Efficiency: Continued

Arnamoy Bhattacharyya

Announcements:

1. Term test 2 marks has been released, good class average and median
2. Lab 8 has been released
    a. Infer big-oh from timing
    b. Almost no-coding
3. Assignment 2: technical glitch for autotesting, should be fixed soon

Agenda:

Worst case performance of Quick Sort

Python recursion limit

Merge Sort performance

In-class exercise for Big-oh assessment

Quick Sort: Recap

What is the average case performance of Quick sort?

What is the worst case performance?

Quick Sort: Average Case

lg(n) splits

n comparisons at each split

qs([4, 2, 6, 1, 3, 5, 7])

qs([2, 1, 3]) + [4] + qs([6, 5, 7])

qs([1]) + [2] + qs([3]) + [4] + qs([5]) + [6] + qs([7])

[1] + [2] + [3] + [4] + [5] + [6] + [7]

[1, 2, 3] + [4] + [5, 6, 7]

[1, 2, 3, 4, 5, 6, 7]

lg(n) concatenation, with n work to concatenate

n x lg n

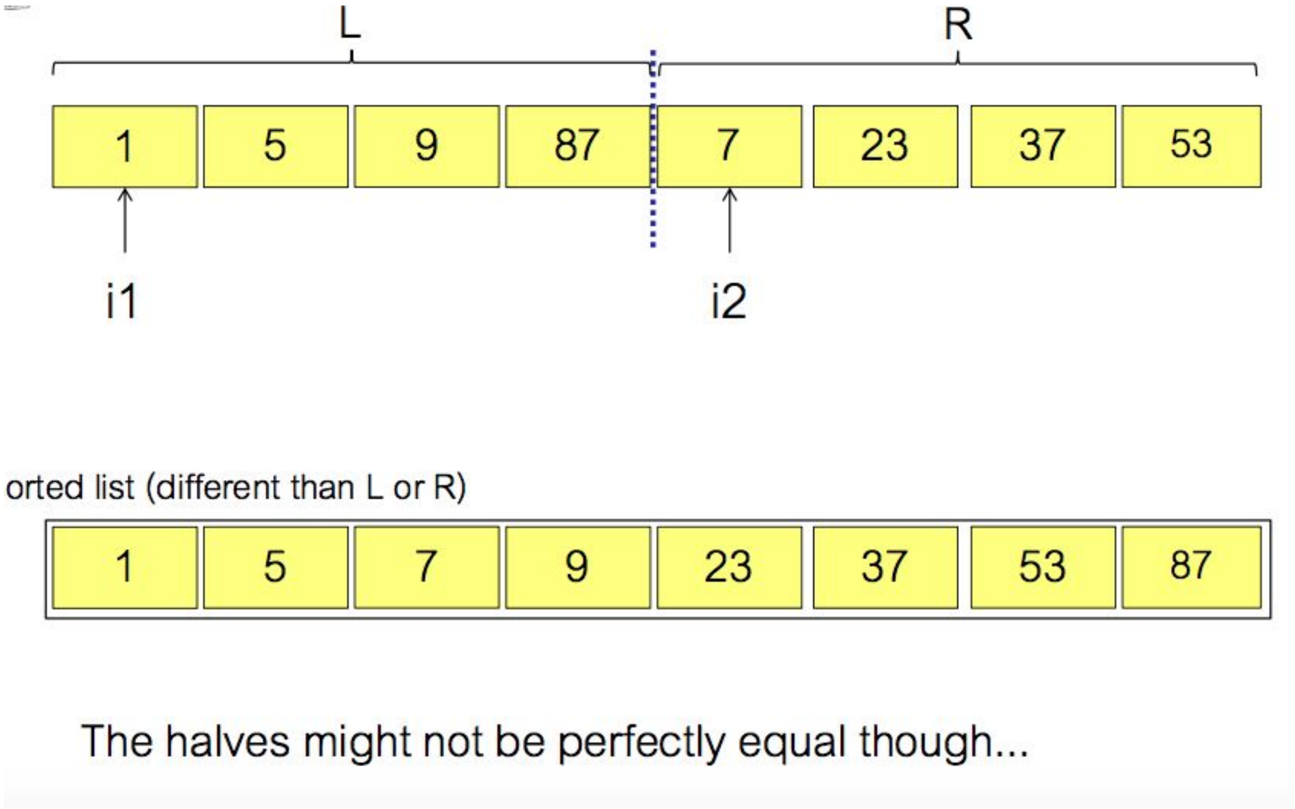
Merge Sort

- Split a list in two halves repeatedly
- Halves with 0 or 1 elements are guaranteed sorted
- Merge the two halves "on the way back"

Merge Sort

9 5 1 87 23 7 53 37
Mergesort First half
Mergesort Second half
9 5 1 87 23 7 53 37
9 5 1 87 23 7 53 37
5 9 1 87 7 23 37 53
1 5 9 87 7 23 37 53
1 5 7 9 23 37 53 87

Merge Step : merge(L, R)



Code for mergesort

idea: break a list up (partition) into two halves, mergesort each half, then recombine (merge) the halves

```
def mergesort(list_):
    """
     Produce a copy of list_ in sorted order.
    """
    if len(list_) < 2:
         return list_[:]
    else:
         return merge(mergesort(list_[:len(list_) // 2]),
                      mergesort(list_[len(list_) // 2 :]))
```

Lists of length < 2 are already sorted

First half

Second half

Merge the two sorted halves, "on the way back". How?

Divide and Conquer

Division step analysis

ms([4, 2, 6, 8, 1, 3, 5, 7]) division steps

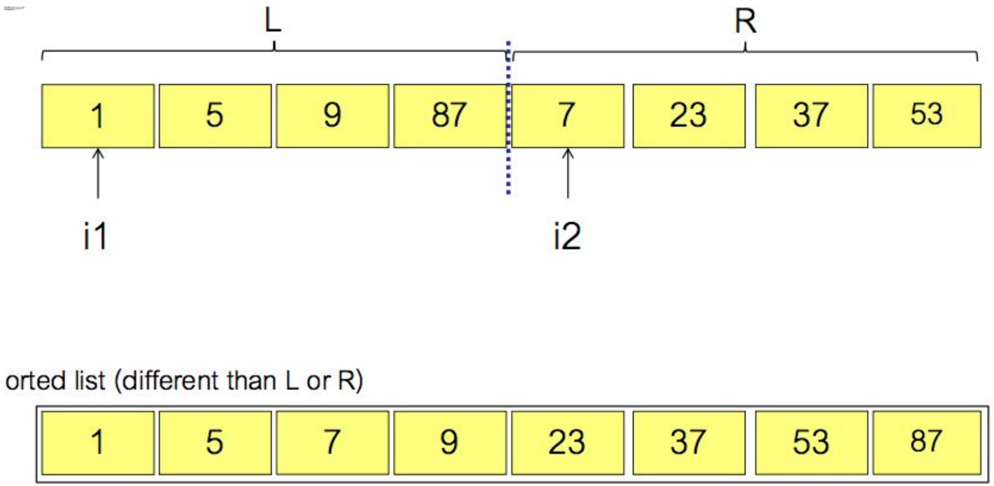
merge(ms([4, 2, 6, 8]) , ms([1, 3, 5, 7]))

merge(merge(ms([4,2]), ms([6,8]) , merge(ms([1,3]), ms ([5,7])

merge(merge(merge(ms([4],ms([2]),merge(ms([6]),ms([8]), merge(merge(ms([1],ms([3]),merge(ms([5]),ms([7])

Divide and Conquer

Conquer (merge step)



Counting mergesort: n = 8

prop. to
lg(n) splits

ms([4, 2, 6, 8, 1, 3, 5, 7])

merge(ms([4, 2, 6, 8]) , ms([1, 3, 5, 7]))

merge(merge(ms([4,2]), ms([6,8]) , merge(ms([1,3]), ms ([5,7])

merge(merge(merge(ms([4],ms([2]),merge(ms([6]),ms([8]), merge(merge(ms([1],ms([3]),merge(ms([5]),ms([7])

merge(merge(merge([4],[2]),merge([6],[8])), merge(merge([1],[3]),merge([5],[7])),

merge(merge([2,4]),[6,8])),merge([1,3]),[5,7]))

merge([2,4,6,8], [1,3,5,7])

[1, 2, 3, 4, 5, 6, 7, 8]

lg(n) merge with
prop. to n copies

n x lg n

Python Recursion Limits

```python
import sys


def fact(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * fact(n - 1)


if __name__ == '__main__':
    fact(999)
```

```
    return n * fact(n - 1)
  File "/Users/macbook15/PycharmProjects/CSC148H1S/week10/limit.py", line 8, in fact
    return n * fact(n - 1)
  [Previous line repeated 994 more times]
  File "/Users/macbook15/PycharmProjects/CSC148H1S/week10/limit.py", line 5, in fact
    if n == 0 or n == 1:
RecursionError: maximum recursion depth exceeded in comparison

Process finished with exit code 1
```

Efficiency: A few tips

1. Two algorithms to solve the same problem -- two Big Oh classes (O(n^2) and O($n \lg n$)), which one to choose?
2. Timing tools might be noisy, take averages of multiple runs
3. A good computer scientist should perform analytical (pen and paper) AND empirical (measuring and plotting) analysis

Big Oh*

Suppose the number of "steps" (operations that don't depend on n, the input size) can be expressed as $t(n)$. We say that $t \in \mathcal{O}(g)$ if:

> *there are positive constants c and B so that for every natural number n no smaller than B,*
> $t(n) \leq cg(n)$

*Taken from Danny's slides

Empirical study

$t(n) = 7n^2$ $\quad$ $t(n) = n^2 + 396$ $\quad$ $t(n) = 3960n + 4000$

Which one will have the largest growth?

Onto gnuplot

Exercise: Analytical determination of Time Complexity

Exercise 1:

```
def silly(n):
    n = 17 * n**(1/2)
    n = n + 3
    print("n is: {}.".format(n))

    if n > 97:
        print('big!')
    else:
        print('not so big!')
```

How does the running time of **silly** depend on **n**?

Exercise 2:

How does the running time of this code fragment depend on **n**?

```
sum = 0
for i in range(n):
    sum += i
```

Exercise 3:

How does the running of this code fragment depend on **n**?

```
i, sum = 0, 0
while i**2 < n:
    j = 0
    while j**2 < n:
        sum += i * j
        j += 1
    i += 1
```

Empirical verification

Onto PyCharm