Computer Science 1001.py

Lecture 10B[†]:
Recursion and Recursive Functions

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Amir Gilad, Michal Kleinbort

Founding Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University, Spring Semester, 2017
http://tau-cs1001-py.wikidot.com

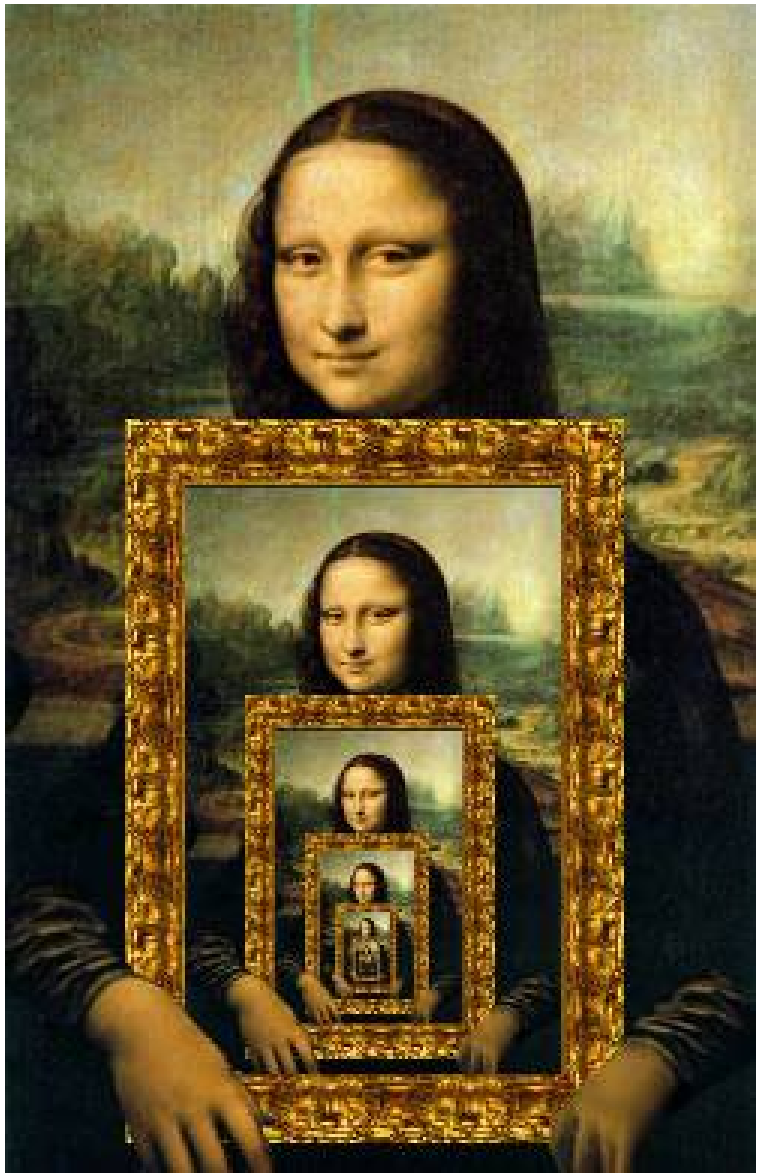
And Now to Something Completely Different: Recursion

'If seven maids with seven mops
Swept it for half a year,
Do you suppose,' the Walrus said,
'That they could get it clear?'
'I doubt it,' said the Carpenter,

And shed a bitter tear.
'O Oysters, come and walk with us!'
The Walrus did beseech.
'A pleasant walk, a pleasant talk,
Along the briny beach:
We cannot do with more than four,
To give a hand to each.'

Through the Looking-Glass and What Alice Found There:
Lewis Carroll, 1871.

And Now For Something Completely Different: Recursion



(originally taken from http://www.dominiek.eu/blog/?m=200711)

Lecture 10–11: Plan

Recursion, and recursive functions

- Basic examples and definition of recursion
  - Fibonacci
  - factorial
- Binary search - revisited
- Sorting
  - QuickSort
  - MergeSort
- Towers of Hanoi
- Improving recursion with memoization

Recursion

A function $f(\cdot)$, whose definition contains a call to $f(\cdot)$ itself, is called recursive.

A simple example is the factorial function, $n! = 1 \cdot 2 \cdot \ldots \cdot n$.
It can be coded in Python, using recursion, as following:

```
def factorial(n):
    if n==1:
        return 1
    else:
        return n*factorial(n-1)
```

A second simple example is the Fibonacci numbers, defined by $F_1 = 1,\ F_2 = 1$, and for $n > 2,\ F_n = F_{n-2} + F_{n-1}$.
A Fibonacci numbers function can be programmed in Python, using recursion, as following:

```
def fibonacci(n):
    if n<=2:
        return 1
    else:
        return fibonacci(n-2)+fibonacci(n-1)
```

Recursion and Convergence

At first sight, one may suspect that the recursive definitions above will lead nowhere. Or in other words, that they are cyclical and will never converge. This surely is not the case, and for specific instances one can simply run the code and get the (correct) answers.

```
>>> factorial(19)
121645100408832000
```

```
>>> fibonacci(21)
10946
```

There are two keys to correct design of recursive functions. The first one is to have a base case (one or more), which is the halting condition (no deeper recursion). In the factorial example, the base case was the condition `n==1`. In the Fibonacci example, it was `n<=2`. The second "design principle" is to make sure that all executions, or "runs", of the recursion lead to one of these base cases.

Recursion and Cyclicity

Recursive definitions that are cyclical will naturally not converge. A famous example is the following "dictionary definition" of recursion:

Recursion
re·cur·sion
n. Mathematics
See "Recursion".
[from Latin recursus, past participle of recurrere, to run back; see **recur**.]

You may also explore Google's version.

Choice of Base Cases

Recursive definitions that are seemingly proper may in fact diverge (lead to an infinite loop), due to unforeseen conditions in which the base cases are never reached. Consider, for example, the factorial function,

```
>>> factorial(1.9)
>>> factorial(0)
```

both lead to an infinite loop.

(How would you fix it? Does it need fixing in the first place?)

Binary Search, Revisited

We saw an iterative version of binary search.

```
def binary_search(key,lst):
    """ iterative binary search
        lst better be sorted for binary search to work"""
    n = len(lst)
    left = 0
    right = n-1
    outcome = None    # default value
    while left<=right:
        middle =(right+left)//2
        if key == lst[middle][0]:     # item found
            outcome = lst[middle]
            break  # gets out of the loop if key was found
        elif key < lst[middle][0]:    # item cannot be in top half
            right = middle-1
        else:          # item cannot be in bottom half
            left = middle+1
    if not outcome:  # holds when the key is not in the list
        print(key, "not found")
    return outcome
```

Binary Search, Recursively

Here is a recursive implementation of the same task. This code follows the iterative code closely. It passes the key, the original list and two indices (`lower, upper`) to the recursive call.

```
def rec_binary_search(key, lst, left, right):
    """ recursive binary search.
        passing lower and upper indices"""
    if left>right:
        return None
    middle = (left+right)//2
    if key == lst[middle][0]:
        return lst[middle]
    elif key < lst[middle][0]:   # item cannot be in top half
        return rec_binary_search(key, lst, left, middle-1)
    else:                        # item cannot be in bottom half
        return rec_binary_search(key, lst, middle+1, right)
```

Recursive Binary Search, Example Runs

Input list contains tuples of tghe form (name, id) representing students with random names and id's.

```
>>> from students import *
>>> stud_list = students(10**5)
>>> srt = sorted(stud_list, key=lambda pair:pair[0]) #sort by name
>>> binary_search(srt[11][0], srt)
('abeq', 843594317)
>>> rec_binary_search(srt[11][0], srt, 0, len(srt)-1)
('abeq', 843594317)

>>> elapsed("binary_search(srt[11][0], srt)",number=10000)
0.297608000000003
>>> elapsed('rec_binary_search(srt[11][0], srt, 0, len(srt)-1)',
            number=10000)
0.42181900000000017  # 40% slower

>>> binary_search('amir', srt) # Amir is not in the list
>>> rec_binary_search('amir', srt, 0, len(srt)-1)
>>>
>>> elapsed("binary_search('amir', srt)",number=10000)
0.271872000000001
>>> elapsed("rec_binary_search('amir', srt, 0, len(srt)-1)",
            number=10000)
0.41831800000000002 # 40% slower
```

Binary Search, Recursively, Using Slicing

Here is a second recursive version of the same task. It looks a bit simpler, as we don't need to give lower and upper bounds as parameters. Instead, we use slicing.

```
def rec_slice_binary_search(key,lst):
    """ recursive binary search.
        passing sliced list"""
    n = len(lst)
    if n<=0:
        return None
    if key == lst[n//2][0]:
        return lst[n//2]
    elif key < lst[n//2][0]:   # item cannot be in top half
        return rec_slice_binary_search(key,lst[0:n//2])
    else:                      # item cannot be in bottom half
        return rec_slice_binary_search(key,lst[n//2+1:n])
```

Binary Search, Recursively, Using Slicing AND Intermediate Printing

```
def rec_slice_disp_binary_search(key,lst):
    """ recursive binary search, displaying intermediate results.
        passing sliced """
    n = len(lst)
    if n<=0:
        print(key," not found")
        return None
    print(n,lst[n//2])
    if key == lst[n//2][0]:
        return lst[n//2]
    elif key < lst[n//2][0]:    # item cannot be in top half
        return rec_slice_disp_binary_search(key,lst[0:n//2])
    else:                       # item cannot be in bottom half
        return rec_slice_disp_binary_search(key,lst[n//2+1:n])
```

Recursive Binary Search, Example Runs

Input list contains objects from the Student class.

```
>>> st_list = students(10**4)  # names, id generated at random
>>> srt = sorted(st_list, key = lambda pair:pair[0])
>>> srt[5000-2:5000+2]  # sample slice
[('bhsg', 865349147), ('bhsiod', 908814636),
('bhsj', 642338206), ('bhsjaud', 365944448)]

>>> rec_slice_disp_binary_search(srt[10**4//2-1][0], srt)
10000 ('muegoc', 213663817)
5000 ('gjzo', 89530911)
2500 ('jnuaqsme', 307975520)
1250 ('legrpw', 447827839)
625 ('lzicmwxh', 80661931)
313 ('miervon', 591363219)
157 ('monxzy', 763580450)
79 ('mrsp', 331551639)
40 ('msuydfbt', 825603635)
20 ('mtq', 978442113)
10 ('mtsoflj', 939877912)
5 ('mtunhog', 688777936)
3 ('mtysbg', 44172980)
2 ('mucxsv', 606969888)
('mucxsv', 606969888)
```

Recursive Binary Search, Two Additional Example Runs

```
>>> rec_slice_disp_binary_search(srt[10**4//2][0], srt)
10000 ('muegoc', 213663817)
('muegoc', 213663817)  # smack in the middle

>>> rec_slice_disp_binary_search("amir",srt)# key not existing
10000 ('ngbtk', 34802425)
5000 ('gqilumdv', 237735464)
2500 ('dfsz', 660962448)
1250 ('bpov', 990162785)
625 ('avw', 632545828)
312 ('akqus', 749218619)
155 ('apv', 644649968)
77 ('anhwriuz', 929848795)
38 ('altwbhf', 227489830)
18 ('amjnt', 17908687)
9 ('alzp', 777721142)
4 ('amgotvrp', 908649465)
1 ('amjbv', 506326306)
amir  not found
```

Binary Search, Iterative vs. Recursive

Lets put the two functions to the test of the clock.

```
>>> from students import *
>>> st_list = students(10**6)  # names, id generated at random
>>> srt = sorted(st_list, key = lambda pair:pair[0])

>>>elapsed('rec_slice_binary_search("not_student",srt)',number=1)
0.07140300000000366
>>>elapsed('rec_slice_binary_search("not_student",srt)',number=100)
6.905284999999992
>>>elapsed('binary_search("not a student",srt)',number=200000)
6.167489000000175

>>>srt[5*10**5+1][0]
'nad'
>>>elapsed('rec_slice_binary_search("nad",srt)',number=1)
0.06951200000000313
>>>elapsed('binary_search("nad",srt)',number=2000)
0.06343100000000845
```

So the iterative version is approximately 2000 times faster than the recursive, sliced version (both for existing and non existing keys).

Food for thought: Why? What is the complexity of the sliced recursive version? Hint: Each slicing allocates new memory locations.

Binary Search w/wo slicing - complexity analysis

On the board.

The essential ingredient is the complexity of a single step in the sliced vs. un-sliced versions.
And the number of iterations. Note that the latter ingredient alone does not suffice.

Next class, we will refine this analysis, using recursion trees.

Envelope functions for recursion

So slicing may be problematic, complexity wise.

On the other hand, the recursive version without slicing used two additional parameters (left and right indices).
This is against a fundamental "rule" in computational problem solving: the user should be required to give only the arguments of the problem, and should not be bothered with additional ones, related to the way the problem is solved.

The common way to sort this out, is to use envelope functions, whose role is to "set the stage" for the "real" recursive functions:

```
def binary_search2(key, lst):
    """ calls recursive binary search
    lst better be sorted for binary search to work"""

    return rec_binary_search(key, lst, 0, len(lst)-1)
```

Envelope functions for recursion

```
def binary_search2(key, lst):
    """ calls recursive binary search
    lst better be sorted for binary search to work"""

    return rec_binary_search(key, lst, 0, len(lst)-1)


def rec_binary_search(key, lst, left, right):
    """ recursive binary search.
        passing lower and upper indices"""
    if left>right:
        return None
    middle = (left+right)//2
    if key == lst[middle][0]:
        return lst[middle]
    elif key < lst[middle][0]:   # item cannot be in top half
        return rec_binary_search(key, lst, left, middle-1)
    else:                        # item cannot be in bottom half
        return rec_binary_search(key, lst, middle+1, right)
```

Sort and Search

As we saw, binary search requires preprocessing – sorting.

We will introduce one approach to sorting (out of very many).

This approach, quicksort, employs both randomization and recursion.

(Contents include Game Board, 6 Moving Pieces, 6 Tile Holders, 30 Colored Tiles, Over 300 Topic Cards, Sand Timer & Instructions. For 2 to 6 players, ages 12 & up.)

Quicksort

Our input is an unsorted list, say
`[28, 12, 32, 27, 10, 12, 44, 20, 26, 6, 20, 21]`.
We choose a pivot element, simply one of the elements in the list.

For example, suppose we chose 20 (the second occurance). We now compare all elements in the list to the pivot. We create three new lists, termed `smaller, equal, greater`. Each element from the original list is placed in exactly one of these three lists, depending on its size with respect to the pivot.

```
smaller = [12, 10, 12, 6].
equal = [20, 20].
greater = [28, 32, 27, 44, 26, 21].
```

Note that the `equal` list contains at least one element, and that both `smaller` and `greater` is strictly shorter than the original list.

Quicksort (cont.)

What do we do next? We recursively sort `smaller` and `greater`, and then we append the three lists, in order (recall that in Python + means append for lists).

Note that `equal` need not be sorted.

```
return quicksort(smaller) + equal + quicksort(greater)
```

`quicksort(smaller) = [6, 10, 12, 12]`.

`equal = [20, 20]`.

`quicksort(greater) = [21, 26, 27, 28, 32, 44]`.

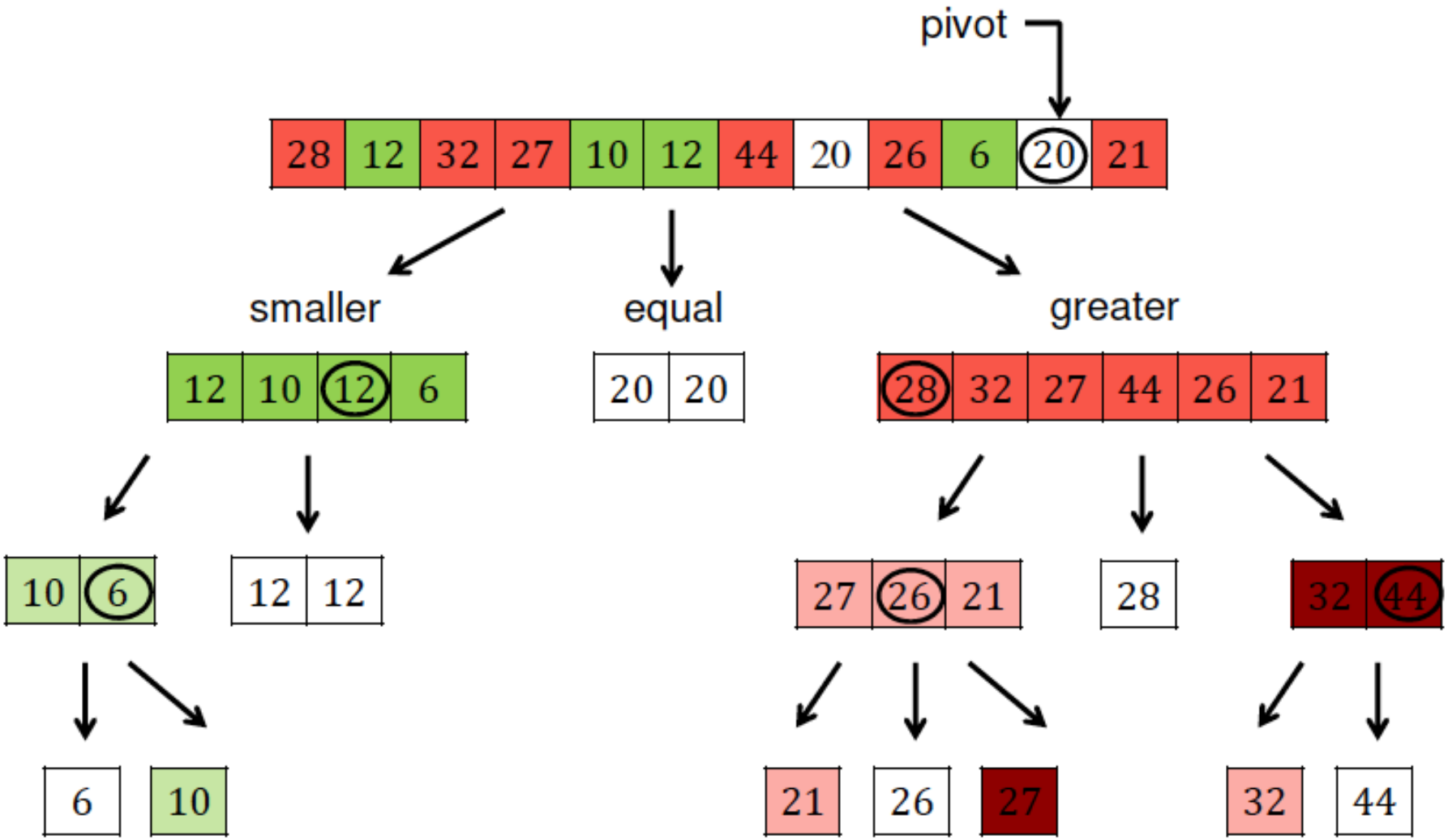
Final result:

```
[6, 10, 12, 12] + [20, 20] + [21, 26, 27, 28, 32, 44]
= [6, 10, 12, 12, 20, 20, 21, 26, 27, 28, 32, 44].
```

(Original list was

`[28, 12, 32, 27, 10, 12, 44, 20, 26, 6, 20, 21]`.)

Quicksort: A Graphical Depiction



(Figure curtesy of Amir Rubinstein.)

Quicksort: Python Code

```
import random     # a package for (pseudo) random generation
def quicksort(lst):
    if len(lst)<=1:      # empty lists or length one lists
        return lst
    else:
        pivot = random.choice(lst)
                # select a random element from the list
        smaller = [elem for elem in lst if elem < pivot]
        equal = [elem for elem in lst if elem == pivot]
        greater = [elem for elem in lst if elem > pivot]
                # ain't these selections neat?
        return quicksort(smaller) + equal + quicksort(greater)
                # two recursive calls
```

List Comprehension, Efficiency, and Correctness

The creation of new lists from an existing one, as done above, is generally known as list comprehension. It provides a very convenient mechanism for writing code, and Python is extremely good at it.

This convenience is good for quickly developing code. It also helps to develop correct code. But this simplicity and elegance do not necessarily imply an efficient execution.

For example, our quicksort algorithm goes three times over the original list. Furthermore, it allocates new memory for the three sublists. There are versions of quicksort that swap original list elements, go over the list only once, and reuse the same memory. They are more efficient, yet more error prone and generally take longer to develop.

Eventually you will choose, on a case by case basis, which style of programming to use.

Quicksort: Convergence and Correctness

Base cases: If the list is either empty or of length one, we do not recurse but return the list itself (in both cases, such lists are sorted).

Convergence: Each time we make a recursive call, its argument (either `smaller` or `greater`) is strictly shorter than the current list). When the length hits zero or one, we are at a base case. Thus the quicksort algorithm always converges (no infinite executions).

Correctness: An inductive argument. If `smaller` and `greater` are sorted, then

```
quicksort(smaller) + equal + quicksort(larger)
```

is a sorted list. Its elements are the same as the original list (multiplicities included), so the outcome is the original list, sorted. ♠

Quicksort: Pivot Selection

We could take the element with the first, last, or middle index in the list as the pivot. This removes the randomness aspect from quicksort, and makes it deterministic. This version would usually work well (assuming some random distribution of input lists). However, in some cases (*e.g.* if the input list is already sorted or close to sorted, and the pivot is the first or last element) this choice would lead to poor performance (even though the algorithm will always converge):

The worst case running time to sort a list of n elements occurs if at each invocation of the function, we choose either the minimum element as the pivot, or the maximum element. This makes either `smaller` or `greater` to be an empty list.

The resulting run time is $O(n^2)$. This follows from the solution to the recurrence relation $\text{WCT}(n) = (n-1) + \text{WCT}(n-1)$ (to be explained and solved on the board, using recursion trees).

Quicksort: Pivot Selection, cont.

We could take the element with the first, last, or middle index in the list as the pivot. This removes the randomness aspect from quicksort and makes it deterministic. This version would usually work well (assuming some random distribution of input lists). However, in some cases (*e.g.* if the input list is already sorted or close to sorted, and the pivot is the first or last element) this choice would lead to poor performance (even though the algorithm will always converge).

Instead of such fixed, deterministic choice, the recommended solution is choosing the pivot at random. With high probability, the randomly chosen pivot will be neither too close to the minimum nor too close to the maximum. This implies that both `smaller` and `greater` are substantially shorter than the original list, and yields good performance with high probability. (At this point this is an intuitive claim, nothing rigorous.)

Quicksort: Asymptotic Run Time Analysis

Instead of a fixed, deterministic choice, the recommended solution is choosing the pivot at random. With high probability, the randomly chosen pivot will be neither too close to the minimum nor too close to the maximum. This implies that both `smaller` and `greater` are substantially shorter than the original list, and yields good performance with high probability.

It can be shown that the best case and the average case running times to sort a list of n elements are both $O(n \cdot \log n)$ (the "best case" constant in the big O notation is slightly smaller than the "average case" constant).

Quicksort: Run Time (best case and average)

The best case occurs if we are lucky at each invocation of the function, and pivot is the median, splitting the list to two equal parts (up to one, if number of list elements is even). The best case run time satisfies the recurrence relation $\mathrm{BCT}(n) = (n-1) + 2 \cdot \mathrm{BCT}(n/2)$.

Complexity analysis for the best case: on the board. Recursion tree will be shown during next class.

A slightly more complicated expression can be written for $\mathrm{ACT}(n)$, the average case running time. (Rigorous analysis is deferred to the data structures course.)

In the `rec_slice_binary_search` function, slicing resulted in $O(n)$ overhead to the time complexity, which is disastrous for searching. Here, however, we deal with sorting, and an $O(n)$ overhead is completely OK.

Running Quicksort

```
def ordlist(n):
    """ generates an ordered list [0,-1,...,-(n-2),-(n-1)] """
    return [-i for i in range(0,n)]

import random
def randlist(n):
    """ generates a list of n random elements from [0,...,n**2] """
    return [random.randint(0,n**2) for i in range(0,n)]

import time

print("Test for an ordered list")
for func in [det_quicksort, quicksort]:
    print(func.__name__)

    for n in [200, 400, 800]: #recall the recursion limit of 1000
        print("n=", n, end=" ")
        olst = ordlist(n)

        t0 = time.clock()
        for i in range(100):
            func(olst) #sort is not inplace, lst remains unchanged
        t1 = time.clock()
        print(t1-t0)
```

Running Quicksort

```
print("\n\nTest for a random list")
for func in [det_quicksort, quicksort]:
    print(func.__name__)

    for n in [200, 400, 800]:
        print("n=", n, end=" ")
        rlst = randlist(n)

        t0 = time.clock()
        for i in range(100):
            func(rlst) #sort is not inplace, lst remains unchanged
        t1 = time.clock()
        print(t1-t0)
```

Running Quicksort

```
Test for an ordered list
det_quicksort
200 1.03312184
400 3.99165228
800 15.13556724 #feels quadratic, doesn't it?
quicksort
200 0.1800147199999907
400 0.3872873599999984
800 0.805047639999998 #"almost linear"
                       #fits the theoretic bound of O(nlogn)


Test for a random list #now both should be O(nlogn)
det_quicksort
200 0.1380252399999975
400 0.291556240000002
800 0.6495146399999996
quicksort
200 0.17874596000000054
400 0.38415788000000006
800 0.817371640000011
```

Can you explain why for random lists the deterministic QS is slightly better?

Quicksort: Closing Remarks

We could further explore and expand upon quicksort:

- Our code allocates additional memory during every recursive call. There are iterative versions that operate in place.
- Add code for tracking total number of recursive calls in a run.
- Other pivot selection strategies (*e.g.* median of three random elements).
- Performance issues: Random vs. sorted inputs.
- Performance issues when sorting huge files.

You may have a chance to look into these issues in your home assignments.

Merge Sort

Mergesort is a recursive, deterministic, sorting algorithm. It follows a divide and conquer approach. An input list (unsorted) is split to two – elements with indices from 0 up to the middle, and those from the middle up to the end of the list.

Each half is sorted recursively.

The two sorted halves are then merged to one, sorted list.

Merge Sort, cont.

Suppose the input is the following list of length 13
`[28, 12, 32, 27, 10, 12, 44, 20, 26, 6, 20, 21, 0]`.
We split the list in half, to
`[28, 12, 32, 27, 10, 12]` and `[44, 20, 26, 6, 20, 21, 0]`.

And recursively sort the two smaller lists, resulting in
`[10, 12, 12, 27, 28, 32]` and `[0, 6, 20, 20, 21, 26, 44]`.
We then merge the two lists, getting the final, sorted list
`[0, 6, 10, 12, 12, 20, 20, 21, 26, 27, 28, 32, 44]`.

The key to the efficiency of merge sort is the fact that merging two lists is done in time O(length of first list + length of second list).

Merge: Python Code

The key to the efficiency of merge sort is the fact that merging two lists is done in time O(length of first list + length of second list), by a simple two pointers algorithm

```
def merge(lst1, lst2):
    """ merging two ordered lists using
        the two pointer algorithm """
    n1 = len(lst1)
    n2 = len(lst2)
    lst3 = [0 for i in range(n1 + n2)]  # alocates a new list
    i = j = k = 0  # simultaneous assignment
    while (i < n1 and j < n2):
        if (lst1[i] <= lst2[j]):
            lst3[k] = lst1[i]
            i = i +1
        else:
            lst3[k] = lst2[j]
            j = j + 1
        k = k + 1  # incremented at each iteration
    lst3[k:] = lst1[i:] + lst2[j:]  # append remaining elements
    return lst3
```

Merge Sort: Python Code & Time Analysis

```
def mergesort(lst):
    """ recursive mergesort """
    n=len(lst)
    if n <= 1:
        return lst
    else:
        return merge(mergesort(lst[0:n//2]),mergesort(lst[n//2:n]))
                #two recursive calls
```

The runtime of `mergesort` on lists with n elements satisfies the recurrence relation $T(n) = c \cdot n + 2 \cdot T(n/2)$, where c is a constant. The solution to this relation is $T(n) = O(n \cdot \log n)$.

Question: Is the last statement true for the worst or for the best case?

In the `rec_slice_binary_search` function, slicing resulted in $O(n)$ overhead to the time complexity, which is disastrous for searching. Here, however, we deal with sorting, and an $O(n)$ overhead is completely OK.

A Three Way Race

Three sorting algorithms left Haifa at 8am, heading south. Which one will get to TAU first?
We will run them on random lists of lengths 200, 400, 800.

```
>>> from quicksort import *
>>> from mergesort import *

3 way race
quicksort
n= 200 0.17896895999999998
n= 400 0.38452376
n= 800 0.87327308
mergesort
n= 200 0.24297283999999997
n= 400 0.49345808000000013
n= 800 1.0856526
sorted #Python's sort
n= 200 0.007834879999999877
n= 400 0.02002811999999965
n= 800 0.04940151999999998 #We have a winner!
```

The results, ahhhm, speak for themselves.