Sorting and Classifying

CAC 210
Spring 2018
Amber Wagner
Birmingham-Southern College

Class Today

- Survey
- Big O/Algorithm Analysis
- Sorting

Big O Notation

- Mathematical way to articulate the running time of an algorithm.
- Why is this important?

Big O Notation

- Mathematical way to articulate the running time of an algorithm.
- Why is this important?

Function	N = 10	N = 50	N = 100	N = 1000	N = 10000	N = 100000
log_2N	3.3 µs	5.65 µs	6.6 µs	9.9 µs	13.3 µs	16.6 µs
N	10 µs	50 µs	100 µs	1000 µs	10 ms	1 s
$Nlog_2N$	.03 ms	.28 ms	.66 ms	.099 s	.132 s	1.66 s
N^2	.1 ms	2.5 ms	10 ms	1 s	100 s	2.7 hours
N^3	1 ms	.125 s	1 s	16.7 min	11.57 days	31.7 years
2^N	.001 s	35.7 years	*	*	*	*

Big O Notation

- Landau's Symbol
- Created by Edmund Landau, a German mathematician focusing on number theory and complex analysis
- The growth of a function is also called *order*

Big O Notation

- Only want the highest order term
- Sometimes described as the worst-case scenario
- Example:
    - Using a linear search, we could find the element we're searching for in the first iteration: O(1)
    - The longest that it will take: O(n)
- While this is the worst-case, what we're really saying is that the growth of the function described by linear search is O(n)

Big O Notation

Notation	Name	Example Python

O(1)	Constant	`def find_min(x, y):`
		`  if x < y:`
		`  return x`
		`  else:`
		`  return y`

O(log N)	Logarithmic	`def binary_search(numbers, key):`
		`  low = 0`
		`  high = len(numbers) - 1`
		
		`  while high >= low:`
		`  mid = (high + low) // 2`
		`  if numbers[mid] < key:`
		`  low = mid + 1`
		`  elif numbers[mid] > key:`
		`  high = mid - 1`
		`  else:`
		`  return mid`
		`  return -1  # not found`

O(N)	Linear	`def linear_search(numbers, key):`
		`  for i in range(len(numbers)):`
		`  if numbers[i] == key:`
		`  return i`
		`  return -1  # not found`

Big O Notation

O(N log N)	Log-linear	`def merge_sort(numbers, i, k):`
		`  if i < k:`
		`  j = (i + k) // 2  # Find midpoint`
		`  merge_sort(numbers, i, j)  # Sort left part`
		`  merge_sort(numbers, j + 1, k)  # Sort right part`
		`  merge(numbers, i, j, k)  # Merge parts`

$O(N^2)$	Quadratic	`def selection_sort(numbers):`
		`  for i in range(len(numbers)):`
		`  index_smallest = i`
		`  for j in range(i + 1, len(numbers)):`
		`  if numbers[j] < numbers[index_smallest]:`
		`  index_smallest = j`
		
		`  temp = numbers[i]`
		`  numbers[i] = numbers[index_smallest]`
		`  numbers[index_smallest] = temp`

$O(c^N)$	Exponential	`def fibonacci(N):`
		`  if (1 == N) or (2 == N):`
		`  return`
		`  return fibonacci(N-1) + fibonacci(N-2)`

Sorting

- We can use built-in methods in Python to sort so what's the point?
- Common sorting algorithms:
  - Selection sort
  - Insertion sort
  - Quicksort
  - Merge sort
  - Bubble sort

How Would You Sort?

- Given the list:

[10, 27, 9, 15, 45, 5]

- How would you sort it?

Selection Sort

```
for i in range(len(numbers) - 1):
    # Find index of smallest remaining element
    index_smallest = i
    for j in range(i + 1, len(numbers)):
        if numbers[j] < numbers[index_smallest]:
            index_smallest = j

    # Swap numbers[i] and numbers[index_smallest]
    temp = numbers[i]
    numbers[i] = numbers[index_smallest]
    numbers[index_smallest] = temp
```

http://www.cs.armstrong.edu/liang/animation/web/SelectionSort.html

Selection Sort

- What is the runtime of selection sort?

```
for i in range(len(numbers) - 1):
    # Find index of smallest remaining element
    index_smallest = i
    for j in range(i + 1, len(numbers)):
        if numbers[j] < numbers[index_smallest]:
            index_smallest = j

    # Swap numbers[i] and numbers[index_smallest]
    temp = numbers[i]
    numbers[i] = numbers[index_smallest]
    numbers[index_smallest] = temp
```

Selection Sort

- What is the runtime of selection sort?
- $O(N^2)$

```
for i in range(len(numbers) - 1):
    # Find index of smallest remaining element
    index_smallest = i
    for j in range(i + 1, len(numbers)):
        if numbers[j] < numbers[index_smallest]:
            index_smallest = j

    # Swap numbers[i] and numbers[index_smallest]
    temp = numbers[i]
    numbers[i] = numbers[index_smallest]
    numbers[index_smallest] = temp
```

Insertion Sort

- Slightly more complicated algorithm, same runtime
- Swap as you go…

```
for i in range(1, len(numbers)):
    j = i
    # Insert numbers[i] into sorted part
    # stopping once numbers[i] in correct position
    while j > 0 and numbers[j] < numbers[j - 1]:

        # Swap numbers[j] and numbers[j - 1]
        temp = numbers[j]
        numbers[j] = numbers[j - 1]
        numbers[j - 1] = temp
        j = j - 1
```

http://cs.armstrong.edu/liang/animation/web/InsertionSort.html

Insertion Sort

- If it's the same runtime, what's the point?
- If a list is nearly sorted, the runtime is now O(N)
- Consider an example of you have a sorted list of students. A student adds the class and needs to be added to the roll. I now resort, but I only really have to find the position for one student within the list.

Quicksort

- Partitions a list into (ideally) two halves: low and high
- Recursively repeats this process until there is one element left in the list; thus, the partitioned list will be sorted
- All partitioned lists are reassembled

https://visualgo.net/bn/sorting?slide=1

Quicksort

```
def partition(numbers, i, k):
    # Pick middle value as pivot
    midpoint = i + (k - i) // 2
    pivot = numbers[midpoint]

    # Initialize variables
    done = False
    l = i
    h = k
    while not done:
        # Increment l while numbers[l] < pivot
        while numbers[l] < pivot:
            l = l + 1
        # Decrement h while pivot < numbers[h]
        while pivot < numbers[h]:
            h = h - 1
```

```
""" If there are zero or one items remaining,
        all numbers are partitioned. Return h """
        if l >= h:
            done = True
        else:
            """ Swap numbers[l] and numbers[h],
            update l and h """
            temp = numbers[l]
            numbers[l] = numbers[h]
            numbers[h] = temp
            l = l + 1
            h = h - 1
    return h
```

Quicksort

```
def quicksort(numbers, i, k):

    """ Base case: If 1 or zero elements,
      partition is already sorted """
    if i >= k:
        return

    """ Partition the array.
        Value j is the location of last
        element in low partition. """
    j = partition(numbers, i, k)

    """ Recursively sort low and high
       partitions """
    quicksort(numbers, i, j)
    quicksort(numbers, j + 1, k)

    return
```

Quicksort

- Runtime is usually O(n log n)
- Worst-case is $O(n^2)$
  - Rarely happens
  - Only occurs when a pivot value that is the lowest or highest value in the list is selected

Merge Sort

- Divide list in half recursively until we have a list with only one element
- By default, this list is sorted
- Compare to other lists to put in sorted order in a temporary list
- Copy back to the original list
- O(n log n) requiring O(n) extra memory

Merge Sort

```
def merge_sort(numbers, i, k):
    j = 0
    if i < k:
        j = (i + k) // 2  #  Find the midpoint in the partition

    #  Recursively sort left and right partitions
    merge_sort(numbers, i, j)
    merge_sort(numbers, j + 1, k)

    #  Merge left and right partition in sorted order
    merge(numbers, i, j, k)
```

Merge Sort

```
def merge(numbers, i, j, k):
    # Create temporary list merged_numbers
    # Initialize position variables

    # Add smallest element to merged numbers
    while left_pos <= j and right_pos <= k:
        if numbers[left_pos] <= numbers[right_pos]:
            merged_numbers[merge_pos] = numbers[left_pos]
            left_pos = left_pos + 1
        else:
            merged_numbers[merge_pos] = numbers[right_pos]
            right_pos = right_pos + 1
        merge_pos = merge_pos + 1
```

```
    # If left partition not empty, add remaining elements
    while left_pos <= j:
        merged_numbers[merge_pos] = numbers[left_pos]
        left_pos = left_pos + 1
        merge_pos = merge_pos + 1

    # If right partition now empty, add remaining elements
    while right_pos <= k:
        merged_numbers[merge_pos] = numbers[right_pos]
        right_pos = right_pos + 1
        merge_pos = merge_pos + 1

    # Copy merge number back to numbers
    for merge_pos in range(merged_size):
        numbers[i + merge_pos] = merged_numbers[merge_pos]
```

Bubble Sort

- Named for the way smaller or larger items ‘bubble’ to the top
- Generally, too slow: $O(n^2)$

```
1  def bubbleSort(alist):
2      for passnum in range(len(alist)-1,0,-1):
3          for i in range(passnum):
4              if alist[i]>alist[i+1]:
5                  temp = alist[i]
6                  alist[i] = alist[i+1]
7                  alist[i+1] = temp
8
9  alist = [54,26,93,17,77,31,44,55,20]
10 bubbleSort(alist)
11 print(alist)
12
```

Next Class

- Read Chapter 2.1-2.7