Scientific Computation

Spring 2019

Lecture 3

Imperial College
London

21 January, 2019

Notes

- Lab 3 solutions have been posted online
- Office hour today is in 6M20 (10-11am)

Python notes

Getting comfortable with python:

- **Have command of *all* of the material in online lectures –use exercises for self-assessment (solutions are online)**

- **Understand structure and purpose of functions**

- **Choose an editor + terminal combination for developing code. Can be *spyder* (distributed w/ anaconda), *canopy*, or *atom* + *jupyter qtconsole*. Use python3.x (e.g. python3.6)**

- **Understand binary search and merge sort codes from week 1**

- **Further help: list of supplementary material on course webpage, office hours**

This week

- Complete merge sort
- Hash tables and constant-time search
- Patterns in gene sequences

Sorting

- A "divide and conquer" approach worked well for search
- Can we do something similar for sorting?
- Test the basic idea:
  - divide array into left and right halves
  - Sort each half
  - Then merge the two halves
- Cost of sorting 2 halves should be half of sorting the full array
- Merge can be done in O(N) operations
- Save N*(N-1)/8 during sorting, need extra O(N) during merging
  - Substantial savings for 'large' N

Merging

- Merge can be done in O(N) operations?
- How do we merge L and R below into a sorted array, M?

6	8	23	32

2	24	31	53

- Fill each element of merged array sequentially
  - 1st element: compare L[0] with R[0]

2								

  - 2nd element: compare L[0] with R[1]

2	6							

ith element: compare leftmost unassigned element of L with leftmost unassigned element of R

Requires N comparisons and N assignments

Merging

i^{th} element: compare leftmost element of L (not in M) with leftmost element of R (not in M)

Python implementation:

- **Outer loop: add one element from either L or R to M each iteration**
- **Keep track of leftmost indices in L and R**

```
indL,indR=0,0

for i in range(n):
    if L[indL]<R[indR]: #add element from L to M
        value = L[indL]
        indL = indL+1
    else: #add element from R to M
        value = R[indR]
        indR = indR+1
    M.append(value)
    #Check if all elements in L or R have been assigned
    if indL>len(L)-1:
        M = M + R[indR:]
        break
    elif indR>len(L)-1:
        M = M + L[indL:]
        break
return M
```

Merge Sort

- But why divide only once?
- The sorting step requires $O(N^2)$ operations, so we want this N to be small as possible
- Merge sort: Keep dividing until N=1, then merge multiple times

$N_s=8$

23	6	8	32	56	31	2	53

$N_s=4$

23	6	8	32

56	31	2	53

$N_s=2$

23	6

8	32

56	31

2	53

$N_s=1$: 8 1-element "arrays"

Merge Sort

23	6	8	32	56	31	2	53

Merging: N_s=1

23 and 6 → [6 23]   8 and 32 → [8 32]   56 and 31 → [31 56] 2 and 53 →[2 53]

N_s=2

[6 23] and [8 32] → [6 8 23 32]   [31 56] and [2 53] → [2 31 53 56]

N_s=4

[6 8 23 32] and [2 31 53 56] → [2 6 8 23 31 32 53 56]

Merge Sort

23	6	8	32	56	31	2	53

Merging: N_s=1

23 and 6 → [6 23]   8 and 32 → [8 32]   56 and 31 → [31 56] 2 and 53 →[2 53]

N_s=2

[6 23] and [8 32] → [6 8 23 32]   [31 56] and [2 53] → [2 31 53 56]

N_s=4

[6 8 23 32] and [2 31 53 56] → [2 6 8 23 31 32 53 56]

What is the cost?

$\log_2 N$ "levels", N/N_s merges per level, ~$4N_s$ operations per merge

approximately $4N \log_2 N$ operations

Imperial College
London

Merge Sort

- **Cost of merge sort is $O(N \log_2 N)$ vs. $O(N^2)$ for insertion sort, so merge sort is clearly superior**
- **How do we implement it?**
- **We should recognize that after the divide step, each of the subarrays needs to be sorted**
  - **Apply merge sort to original array and sequence of subarrays**
  - **Best implemented using *recursion:***

```
def msort(A):
    n = len(A)
    if n==1:
        return A
    else: #call msort on Left and Right lists, then merge
        nh = int(n/2)
        L = msort(A[:nh])
        R = msort(A[nh:])
        M = merge(L,R)
        return M
```

Recursion

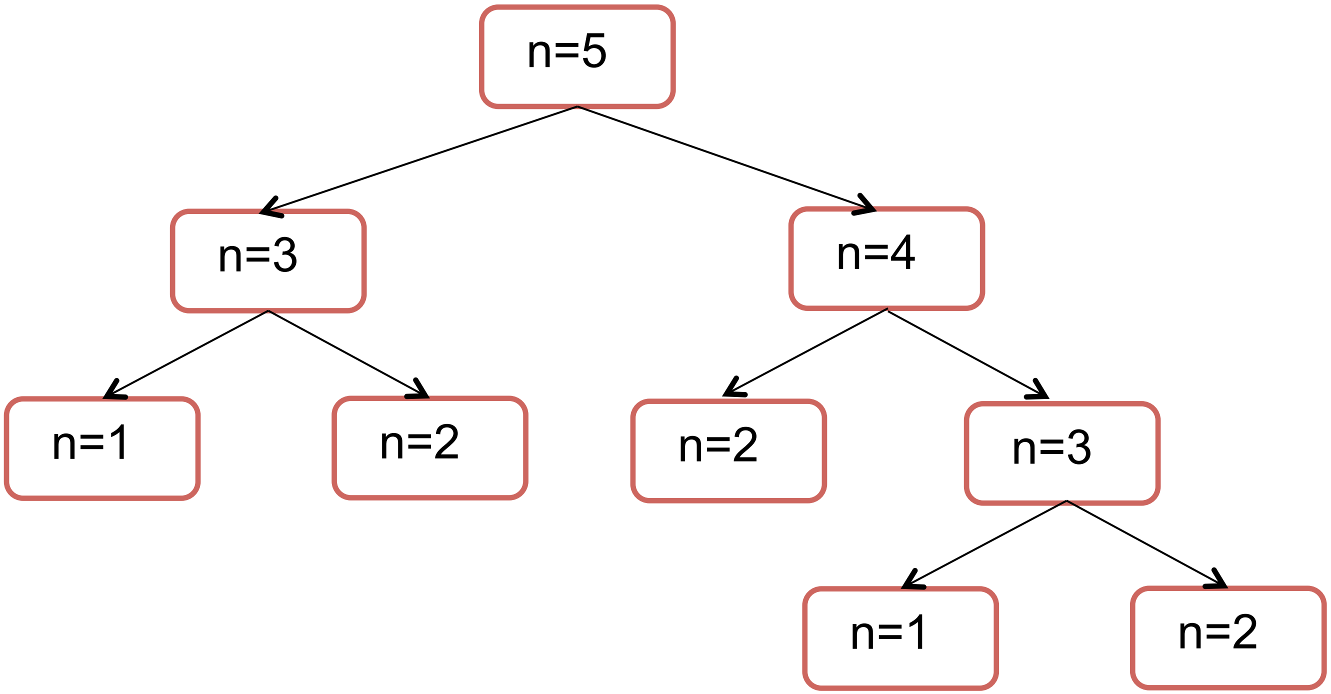
- Lab 1 asked you to work with recursive functions
- It can be helpful to put together a *recursion tree*
- For the Fibonacci example:

```
In [2]: def fib(n):
            """Find nth term in Fibonacci sequence start from 0,1
            """
            print("n=",n)
            if n==1:
                return 0
            elif n==2:
                return 1
            else:
                return fib(n-2) + fib(n-1)
```

```
In [2]: fib(5)
n= 5
n= 3
n= 1
n= 2
n= 4
n= 2
n= 3
n= 1
n= 2
Out[2]: 3
```

Recursion

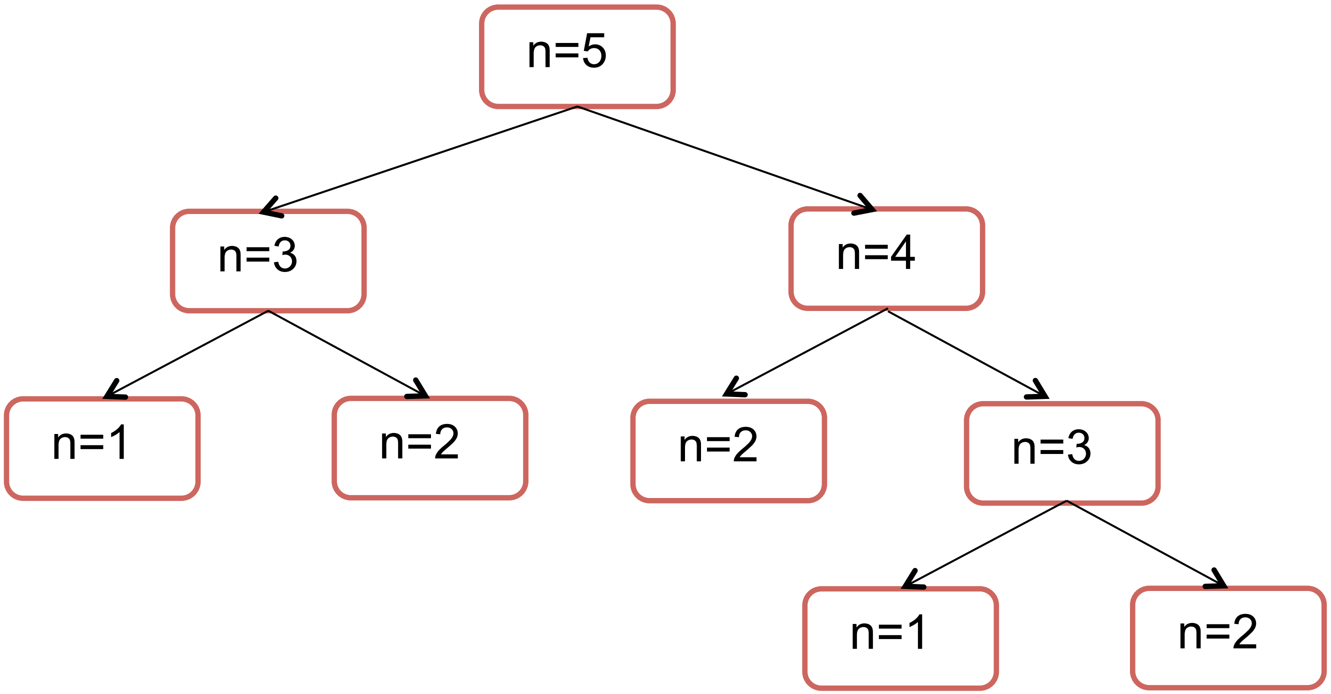
- **Lab 1 asked you to work with recursive functions**
- **It can be helpful to put together a *recursion tree***
- **For the Fibonacci example:**




```
In [2]: fib(5)
n= 5
n= 3
n= 1
n= 2
n= 4
n= 2
n= 3
n= 1
n= 2
Out[2]: 3
```

Recursion

- **Lab 1 asked you to work with recursive functions**
- **It can be helpful to put together a *recursion tree***
- **For the Fibonacci example:**




```
In [2]: fib(5)
n= 5
n= 3
n= 1
n= 2
n= 4
n= 2
n= 3
n= 1
n= 2
Out[2]: 3
```

- **This is just a simple example to illustrate recursion**
- **It is very inefficient! (why?)**

Merge Sort

- Won't go into details of how Python interpreter deals with recursive functions
- Can have problems with efficiency when number of recursive calls becomes large ("call stack" is too large)
- Recursion isn't usually necessary, some find it to be elegant or natural
- Key idea here: "divide and conquer"
  - General approach used in many algorithms
  - Need total work for sub-problems after division to be sufficiently small relative to work for original problem
  - E.g. won't help when summing elements in array

Sorting and searching

- **Binary search and merge sort are two of three (or so) algorithms which any programmer must master to have finite credibility**

- **Quick sort is the third – it is a randomized algorithm, and on average is faster than merge sort (though both are $O(N\log_2 N)$) and used more often**

- **Many other sorting algorithms are out there: selection, bubble, heap, …**

- **See https://www.toptal.com/developers/sorting-algorithms for nice visualizations**

- **In practice, no need to code your own sorting routine. Cf. np.sort or the sort function in Unix**

- **Understanding how to implement and analyze sorting and searching algorithms is essential to make progress on more complicated problems**

- Lecture 1: Binary search applied to sorted array requires $\log_2 N$ operations

- Can we do better?

Searching

- Lecture 1: Binary search applied to sorted array requires $\log_2 N$ operations

- Can we do better?

- Real-world problem:
    - A startup is keeping track of unique visitors to its website each month
    - Each visitor must be assessed as “new” or “repeat”
    - Visitors who have not visited in 30+ days must be removed
    - Everything must be fast and accurate!