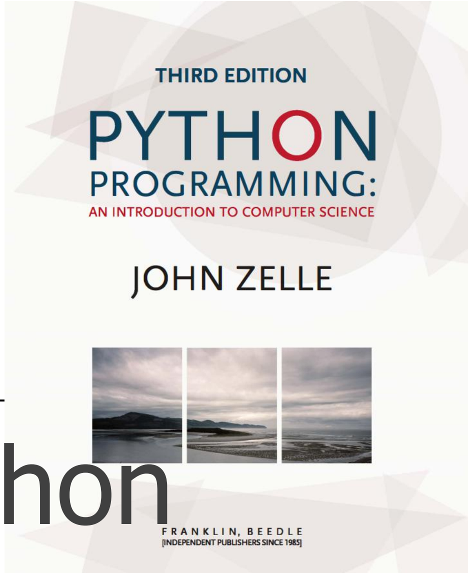


An Introduction to Python Programming

Chapter 13: Algorithm Design and Recursion

Objectives

- To understand the basic techniques for **analyzing** the **efficiency** of algorithms.
- To know what **searching** is and linear or binary search.
- To understand the basic principles of **recursive definitions and functions**.
- To understand **sorting in depth** and know selection sort and merge sort.

Searching

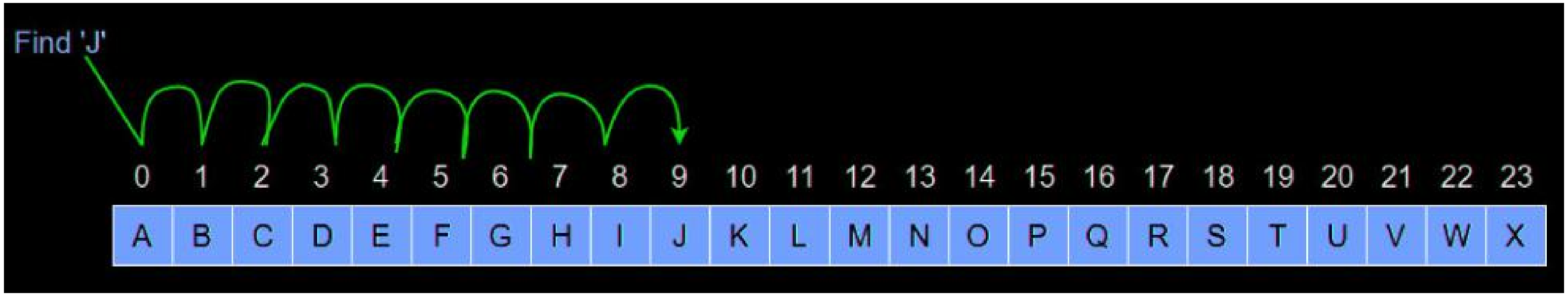
- ***Searching*** is the process of looking for a particular value in a collection.

```
def search(x, nums):
    # nums is a list of numbers and x is a number
    # Returns the position in the list where x occurs or -1 if
    #    x is not in the list.
```

```
>>> search(4, [3, 1, 4, 2, 5])
2
>>> search(7, [3, 1, 4, 2, 5])
-1
```

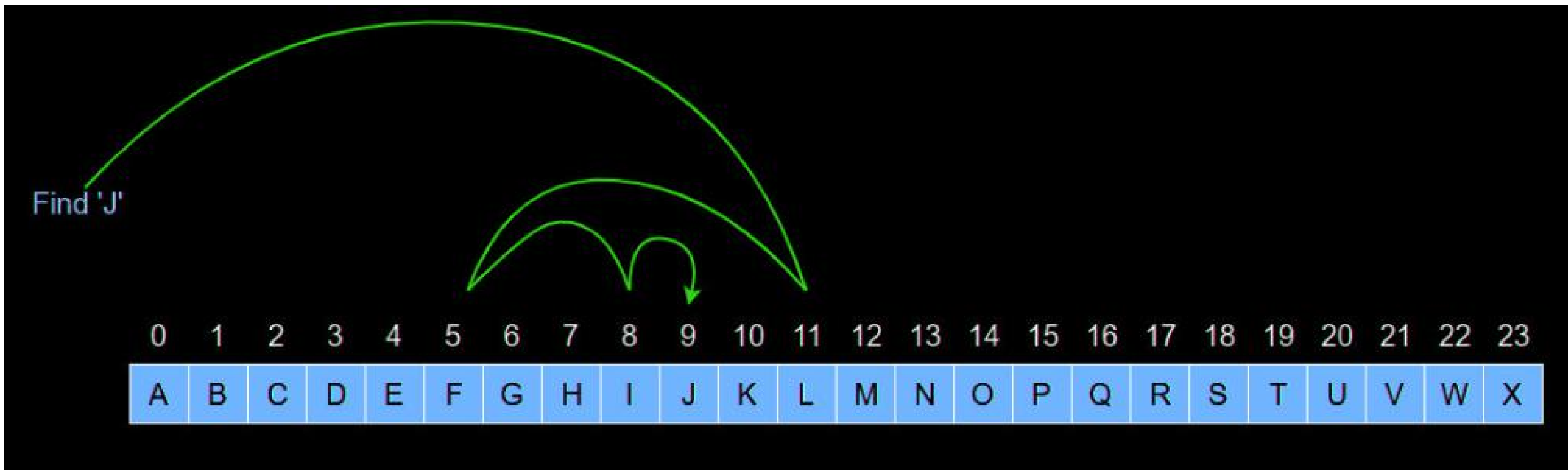
Using somebuilt-in search-related methods

Strategy 1: Linear Search

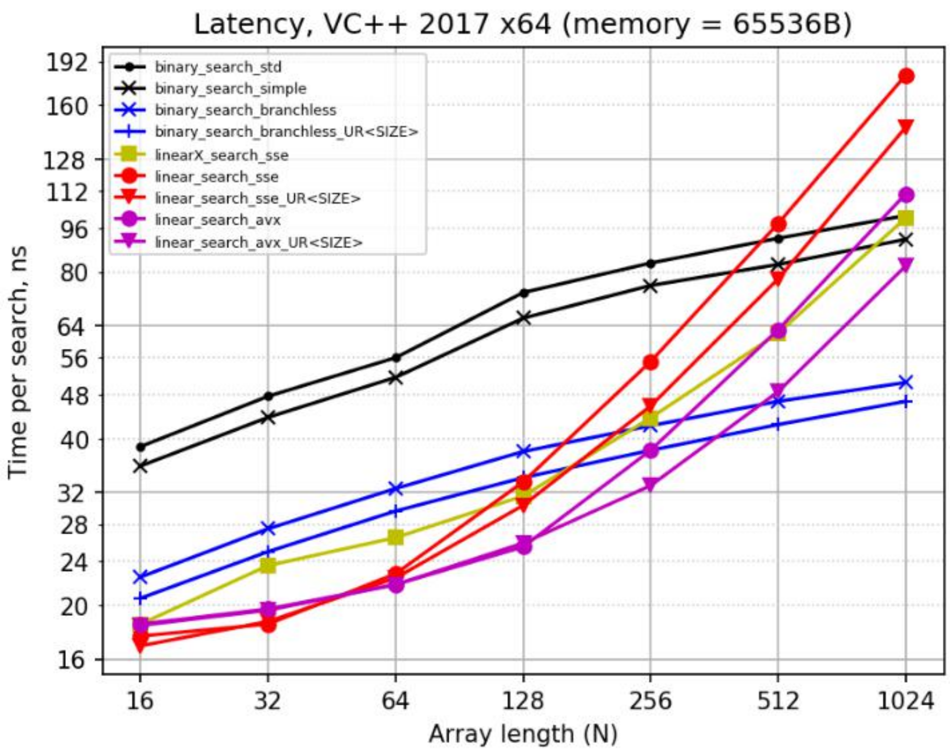


- If we have **a very large collection of data**, we might **don't** want to **look at every single item** to determine a particular value appears in the list.

Strategy 2: Binary Search



Comparing Algorithms



- for lists of length 10 or fewer, linear search was faster.
- In the range of length 10-1000, there was no noticeable difference .
- For a list of a million elements, linear search cost 2.5 seconds whereas binary search only 0.0003 seconds.

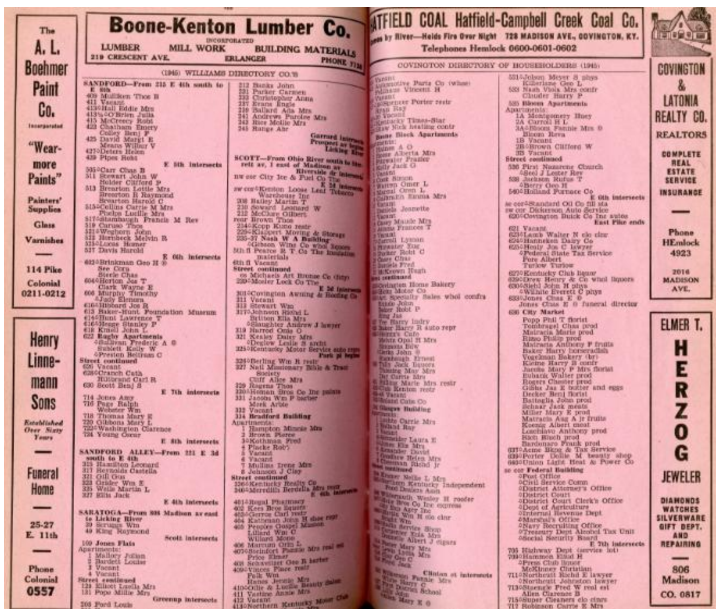
Comparing Algorithms

- For the linear search, the time required is **linearly** related to the size of the list ***n***.
- For the binary search, loop ***i*** times, 2^i items are examined, $i = \log_2 n$

list size	halvings
1	0
2	1
4	2
8	3
16	4

Comparing Algorithms

- Suppose a **city phone book**



- Searching a million(2^{20}) items requires only **20 guesses**.
- By comparison, a linear search would require **a million guesses**.

Recursive Problem-Solving

- The **Recursive** technique is known as ***a divide and conquer*** approach.
- **Algorithm**: binarySearch

```
mid = (low + high) // 2
if low > high
    x is not in nums
elif x < nums[mid]
    perform binary search for x in nums[low]...nums[mid-1]
else
    perform binary search for x in nums[mid+1]...nums[high]
```

Recursive Problem-Solving

- **Recursive Definition**: a description of something that **refers to itself** is called a recursive definition.
- For example,in mathematics

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

- But what is **(n-1)!** ? To find out, we apply the definition again.

$$4! = 4(3!) = 4(3)(2!) = 4(3)(2)(1!) = 4(3)(2)(1)(0!) = 4(3)(2)(1)(1) = 24$$

Recursive Problem-Solving

- **Recursive functions**
  - the **factorial** of n:

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

```
>>> from recfact import fact
>>> fact(4)
24
>>> fact(10)
3628800
```

Recursive Problem-Solving

- Example: **String Reversal**

1st way: **STRING** $\longrightarrow$ **List of Charactors**

Reverse the list

2nd way: **Think of a string as a recursive object.**

STRING=its first character + "all the rest."

```
def reverse(s):
    return reverse(s[1:]) + s[0]
```

Recursive Problem-Solving

- **NOTE:** This function doesn't quite work.1000 lines is like the following!

```
>>> reverse("Hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in reverse
  File "<stdin>", line 2, in reverse
...
  File "<stdin>", line 2, in reverse
RuntimeError: maximum recursion depth exceeded
```

- Need a **base case!**
- No return, **infinite recursion!**

Recursive Problem-Solving

- A correct version of reverse:

```
def reverse(s):
    if s == "":
        return s
    else:
        return reverse(s[1:]) + s[0]

>>> reverse("Hello")
'olleH'
```

Recursive Problem-Solving

- Example: **Anagrams**

Let's write a function generating a list of all the possible anagrams of a string.

"abc" → *"bac", "bca", "acb", "cab", "cba"*

Recursive Problem-Solving

```
def anagrams(s):
    if s == "":
        return [s]
    else:
        ans = []
        for w in anagrams(s[1:]):
            for pos in range(len(w)+1):
                ans.append(w[:pos]+s[0]+w[pos:])
        return ans
```

- The **outer loop** iterates through each anagram of the tail of s
- The **inner loop** goes through each position in the anagram

Recursive Problem-Solving

- Example: **Binary Search**

```
def recBinSearch(x, nums, low, high):
    if low > high:                  # No place left to look, return -1
        return -1
    mid = (low + high) // 2
    item = nums[mid]
    if item == x:                   # Found it! Return the index
        return mid
    elif x < item:                  # Look in lower half
        return recBinSearch(x, nums, low, mid-1)
    else:                           # Look in upper half
        return recBinSearch(x, nums, mid+1, high)
```

calling functions is generally slower than iterating a loop.

Recursive Problem-Solving

- **Recursion** vs. **Iteration**

- In fact, recursive functions are a **generalization** of loops.
- Be careful of some **very ineffcient recursive algorithms**.

Calculating the nth Fibonacci number ：

$$F_0=0\text{，}F_1=1\text{，}F_n=F_{n-1}+F_{n-2}\ (n>=2\text{，}n\in N^*)$$

Recursive Problem-Solving

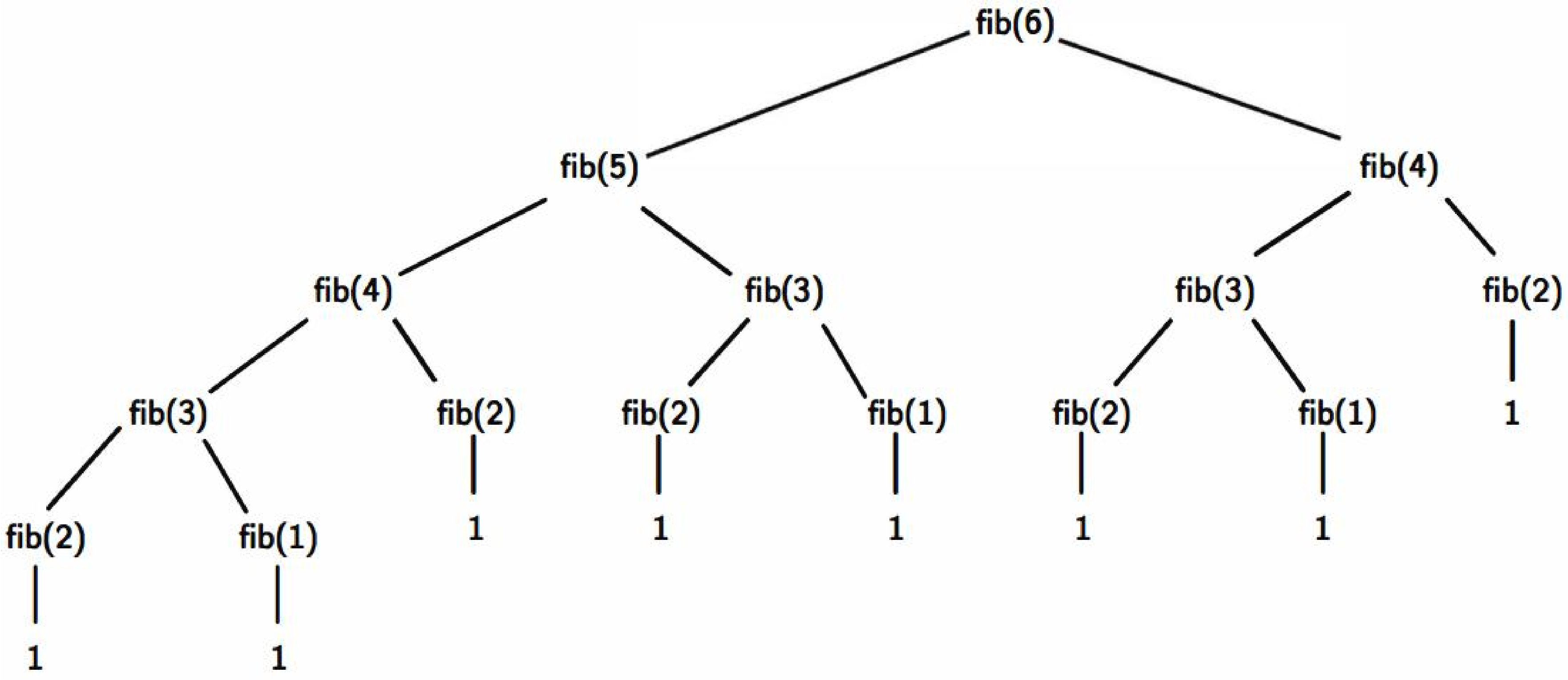
The loop function:

```
def loopfib(n):
    # returns the nth Fibonacci number

    curr = 1
    prev = 1
    for i in range(n-2):
        curr, prev = curr+prev, curr
    return curr
```

The recursive function:

```
def fib(n):
    if n < 3:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Recursive Problem-Solving



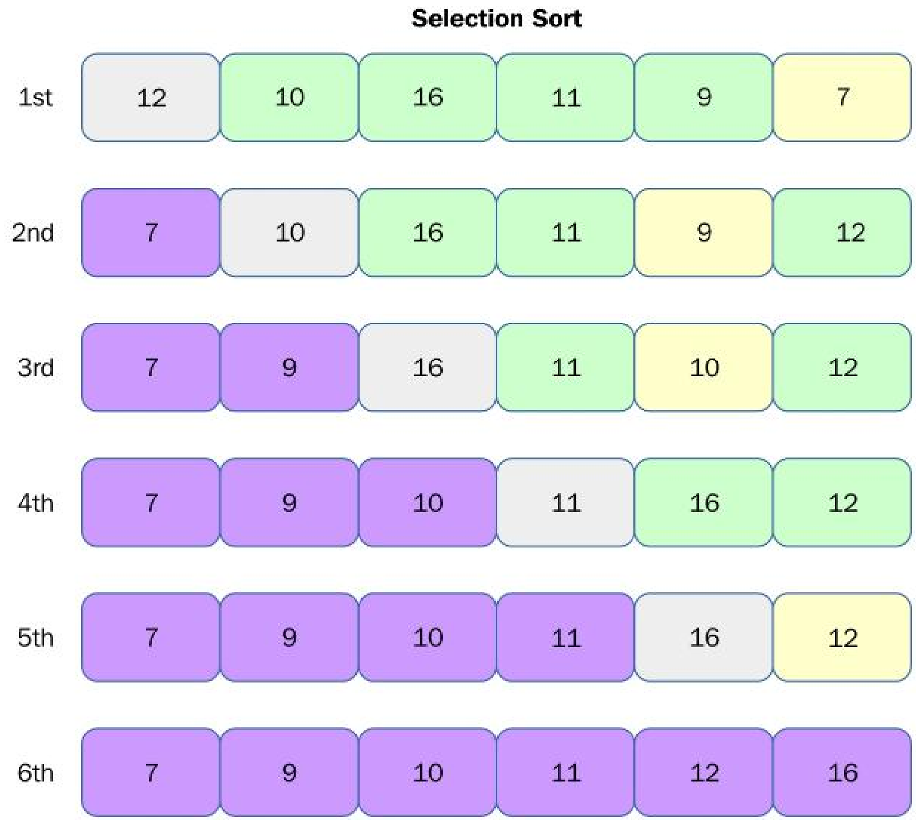
This redundancy piles up!

Sorting Algorithms

- We've written a function that calls *itself*, a *recursive function*.
- The function first checks to see if we're at the base case (`n==0`). If so, return 1. Otherwise, return the result of multiplying *n* by the factorial of *n-1*, `fact(n-1)`.

Sorting Algorithms

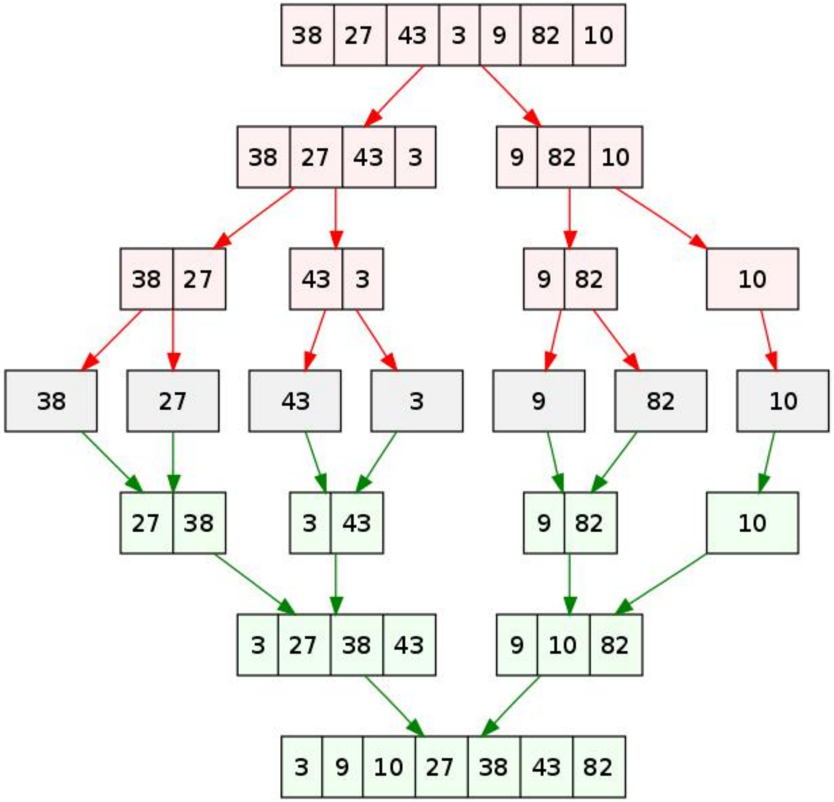
- Naive Sorting: **Selection Sort**



```python
def selSort(nums):
    # sort nums into ascending order

    n = len(nums)

    # For each position in the list (except the very last)
    for bottom in range(n-1):
        # find the smallest item in nums[bottom]..nums[n-1]

        mp = bottom                      # bottom is smallest initially
        for i in range(bottom+1,n):      # look at each position
            if nums[i] < nums[mp]:       # this one is smaller
                mp = i                   #     remember its index

        # swap smallest item to the bottom
        nums[bottom], nums[mp] = nums[mp], nums[bottom]
```

Sorting Algorithms

- Divide and Conquer: **Merge Sort**



Sorting Algorithms

```
Algorithm: merge sort nums

split nums into two halves
sort the first half
sort the second half
merge the two sorted halves back into nums
```

- The Python implementation of the merge process is shown on P479.

Sorting Algorithms

- Update the **merge sort algorithm** by **recursive**:

```
def mergeSort(nums):
    # Put items of nums in ascending order
    n = len(nums)
    # Do nothing if nums contains 0 or 1 items
    if n > 1:
        # split into two sublists
        m = n // 2
        nums1, nums2 = nums[:m], nums[m:]
        # recursively sort each piece
        mergeSort(nums1)
        mergeSort(nums2)
        # merge the sorted pieces back into original list
        merge(nums1, nums2, nums)
```

Sorting Algorithms

- **Comparing Sorts**

list :

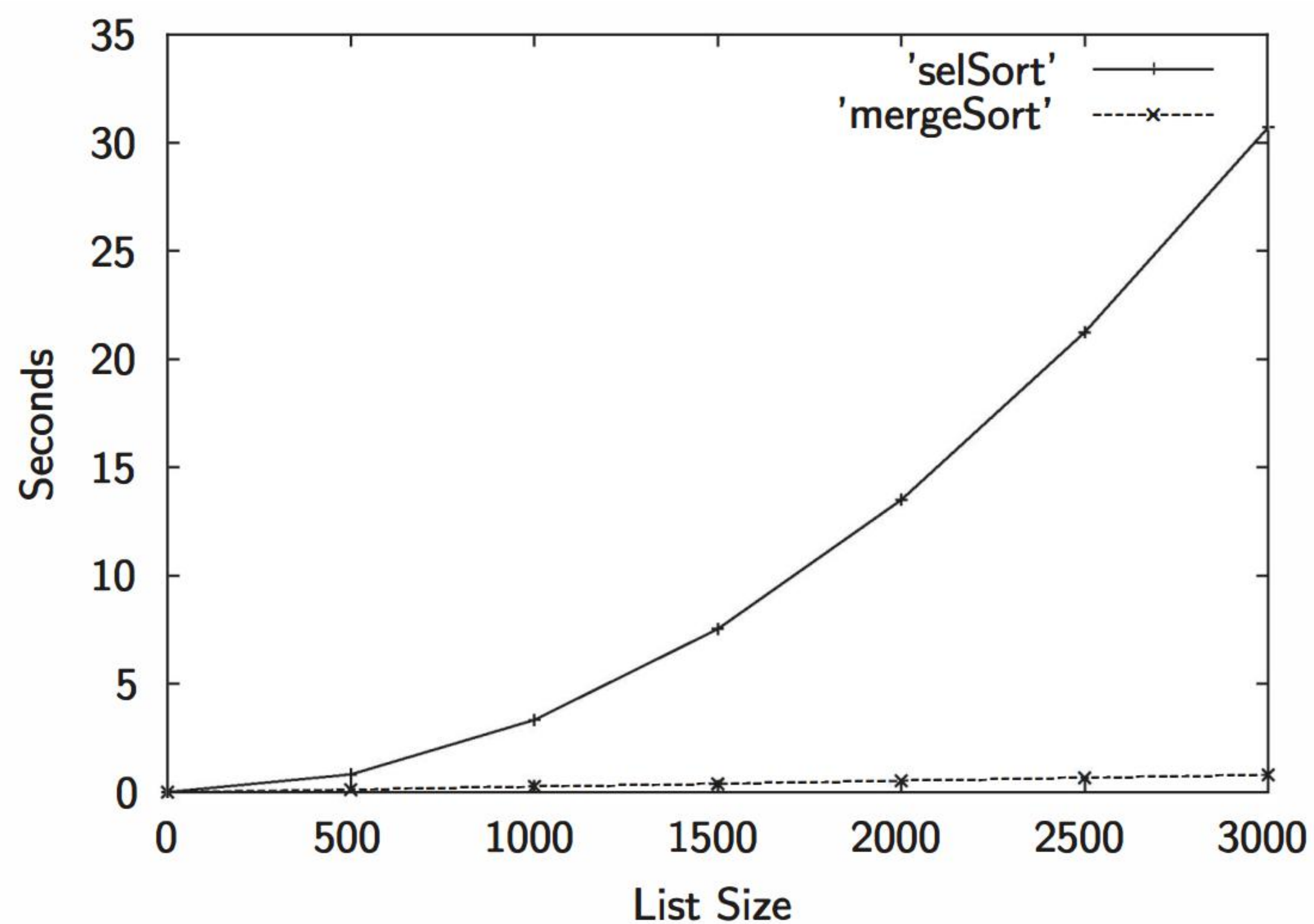
9	5	4	……	i	……	n

The time required by selection sort：

$$n + (n-1) + (n-2) + (n-3) + \ldots + 1$$

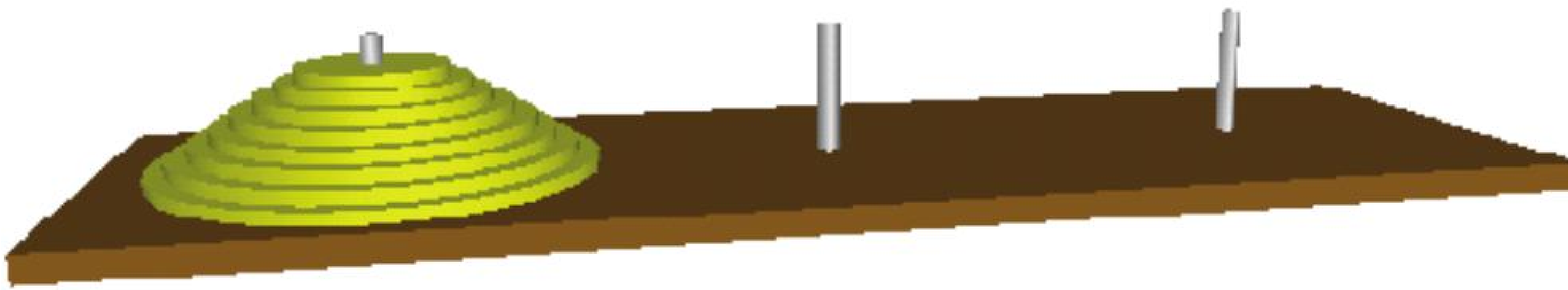
The total work required by merge sort： $n \log_2 n$

Sorting Algorithms



Hard Problems

- **Towers of Hanoi**



① Only one disk may be moved at a time.
② A disk may not be "set aside."
③ A larger disk may never be placed on top of a smaller one.

Hard Problems

For 1 disk	Move it fom A to C
For 2 disks	1. Move the smaller disk from A to B 2. Move the larger from A to C 3. Move the smaller disk from B to C
For 3 disks	1. Move a tower of two from A to B. 2. Move one disk from A to C. 3. Move a tower of two from B to C.

Hard Problems

- Using recursive algorithm

```
move n-1 disk tower from source to resting place
move 1 disk tower from source to destination
move n-1 disk tower from resting place to destination
```

```
def moveTower(n, source, dest, temp):
    if n == 1:
        print("Move disk from", source, "to", dest+".")
    else:
        moveTower(n-1, source, temp, dest)
        moveTower(1, source, dest, temp)
        moveTower(n-1, temp, dest, source)
```

Hard Problems

```
def hanoi(n):
    moveTower(n, "A", "C", "B")
```


```
>>> hanoi(4)
Move disk from A to B.
Move disk from A to C.
Move disk from B to C.
Move disk from A to B.
Move disk from C to A.
Move disk from C to B.
Move disk from A to B.
Move disk from A to C.
Move disk from B to C.
Move disk from B to A.
Move disk from C to A.
Move disk from B to C.
Move disk from A to B.
Move disk from A to C.
Move disk from B to C.
```

Hard Problems

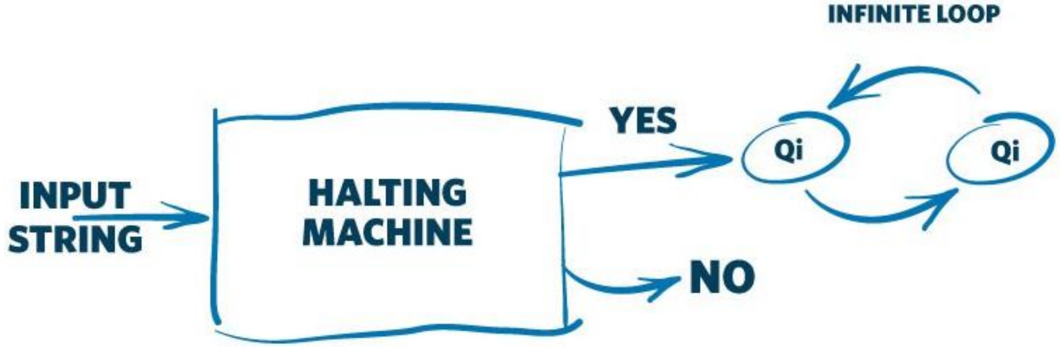
- ***How many steps*** *does it take to move a tower of* ***size n?***

number of disks	steps in solution
1	1
2	3
3	7
4	15
5	31

- *Solving a puzzle of size **n** will require* $2^n - 1$ *steps.*

Hard Problems

- **The Halting Problem**



Program: Halting Analyzer

Inputs: A Python program file.
The input for the program.

Outputs: "OK" if the program will eventually stop.
"FAULTY" if the program has an infinite loop.

Hard Problems

- *How do I know that there is **no solution** to this problem?*
  - **Proof by Contradiction**

 # If such an algorithm could be written

```
def terminates(program, inputData):
    # program and inputData are both strings
    # Returns true if program would halt when run with inputData
    #    as its input.
```

Hard Problems

- **Using the terminates function**

```
def main():
    # Read a program from standard input
    lines = []
    print("Type in a program (type 'done' to quit).")
    line = input("")
    while line != "done":
        lines.append(line)
        line = input("")
    testProg = "\n".join(lines)

    # If program halts on itself as input, go into an infinite loop
    if terminates(testProg, testProg):
        while True:
            pass    # a pass statement does nothing

main()
```

Hard Problems

- Does **turing. py** halt when given itself as its input?

  Turing. py can't both halt and not halt.

- ***Conclusion***

Hope these helped you on the road to becoming a computer programmer!