Part 3

SEARCHING AND SORTING

Searching

- Searching is the process of finding particular information from a collection of data based on specific criteria
- Search operations can be performed on every collection data structure (string, array, list, stack, dictionary, set, …)
- Search operation accepts two inputs:
  - Collection (or sequence) object
  - Search key
- Search key can have several forms
  - An item that we want to find in a list
  - Part of an item to search
  - Multiple parts for searching matching items (Google search)

Search Modes

- There are four different types of search operations
- **<u>In or out</u>:** Checking if the collection contains or does not contain the item
  Example: **item in L**
- **<u>First match</u>:** Finding the first occurrence of the key and reporting its location in the collection
  Example: **List.index(item)**
- **<u>All matches</u>:** Finding all the items in the collection that match the key
  Example: **fnmatch.filter(Names, “Dan*”)**
- **<u>Partial matches</u>:** Find the first n items that match the key

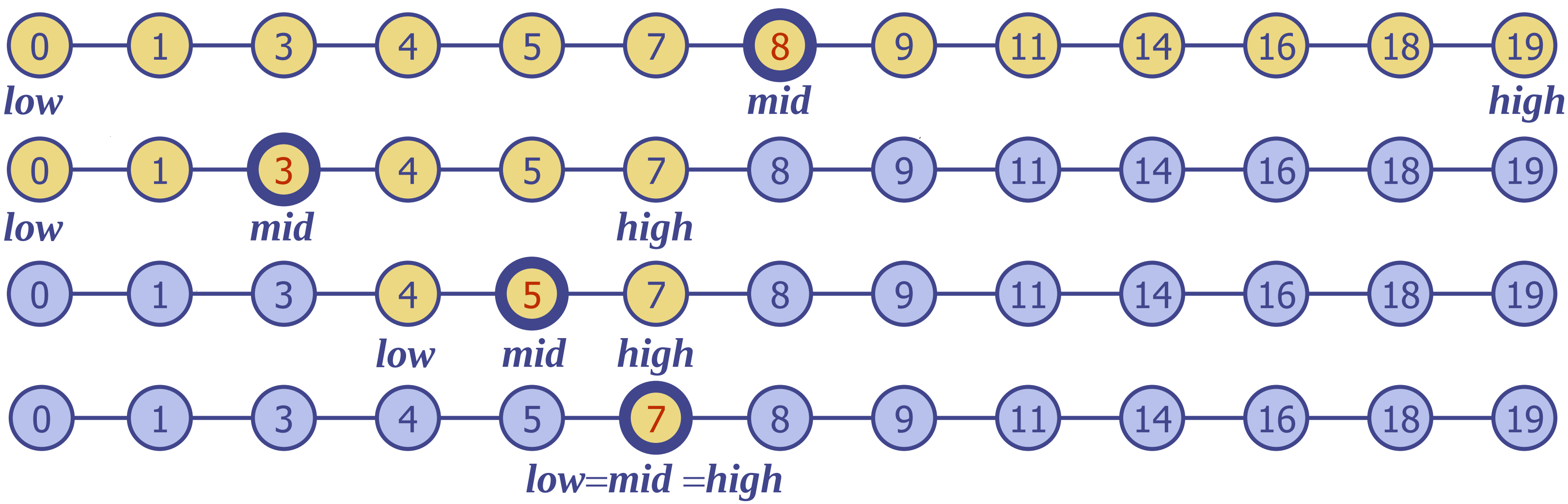
Linear Search (return first match)

```
def linear_search(List, item):
    n = len(List)
    for i in range(n):
        if item == List[i]:
            return i
    return -1
```

- Linear search is already implemented by the list index method except that when the item is not in the list you get an error
- The run time order of the linear search algorithm is O(n)
- Question: suppose that our sequence is sorted, could this help to speed the search process?

Binary Search

```
L = [0, 1, 3, 4, 5, 7, 8, 9, 11, 14, 16, 18, 19]
L is a sorted list in increasing order!
binary_search(L, 7)
low = 0, high = len(L) = 12
mid = (low+high) / 2 = 6
```



Binary Search Algorithm (Recursive)

```
def binary_search_rec(List, item, low=0, high=None):
    if high is None:
        high = len(List)

    if low >= high:    # empty list
        return -1

    mid = (low + high) / 2
    mid_value = List[mid]
    if item < mid_value:
        return binary_search_rec(List, item, low, mid)
    elif item > mid_value:
        return binary_search_rec(List, item, mid+1, high)
    else:
        return mid
```

Binary Search Algorithm

```
def binary_search(List, item, low=0, high=None):
    if high is None:
        high = len(List)

    while low < high:
        mid = (low + high) / 2
        mid_value = List[mid]
        if mid_value < item:
            low = mid+1
        elif mid_value > item:
            high = mid
        else:
            return mid
    return -1
```

SORTING

- Although binary search run time is fast O(log n), it depends on <u>sorting the sequence </u>!!!
- **Questions:**
  - What is the cost of sorting a sequence container?
  - What sorting algorithms do we have?
  - And which are the best sorting algorithms?
- In the next slides we will explore several (out of many) sorting algorithms and check their run time and quality

Why Sorting?

- Sorting is among the most important, and well studied computational problems
- Data sets are often stored in sorted order, for example, to allow for efficient searches with the binary search algorithm
- Many advanced algorithms rely on sorting as a subroutine

Bubble Sort

- YouTube Bubble Sort Dance
- The simplest and most intuitive sorting algorithm

```
# L is a list of integers that we want to sort

def bubble_sort(L):
    N = len(L)
    while True:
        sorted = True
        for i in range(0,N-1):
            if L[i+1] < L[i]:
                sorted = False
                L[i], L[i+1] = L[i+1], L[i]
        if sorted:
            return
```

Bubble Sort – version 2

- Here is a different version of Bubble Sort:

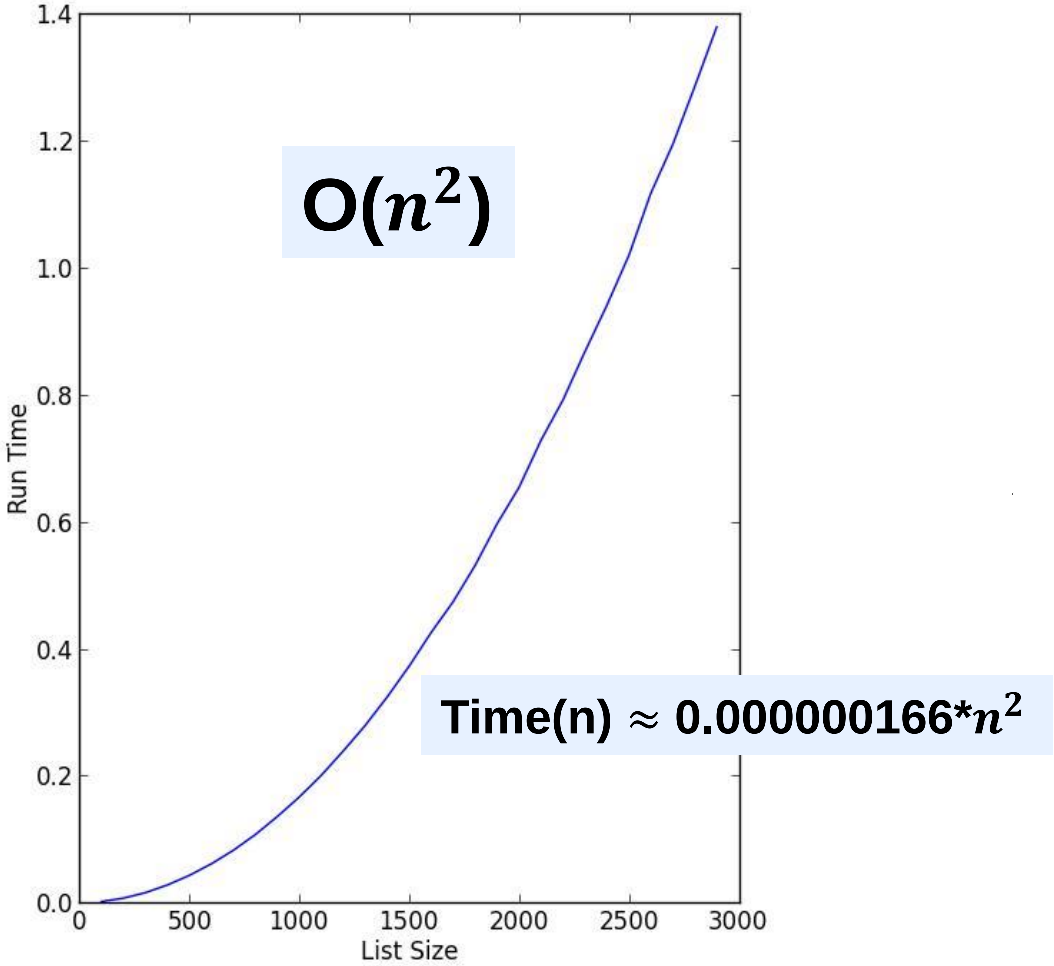
```
# L is a list of integers

def bubble_sort2(L):
    N = len(L)
    for i in range(0,N-1):
        for j in range(i+1, N):
            if L[j] < L[i]:
                L[i], L[j] = L[j], L[i]
```

Bubble Sort Test

```
def bubble_sort_test():
    for i in range(24):
        L = range(0,10)
        random.shuffle(L)
        print "L = ", L
        bubble_sort(L)
        print "Bubble sort:", L
        assert L == range(0,10)
        raw_input("Press any key to continue:")
```

Bubble Sort Run Time Data

Run time results obtained by running
Python 2.7.5 on a core-i7 ASUS laptop



List Size	Run Time (seconds)
100	0.0017
200	0.007
300	0.0157
400	0.0278
500	0.0429
600	0.0611
700	0.0824
800	0.1071
900	0.1355
1000	0.1663
1100	0.2003
1200	0.2387
1300	0.2789
1400	0.3238
1500	0.3723
1600	0.4252
1700	0.4737
1800	0.5308
1900	0.5964
2000	0.6538
2100	0.7279
2200	0.7914
2300	0.8676
2400	0.9406
2500	1.0191
2600	1.1171
2700	1.1941
2800	1.2853
2900	1.3791

Bubble Sort – Run Time Analysis

- Another name for **O(n^2) i**s "Quadratic Time Complexity" which is considered industry-bad unless the input size is expected to be small in almost all practical cases
- The above 30 experiments allows us to predict what will happen if our list size grows
- Lists of size 10M are not very rare. For example, chip floor-plan models may contain more than 1 billion transistors - 6 months run time for a 10M size list is of course unacceptable

List Size	Run Time (seconds)
10000	16.6 seconds
100000	1660 seconds
1000000	166000 seconds
10M	16600000 seconds ~ 6 months

Time(n) $\approx$ 0.000000166*n^2

Bubble Sort – Average Time Tests

- Python code for the Bubble sort algorithm and the tests code can be downloaded from:
  http://brd4.braude.ac.il/~samyz/cgi-bin/view_file.py?file=DSAL/CODE/bubble_sort.py
- Here is a typical routine for calculating average run time by generating many random shuffles of a list

```
import random

def bubble_sort_average_time(list_size, num_tests):
    times = list()
    L = range(0, list_size)

    for i in range(num_tests):
        random.shuffle(L)
        t0 = time.time()
        bubble_sort(L)
        t1 = time.time()
        t = t1-t0
        times.append(t)

    return sum(times)/num_tests
```

Bubble Sort – Average Time

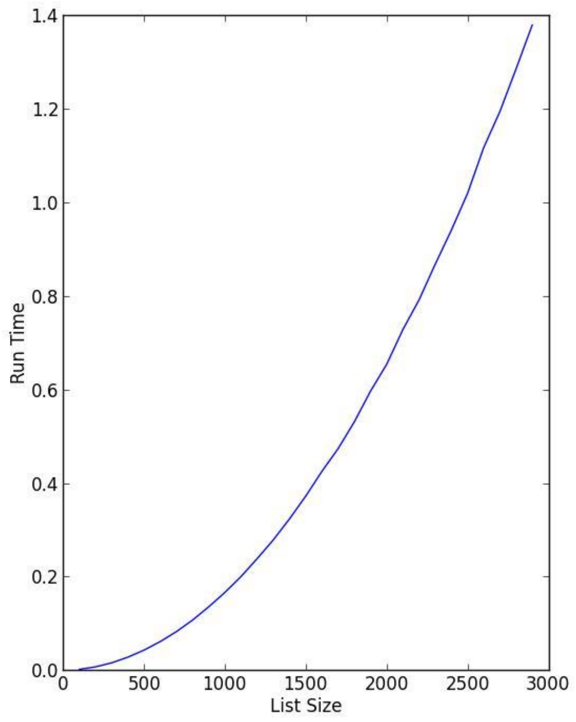
- Code for computing average time is also in: http://brd4.braude.ac.il/~samyz/cgi-bin/view_file.py?file=DSAL/CODE/bubble_sort.py
- We expect the student to copy paste and apply it to other algorithms!

```
# Create num_tests lists of size list_size and compute
# average time for doing bubble_sort on these lists

def bubble_sort_average_time(list_size, num_tests):
    times = list()
    L = range(0, list_size)

    for i in range(num_tests):
        random.shuffle(L)
        t0 = time.time()
        bubble_sort(L)
        t1 = time.time()
        t = t1-t0
        times.append(t)

    return sum(times)/num_tests
```

Bubble Sort – Average Time Graph

- Code for drawing average time graphs is also in: https://samyzaf.com/braude/DSAL/CODE/bubble_sort.py
- We expect the student to apply it to other algorithms!

```
def bubble_sort_runtime_graph():
    import matplotlib.pyplot as pyplot
    Size = [100*i for i in range(1,30)]
    Time = list()
    for N in Size:
        print "N=", N
        t = bubble_sort_average_time(N,16)
        t = round(t,4)
        Time.append(t)

    pyplot.plot(Size,Time)
    pyplot.xlabel('List Size')
    pyplot.ylabel('Run Time')
    pyplot.show()
    header = ('List Size', 'Run Time (seconds)')
```



The Halting Problem

- Could there be a special list on which Bubble sort runs forever ?
- The general halting problem: given an algorithm and an input, can we determine whether the algorithm will eventually halt or will run forever?
- Being able to prove that a given algorithm will halt for all its possible inputs is a critical !
- Proving that an algorithm must halt for all its inputs is usually very hard, and in many cases impossible.
- It may involve very complicated mathematical proofs and/or very long and expensive computations (e.g., QA, verification of an VLSI unit)

Why Bubble Sort Always Halt?

- We'll prove that for the second version
- Idea: prove an **<u>invariant</u>** is true for all iterations
  - It holds initially
  - If it holds at stage i, then it holds for stage i+1
  - Eventually must hold for all the list
- For bubble sort 2, the invariant is:
  at iteration i, the sub-list L[0:i] is sorted and any element in L[i:n] is greater or equal to any element in L[0:i]
- Since i is increasing, it eventually reaches n, and the algorithm halts

Why Bubble Sort Always Halt?

- For bubble sort 1, the invariant starts from the end (watch the Hungarian dance again …)
- The largest element must always “float” to the top, after which it will never move again!
- Therefore the problem is reduced to L[0,n-1]
- This proves that by at most n iterations of the loop, the list must be sorted. The inner loop also has n iterations, so by a total of n**2 steps the sorting is done
- Example: how many swaps are needed to sort the list
  L = [n, n-1, n-2, n-3, …, 2, 1, 0] ?
- This example demonstrates why bubble sort is O(n**2)

Selection Sort

- Yet one more intuitive method for sorting a list
- For simplicity, let L be a list of integers whose size is n=len(L)
- The idea in selection sort is:
  - Find the minimal element of L[0], L[1], …, L[n-1] and then make it the first (L[0])
  - Find the minimal element of L[1], L[2], …,L[n-1] and make it the second element (L[1])
  - Find the minimal element of L[2], L[3], …,L[n-1] and make it the third element (L[2])
  - Repeat this process until the list is fully sorted

Selection Sort: the idea

```
L = [7, 2, 8, 4, 6, 5, 1, 3]
    [1, 2, 8, 4, 6, 5, 7, 3]
    [1, 2, 8, 4, 6, 5, 7, 3]
    [1, 2, 3, 4, 6, 5, 7, 8]
    [1, 2, 3, 4, 6, 5, 7, 8]
    [1, 2, 3, 4, 5, 6, 7, 8]
    [1, 2, 3, 4, 6, 5, 7, 8]     Sorted!
```

Selection Sort: simpler version

```
1. Start with i=0
2. For every j from i+1 until n-1, if L[j] is
   smaller than L[i], swap L[i] and L[j]
3. Increment i (i = i+1)
4. Repeat step 2 until i=n-1
```

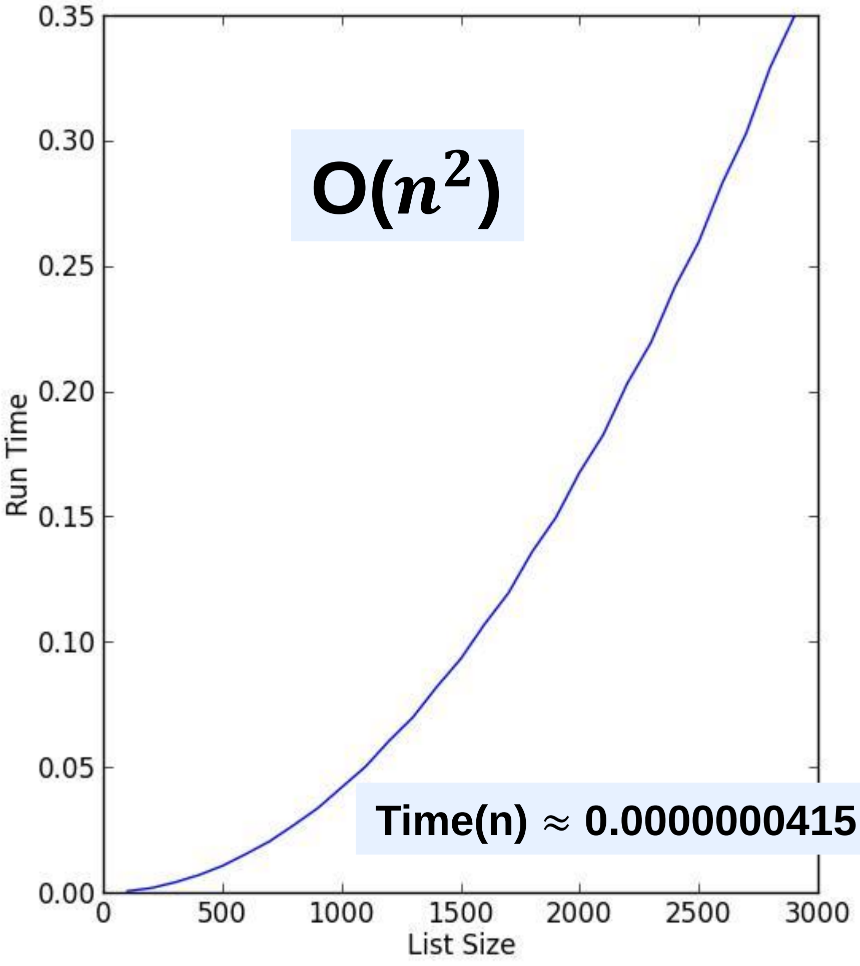
- This is a slightly different version than the heuristic one (two slides back)
- In this version we also compute the minimal value as part of the algorithm (instead of relying on an external method)

Selection Sort: Algorithm

```
def selection_sort(L):
    n = len(L)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if L[j] < L[min_index]:
                min_index = j
        L[i], L[min_index] = L[min_index], L[i]
```

Selection Sort Run Time

Run time results obtained by running
Python 2.7.5 on a core-i7 ASUS laptop



$O(n^2)$

Time(n) ≈ 0.0000000415 * n^2

List Size	Run Time (seconds)
100	0.0005
200	0.0017
300	0.004
400	0.0069
500	0.0106
600	0.0154
700	0.0205
800	0.0269
900	0.0336
1000	0.0419
1100	0.0501
1200	0.0605
1300	0.0699
1400	0.082
1500	0.0931
1600	0.1069
1700	0.1193
1800	0.1358
1900	0.1495
2000	0.1676
2100	0.1827
2200	0.203
2300	0.2194
2400	0.2415
2500	0.2594
2600	0.2831
2700	0.3029
2800	0.329
2900	0.3491

Selection Sort – Run Time Analysis

- Although Selection sort is 4x faster that Bubble sort, it's time complexity is still **O(n^2)** ("Quadratic Time Complexity") which is means it is essentially as bad as Bubble sort ☹
- This is obvious from the following table, which shows that for sorting a 40M random list may take about 2 years

List Size	Run Time (seconds)
10000	**4.15 seconds**
100000	**415 seconds**
1000000	**41510 seconds**
40M	**66,416,171 seconds ~ 2 years**

Time(n) ≈ 0.0000000415 * n^2

Average Time Tests

- Python code for the Selection sort algorithm and the tests code can be downloaded from:
  http://brd4.braude.ac.il/~samyz/cgi-bin/view_file.py?file=DSAL/LAB/selection_sort.py
- Here we introduce a more general function for computing average time which can be used by any other sorting algorithm!

```
import random

def sort_average_time(sorter, list_size, num_tests):
    times = list()
    L = range(0, list_size)

    for i in range(num_tests):
        random.shuffle(L)
        t0 = time.time()
        sorter(L)
        t1 = time.time()
        t = t1-t0
        times.append(t)

    return sum(times)/num_tests
```

Sort Average Time

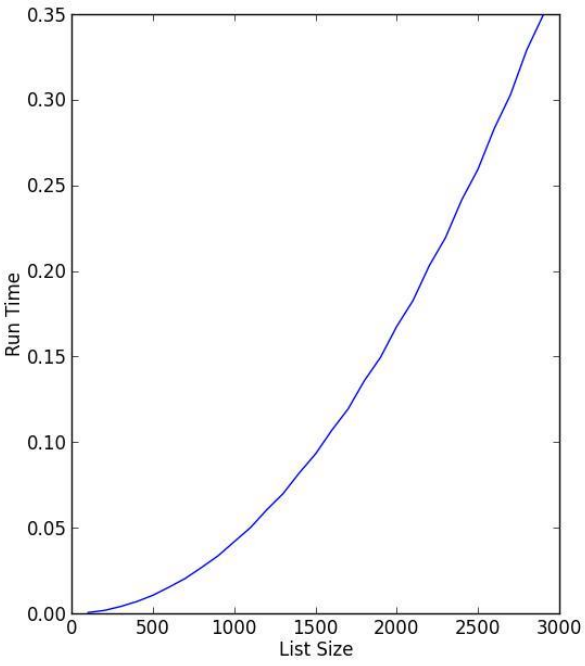
- Code for computing average time is also in:
  http://brd4.braude.ac.il/~samyz/cgi-bin/view_file.py?file=DSAL/LAB/sort_bench.py
- The following code can be used for any sort algorithm !

```python
# sorter is any function that sorts a list
# Create num_tests lists of size list_size and compute
# average time for doing bubble_sort on these lists

def sort_average_time(sorter, list_size, num_tests):
    times = list()
    L = range(0, list_size)

    for i in range(num_tests):
        random.shuffle(L)
        t0 = time.time()
        sorter(L)
        t1 = time.time()
        t = t1-t0
        times.append(t)

    return sum(times)/num_tests
```

Sort – Average Time Graph

- Code for drawing average time graphs is also in: http://brd4.braude.ac.il/~samyz/cgi-bin/view_file.py?file=DSAL/CODE/sort_bench.py
- The following code can be used for any sort algorithm !

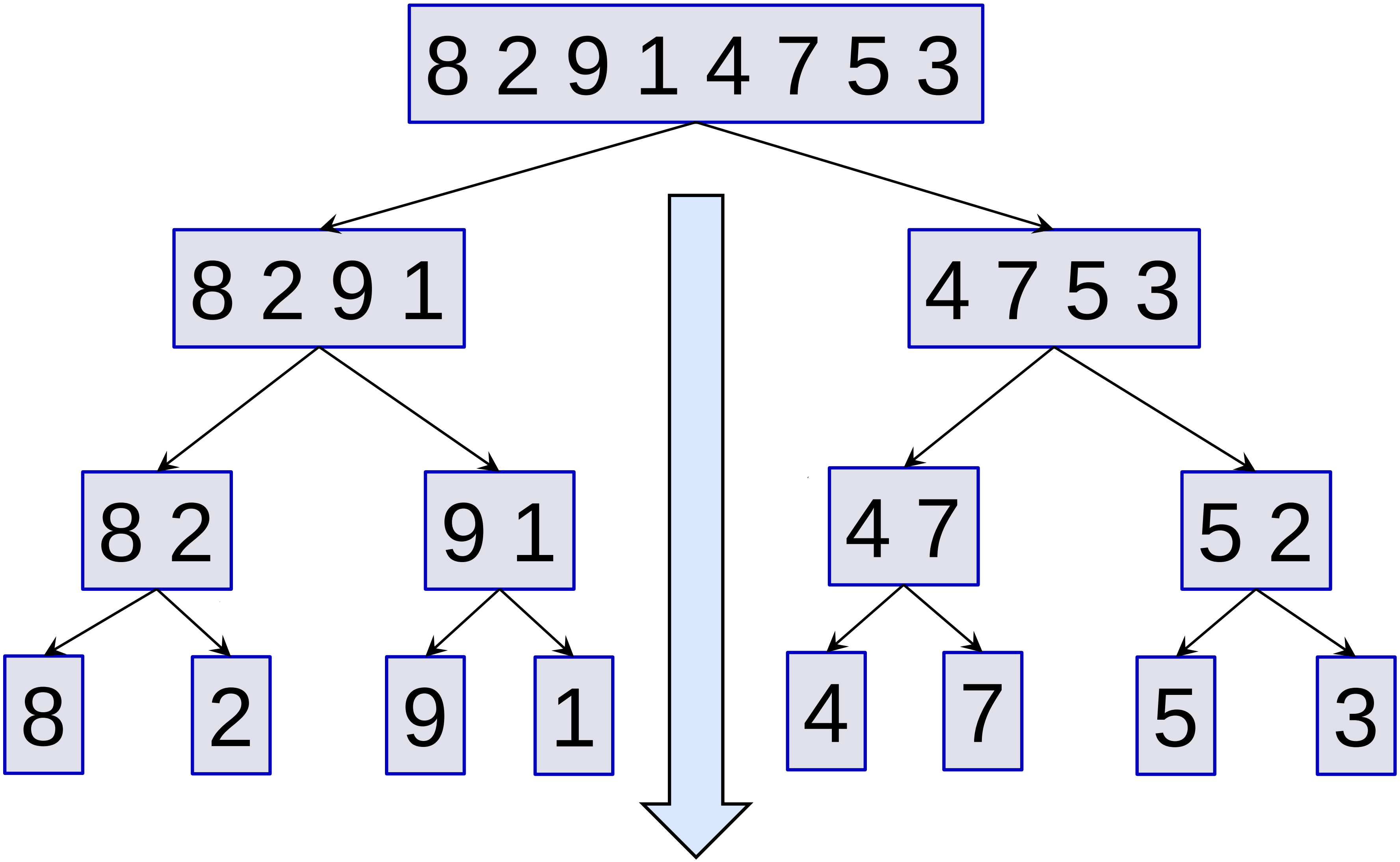
```
def sort_runtime_graph(sorter, n=30, ntests=16):
    import matplotlib.pyplot as pyplot
    import sys
    Sizes = [100*i for i in range(1,n)]
    Times = list()
    for N in Sizes:
        print "N=", N
        t = sort_average_time(sorter, N, ntests)
        t = round(t,4)
        Times.append(t)

    pyplot.plot(Sizes, Times)
    pyplot.xlabel('List Size')
    pyplot.ylabel('Run Time')
    pyplot.show()
```



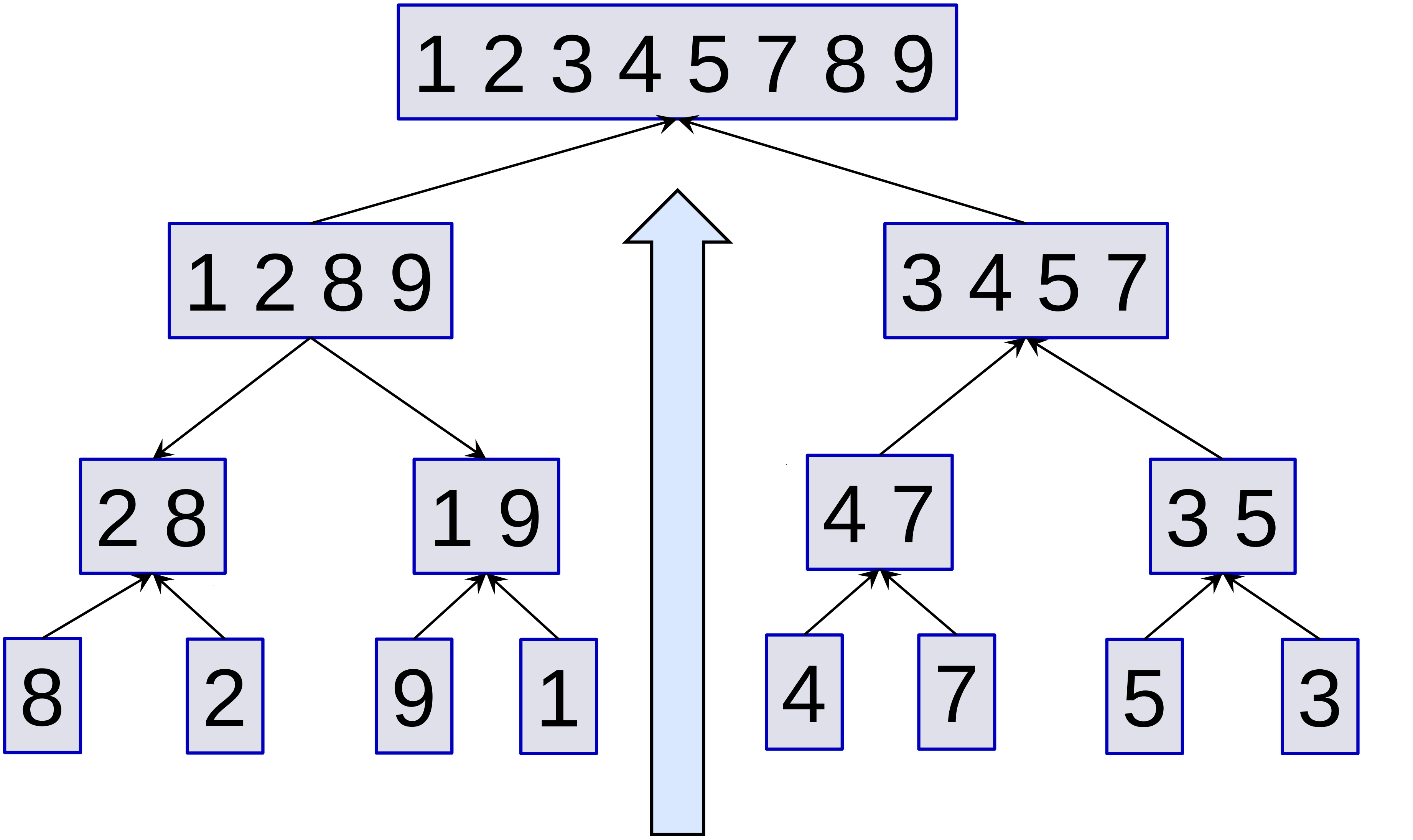
MERGE SORT / Divide and Conquer

- **Divide**
  - If the sequence is too small (1 or two elements) then sorting is easy
  - If the sequence is big, divide it to two parts and solve each part separately
- **Conquer**
  Recursively solve the subproblems associated with the subsets
- **Combine**
  Take the solutions to the sub problems and merge them into a solution to the original problem

Example: Divide



Example: Merge



The merge_sort algorithm

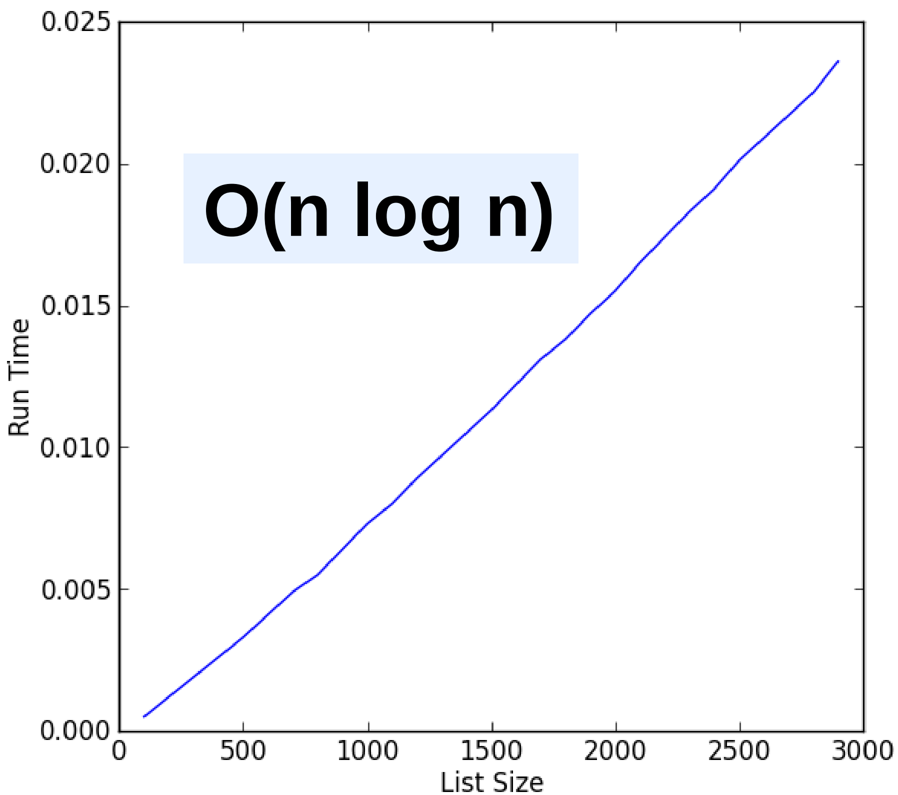
```
def merge_sort(L):
    n = len(L)
    if n <= 1:
        return
    mid = n / 2
    A = L[0:mid]
    B = L[mid:]
    merge_sort(A)
    merge_sort(B)
    M = merge(A,B)
    for i in range(n):
        L[i] = M[i]
```

The merge algorithm

```
def merge(A, B):
    "merge sorted lists A and B. Return a sorted result"
    result = []
    i = 0
    j = 0

    while True:
        if i >= len(A):             # If A is done,
            result.extend(B[j:])    # Add remaining items from B
            return result           # And we're totally done

        if j >= len(B):             # Same again, but swap roles
            result.extend(A[i:])
            return result

        # Both lists still have items, copy smaller item to result.
        if A[i] <= B[j]:
            result.append(A[i])
            i += 1
        else:
            result.append(B[j])
            j += 1
```

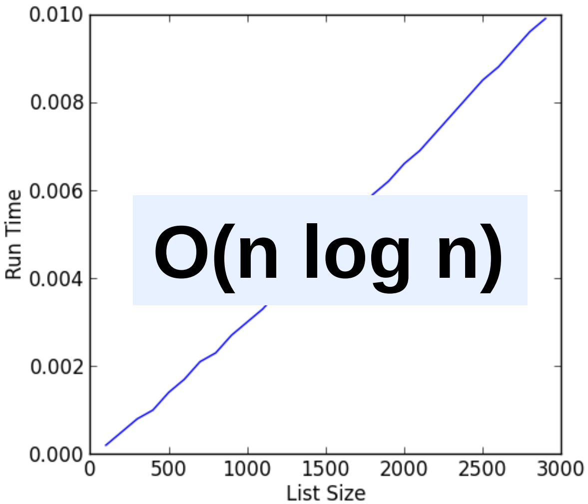
Merge Sort Run Time Benchmark

Merg Sort	Algorithm
List Size	Run Time (seconds)
600	0.0041
700	0.0049
800	0.0055
900	0.0064
1000	0.0073
1100	0.008
1200	0.0089
1300	0.0097
1400	0.0105
1500	0.0113
1600	0.0122
1700	0.0131
1800	0.0138
1900	0.0147
2000	0.0155
2100	0.0165
2200	0.0174
2300	0.0183
2400	0.0191
2500	0.0201
2600	0.0209
2700	0.0217
2800	0.0225
2900	0.0236



Time(n) ≈ 0.000001021 * n * log n

Merge Sort Run Time



Time(n) ≈ 0.0000004282 * n * log n

List Size	Run Time (seconds)
10000	0.0940 seconds
100000	1.1754 seconds
1000000	14.1056 seconds
10M	164.5657 seconds (bubble was 6 months !!!)
1000M	21158 seconds - less than 6 hours vs. 5200 years with bubble sort

QUICK SORT

- Invented by Tony Hoare 1960 (Moscow Univ.)
- **Divide**
  - The first item is selected as the pivot, p. The pivot value is used to partition the list to two sub-lists A and B, such that
    - A consists of all elements less than p
    - B consists of all elements bigger or equal to p
- **Conquer**
  Recursively solve the sub-problems by applying **`quick_sort`** to A and B
- **Combine**
  Combine the solutions of **`quick_sort(A)`** and **`quick_sort(B)`** by a simple concatenation (A then B)

The partition algorithm

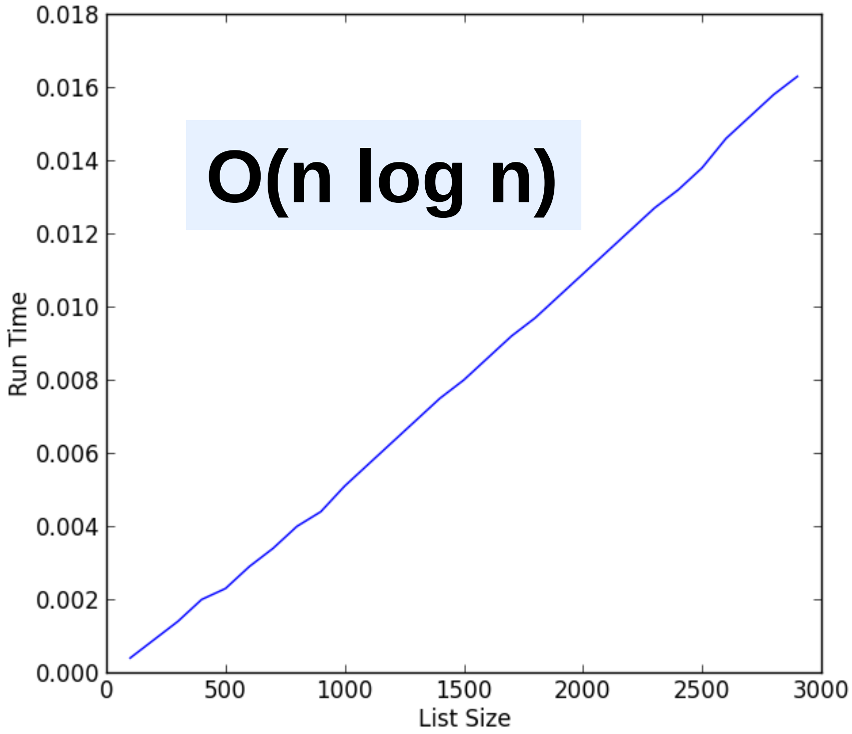
```
def partition(L, pivot):
    A = []
    B = []
    for element in L:
        if element < pivot:
            A.append(element)
        else:
            B.append(element)
    return A, B
```

The qsort algorithm

```
def qsort(L):
    n = len(L)
    if n <= 1:
        return
    pivot = max(L[0], L[-1])
    A, B = partition(L, pivot)
    qsort(A)
    qsort(B)
    A.extend(B)
    for i in range(n):
        L[i] = A[i]
```

Run Time Benchmark

List Size	Run Time (seconds)
500	0.0023
600	0.0029
700	0.0034
800	0.004
900	0.0044
1000	0.0051
1100	0.0057
1200	0.0063
1300	0.0069
1400	0.0075
1500	0.008
1600	0.0086
1700	0.0092
1800	0.0097
1900	0.0103
2000	0.0109
2100	0.0115
2200	0.0121
2300	0.0127
2400	0.0132
2500	0.0138
2600	0.0146
2700	0.0152
2800	0.0158
2900	0.0163



Time(n) ≈ 0.0000007050 * n * log n

In Place Sorting

- The quick sort algorithm from last slide, although very fast as compared to the previous algorithms, suffers from one major problem:
- The partition routine I using additional memory (except of L) to generates the two sub-lists (which are returned to the caller)
- The amount of extra space used for an algorithm as a function of its input size is called is space complexity
- Exercise: what is the space complexity of this version of qsort?
- A more efficient approach is to perform the partition “in place” – that is perform partition on the list itself

Tony Hoare Partition Algorithm (1960)

```
def partition(L, start, end):
    pivot = L[start]
    i = start+1
    j = end
    while True:
        while i <= j and L[i] <= pivot:
            i += 1
        while i <= j and pivot <= L[j]:
            j -= 1
        if j < i:
            break
        else:
            L[i], L[j] = L[j], L[i]

    # pivot should move to the middle
    L[start], L[j] = L[j], pivot
    return j
```

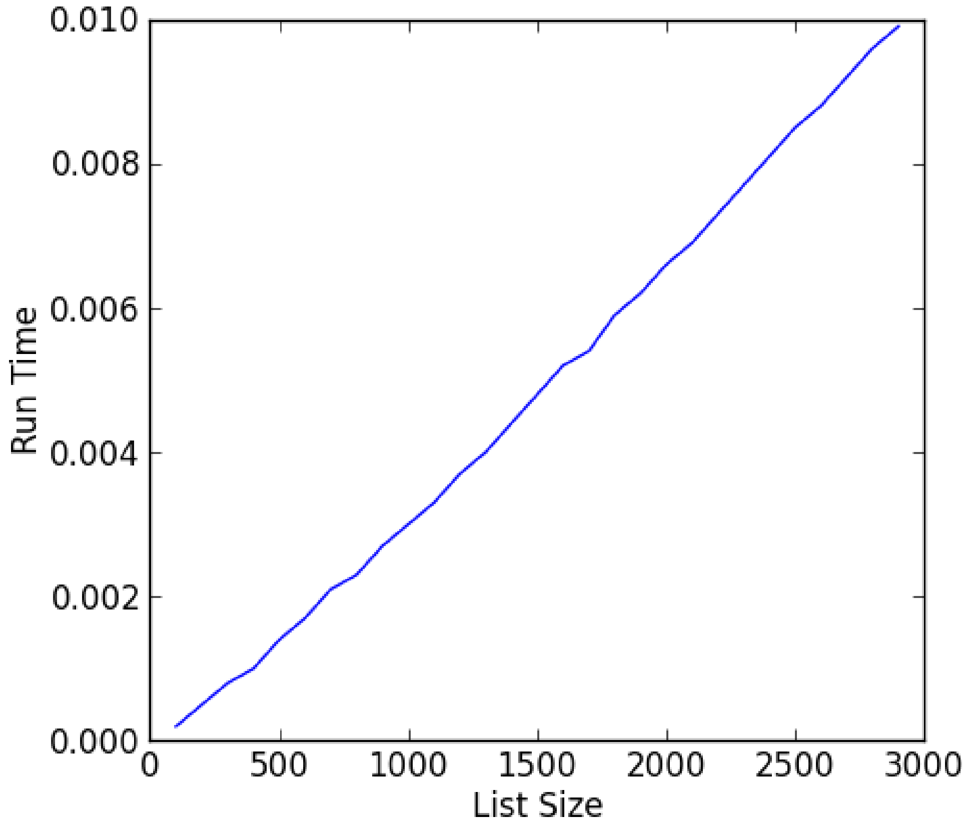
Tony Hoare qsort Algorithm

```
def qsort(L, start=0, end=None):
    if end is None: end = len(L) - 1
    if start < end:
        pivot = partition(L, start, end)
        qsort(L, start, pivot-1)
        qsort(L, pivot+1, end)
```

Quick Sort 2 (Tony Hoare)

List Size	Run Time (seconds)
500	0.0013
600	0.0017
700	0.002
800	0.0023
900	0.0027
1000	0.0029
1100	0.0033
1200	0.0036
1300	0.0041
1400	0.0043
1500	0.0048
1600	0.0052
1700	0.0055
1800	0.0058
1900	0.0063
2000	0.0066
2100	0.007
2200	0.0073
2300	0.0077
2400	0.008
2500	0.0085
2600	0.0089
2700	0.0092
2800	0.0096
2900	0.0099

O(n log n) Average Time

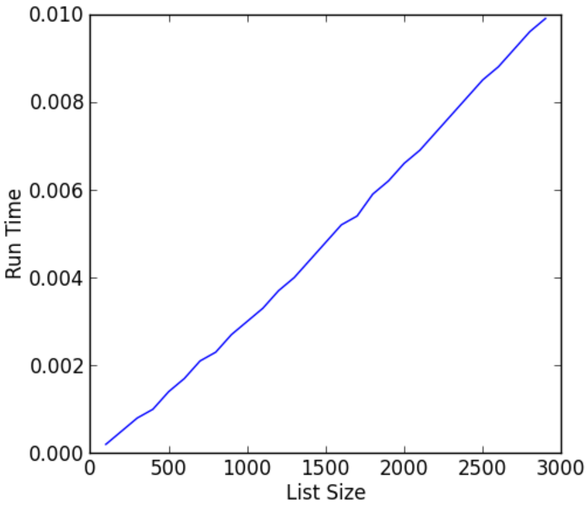


Time(n) ≈ 0.0000004283 * n * log n

O(n2) worst case !!**

Quick Sort 2 (Tony Hoare)

O(n log n) Average Time
O(n2) worst case!**



Time(n) ≈ 0.0000004283 * n * log n

List Size	Run Time (seconds)
10000	0.0394 seconds
100000	0.4930 seconds
1000000	5.9171 seconds
10M	69.0176 seconds (bubble was 6 months !!!)
1000M	8875.7747 seconds, less than 3 hours vs. 5200 years with bubble sort

So Why Bubble Sort is Important?

- Bubble is a very important example of an algorithm which is very intuitive, very easy to understand, and very easy to prove its correctness, yet this is the worst algorithm with respect to run time complexity
- It proves that an easy and elegant algorithm is not necessarily good!
- It is also a great example to Tim Peters Zen principles:

If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.

RADIX SORT

- Intuitively method based on alphabetizing a large list of names (like in a dictionary)
- The list of names is first sorted according to the first letter: the names are arranged in 26 buckets
- Similarly we can sort numbers according to the most significant digit
- But Radix sort goes by sorting on the least significant digit first. Then on the second pass, the entire numbers are sorted again on the second least-significant digit and so on

Radix Sort

It works great for decimal numbers with equal decimal length

INPUT	1^{st} pass	2^{nd} pass	3^{rd} pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix Sort

But if our numbers do not have equal length?
In such case we fill "empty digits" as zeros

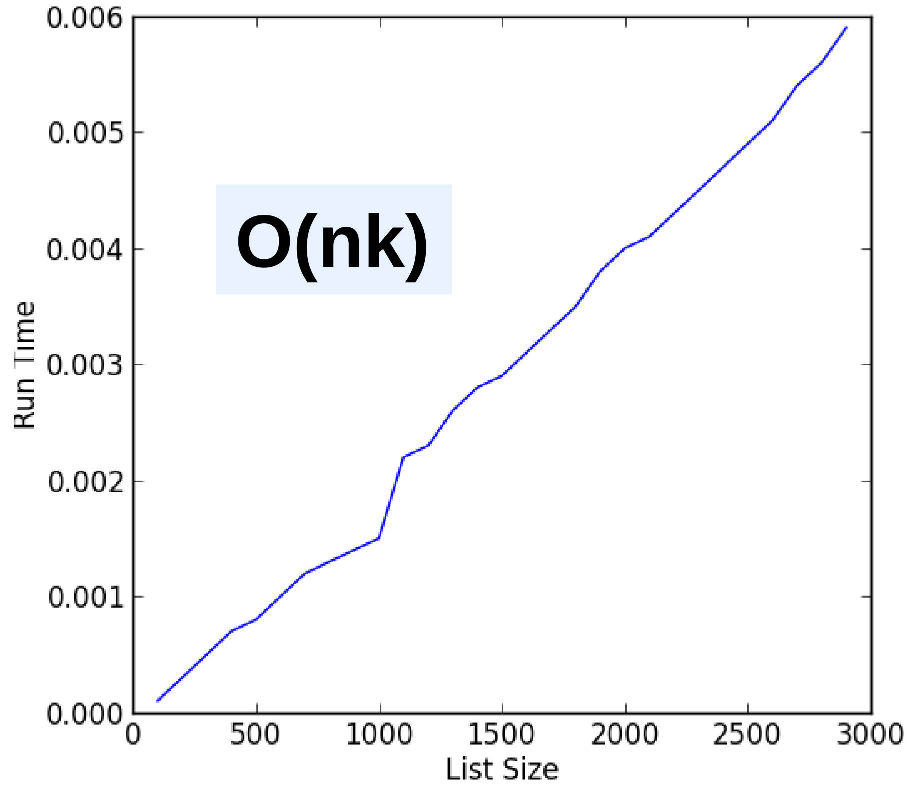
INPUT	VIEW	1^{st} pass	2^{nd} pass	3^{rd} pass	4^{th} pass	5^{th} pass
29	00029	06720	06720	00029	00029	00029
1457	01457	00355	00029	00057	00057	00057
57	00057	00436	00436	00355	00355	00355
31839	31839	01457	31839	00436	00436	00436
436	00436	00057	00355	01457	01457	01457
6720	06720	00029	01457	06720	31839	06720
355	00355	31839	00057	31839	06720	31839

Radix Sort Algorithm (2002)

```
def radix_sort(L):
    RADIX = 10
    deci = 1

    while True:
        buckets = [list() for i in range(RADIX)]
        done = True

        for n in L:
            q = n / deci      # q = quotient
            r = q % RADIX     # r = remainder = last digit
            buckets[r].append(n)
            if q > 0:
                done = False  # i has more digits

        i = 0   # Copy buckets to L (so L is rearranged)
        for r in range(RADIX):
            for n in buckets[r]:
                L[i] = n
                i += 1

        if done: break
        deci *= RADIX      # move to next digit
```

Radix Sort Run Time Benchmark

List Size	Run Time (seconds)
500	0.0008
600	0.001
700	0.0012
800	0.0013
900	0.0014
1000	0.0015
1100	0.0022
1200	0.0023
1300	0.0026
1400	0.0028
1500	0.0029
1600	0.0031
1700	0.0033
1800	0.0035
1900	0.0038
2000	0.004
2100	0.0041
2200	0.0043
2300	0.0045
2400	0.0047
2500	0.0049
2600	0.0051
2700	0.0054
2800	0.0056



Time(n) ≈ 0.0000019 * n
k = average num digits

Radix Sort Run Time

Time(n) ≈ 0.0000019 * n
k = average num digits

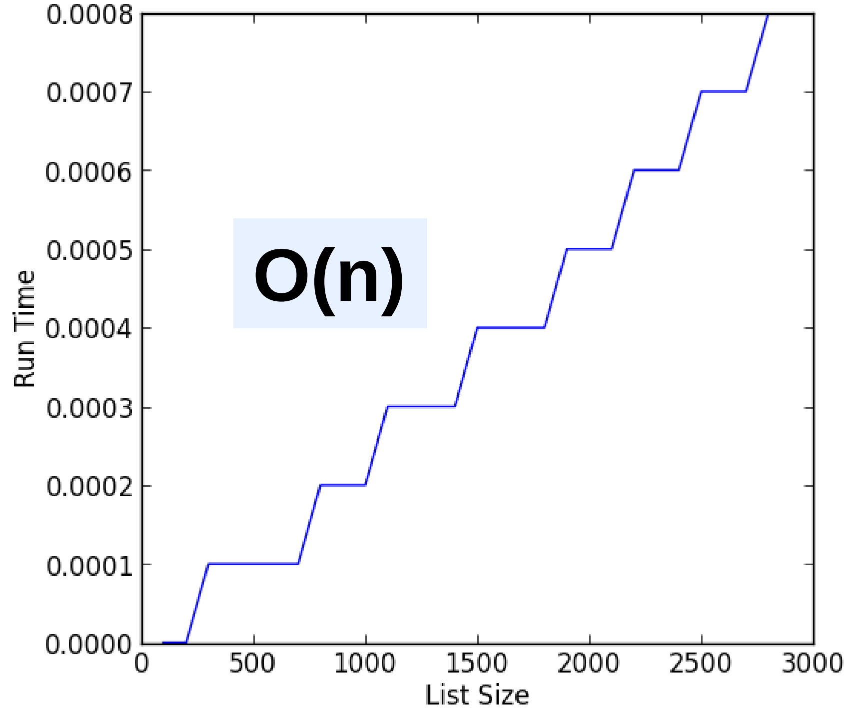
List Size	Run Time (seconds)
10000	0.019 seconds
100000	0.19 seconds
1000000	1.9 seconds
10M	19 seconds (bubble was 6 months !!!)
1000M	1900 seconds – half hour vs. 5200 years with bubble sort

TIM SORT

- Python's built-in sort algorithm was invented by Tim Peters around 2002
- It is considered to be one of the best sort algorithms in use
- We will not cover it in this preliminary course, but if you're interested, here are a few interesting links:
  http://en.wikipedia.org/wiki/Timsort
  http://www.youtube.com/watch?v=NVIjHj-lrT4
- Link to a simple test of Tim sort

Tim Sort Run Time Benchmark

List Size	Run Time (seconds)
500	0.0001
600	0.0001
700	0.0001
800	0.0002
900	0.0002
1000	0.0002
1100	0.0003
1200	0.0003
1300	0.0003
1400	0.0003
1500	0.0004
1600	0.0004
1700	0.0004
1800	0.0004
1900	0.0005
2000	0.0005
2100	0.0005
2200	0.0006
2300	0.0006
2400	0.0006
2500	0.0007
2600	0.0007
2700	0.0007
2800	0.0008



$$\text{Time}(n) \approx 0.0000002857 * n$$

Tim Sort Run Time (average)

Time(n) $\approx$ 0.0000002857 * n
Worst case is still O(n * log n)

List Size	Run Time (seconds)
10000	0.00286 seconds
100000	0.0286 seconds
1000000	0.286 seconds
10M	2.86 seconds (bubble was 6 months !!!)
1000M	286 seconds – 5 minutes vs. 5200 years with bubble sort