Introduction to Computational and Algorithmic Thinking

CHAPTER 5 – DIVIDE AND CONQUER ALGORITHMS. RECURSION

1

Announcements

This lecture: Divide and conquer algorithms. Recursion

Reading: Read Chapter 5 of Conery

Acknowledgement: Some of this lecture slides are based on CSE 101 lecture notes by Prof. Kevin McDonald at SBU

 2

2

Divide and Conquer

- The strategy for the linear search and insertion sort algorithms was the same: iterate over every location in the list and perform some operation
- We will now look at a different strategy: **divide and conquer**
  - The idea: break a problem into smaller sub-problems and solve the smaller sub-problems
  - Sub-problems are chosen in such a way that their solutions can be combined to provide the solution to the original problem
- It may not seem like that big a deal, but the improvement can be dramatic, as we will see

Example: Searching a Dictionary

- To get a general sense of how the divide-and-conquer strategy improves search, consider how people find information in a (physical) phone book or dictionary
- Suppose you want to find “janissary” in a dictionary
  - Open the book near the middle
  - The heading on the top left page is “kiwi”, so move back a small number of pages
  - Here you find “hypotenuse”, so move forward
  - Find “ichthyology”, move forward again
- The number of pages you move gets smaller (or at least adjusts in response to the words you find)

Example: Searching a Dictionary

- A more detailed specification of this process:
  1. The goal is to search for a word *w* in a 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).
- Note: at first a "region" is a group of pages, but eventually a region is a set of words on a single page

A Note About Data Organization

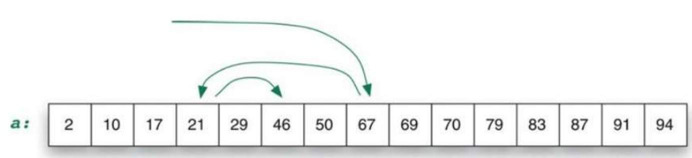
- An important note: an efficient search depends on having the data organized in some fashion
- If books in a library are scattered all over the place we have to do an iterative search
  - Start at one end of the room and progress toward the other
- If books are sorted or carefully cataloged we can try a more efficient method that exploits the sorted nature of the books

Binary Search: Overview

- The **binary search** algorithm uses the divide-and-conquer strategy to search through a list
- The list must be sorted for this algorithm to work properly
  - The “zeroing in” strategy for looking up a word in the dictionary won’t work if the words are not in alphabetical order
  - Similarly, binary search will not work for a list of values unless the list is sorted
- To search a list of n items, first look at the item in location $n/2$
- If this is the item we want, then the search has ended successfully
- Otherwise, search either the region from 1 to $(n/2)-1$ or the region from $(n/2)+1$ to n

Binary Search: Example

- Example: searching for 46 in a sorted list of 15 numbers:



- Note how the search moves backward and forward, quickly finding the **target element**

Binary Search: the Details

- The algorithm uses two variables to keep track of the boundaries of the region to search
  - **lower**: the index one position below the leftmost item in the region
  - **upper**: the index one position above the rightmost item in the region
- Initial values when searching a list of *n* items:

```
lower = -1
upper = n
```


9

Binary Search: the Details

- The algorithm is based on an iteration (loop) that keeps making the search region smaller and smaller
  - The initial region is the complete list
  - The next one is either the upper half or lower half
  - The one after that is one quarter, then one eighth, then…
- The heart of the algorithm contains these operations:
  - Set **mid** to a location halfway between lower and upper:

```
mid = (lower + upper) // 2
```

  - If the item is at this location, then we're done:

```
if a[mid] == x:
    return mid
```


10

Binary Search: the Details

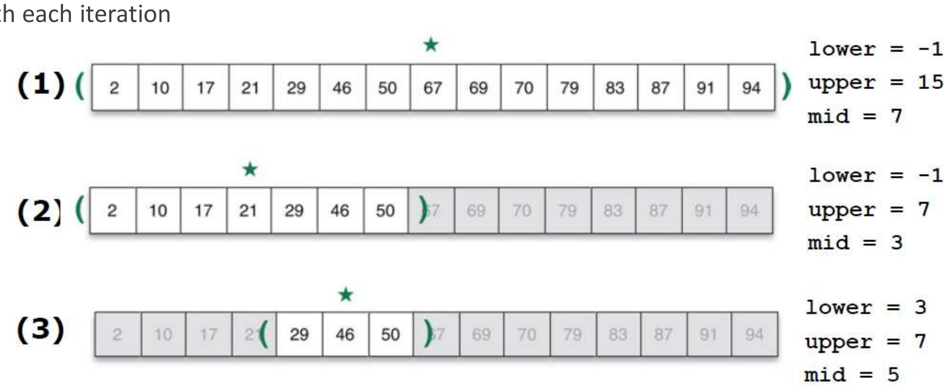
- The heart of the algorithm (continued):
  - Otherwise, move one of the "brackets" to the current mid-point for the next iteration:

```
if x < mid:
    upper = mid
else:
    lower = mid
```


11

Binary Search: the Details

- Let's revisit our example from earlier. The star in the figure shows how the value of mid changes with each iteration



12

Binary Search: the Details

- How do we handle the case when the target item is not in the list?
  - We have to add a condition that makes sure that **lower** is still to the left of **upper**
  - If the **upper** and **lower** pointers meet each other, this means that the search region has no elements in it – the search has failed
- We can now write the complete **bsearch** function, which returns:
  - The index of the target item in the list, when the search is successful, or
  - **None**, if the target item is not in the list

13

Completed bsearch() Function

```
def bsearch(a, x):
   lower = -1
   upper = len(a)
   while upper > lower + 1:
      mid = (lower + upper) // 2
      if a[mid] == x:
         return mid
      if x < a[mid]:
         upper = mid
      else:
         lower = mid
   return None
```

- See bsearch_tests.py for fully commented code

14

Trace Execution: bsearch()

a: [1, 2, 3, 5, 6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
→ lower = -1
  upper = len(a)
  while upper > lower + 1:
    mid = (lower + upper) // 2
    if a[mid] == x:
      return mid
    if x < a[mid]:
      upper = mid
    else:
      lower = mid
  return None
```

Variable	Value
x	8

Trace Execution: bsearch()

a: [1, 2, 3, 5, 6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
→ lower = -1
  upper = len(a)
  while upper > lower + 1:
    mid = (lower + upper) // 2
    if a[mid] == x:
      return mid
    if x < a[mid]:
      upper = mid
    else:
      lower = mid
  return None
```

Variable	Value
x	8
lower	-1

Trace Execution: bsearch()

```
a: [1, 2, 3, 5, 6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:
        mid = (lower + upper) // 2
        if a[mid] == x:
            return mid
        if x < a[mid]:
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	-1
upper	10

17

Trace Execution: bsearch()

```
a: [1, 2, 3, 5, 6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:      # True
        mid = (lower + upper) // 2
        if a[mid] == x:
            return mid
        if x < a[mid]:
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	-1
upper	10

18

Trace Execution: bsearch()

```
a: [1, 2, 3, 5, 6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9
def bsearch(a, x):
  lower = -1
  upper = len(a)
  while upper > lower + 1:
→   mid = (lower + upper) // 2
    if a[mid] == x:
      return mid
    if x < a[mid]:
      upper = mid
    else:
      lower = mid
  return None
```

Variable	Value
x	8
lower	-1
upper	10
mid	4

19

Trace Execution: bsearch()

```
a: [1, 2, 3, 5, 6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9
def bsearch(a, x):
  lower = -1
  upper = len(a)
  while upper > lower + 1:
    mid = (lower + upper) // 2
→   if a[mid] == x:         # False
      return mid
    if x < a[mid]:
      upper = mid
    else:
      lower = mid
  return None
```

Variable	Value
x	8
lower	-1
upper	10
mid	4
a[mid]	6

20

Trace Execution: bsearch()

a: [1, 2, 3, 5, 6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:
        mid = (lower + upper) // 2
        if a[mid] == x:
            return mid
        if x < a[mid]:           # False
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	-1
upper	10
mid	4
a[mid]	6

Trace Execution: bsearch()

a: 1, 2, 3, 5, [6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:
        mid = (lower + upper) // 2
        if a[mid] == x:
            return mid
        if x < a[mid]:
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	4
upper	10
mid	4
a[mid]	6

Trace Execution: bsearch()

a: 1, 2, 3, 5, [6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:      # True
        mid = (lower + upper) // 2
        if a[mid] == x:
            return mid
        if x < a[mid]:
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	4
upper	10
mid	4
a[mid]	6

Trace Execution: bsearch()

a: 1, 2, 3, 5, [6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:
        mid = (lower + upper) // 2
        if a[mid] == x:
            return mid
        if x < a[mid]:
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	4
upper	10
mid	7
a[mid]	11

Trace Execution: bsearch()

a: 1, 2, 3, 5, [6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:
        mid = (lower + upper) // 2
        if a[mid] == x:        # False
            return mid
        if x < a[mid]:
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	4
upper	10
mid	7
a[mid]	11

25

Trace Execution: bsearch()

a: 1, 2, 3, 5, [6, 8, 9, 11, 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:
        mid = (lower + upper) // 2
        if a[mid] == x:
            return mid
        if x < a[mid]:        # True
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	4
upper	10
mid	7
a[mid]	11

26

Trace Execution: bsearch()

a: 1, 2, 3, 5, [6, 8, 9, 11], 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:
        mid = (lower + upper) // 2
        if a[mid] == x:
            return mid
        if x < a[mid]:
→           upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	4
upper	7
mid	7
a[mid]	11

27

Trace Execution: bsearch()

a: 1, 2, 3, 5, [6, 8, 9, 11], 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
→   while upper > lower + 1:      # True
        mid = (lower + upper) // 2
        if a[mid] == x:
            return mid
        if x < a[mid]:
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	4
upper	7
mid	7
a[mid]	11

28

Trace Execution: bsearch()

a: 1, 2, 3, 5, [6, 8, 9, 11], 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:
        mid = (lower + upper) // 2
        if a[mid] == x:
            return mid
        if x < a[mid]:
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	4
upper	7
mid	5
a[mid]	8

Trace Execution: bsearch()

a: 1, 2, 3, 5, [6, 8, 9, 11], 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:
        mid = (lower + upper) // 2
        if a[mid] == x:          # True
            return mid
        if x < a[mid]:
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	4
upper	7
mid	5
a[mid]	8

Trace Execution: bsearch()

a: 1, 2, 3, 5, [6, 8, 9, 11], 14, 17]
index: 0 1 2 3 4 5 6 7 8 9

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:
        mid = (lower + upper) // 2
        if a[mid] == x:
➡           return mid
        if x < a[mid]:
            upper = mid
        else:
            lower = mid
    return None
```

Variable	Value
x	8
lower	4
upper	7
mid	5
a[mid]	8

Completed bsearch() Function

In the PythonLabs module called RecursionLab there is a function named **print_bsearch_brackets** that will let us visualize how the **lower** and **upper** pointers change as the search progresses

The call to this function goes near the top of the loop

See the code on the next slide

bsearch() with Print-outs

```
def bsearch(a, x):
  lower = -1
  upper = len(a)
  while upper > lower + 1:
    mid = (lower + upper) // 2
    print_bsearch_brackets(a,lower,mid,upper)
    if a[mid] == x:
      return mid
    if x < a[mid]:
      upper = mid
    else:
      lower = mid
  return None
```


bsearch() Example

- list: **[5, 12, 16, 40, 58, 62, 72, 84, 88, 90]**
- target element: 72
- In the sample visualizations below, the brackets indicate the current
- search region, and * indicates the middle element

  [5 12 16 40 *58 62 72 84 88 90]
  5 12 16 40 58 [62 72 *84 88 90]
  5 12 16 40 58 *62 72] 84 88 90
  5 12 16 40 58 62 [72] 84 88 90
- Result: 6

bsearch() Example

- list: **[5, 12, 16, 40, 58, 62, 72, 84, 88, 90]**
- target element: 65 (not present in list)
- In the sample visualizations below, the brackets indicate the current search region, and * indicates the middle element

  [5 12 16 40 *58 62 72 84 88 90]

  5 12 16 40 58 [62 72 *84 88 90]

  5 12 16 40 58 *62 72] 84 88 90
- Result: **None**

Cutting the Problem Down to Size

- It should be clear why we say that the binary search algorithm uses a divide-and-conquer strategy
  - The problem is to find an item within a given range
  - At each step, the problem is split into two equal sub-problems
  - Focus then turns to one sub-problem for the next step

Number of Comparisons

- The number of iterations made by this algorithm when it searches a list containing *n* items is roughly $\log_2 n$
- To see why, consider the question from the other direction
  - Let's start with a list containing 1 item. Suppose each step of an iteration doubles the size of the list
  - After *n* steps, we will have 2^n items in the list
- By definition of logarithm, if $x=2^y$, then $y=\log_2 x$
- During searching we're reducing a region of size n down to a region of size 1
- A successful search might return after the first comparison
- An unsuccessful search does all $\log_2 n + 1$ comparisons
- Exampe: 8->4->2->1, or 4 comparisons (note: $\log_2 8 = 3$)

Searching Long Lists

- Divide-and-conquer might seem like a lot of extra work for such a simple problem (searching)
- For large lists, however, that work leads to a very efficient search
- We would need at most 30 comparisons to find something in a list of 1 billion items
- The worst case for linear search would be 1 billion comparisons!

n	log↓2 *n* (rounded up)
2	1
4	2
8	3
16	4
1,000	10
2,000	11
4,000	12
1,000,000	20
1,000,000,000	30
1,000,000,000,000	40

Divide and Conquer Sorting

- The divide-and-conquer strategy used to make a more efficient search algorithm can also be applied to **sorting**
- Two well-known sorting algorithms:
  - **Merge Sort**: sort subgroups of size 2, merge them into sorted groups of size 4, merge those into sorted groups of size 8, and so on
  - **Quicksort**: divide a list into big values and small values, then sort each part
- Let's first explore merge sort and see how it can be implemented in Python

Merge Sort

- The merge sort algorithm works from "the bottom up"
  - Start by solving the smallest pieces of the main problem
  - Keep combining their results into larger solutions
  - Eventually the original problem will be solved
- Example: sorting playing cards
  - Divide the cards into groups of two
  - Sort each group, putting the smaller of the two on the top
  - Merge groups of two into groups of four
  - Merge groups of four into groups of eight
  - and so on ...

Merge Sort: Example

- Let's try an example of merge sort with 7 playing cards:

2 Q J 7 A 10 5	Original list
2 Q 7 J 10 A 5	Sorted pairs
2 7 J Q 5 10 A	Merged pairs into sorted groups of 3 or 4
2 5 7 10 J Q A	Merged smaller groups into 1 large sorted group

Merge Sort

- What makes this method more effective than simple insertion sort?
  - **Merging** two piles is a very simple operation
  - Only need to look at the two cards currently on the top of each pile
  - No need to look deeper into either group
- In our example, we had these two piles at one point:
  - **[2 7 J Q]** and **[5 10 A]**
  - Compare 2 with 5, pick up the 2
  - Compare 5 with 7, pick up the 5
  - Compare 7 with 10, pick up the 7
  - and so on ...

Merge Sort Visualization

- See msort_visualization.py for an animation of merge sort
- Also see visualgo.net/en/sorting
- Watching a few animations of merge sort in action will give you a stronger sense of how the algorithm sorts a list of values

Implementing Merge Sort

- We will now see how to implement merge sort as a function called **msort**
- **msort** depends on two helper functions:
  - **merge**, which merges two sorted lists into one. This function is already implemented in the built-in **heapq** module in Python.
  - **merge_groups**, which calls **merge** and tells it exactly which sub-lists of the original list to merge

Implementing Merge Sort (next)

- Let's look at an example of the **merge** function so we understand how it works

```
import heapq
list1 = [1, 4, 6, 8]
list2 = [2, 5, 7, 9, 10, 13, 19]
merged_list = heapq.merge(list1, list2)
```

- **merged_list** will be:
  [1, 2, 4, 5, 6, 7, 8, 9, 10, 13, 19]

The merge_groups Function

- A helper function which we will write ourselves is **merge_groups**
- The **merge_groups** function takes two arguments: the list and the size of a group, **group_size** (e.g., 2, 4, 8, ...)
  - It takes adjacent groups of *sorted* values two at a time and merges them into single groups
  - For example, if the group size is 2, this means that **merge_groups** will merge adjacent pairs into quartets
- The function depends on Python's ***slicing notation***, which works with lists and strings
  - Code like **nums[i:j]** means "create a new list containing elements **i** through **j-1** of **nums**"

Slicing Examples

- Example of slicing:

```
nums = [23, 6, 21, 45, 82, 4, 10]
         0  1 2   3   4  5  6
print(nums[2:6])
```

- Output: **[21, 45, 82, 4]**
- Slicing notation can be used to change the contents of a list:

```
nums[1:3] = [11, 22, 33]
```

- **nums** becomes:

  [23, 11, 22, 33, 45, 82, 4, 10]
- Note: **6** and **21** have been replaced with the numbers in red

(C) ARTHUR LEE, TONY MIONE, PRAVIN PAWAR, ALEX KUHN - SUNY KOREA - CSE 101 47

47

Slicing Examples 2

- Example of slicing:

```
names = ['Abe', 'Barbara', 'Chris', 'Dave', 'Erin', 'Frank', 'Harry']
print(names[2:6])
```

- Output: **['Chris', 'Dave', 'Erin', 'Frank']**
- Slicing notation can be used to change the contents of a list:

```
names[1:3] = ['Mike', 'Nathan', 'Opal']
```

- **names** becomes:

  ['Abe', 'Mike', 'Nathan', 'Opal', 'Dave', 'Erin', 'Frank', 'Harry']
- Note: **'Barbara'** and **'Chris'** have been replaced with the words in red

(C) ARTHUR LEE, TONY MIONE, PRAVIN PAWAR, ALEX KUHN - SUNY KOREA - CSE 101 48

48

The merge_groups Function

- To understand how **merge_groups** needs to work, consider the task of merging two quartets into one octet ("quartets" means **group_size = 4**)
  - The two quartets are adjacent to each other in the list
  - Generally, there are several or many such pairs of quartets we need to merge together, and we have to merge all such pairs of quartets into octets
- We need variables to tell us where each pair of quartets begins
- Call these variables **i** and **j**
  - **i** is the starting index of the first quartet
  - **j** is the starting index of the second quartet

The merge_groups Function

- After merging those quartets together, we need to move to the next two quartets
- They can be found at indexes **i+8** and **j+8** since we need to skip over the octet we just created
- Initially, **i = 0** and **j = 4**
- For the second iteration, **i = 8** and **j = 12**
  - Note that **j = i + 4** which means that **j = i + group_size**
- Next, **i = 16**, **j = 20** (again, **j = i + group_size**)
- In general, after merging two groups together,
  - **i** will increase by 2 × **group_size** and
  - **j** will simply become **j = i + group_size**

The merge_groups Function

- Now that we have worked out this logic, we can implement the **merge_groups** function:

```
def merge_groups(a, group_size):
  for i in range(0, len(a), 2*group_size):
    j = i + group_size
    k = j + group_size
    a[i:k] = list(heapq.merge(a[i:j], a[j:k]))
```

- The for-loop doubles the group size until the list is just one large group
- See merge_groups_tests.py for examples of this function in action

Trace Execution: merge_groups

```
def merge_groups(a, group_size):
  for i in range(0, len(a), 2*group_size):
    j = i + group_size
    k = j + group_size
    a[i:k] = list(heapq.merge(a[i:j], a[j:k]))
```

- **nums: [8, 6, 7, 5, 3, 1, 2, 4]**
- Function call: **merge_groups(nums, 1)**
- a: **[8, 6, 7, 5, 3, 1, 2, 4] group_size = 1**
- The loop will iterate as: **i = 0, 2, 4, 6**

Trace Execution: merge_groups

```
def merge_groups(a, group_size):
    for i in range(0, len(a), 2*group_size):
        j = i + group_size
        k = j + group_size
        a[i:k] = list(heapq.merge(a[i:j], a[j:k]))
```

- **nums: [8, 6, 7, 5, 3, 1, 2, 4]**
- Function call: **merge_groups(nums, 1)**
- **a**: **[8, 6, 7, 5, 3, 1, 2, 4] group_size = 1**
- **i = 0**
- **j = i + 1 = 1**
- **k = j + 1 = 2**
- **a[0:2] = merge(a[0:1], a[1:2])**
- **a**: **[6, 8, 7, 5, 3, 1, 2, 4]**

Trace Execution: merge_groups

```
def merge_groups(a, group_size):
    for i in range(0, len(a), 2*group_size):
        j = i + group_size
        k = j + group_size
        a[i:k] = list(heapq.merge(a[i:j], a[j:k]))
```

- **nums: [8, 6, 7, 5, 3, 1, 2, 4]**
- Function call: **merge_groups(nums, 1)**
- **a**: **[6, 8, 7, 5, 3, 1, 2, 4] group_size = 1**
- **i = 2**
- **j = i + 1 = 3**
- **k = j + 1 = 4**
- **a[2:4] = merge(a[2:3], a[3:4])**
- **a**: **[6, 8, 5, 7, 3, 1, 2, 4]**

Trace Execution: merge_groups

```
def merge_groups(a, group_size):
    for i in range(0, len(a), 2*group_size):
        j = i + group_size
        k = j + group_size
        a[i:k] = list(heapq.merge(a[i:j], a[j:k]))
```

- **nums: [8, 6, 7, 5, 3, 1, 2, 4]**
- Function call: **merge_groups(nums, 1)**
- **a**: **[6, 8, 5, 7, 3, 1, 2, 4] group_size = 1**
- **i = 4**
- **j = i + 1 = 5**
- **k = j + 1 = 6**
- **a[4:6] = merge(a[4:5], a[5:6])**
- **a**: **[6, 8, 5, 7, 1, 3, 2, 4]**

55

Trace Execution: merge_groups

```
def merge_groups(a, group_size):
    for i in range(0, len(a), 2*group_size):
        j = i + group_size
        k = j + group_size
        a[i:k] = list(heapq.merge(a[i:j], a[j:k]))
```

- **nums: [8, 6, 7, 5, 3, 1, 2, 4]**
- Function call: **merge_groups(nums, 1)**
- **a**: **[6, 8, 5, 7, 1, 3, 2, 4] group_size = 1**
- **i = 6**
- **j = i + 1 = 7**
- **k = j + 1 = 8**
- **a[6:8] = merge(a[6:7], a[7:8])**
- **a**: **[6, 8, 5, 7, 1, 3, 2, 4]**

56

Trace Execution: merge_groups

```
def merge_groups(a, group_size):
    for i in range(0, len(a), 2*group_size):
        j = i + group_size
        k = j + group_size
        a[i:k] = list(heapq.merge(a[i:j], a[j:k]))
```

- **nums: [8, 6, 7, 5, 3, 1, 2, 4]**
- Function call: **merge_groups(nums, 2)**
- **a**: **[6, 8, 5, 7, 1, 3, 2, 4] group_size = 2**
- The loop will iterate as: **i = 0, 4**

57

Trace Execution: merge_groups

```
def merge_groups(a, group_size):
    for i in range(0, len(a), 2*group_size):
        j = i + group_size
        k = j + group_size
        a[i:k] = list(heapq.merge(a[i:j], a[j:k]))
```

nums: [8, 6, 7, 5, 3, 1, 2, 4]
Function call: **merge_groups(nums, 2)**
a: **[6, 8, 5, 7, 1, 3, 2, 4] group_size = 2**
i = 0
j = i + 2 = 2
k = j + 2 = 4
a[0:4] = merge(a[0:2], a[2:4])
a: **[5, 6, 7, 8, 1, 3, 2, 4]**

58

Trace Execution: merge_groups

```
def merge_groups(a, group_size):
  for i in range(0, len(a), 2*group_size):
    j = i + group_size
    k = j + group_size
    a[i:k] = list(heapq.merge(a[i:j], a[j:k]))
```

- **nums: [8, 6, 7, 5, 3, 1, 2, 4]**
- Function call: **merge_groups(nums, 2)**
- **a**: **[5, 6, 7, 8, 1, 3, 2, 4] group_size = 2**
- **i = 4**
- **j = i + 2 = 6**
- **k = j + 2 = 8**
- **a[4:8] = merge(a[4:6], a[6:8])**
- **a**: **[5, 6, 7, 8, 1, 2, 3, 4]**

59

Trace Execution: merge_groups

```
def merge_groups(a, group_size):
  for i in range(0, len(a), 2*group_size):
    j = i + group_size
    k = j + group_size
    a[i:k] = list(heapq.merge(a[i:j], a[j:k]))
```

- **nums: [8, 6, 7, 5, 3, 1, 2, 4]**
- Function call: **merge_groups(nums, 4)**
- **a**: **[5, 6, 7, 8, 1, 2, 3, 4] group_size = 4**
- The loop will iterate as: **i = 0, 8** (iterate once only)

60

Trace Execution: merge_groups

```
def merge_groups(a, group_size):
    for i in range(0, len(a), 2*group_size):
        j = i + group_size
        k = j + group_size
        a[i:k] = list(heapq.merge(a[i:j], a[j:k]))
```

- **nums: [8, 6, 7, 5, 3, 1, 2, 4]**

Function call: **merge_groups(nums, 4)**

a: **[5, 6, 7, 8, 1, 2, 3, 4] group_size = 4**

i = 0

j = i + 4 = 4

k = j + 4 = 8

a[0:8] = merge(a[0:4], a[4:8])

a: **[1, 2, 3, 4, 5, 6, 7, 8]**

Exercise: write merge Function

- Our **merge_groups** function uses **merge** from the **heapq** module
- Let's write our own **merge** function
  - It takes two parameters: list **u** and list **v**, both are sorted in increasing order
  - It returns a sorted list containing all the elements in **u** and **v**
- Write a **main** function that tests **merge** that you write

Completed msort Function

- All that remains now is to write **msort**, which will be straightforward with the help of **merge_groups** which in turn uses **merge**
- The main thing that **msort** needs to do is tell **merge_groups** how large each group is. But that's easy:
  - First, we take single elements and merge them into sorted pairs
  - Then, merge all the sorted pairs into sorted quartets
  - Next, merge all the sorted quartets into sorted octets
  - and so on ...
- Two implementations of **msort** (make sure you study these two)
  - msort.py using **heapq.merge**
  - msort2.py using our own **merge** (see the previous slide)
- If you want to visualize the progress of **msort**, you can call a function from RecursionLab called **print_msort_brackets** (continued on the next slides)

Completed msort Function

```
from PythonLabs.RecursionLab import
print_msort_brackets
def msort(a):
  size = 1
  while size < len(a):
    print_msort_brackets(a, size)      # optional
    merge_groups(a, size)
    size = size * 2
    print_msort_brackets(a, len(a))    # optional
```

- See msort_tests.py for a test run of the **msort** function

Completed msort Function

Example run of **msort**, with **print_msort_brackets**:

nums:
[33, 93, 7, 15, 50, 11, 65, 43]
[33] [93] [7] [15] [50] [11] [65] [43]
[33 93] [7 15] [11 50] [43 65]
[7 15 33 93] [11 43 50 65]
[7 11 15 33 43 50 65 93]

Comparisons in Merge Sort

- To completely sort a list with n items requires $\log_2 n$ iterations
  - Why? The group size starts at 1 and doubles with each iteration. The group size equals or exceeds n after $\log_2 n$ rounds of doubling
- During each iteration of **msort** there are at most n comparisons. Why?
  - Comparisons occur in the built-in **merge** method
  - It compares values at the front of each group
  - It may have to work all the way to the end of each group, but might stop early
  - So, the total number of comparisons is roughly $n \log_2 n$

Scalability of Merge Sort

- So, merge sort is a O($n\log_2 n$) algorithm
- Is the new formula that much better than the comparisons made by insertion sort?
- Not that big of a difference for small lists
- But as the length of the list increases, the savings start to add up

n	$n↑2$ /2	$n\log n$
8	32	24
16	128	64
32	512	160
1,000	500,000	10,000
5,000	12,500,000	65,000
10,000	50,000,000	140,000

Recursion

- An algorithm that uses divide and conquer can be written using iteration or **recursion**
- A **recursive** solution to a problem features "self-similarity", meaning that a function that solves a problem *calls itself*
- You actually already have familiarity with this concept
  - Consider the factorial operation in mathematics
  - n! = n x (n-1)! for integers n≥1, where 0! = 1
  - Note how factorial is defined in terms of itself (i.e., the ! Symbol appears on both sides of the equals sign)
  - This is a **recursive definition** of factorial
  - The simplest case of a recursive definition is called the *base case*

Recursion Example: Factorial

- Writing a recursive Python function that implements factorial is very straightforward
- We need to define both the recursive part (which is when the factorial function calls itself), and the base case

```
def factorial(n):
    if n == 0:          # base case
        return 1
    else:
        return n * factorial(n-1)
```

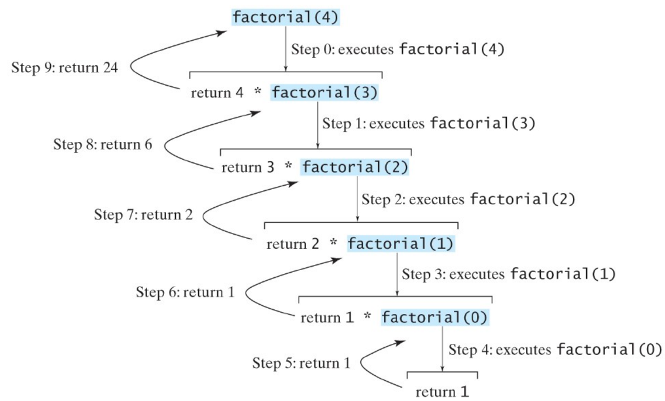
Recursive call to `factorial`

- See recursion_examples.py for code for many of the example recursive functions from these notes

Recursion

- All recursive functions have the following characteristics:
  - One or more **base cases** (the simplest cases) are used to stop recursion
  - One or more a **recursive calls** that reduce the original problem in size, bringing it increasingly closer to a base case until it becomes that case
  - A recursive call can result in many more recursive calls, because the method keeps on dividing a sub-problem into new sub-problems *that are of smaller size than the original*
  - These sub-problems are of the same nature as the original
- Please note: *solutions* can be recursive, not problems!

Trace: factorial(4)



Trace: factorial(4)

```
factorial(4) = 4*factorial(3)
  factorial(3) = 3*factorial(2)
    factorial(2) = 2*factorial(1)
      factorial(1) = 1*factorial(0)
        factorial(0) = 1
      factorial(1) = 1*factorial(0) = 1*1 = 1
    factorial(2) = 2*factorial(1) = 2*1 = 2
  factorial(3) = 3*factorial(2) = 3*2 = 6
factorial(4) = 4*factorial(3) = 4*6 = 24
```

recursive function calls

functions returning values

A Disclaimer

- The true benefit of *recursive thinking* is not realized until one starts trying to solve challenging problems that are more complicated than what we will explore in CSE 101
- Some (but not all) of the problems described in these lecture notes would be better solved using iterative, non-recursive functions
  - One notable exception is sorting, which *can* be solved efficiently using recursive algorithms like merge sort or Quicksort
- The purpose of these examples, therefore, is to help you understand *how to think recursively* when solving problems, not necessarily how to solve the stated problems in the most efficient manner

Example: Fibonacci Numbers

- Suppose we have one pair of rabbits (male and female) at the beginning of a year
- Rabbit pairs are not fertile during their first month of life but thereafter give birth to one new male and female pair at the end of every month
- Also, these are immortal rabbits and never die

Example: Fibonacci Numbers

- So we can now compute how many rabbit *pairs* will be alive at the end of month k:

$$\begin{bmatrix}\text{the number}\\ \text{of rabbit}\\ \text{pairs alive}\\ \text{at the end}\\ \text{of month } k\end{bmatrix} = \begin{bmatrix}\text{the number}\\ \text{of rabbit}\\ \text{pairs alive}\\ \text{at the end}\\ \text{of month } k-1\end{bmatrix} + \begin{bmatrix}\text{the number}\\ \text{of rabbit}\\ \text{pairs born}\\ \text{at the end}\\ \text{of month } k\end{bmatrix}$$

$$= \begin{bmatrix}\text{the number}\\ \text{of rabbit}\\ \text{pairs alive}\\ \text{at the end}\\ \text{of month } k-1\end{bmatrix} + \begin{bmatrix}\text{the number}\\ \text{of rabbit}\\ \text{pairs alive}\\ \text{at the end}\\ \text{of month } k-2\end{bmatrix}$$

Example: Fibonacci Numbers

- At the start of the year (after 0 months), we have F_0 = 1 pair of rabbits
- At the end of the first month we still have only F_1 = 1 pair of rabbits
- At the end of k months there will be $F_k = F_{k-1} + F_{k-2}$ pairs of rabbits
  - F_{k-1} is how many rabbits were alive the previous month
  - F_{k-2} is how many rabbits were alive two months ago, which equals how many rabbit will be born in month k
- By now you have probably guessed that F is the Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, 21,…)
- Let's see a function that returns the nth Fibonacci number

Example: Fibonacci Numbers

```
def fib(n):
  if n == 0 or n == 1: # two base cases
    return 1
  return fib(n - 1) + fib(n - 2)
```

- Examples:

  fib(0) = 1

  fib(1) = 1

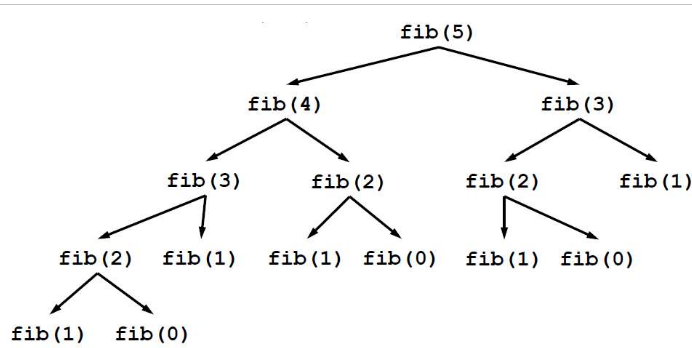
  fib(2) = fib(1) + fib(0) = 1 + 1 = 2

  fib(3) = fib(2) + fib(1) = 2 + 1 = 3

  fib(4) = fib(3) + fib(2) = 3 + 2 = 5

See recursion_examples.py

Trace: fib(5)



- This *call tree* diagram illustrates how the initial call to **fib(n)** generates a large number of recursive calls, even for a small value for **n**

Recursive Binary Search

- For recursive binary search (**rsearch**), the idea is basically the same as iterative binary search (**bsearch**, page 14)
- But, the while-loop is replaced with a recursive call to the function
- The algorithm checks the middle element to see if it equals the target
- If not, the function calls itself on the first half or second half, depending on whether the middle element is greater than or less than the target (respectively)

Completed rsearch Function

```
def rsearch(a, x, lower, upper):
    if upper == lower + 1:
        return None
    mid = (lower + upper) // 2
    if a[mid] == x:
        return mid
    if x < a[mid]:
        return rsearch(a, x, lower, mid)
    else:
        return rsearch(a, x, mid, upper)
```

- See rsearch_tests.py

Binary Search Algorithms

Iterative version:

```
def bsearch(a, x):
    lower = -1
    upper = len(a)
    while upper > lower + 1:
        mid = (lower + upper) // 2
        if a[mid] == x:
            return mid
        if x < a[mid]:
            upper = mid
        else:
            lower = mid
    return None
```

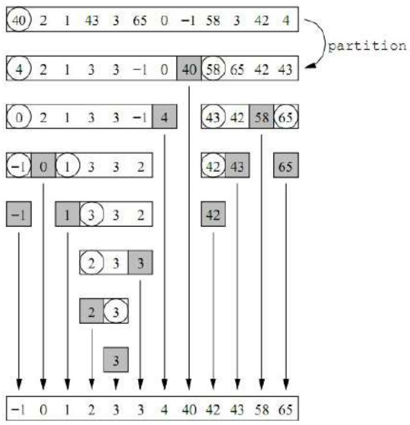
Recursive version:

```
def rsearch(a, x, lower, upper):
    if upper == lower + 1:
        return None
    mid = (lower + upper) // 2
    if a[mid] == x:
        return mid
    if x < a[mid]:
        return rsearch(a, x, lower, mid)
    else:
        return rsearch(a, x, mid, upper)
```


Quicksort

- Quicksort is a recursive sorting algorithm
- Like merge sort, quicksort breaks a list into smaller sub-lists and sorts the smaller lists
  - It divides the list into sub-lists in a different manner, however
  - The first element in a region to be sorted is chosen as the **pivot element**
  - The region is then **partitioned** into two sub-regions with a helper function called **partition**

Quicksort Example



Note: Once an element is picked as a pivot, it is already in its eventual place!

Quicksort

- The **partition** function performs this work:
  - Elements less than the pivot element are put in the left sub-region
  - Elements greater than the pivot element are put in the right sub-region
  - The pivot element is placed between the two sub-regions
  - The pivot element is now in its final position
- Quicksort works in a “top-down” approach by repeatedly splitting largest lists into smaller ones, whereas merge sort works in a “bottom-up” manner to recombine smaller lists into larger ones.
- See visualgo.net/en/sorting for animations

Quicksort

- The **partition** function partitions only a portion of a list
- The function takes three arguments:
  - A list of numbers
  - The starting index of the region to partition
  - The ending index of the region to partition
- For instance, **partition(nums, 4, 15)** means that **nums[4]** is the pivot element and that we want to partition elements in the range **nums[4:16]**
- Example: **nums = [62 88 6 85 39 19 82 23]**
- Function call: **partition(nums, 0, 7)**
- Pivot element: **62** (element **[0]** is always the pivot element)
- After partition: **[23 6 39 19 62 85 82 88]**

Completed partition Function

```
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

- The function returns the index of where the pivot element eventually winds up in a[]. That number also happens to be the number of elements ≤ the pivot element.

Completed partition Function

```
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

`x` stores a copy of the pivot element

87

Completed partition Function

```
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

`i` will eventually store the final position of the pivot element

88

Completed partition Function

```
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

← For each element in the region, except the pivot element...

89

Completed partition Function

```
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

← Compare each element in the region with **x**

90

Completed partition Function

```
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

- a[i] ≤ x, so we found another element that will go in the first half of

- The i variable essentially counts the number of elements that are ≤ the pivot element

Completed partition Function

```
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

Move the small element to the front half of the region

Completed partition Function

```
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

← Move the pivot element into its final position

Completed partition Function

```
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

← Return final index of where pivot element was moved to

Trace Execution: partition()

```
a: [5, 8, 1, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
→   x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5

Trace Execution: partition()

```
a: [5, 8, 1, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
    x = a[p]
→   i = p
    for j in range(p+1, r+1):
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	0

Trace Execution: partition()

a: [5, 8, 1, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6

```
def partition(a, p, r):
    x = a[p]
    i = p
➡   for j in range(p+1, r+1):
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	0
j	1

97

Trace Execution: partition()

a: [5, 8, 1, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6

```
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
➡       if a[j] <= x:        # False
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	0
j	1
a[j]	8

98

Trace Execution: partition()

a: [5, 8, 1, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6

```
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):      ➡
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	0
j	2
a[j]	1

Trace Execution: partition()

a: [5, 8, 1, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6

```
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:      # True      ➡
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	0
j	2
a[j]	1

Trace Execution: partition()

a: [5, 8, 1, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6

```
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:
➡           i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	1
j	2
a[j]	1

Trace Execution: partition()

a: [5, 1, 8, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6

```
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:
            i += 1
➡           a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	1
j	2
a[j]	8

Trace Execution: partition()

```
a: [5, 1, 8, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	1
j	3
a[j]	6

103

Trace Execution: partition()

```
a: [5, 1, 8, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:        # False
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	1
j	3
a[j]	6

104

Trace Execution: partition()

```
a: [5, 1, 8, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	1
j	4
a[j]	3

Trace Execution: partition()

```
a: [5, 1, 8, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:      # True
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	1
j	4
a[j]	3

Trace Execution: partition()

a: [5, **1**, **8**, 6, 3, 7, 2]
index: 0 1 2 3 4 5 6

```
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1          ⟵
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

Variable	Value
p	0
r	6
x	5
i	2
j	4
a[j]	3

107

Trace Execution: partition()

a: [5, **1**, 3, 6, 8, 7, 2]
index: 0 1 2 3 4 5 6

```
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
      a[i], a[j] = a[j], a[i]          ⟵
  a[p], a[i] = a[i], a[p]
  return i
```

Variable	Value
p	0
r	6
x	5
i	2
j	4
a[j]	8

108

Trace Execution: partition()

```
a: [5, 1, 3, 6, 8, 7, 2]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	2
j	5
a[j]	7

Trace Execution: partition()

```
a: [5, 1, 3, 6, 8, 7, 2]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:      # False
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	2
j	5
a[j]	7

Trace Execution: partition()

```
a: [5, 1, 3, 6, 8, 7, 2]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
  x = a[p]
  i = p
➡ for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

Variable	Value
p	0
r	6
x	5
i	2
j	6
a[j]	2

111

Trace Execution: partition()

```
a: [5, 1, 3, 6, 8, 7, 2]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
➡   if a[j] <= x:     # True
      i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

Variable	Value
p	0
r	6
x	5
i	2
j	6
a[j]	2

112

Trace Execution: partition()

```
a: [5, 1, 3, 6, 8, 7, 2]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
➡     i += 1
      a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

Variable	Value
p	0
r	6
x	5
i	3
j	6
a[j]	2

Trace Execution: partition()

```
a: [5, 1, 3, 2, 8, 7, 6]
index: 0 1 2 3 4 5 6
def partition(a, p, r):
  x = a[p]
  i = p
  for j in range(p+1, r+1):
    if a[j] <= x:
      i += 1
➡     a[i], a[j] = a[j], a[i]
  a[p], a[i] = a[i], a[p]
  return i
```

Variable	Value
p	0
r	6
x	5
i	3
j	6
a[j]	2

Trace Execution: partition()

a: [2, 1, 3, 5, 8, 7, 6]
index: 0 1 2 3 4 5 6

```
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
➡   a[p], a[i] = a[i], a[p]
    return i
```

Variable	Value
p	0
r	6
x	5
i	3

115

Trace Execution: partition()

a: [2, 1, 3, 5, 8, 7, 6]
index: 0 1 2 3 4 5 6

```
def partition(a, p, r):
    x = a[p]
    i = p
    for j in range(p+1, r+1):
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[p], a[i] = a[i], a[p]
➡   return i
```

Variable	Value
p	0
r	6
x	5
i	3

116

Quicksort

- The **partition** function will do most of the work in the **quicksort algorithm**:
  - First, partition the entire list. The first pivot element will now be at its final position.
  - Take the first sub-region of the list and partition it, and likewise for the second sub-region
  - By now, 3 elements (the 3 pivot elements) are in their final positions and we have 4 small regions
  - We partition those 4 sub-regions, causing 4 more pivot elements to be finally positioned (7 total pivots)
- This process continues until a region is so small that there is nothing to partition (zero elements in the region)

Completed qsort Function

- The top-level function is **qsort**, which depends on a helper
- function called **qs**, which in turn calls **partition**:

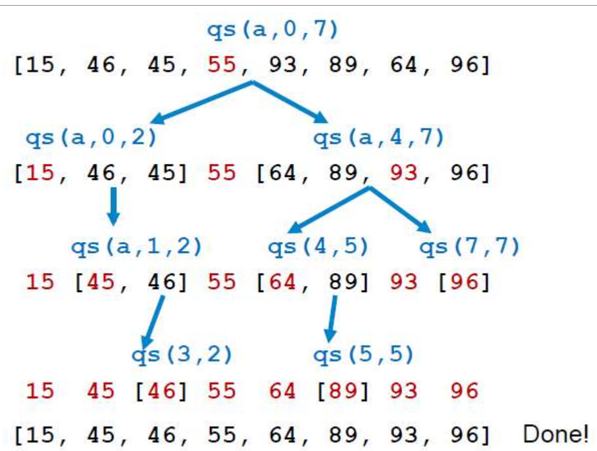
```
def qsort(a):
    qs(a, 0, len(a)-1) # sort the entire list

def qs(a, p, r):
    if p < r:
    q = partition(a, p, r)
    qs(a, p, q-1) # recursively sort first
    qs(a, q+1, r) # and second sub-regions

regions: [... p p+1 ... q-1 q q+1 ... r r + 1 ...]
```

Trace Execution: qsort()

- Input: **[55, 46, 89, 64, 93, 45, 15, 96]**
- Red indicates a pivot element
- Values in brackets are parts of sub-regions that are being partitioned
- A red value outside brackets is a pivot element that was positioned during an earlier round of partitioning
- **[15, 46, 45, 55, 93, 89, 64, 96]** 1st partition
- **[15, 46, 45] 55 [64, 89, 93, 96]** 2nd partitions
- **15 [45, 46] 55 [64, 89] 93 [96]** 3rd partitions
- **15 45 [46] 55 64 [89] 93 96** 4th partitions
- **[15, 45, 46, 55, 64, 89, 93, 96]** Done!

119

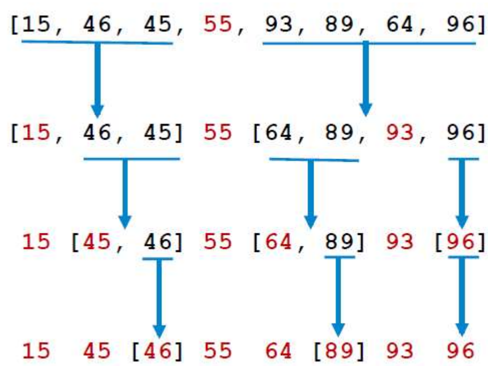
Trace Execution: qsort()

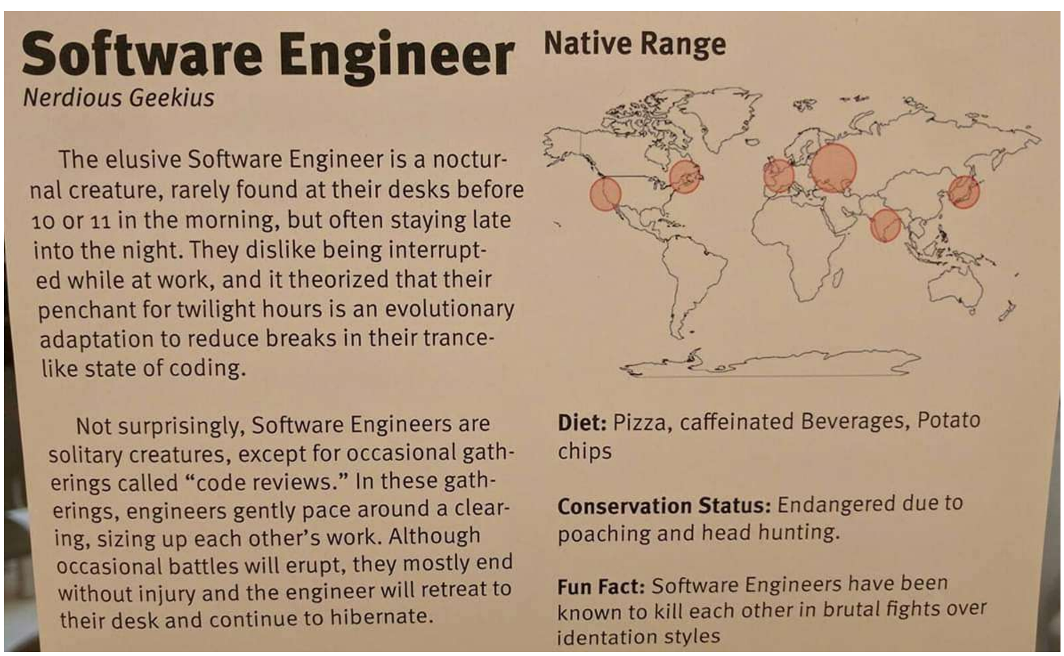


120

Trace Execution: qsort()

Another way to visualize the partitioning steps:





Additional Examples of Recursive Solutions to Problems

Example: Sum of Fractions

- Although computing a sum is computed most efficiently with a loop, it is a simple problem to understand, which makes it a good candidate for solving with recursion.
- Consider the problem of trying to compute the following sum, where n is a positive integer: 1 + 1/2 + 1/3 + … + 1/*n*
- Let's consider a function **sum_fracs()** that computes and returns this sum
- The simplest case (base case) is where *n* = 1
- For *n*>1 we can compute the sum as 1/n plus the sum of 1 + 1/2 + 1/3 + … + 1/n-1, *which we will compute recursively*

Example: Sum of Fractions

```
def sum_fracs(n):
  if n == 1:
    return 1
  return 1/n + sum_fracs(n - 1)
```

- See recursion_examples.py

125

Trace: sum_fracs(4)

```
sum_fracs(4) = 1/4 + sum_fracs(3)
  sum_fracs(3) = 1/3 + sum_fracs(2)
    sum_fracs(2) = 1/2 + sum_fracs(1)
      sum_fracs(1) = 1
```

recursive function calls

```
    sum_fracs(2) = 1/2 + sum_fracs(1)
                 = 1/2 + 1 = 1.5

  sum_fracs(3) = 1/3 + sum_fracs(2)
               = 1/3 + 1.5 = 1.833…

sum_fracs(4) = 1/4 + sum_fracs(3)
             = 1/4 + 1.833… = 2.0833…
```

functions returning values

126

Example: Sum a List

- Suppose we want to write a function **rsum** that computes the sum of the values in the list **nums**
- If **nums** has just one item, then the sum is just the value of **nums[0]**
- Otherwise, the sum is **nums[0]** plus the sum of the rest of the values, which is computed by a recursive call to the function

Example: Sum a List

```
def rsum(nums):
  if len(nums) == 1:
    return nums[0]
  return nums[0] + rsum(nums[1:len(nums)])
```

- Note the following two equivalent expressions:
  - **nums[1:]**
  - **nums[1:len(nums)]**

- See recursion_examples.py

Trace: rsum([8,1,4,5])

```
rsum([8,1,4,5]) = 8 + rsum([1,4,5])
  rsum([1,4,5]) = 1 + rsum([4,5])
    rsum([4,5]) = 4 + rsum([5])
      rsum([5]) = 5
    rsum([4,5]) = 4 + rsum([5]) = 4 + 5 = 9
  rsum([1,4,5]) = 1 + rsum([4,5]) = 1 + 9 = 10
rsum([8,1,4,5]) = 8 + rsum([1,4,5]) = 8 + 10 = 18
```


Example: Exponentiation

- One way to compute a^n for integer n is to multiply a by itself n times: $a^n = a*a* \ldots *a$
  - This is easy to implement using a loop, but it I somewhat inefficient
  - A more efficient approach ues recursion
- Example: suppose we want to compute 2^8
  - From the laws of exponents we know $2^8 = 2^4 * 2^4$
  - If we compute the value of 2^4 once, we can simply multiply the value of 24 by itself
  - Likewise, $2^4 = 2^2 * 2^2$
- In general, $a^n = a^{n/2} * a^{n/2}$ when *n* is even and $a^n=a^{(n-1)/2} * a^{(n-1)/2}*a$ when *n* is odd

Example: Exponentiation

- Let's use these formulas to write a function that recursively computes the *n*th power of any nonzero integer
- For the base case we will use the fact that any non-zero value raised to the 0th power is 1

Example: Exponentiation

```
def power(base, exponent):
  if exponent == 0:
    return 1                        # base case
  elif exponent % 2 == 0:       # even exponent
    temp = power(base, exponent // 2)
    return temp * temp
  else:                             # odd exponent
    temp = power(base, exponent // 2)
    return temp * temp * base
```

- See recursion_examples.py

Trace: power(3,5)

```
power(3,5) = power(3,2) * power(3,2) * 3

  power(3,2) = power(3,1) * power(3,1)

    power(3,1) = power(3,0) * power(3,0) * 3

      power(3,0) = 1     power(3,0) is computed only
                         once and stored in temp

    power(3,1) = power(3,0) * power(3,0) * 3
               = 1 * 1 * 3 = 3
                                 power(3,1) is computed only
                                 once and stored in temp
  power(3,2) = power(3,1) * power(3,1)
             = 3 * 3 = 9
                                 power(3,2) is computed only
                                 once and stored in temp
power(3,5) = power(3,2) * power(3,2) * 3
           = 9 * 9 * 3 = 243
```


Example: Reverse a String

- Consider the problem of taking a string and reversing its characters
  - Example: convert 'stony' to 'ytons'
- Let's explore a recursive function **rev** that solves this problem
- Let n be the length of a string **s**
- In the base case *n*=1, since **s** has only one character, just return s ("reversing" a single letter requires no work)
- Otherwise, when *n*>1, return a string consisting of the last letter in s, followed by the reverse of the first *n*-1 characters
- Doing this will require a recursive call to the **rev** function.

Example: Reverse a String

- Some slicing notation that will help us:
  - **s[-1]** means "get the last character of string **s**"
  - **s[:-1]** means "get all but the last character of string **s**"
  - The same syntaxes can also be used to with lists

```
def rev(s):
  if len(s) == 1:
    return s
  return s[-1] + rev(s[:-1])
```


Trace: rev('stony')

```
rev('stony') = 'y' + rev('ston')
  rev('ston') = 'n' + rev('sto')
    rev('sto') = 'o' + rev('st')
      rev('st') = 't' + rev('s')
        rev('s') = 's'
      rev('st') = 't' + rev('s') = 't'+'s'= 'ts'
    rev('sto') = 'o' + rev('st') = 'o'+'ts' = 'ots'
  rev('ston') = 'n' + rev('sto') = 'n'+'ots' = 'nots'
rev('stony') = 'y' + rev('ston') = 'y'+'nots' = 'ynots'
```

Example: Count Occurrences

- Python has a method named **count()**, which counts the number of times a target character appears in a string
- For example, **'stonybrook'.count('o')** is 3 because there are three lowercase o's in **'stonybrook'**
- How might we implement a recursive function that solves the same problem?
- First, inspect the first character of the string
  - If the character matches the target, we need to add 1 to the number of matches in the *remainder* of the string
  - Otherwise, we simply continue by counting the number of matches in the remainder of the string

Example: Count Occurrences

- But, how do we know how many times the target character appears in the remainder of the string?
  - We perform a recursive call to the function!
- So here's our algorithm:

If the string has at least one character in it then:

  If the first character matches, then return

    (1 + the # of matches of the target in the rest of the string)

  Otherwise, return the # of matches in the rest of the string

Otherwise, return 0

Example: Count Occurrences

```
def count_occurrences(string, ch):
    if len(string) > 0:
        if string[0] == ch:
            return 1 + count_occurrences(string[1:], ch)
        return count_occurrences(string[1:], ch)
    return 0
```

- See recursion_examples.py

Trace: count_occurrences()

- Example: `count_occurrences('stat', 't')`
- Abbreviating `count_occurrences` as `count`:

```
count('stat') = count('tat')
  count('tat') = 1 + count('at')
    count('at') = count('t')
      count('t') = 1 + count('')
        count('') = 0
      count('t') = 1 + count('') = 1 + 0 = 1
    count('at') = count('t') = 1
  count('tat') = 1 + count('at') = 1 + 1 = 2
count('stat') = count('tat') = 2
```

Example: Find Palindromes

- A palindrome is a word or phrase that can be read backwards and forwards
- Examples: radar, dad, toot, e
- Let's consider a function, **is_palindrome**, which returns **True** if its argument is a palindrome, and **False** if not
- How could we formulate a recursive solution to this problem?
  - We need to consider the base case(s) and recursive step(s)

Example: Find Palindromes

- The simplest case (base case) would be a string with exactly one character, which, by definition, would be a palindrome
- For the more general case we have two sub-problems:
  1. Verify that the first character and the last character of the string are equal
  2. If they match, ignore the two end characters and check whether the rest of the substring is a palindrome
  - If the first and last characters don't match, then the string (or sub-string) is not a palindrome
- The notation to slice out the first and last elements of string **s** and keep the remaining characters is **s[1:-1]**

Example: Find Palindromes

```
def is_palindrome(s):
  if len(s) <= 1: # a string of 0 or 1 characters
    return True # is a palindrome
  elif s[0] != s[-1]: # the first and last
    return False # characters don't match
  else:
    return is_palindrome(s[1:-1])
```

- See recursion_examples.py

143

Trace: is_palindrome()

- Example: `is_palindrome('racecar')`
- Abbreviating `is_palindrome` as `is_pal`:

```
is_pal('racecar') = is_pal('aceca')
  is_pal('aceca') = is_pal('cec')
    is_pal('cec') = is_pal('e')
      is_pal('e') = True
    is_pal('cec') = is_pal('e') = True
  is_pal('aceca') = is_pal('cec') = True
is_pal('racecar') = is_pal('aceca') = True
```

Keep making recursive calls while the first and last characters match

144

Trace: is_palindrome()

- Example: `is_palindrome('hannah')`
- Abbreviating `is_palindrome` as `is_pal`:

```
is_pal('hannah') = is_pal('anna')
  is_pal('anna') = is_pal('nn')
    is_pal('nn') = is_pal('')
      is_pal('') = True
    is_pal('nn') = is_pal('') = True
  is_pal('anna') = is_pal('cec') = True
is_pal('hannah') = is_pal('anna') = True
```

Keep making recursive calls while the first and last characters match

Trace: is_palindrome()

- Example: `is_palindrome('struts')`
- Abbreviating `is_palindrome` as `is_pal`:

```
is_pal('struts') = is_pal('trut')
  is_pal('trut') = is_pal('ru')
 X is_pal('ru') = False
  is_pal('trut') = is_pal('ru') = False
is_pal('struts') = is_pal('trut') = False
```

Keep making recursive calls while the first and last characters match

Example: Replace Multiples of 5

- Consider a peculiar function named **replace_mult5** that takes a list of numbers and replaces all multiples of 5 with a substitute number
  - The list and the substitute are passed as arguments
- Here's an example:

  nums = [5,3,15,50,2,4,6,60]

  replace_mult5(nums, 77)
  - **nums** becomes: **[77,3,77,77,2,4,6,77]**
  - Since this function does not return a value, it's not entirely clear how to write it recursively
    - Consider: how do we keep track of what part of the list we have processed so far?

147

Example: Replace Multiples of 5

- We can implement **replace_mult5** more easily if we use a **helper function**
- Recall the **qs** helper function that helped us implement the **qsort** function earlier in this Unit
- Our helper function, **replace_mult5_helper**, will take the same two arguments as **replace_mult5**, plus a third argument that tracks what part of the list we have already processed:

  def replace_mult5 (nums, sub)

  def replace_mult5_helper(nums, sub, i)
- In a certain sense, the helper function will simulate the behavior of a loop, as we can see in the implementation on the next slide

148

Example: Replace Multiples of 5

```
def replace_mult5(nums, sub):
   replace_mult5_helper(nums, sub, 0)

def replace_mult5_helper(nums, sub, i):
   if i == len(nums): # base case
      return
   if nums[i] % 5 == 0:
      nums[i] = sub
   replace_mult5_helper(nums, sub, i+1)
```


Example: Replace Multiples of 5

- The recursive helper function could be written iteratively using the code below

```
def replace_mult5_helper(nums, sub, i):
   for i in range(len(nums)):
      if nums[i] % 5 == 0:
         nums[i] = sub
```

- Compare this code with the recursive version. Do you see how the recursive version is essentially simulating a for-loop?

Trace: replace_mult5_helper

- Example: **nums = [4,10,2,5]**
  replace_mult5_helper(nums, 8, 0)
- Abbreviating **replace_mult5_helper** as **rmh**
  rmh([4,10,2,5],8,0) → rmh([4,10,2,5],8,1)
    rmh([4,10,2,5],8,1) → rmh([4,8,2,5],8,2)
      rmh([4,8,2,5],8,2) → rmh([4,8,2,5],8,3)
        rmh([4,8,2,5],8,3) → rmh([4,8,2,8],8,3)
          rmh([4,8,2,8],8,3) → do nothing & return
- Since the recursive call is the last statement in the function, the four recursive calls now simply return to each other, in sequence, performing no additional work
- The final contents of **nums** is **[4,8,2,8]**

Example: Find Index of Character

- Python has a built-in string method called **index** that returns the index of the first occurrence of a character (or substring, actually) in a string
- Example:
  school = 'stony brook'
  pos = school.index('o') # pos will be 2
- If the target character or substring does not appear in the string, the program crashes
- Let's consider a recursive solution to this problem and implement it in a function **rindex**
- In cases where the target string is not found, the **rindex** function will simply return **None** instead of crashing the program

Example: Find Index of Character

- One challenge we face is that somehow we need to keep track of what part of the string we have searched so far
- We will write a helper function, **rindex_helper**, that will assist with this task
- The helper function will ultimately solve the problem
- All that **rindex** will need to do is call **rindex_helper** with the correct arguments

Example: Find Index of Char.

```
def rindex(string, target):
  return rindex_helper(string, target, 0)

def rindex_helper(string, target, i):
  if i >= len(string):
    return None
  elif string[i] == target:
    return i
  else:
    return rindex_helper(string, target, i+1)
```

Trace: rindex_helper

- Example: `rindex_helper('stony','n',0)`
- Abbreviating `rindex_helper` as `rh`

```
rh('stony','n',0) = rh('stony','n',1)
  rh('stony','n',1) = rh('stony','n',2)
    rh('stony','n',2) = rh('stony','n',3)
      rh('stony','n',3) = 3  # found match!
    rh('stony','n',2) = rh('stony','n',3) = 3
  rh('stony','n',1) = rh('stony','n',2) = 3
rh('stony','n',0) = rh('stony','n',1) = 3
```


155

Trace: rindex_helper

- Example: `rindex_helper('stop','z',0)`
- Abbreviating `rindex_helper` as `rh`

```
rh('stop','z',0) = rh('stop','z',1)
  rh('stop','z',1) = rh('stop','z',2)
    rh('stop','z',2) = rh('stop','z',3)
      rh('stop','z',3) = rh('stop','z',4)
        rh('stop','z',4) = None
      rh('stop','z',3) = rh('stony','n',4) = None
    rh('stop','z',2) = rh('stony','n',3) = None
  rh('stop','z',1) = rh('stop','z',2) = None
rh('stop','z',0) = rh('stop','z',1) = None
```


156

Recursive Helper Functions

- The following material is based on notes by Jayesh Ranjan, a computer science major who served as a TA for CSE 101 many times during his studies at Stony Book University
- The main focus on this guide is on understanding the relationship between iteration and recursion: both are forms of repetition, but each implements the repetition in a different way
- Suppose you wanted to use a loop find the sum of all integers from 0 through *n*, inclusive
- One possible solution is given on the next slide
  - A while-loop is used because it will match up more closely with the recursive version

- See iter_to_rec.py

Recursive Helper Functions

```
def iter_sum(n):
  i = 1
  total = 1
  while i <= n:
    total += i
    i += 1
  return total
```

- Somehow we need to map this iterative algorithm to a recursive implementation, specifically:
  - the **i** and **total** variables
  - the while-loop condition and body
  - the return statement

Recursive Helper Functions

- Ultimately, we want a function **rec_sum(n)** we can call that will return the correct value
- We will use a recursive helper function to keep track of the **i** variable by taking it as an argument to the helper
- The **total** variable will be implemented as the return value of the helper function
- The while-loop's condition in the iterative solution will need to be replaced with a condition for an if-statement that will terminate the recursion
  - So, in both the iterative and recursive solutions we need a carefullywritten condition to stop the repetition
- The recursive implementation is given on the next slide

Recursive Helper Functions

```
def rec_sum(n):
    return rec_sum_helper(n, 1)

def rec_sum_helper(n, i):
    if i == n:
        return n
    return i + rec_sum_helper(n, i+1)
```

- Let's try to understand now how this code matches up with the iterative solution

Recursive Helper Functions

Iterative version:

i = 1

Recursive version:

rec_sum_helper(n, 1)

Initializing **i** to 1 in the iterative version is akin to calling the recursive helper function with an **i** argument of 1.

Recursive Helper Functions

Iterative version:

while i <= n

Recursive version:

if i == n

- The while-loop will stop iterating (repeating) once **i > n**. Similarly, the recursive version will stop making function calls once **i == n**.
- When **i == n** in the recursive version, we return **n** itself. This means that **n** will be added to the running total that the function is recursively computing.

Recursive Helper Functions

Iterative version:

total += I

i += 1

Recursive version:

return i +

rec_sum_helper(n, i+1)

The two **+=** statements from the iterative version are captured in the single line from the recursive version.

Color is used to show the connection between versions. There is no **total** variable for the recursive function. Rather, the function's return value serves this purpose.

Recursive Helper Functions

```
def iter_sum(n):
  i = 1
  total = 0
  while i <= n:
    total += I
    i += 1
  return total
```

```
def rec_sum(n):
  return rec_sum_helper(n, 1)

def rec_sum_helper(n, i):
  if i == n:
    return n
  return i +
        rec_sum_helper(n, i+1)
```

- It's not really possible to make a perfect one-to-one matching between the code in both versions, but I've attempted to do so here using color

Questions?