The recursive book of recursion - Divide and conquer algorithms

Tuesday, 22 August 2023 6:45 PM

Binary Search

```
def binarySearch(needle, haystack, left=None, right=None):
    # By default, `left` and `right` are all of `haystack`:
    if left is None:
        left = 0 # `left` defaults to the 0 index.
    if right is None:
        right = len(haystack) - 1 # `right` defaults to the last index.
    print('Searching:', haystack[left:right + 1])
    if left > right: # BASE CASE
        return None # The `needle` is not in `haystack`.
    mid = (left + right) // 2
    if needle == haystack[mid]: # BASE CASE
        return mid # The `needle` has been found in `haystack`
    elif needle < haystack[mid]: # RECURSIVE CASE
        return binarySearch(needle, haystack, left, mid - 1)
    elif needle > haystack[mid]: # RECURSIVE CASE
        return binarySearch(needle, haystack, mid + 1, right)
print(binarySearch(13, [1, 4, 8, 11, 13, 16, 19, 19]))
```

Dry run:

Initial case: needle = 13, haystack = [el ... lements], left = none, right = none

1st case: left = None ⇒ left = 0
right = None ⇒ right = len (... haystack) ⇒ right = 9

Searching → printed, haystack ... [0:9+1] ⇒ 0 to 10-1

if 0>9 (false) # Base case

mid = $\frac{0+9}{2}$ = 9/2 = $\frac{4.5}{2}$ = ...

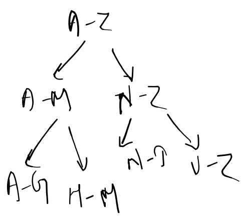
if 13 == haystack[mid] = ... ⇒ 13 == haystack[5] ⇒ 13 == 16 (fal

elif 13 < 16 (true) ⇒ re... turn (13, haystack, 0, 5-1 (4))

if 13 > 16 (false)

je)

Quicksort




```
def quicksort(items, left=None, right=None):
    # By default, `left` and `right` span the entire range of `items`:
    if left is None:
        left = 0 # `left` defaults to the 0 index.
    if right is None:
        right = len(items) - 1 # `right` defaults to the last index.
print('\nquicksort() called on this range:', items[left:right + 1])
    print('................The full list is:', items)
if right <= left: 1
# With only zero or one item, `items` is already sorted. return # BASE CA
SE

# START OF THE PARTITIONING
i = left # i starts at the left end of the range. 2
pivotValue = items[right] # Select the last value for the pivot.
```

Dry run

Initialisation: items = [0, 7, 6, 3, 1, 2

1st case: if left = none (true), left
if right = none (true), rig

print (quicksort is called
print (........ The full

- if right <= left: , 7 <=
Base case
return (the lis

start partitioning

-,5,4] , left = none, right = none

= 0

ult = len(items), right = 8-1 = 7

on this range: items[0, 8]
list is: [0,7,6,3,1,2,5,4]

0 (false)

t is already sorted)

```
    print('....................The pivot is:', pivotValue)
    # Iterate up to, but not including, the pivot:
    for j in range(left, right):
        # If a value is less than the pivot, swap it so that it's on the
        # left side of `items`:
        if items[j] <= pivotValue:
            # Swap these two values:
            items[i], items[j] = items[j], items[i] 3
            i += 1

    # Put the pivot on the left side of `items`:
    items[i], items[right] = items[right], items[i]
    # END OF THE PARTITIONING
    print('....After swapping, the range is:', items[left:right + 1])
    print('Recursively calling quicksort on:', items[left:i], 'and', items[i + 1:
right + 1])
    # Call quicksort() on the two partitions:
    quicksort(items, left, i - 1)   # RECURSIVE CASE
    quicksort(items, i + 1, right)  # RECURSIVE CASE
myList = [0, 7, 6, 3, 1, 2, 5, 4]
quicksort(myList)
print(myList)
```

i = left , i = 0
pivot = items[right]
print (The pivot value
for j in range(left
if a value is
if item[j] <
item[i], item[
i += 1 , i =
7 <= 4 false
6 <= 4 false
3 <= 4 true
7, 3 = 3, 7

1 , pivot = 4

- is 4)

e, pivot):
less than the pivot, swap it to the le

= pivot : item[0] , 0 <= 4 true, -

j] = item[j], item[i] , 0,0 = 0,0

= 1

ft of the items

(i)

Alternative approach:

```
def quicksort(arr):
    if len(arr) <= 1:
        # base case
        return arr

    # recursive case
    pivot = arr[len(arr)//2]
    left = [i for i in arr if i < piv
    middle = [i for i in arr if i ==
    right = [i for i in arr if i >
```

items [i] , it ... ems[right] = items[right] , items[i ...]

7 , 4 = ... 4, 7

ot]

pivot]

pivot]

return quicksort(left) + middle + quicksort(right)

Dry run

base case : arr = [ ] or arr = [1] or len(arr) = 1

arr = [5, 4, 1, 0, 3]

1) pivot = arr[len(arr)//2] = arr[2] = 1

left = [0]
middle = [1]
right = [5, 4, 3]

return quicksort([0]) + [1] + quicksort([5,4,3])

2) pivot = arr[len([5,4,3])//2] = 3//2 = 1
= arr[1] = 4

left = [3]
middle = [4]
right = [5]

return [3] + [4] + [5]

3: Final: [0]+[1]+[3]+[4]+[5]

[0, 1, 3, 4, 5]

Merge Sort

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    return merge(left_half, right_half)

def merge(left, right):
    result = []
    left_index, right_index = 0, 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            result.append(left[left_index])
            left_index += 1
        else:
            result.append(right[right_index])
```

Dry run

arr = [3, 6, 8, 10, 1, 2, 1]

1) mid = len(arr)//2 = len(7)//2 = 3
left_half = arr[: mid] = [3, 6, 8]
right_half = arr[mid:] = [10, 1, 2, 1]

left half = merge_sort([3, 6, 8])

mid = 3//2 = 1
left_half = [3]
right_half = [6, 8]

```
            right_index += 1

    result.extend(left[left_index:])
    result.extend(right[right_index:])

    return result

# Example usage
input_array = [3, 6, 8, 10, 1, 2, 1]
sorted_array = merge_sort(input_array)
print(sorted_array)
```

left-half = merge-sort([3])

right-half = merge-sort([6, 8])

mid = 2//2 = 1

left-half = [6]