CS5800 Module 2

How Algorithms Can Make Differences To Any Problem

Multiple Ways of Computing Fibonacci Numbers

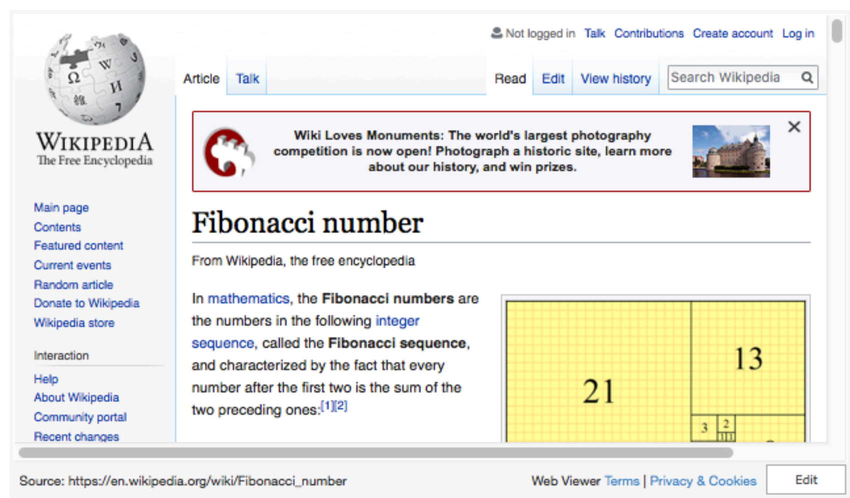
Lesson Objectives

- Calculate correct Fibonacci numbers manually
- Write recursive and iterative algorithms to calculate arbitrary Fibonacci numbers
- Distinguish time complexities of recursive and iterative Fibonacci calculation algorithms
- Argue what makes recursive Fibonacci calculation algorithm very slow

Fibonacci Numbers

- $Fib_0 = 0, Fib_1 = 1$
- $Fib_n = Fib_{n-1} + Fib_{n-2}$ for any integer $n \geq 2$
- 0, 1, ___, ___, ___, ___, ___, ___, ___, ___, …

Fibonacci Numbers Wiki Page



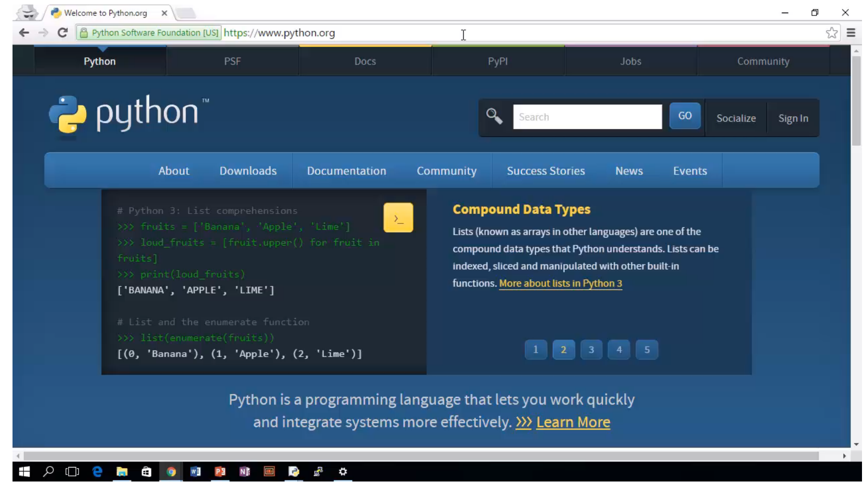
Simple (Naive) Fibonacci Computation Algorithm

- How did you find Fib_{15} in the previous quiz problem?
- Will the following simple Fibonacci computation code be good enough for finding Fib_{50}?

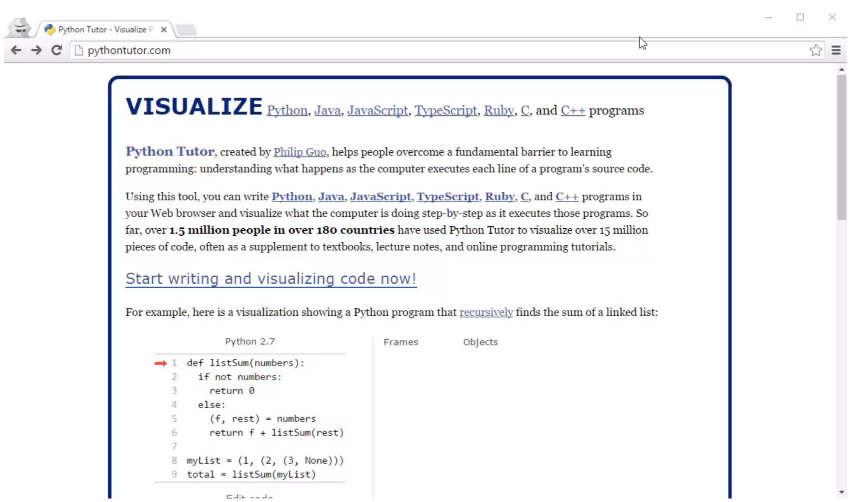
```
# Simple Python code to compute/print Fib_50.
# (should be readable to everyone)
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

print fib(50)
```

Recursive Fibonacci Code Run Screencast

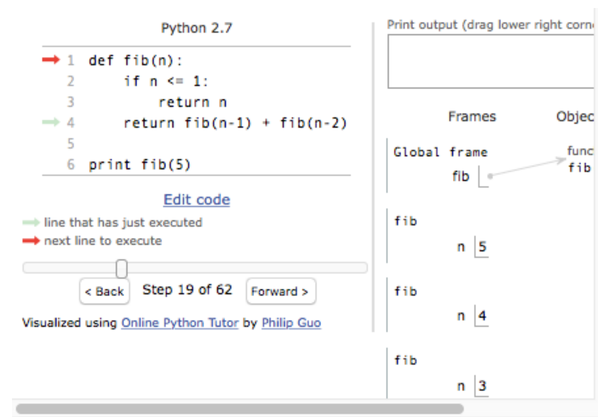


Python Online Tutor Screencast



Why So Slow?

- How recursive calls are executed?
- Did you see how many times fib(3) was called?
- How about fib(2)? fib(1)? fib(0)?
- Edit code by changing 5 in fib(5) and check # steps.



Unnecessary Redundant Work

- When computing Fib_{50},
  - How many times fib(50) was called?
  - How many times fib(49) was called?
  - How many times fib(48) was called?
  - How many times fib(47) was called?
  - How many times fib(46) was called?
  - How many times fib(45) was called?
  - …
  - How many times fib(3) was called?
  - How many times fib(2) was called?

Analysis of Recursive Fibonacci

- $T(n)$: # execution steps when calling fib(n)
  - "Execution steps" include comparisons, arithmetic operations (e.g., +), assignments, return, … . It may not be the same as # lines of code.
- $T(0)$? $T(1)$?
- For $n \geq 2$, $T(n)$?

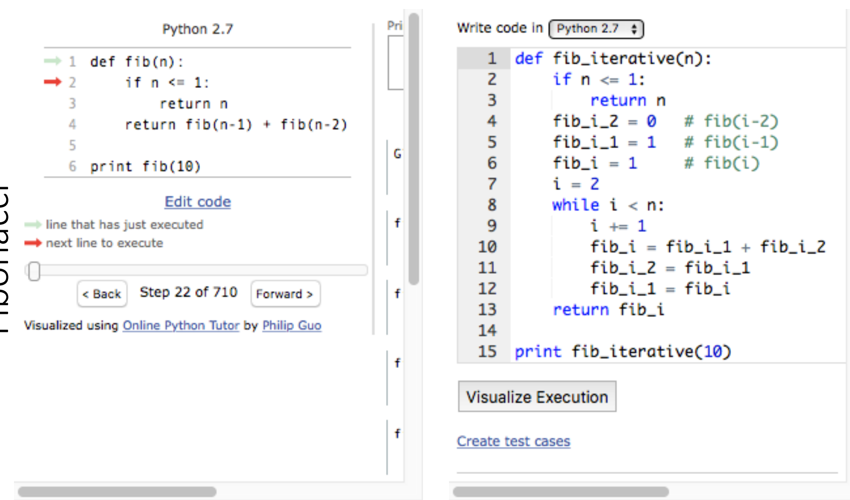
```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

- An equation like above is called a ***recurrence relation***.
- Closed form solution (non-recurrence):

Can We Do Any Better?

- How did you find Fib_{15} in the Checkpoint Quiz?
- Can you implement that idea in code?
- Check the interactive codes in the next slides
  - Click 'Visualize Execution' to visualize/trace the algorithm's execution.
  - Trace each execution by clicking Forward. Make sure to think about what the next step does before clicking Forward.
  - Compare the two different numbers of steps for the same n.
    - Click 'Edit Code', replace 5 in 'print fib(5)' with another number, and click 'Visualize Execution' again to see the new number of steps for the different n.

Recursive vs. Iterative Fibonacci



Analysis of Iterative Fibonacci

- $T(0), T(1)$ are the same as recursive.
- For $n \geq 2$, the while loop iterates exactly $n-2$ times
  - $T(n) =$
- Asymptotic notation for $T(n)$

```python
def fib_iterative(n):
    if n <= 1:
        return n

    fib_i_2 = 0  # fib(i-2)
    fib_i_1 = 1  # fib(i-1)
    fib_i = 1    # fib(i)
    i = 2
    while i < n:
        i += 1
        fib_i = fib_i_1 + fib_i_2
        fib_i_2 = fib_i_1
        fib_i_1 = fib_i

    return fib_i
```

Comparing Growths of Linear Time and Exponential Time Complexities

- Assuming each execution step takes $1\mu s$ (microsecond = 10^{-6}s, frequently typed as ‘us’), $T_{rec}(n) = 2^n$ and $T_{iter}(n) = 10000n$ (some big coefficient),

n	5	10	20	30	40	50	100	1000	10000
Iter. time									
Recur. time									

- See for yourself by computing/printing first 50 Fibonacci numbers using fib_iterative(n)
- Which algorithm would you use? Can we do better?

Are All Recursive Algorithms Bad?

- Fibonacci is an extreme case
- Usually same time complexity
  - But still bigger coefficient
- Higher space complexity
  - Iterative is usually $O(1)$, whereas recursive is usually $O(n)$.
  - If memory is limited, can’t use recursive.
- On the other hand, recursive algorithm can be:
  - Easier to understand
  - Smaller code: Iterative code can be extremely complicated in many cases
  - Easier to analyze time complexity

Time Complexities of Quadratic Sorting Algorithms

Selection, Insertion, Bubble Sorts

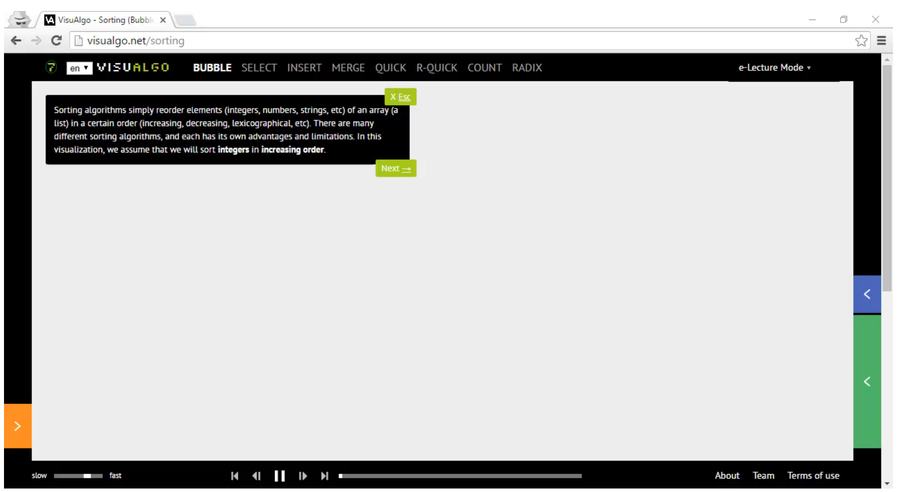
Lesson Objectives

- Utilize algorithm visualization tool to understand how well-known elementary sorting algorithms work
- Utilize algorithm visualization tool to gain understanding on time complexities of well-known elementary sorting algorithms
- Task: Make sure to click the link in the following page and experiment as instructed.
  - There's also a screencast introducing the algorithm visualization tool

VisuAlgo.Net/sorting

- Click the link above and do the following experiments:
- For each 'BUBBLE', 'SELECT', 'INSERT', do the following:
  - Click Create. Then for each 'Random', 'Sorted-Increasing', 'Sorted-Decreasing':
    - Click 'Sort-Go'.
- See how each algorithm works visually
  - There are checkpoint quiz problems about the sorting algorithms
- Think about time complexity of each case
  - "Complexity": Since we are not using the raw running time, the word "complexity" is used to refer to the performant nature of an algorithm (its difficulty)
  - Complexity is always stated in asymptotic notation (big-Oh, Omega, Theta)

VisuAlgo.Net/sorting Screencast



Time Complexities of Quadratic Sorts

- "Quadratic" because their worst-case time complexities are all $O(n^2)$.
- We saw in Module 1 that in fact, selection sort is $\Theta(n^2)$!
  - Its best-case time complexity is still $\Omega(n^2)$.
- What is the best case time complexity for bubble & insertion sort?
- Can you prove formally your claim for the question above?
- Can we do better?

Merge Sort And Intro To Divide-And-Conquer

Improving Sorting Performance From Quadratic To Linear-Logarithmic

Lesson Objectives

- Identify the result of merging given two sorted subarrays, by applying the merge operation correctly
- Write the merge sort pseudocode fluently
- Derive merge sort's asymptotic time complexity by establishing and solving a recurrence relation

Sub-Problem Property

- Recall selection sort:
  - After the first pass, we got a smaller problem of the same type (sorting).
    - Sorting an array with one fewer items, though the indexing structure is different.
  - Or we could say that we deliberately seek to reduce the original problem into a smaller sub-problem and the related reduction process.
    - In this case, the reduction process is a pre-process of finding the smallest and swapping it with the first entry.
  - Very similar nature in bubble sort: Reducing problem size by one every pass
- Can we think of a different sub-problem structure and reduction?
  - If we are reducing the problem size by one, why not reducing bigger?
  - How about *dividing* the original problem by halves?

Merge Sort: Divide-And-Conquer Sorting

- Divide the original array into two halves.
  - Sort each half.
    - Need to use recursion. Recall the recursive algorithm for Fibonacci calculation.
    - For now, do not consider jumping into the recursive calls.
      - Just assume that the recursive call returned with the sorted sub-array (half).
- Then merge the two sorted halves. E.g.:
  - Sorted first half: 11, 33, 44, 66, 88
  - Sorted second half: 22, 35, 40, 77, 80
  - Merging two sorted subarrays:
- CLRS Section 2.3 Designing Algorithms

Merge Sort

http://visualgo.net/sorting

Code: Merge Sort

- Top-level sort is easy: (CLRS pp. 34)

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

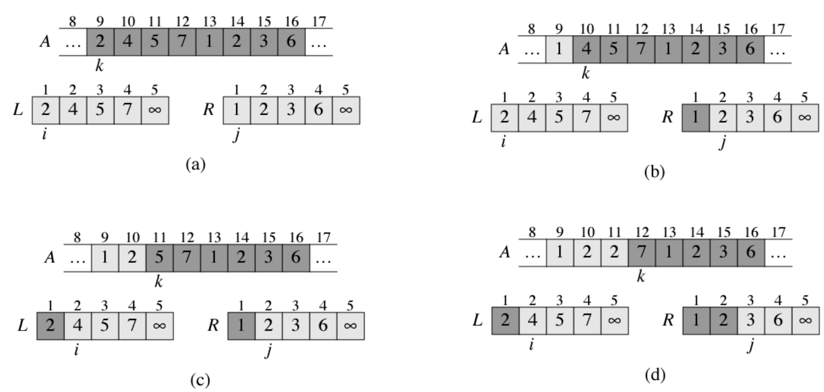
- Top-level call is MERGE-SORT(A, 1, A.length) for $A = \langle A[1], A[2], \dots, A[n] \rangle$ where A.length = n.
- What's difficult is MERGE(A, p, q, r)

```
MERGE-SORT(A)
MERGE-SORT(A, 1, A.length)
```

Code: Merge (CLRS pp. 31)

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Merge Operations (CLRS Fig. 2.3 in pp. 32)



Sidebar: MERGE() Implementation

- What if we can't depend on the availability of ∞?
- Creating "new arrays" in Line 3 of MERGE(A,p,q,r) is computationally expensive. How can we avoid creating new arrays every time MERGE() is called?
- Can you re-implement MERGE(A,p,q,r) with the two constraints above?
  - This is a homework problem, and may be an exam problem!

Analysis of Merge Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

- Straightforward top-level recurrence for $T_{MergeSort}(n)$:

$$T_{MergeSort}(n) = 2T_{MergeSort}\left(\frac{n}{2}\right) + T_{Merge}(n) + c_1$$

  - Of course $T_{MergeSort}(0) = T_{Merge}(0) = c_2$ (all c_i's are some constants)
- What is $T_{Merge}(n)$?
  - Counting the number of steps executed in MERGE(A,p,q,r) when r-p+1=n, we get:

$$T_{Merge}(n) = c_3 n + c_4$$

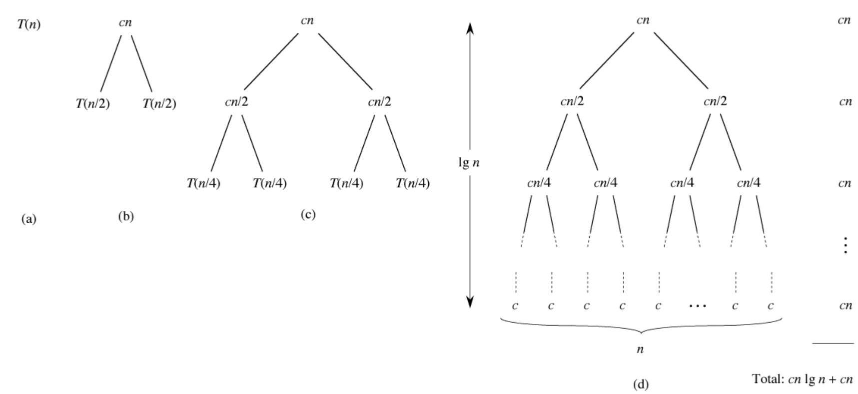
- Therefore, $T_{MergeSort}(n) = 2T_{MergeSort}\left(\frac{n}{2}\right) + cn + c'$

Solving $T(n) = 2T\left(\frac{n}{2}\right) + cn$

- Expansion method:

- Solution: $T(n) =$

Recursion Tree Method for Solving Recurrence (CLRS Fig. 2.5 in pp. 38)



Merge Sort vs. Quadratic Sorts

- $\Theta(n \log_2 n)$ merge sort vs. $O(n^2)$ quadratic (selection, insertion, bubble) sorts
- Assuming each execution step takes $1\mu s$ (microsecond = 10^{-6}s, frequently typed as 'us'), $T_{quad}(n) = n^2$ and $T_{merge}(n) = 10000n \log_2 n$ (some big coefficient),

n	10	100	1000	10000	100000	10^6	10^7	10^8
n^2	0.1ms	0.01s	1s	100s	~2.8h	~11.6 days	~3.17 years	~317 years
$10000n \log_2 n$	~0.33s	~6.6s	~100s	22m	~4.6h	~2.3 days	~26.7 days	~307.6 days

- Which algorithm would you use? Can we do better?