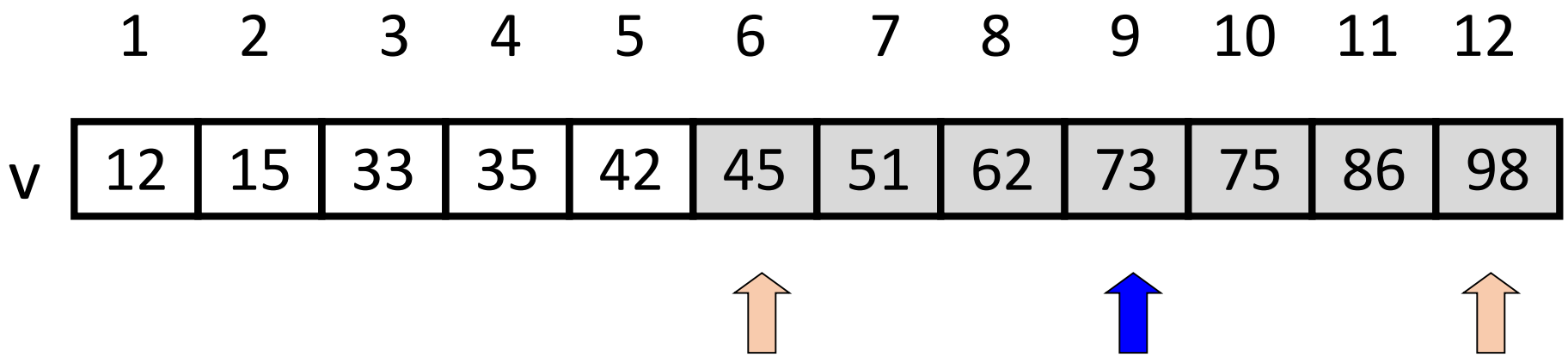
R: 12

v(Mid) <= x

So throw away the left half…

Binary search: Sorted list

target x = 65



L: 6

Mid: 9

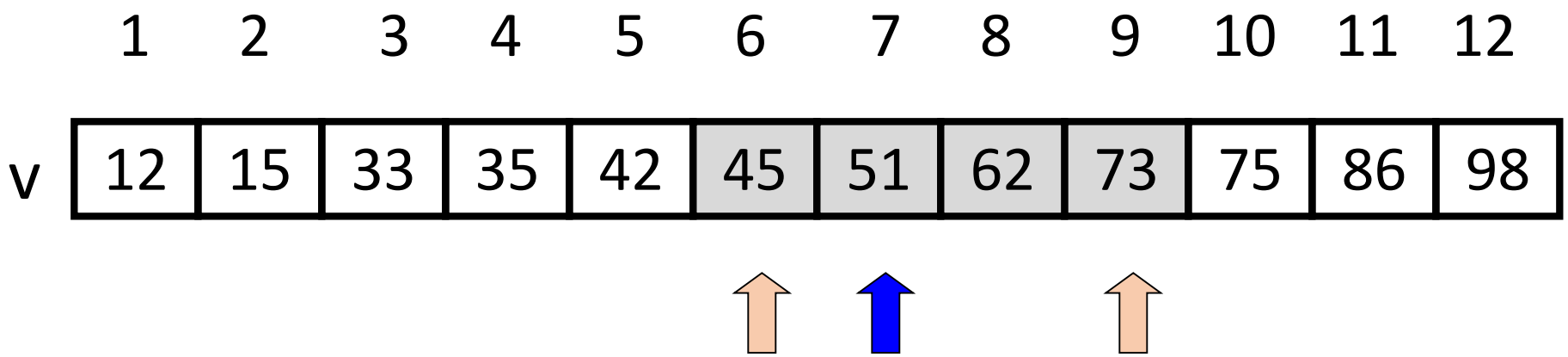
R: 12

x < v(Mid)

So throw away the right half…

Binary search: Sorted list

target x = 65



L: 6

Mid: 7

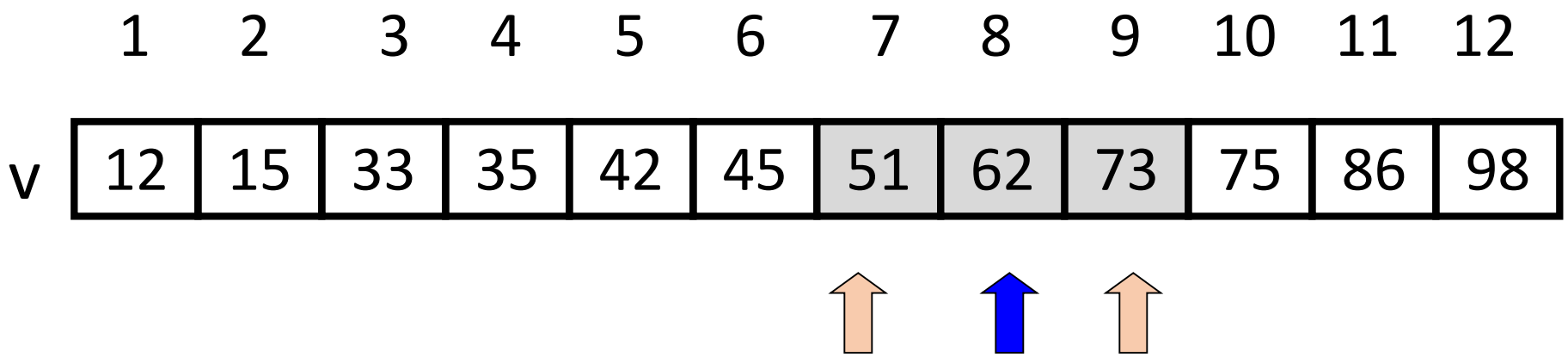
R: 9

v(Mid) <= x

So throw away the left half…

Binary search: Sorted list

target x = 65



L: 7

Mid: 8

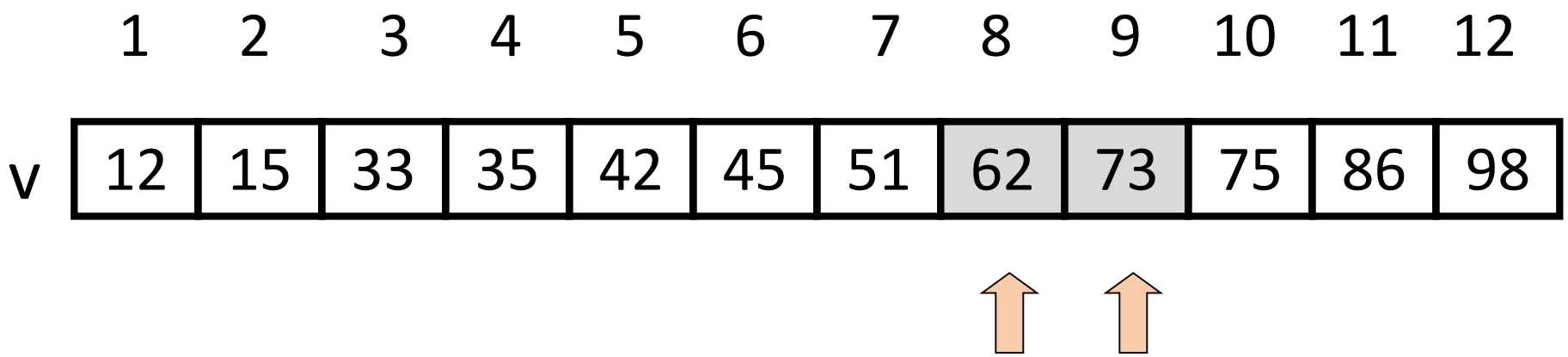
R: 9

v(Mid) <= x

So throw away the left half…

Binary search: Sorted list

target x = 65



L: 8

Mid: 8

R: 9

Done because

R-L = 1

Binary search: Sorted list

```
def bisect_search(L, e):
    def bisect_search_helper(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                 return bisect_search_helper(L, e, low, mid - 1)
        else:
            return bisect_search_helper(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bisect_search_helper(L, e, 0, len(L) - 1)

print(bisect_search([1,4,7,9,13,25,45,76], 45))
```

Every comparison reduces our "search space" by a factor 2, hence we are done after O(log n) time.

Sort Algorithms

- Goal: efficiently sort a list of elements

- Several methods…

Sort algorithms: Bogosort

Bogosort (permutation sort, stupid sort, slowsort, shotgun sort, random sort, monkey sort, bobosort or shuffle sort) :

- is a highly inefficient sorting algorithm based on the generate and test paradigm.
- The function successively generates permutations of its input until it finds one that is sorted.

Sort algorithms: Bogosort (Poll)

```
import random

def bogoSort(l):
    n = len(l)
    nb_it =0
    while (is_sorted(l) == False):
        shuffle(l)
        nb_it+=1
    return (l, nb_it)

def is_sorted(l):
    n = len(l)
    for i in range(0, n - 1):
        if (l[i] > l[i + 1]):
            return False
    return True

def shuffle(l):
    n = len(l)
    for i in range(0, n):
        r = random.randint(0, n - 1)
        l[i], l[r] = l[r], l[i]
```

best case: **O(n) where n is len(l)** to check if sorted
worst case: O(?) it is **unbounded** if really unlucky

Sort algorithms: Bubble sort

Bubble sort: it compares repeatedly adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

- **compare consecutive pairs** of elements
- **swap elements** in pair such that smaller is first
- when reach end of list, **start over** again
- stop when **no more swaps** have been made

Sort algorithms: Bubble sort (Poll)

```
def bubbleSort(l):
    n = len(l)
    for i in range(n-1):
        for j in range(0, n-i-1):
            if l[j] > l[j+1] :
                l[j], l[j+1] = l[j+1], l[j]
```

O(len(l))
O(len(l))

→ O(n^2), where n is the length of the list

Sort algorithms: Selection sort

Selection sort: is an in-place comparison sorting algorithm

- first step:
  - extract **minimum element**
  - **swap it** with element at **index 0**
- subsequent step
  - in remaining sublist, extract **minimum element**
  - **swap it** with the element at **index 1**
- keep the left portion of the list sorted
  - at i'th step, **first i elements in list are sorted**
  - all other elements are bigger than first i elements

Sort algorithms: Selection sort (Poll)

```
def selection_sort(l):
    for i in range(len(l)):
        min_idx = i
        for j in range(i + 1, len(l)):
            if l[min_idx] > l[j]:
                min_idx = j
        l[i], l[min_idx] = l[min_idx], l[i]
    return l
```

O(len(l))

O(len(l))

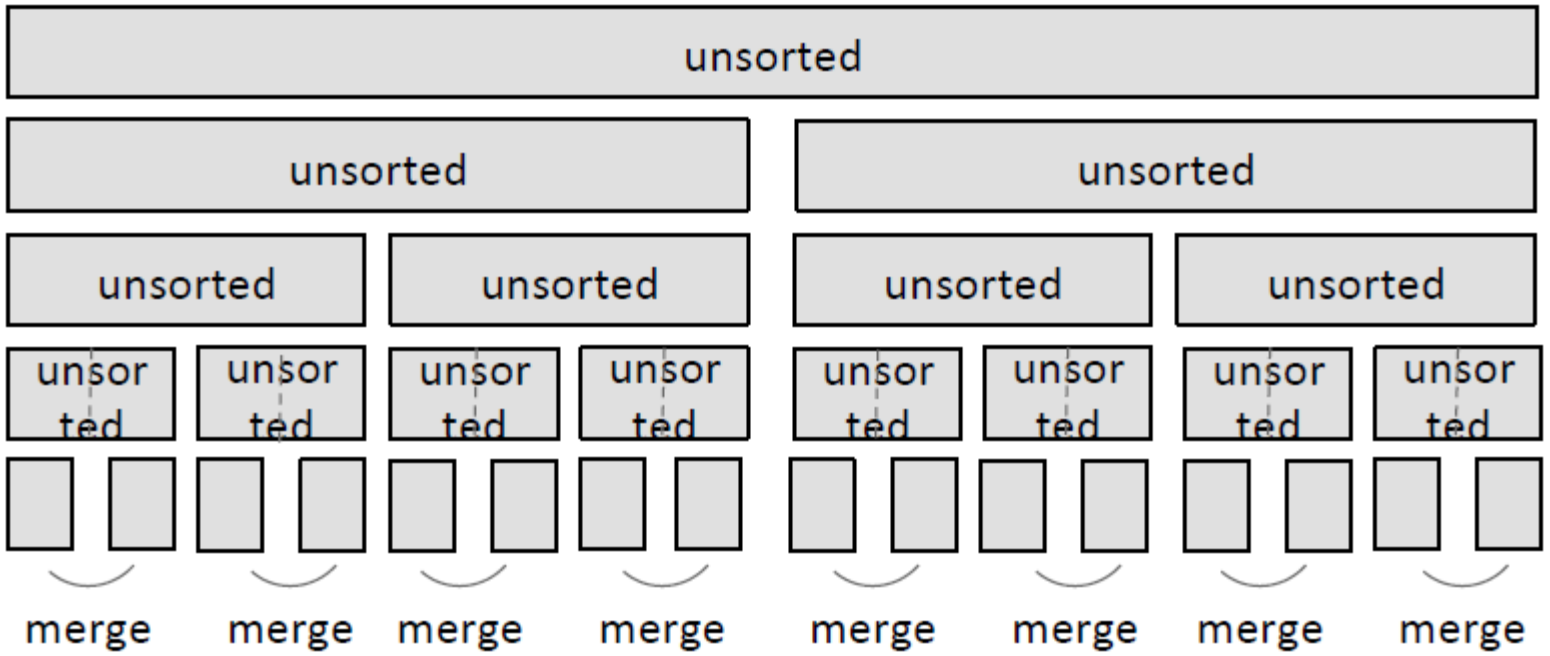
→ O(n^2), where n is the length of the list

Sort algorithms: Merge sort

Merge sort is a divide and conquer algorithm, it is a comparison-based sorting algorithm

- if list is of length 0 or 1, already sorted
- if list has more than one element, split into two lists, and sort each
- merge sorted sublists
  - look at first element of each, move smaller to end of the result
  - when one list empty, just copy rest of other list

Sort algorithms: Merge sort



Sort algorithms: Merge sort

```
def mergeSort(l):
    if len(l) > 1:
        mid = len(l) // 2
        L = l[:mid]
        R = l[mid:]
        mergeSort(L)
        mergeSort(R)

        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                l[k] = L[i]
                i += 1
            else:
                l[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            l[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            l[k] = R[j]
            j += 1
            k += 1
    return l
```

Use http://www.pythontutor.com/visualize.html#mode=display to better understand each step

Sort algorithms: Merge sort

- at **first recursion level**
  - n/2 elements in each list
  - O(n) + O(n) = O(n) where n is len(L)
- at **second recursion level**
  - n/4 elements in each list
  - two merges O(n) where n is len(L)
- each recursion level is O(n) where n is len(L)
- **dividing list in half** with each recursive call
  - O(log(n)) where n is len(L)
- overall complexity is **O(n log(n)) where n is len(L)**

Sorting algorithms

- bogo sort
  - randomness, unbounded O()
- bubble sort
  - O(n^2)
- selection sort
  - O(n^2)
  - guaranteed the first i elements were sorted
- merge sort
  - O(n log(n))

→ O(n log(n)) is the fastest a sort can be