Introduction to Information Science and Technology

(SI100B Python Part)

Fu Song
School of Information Science and Technology
ShanghaiTech University

Objective

To understand basic techniques for *analyzing algorithms*.

  - CS140 Algorithm

- To understand the algorithms for *linear* and *binary search*.
- To understand *sorting* and know the algorithms for ***selection*** sort and ***merge*** sort.
- To understand some problems are ***intractable*** and others are ***unsolvable***.

Roadmap

- Searching problem
  - Linear and binary search
- Recursion and Divide-and-Conquer
- Sorting problem
  - Selection sort and merge sort
- Intractability and unsolvability

Search problem

- Searching is a basic, but very important problem.
- Specification:

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

Built-in search methods

- Using `in`:

```
if x in nums:
    # do something
```

- The `index` method (throws an exception if x is not in the list.):

```
>>> nums = [3, 1, 4, 2, 5]
>>> nums.index(4)
2
```

- You can implement search by using `index`, however, we want to know how `index` is implemented.

Linear search

- Pretend you were given a page full of randomly ordered numbers and were asked where 13 was.
- How would you do it?
- Scanning downward from the top to the bottom!

Linear search

```
def search(x, nums):
    for i in range(len(nums)):
        if nums[i] == x:
            return i
    return -1
```

- The Python `in` and `index` operations both implement linear searching algorithms.

Binary search

- If numbers are in ascending order, what would you do?
- Like finding some page number in a book.
- You can “throw away” some parts of the list after one comparison.

Binary search

```
def search(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high)/2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

Comparing algorithms

- Intuitively, we might expect the linear search to work better for small lists, and binary search for longer lists. But how can we be sure?
- Method#1: Test them!
  1. n <= 10, linear search faster
  2. 10 < n <= 1000, little difference
  3. n > 1000, binary search faster
  4. n = 100,000,000, linear 2.5s, binary .0003s

Comparing algorithms

- Method#1 is dependent on hardware (cpu, etc.) and software (compilers, etc.).
- Method#2: counting the steps!
- *How many steps are needed to find a value in a list of size n?*
- Linear: the steps required are linearly related to n
- Binary: the steps required are related to log n

Comparing algorithms

- Binary search is much faster. But why the built-in functions use linear search?
- Before carrying out binary search, you need to sort the list!
- How efficient can we solve sorting problem?
- Answer is at least *nlog n* steps for lists sized n.

Roadmap

- Searching problem
  - Linear and binary search
- Recursion and Divide-and-Conquer
- Sorting problem
  - Selection sort and merge sort
- Intractability and unsolvability

Recursive version of Binary search

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

- A description of something that refers to itself is called a *recursive* definition.
- Binary search is a kind of Divide-and-conquer algorithm.

Divide-and-Conquer

- *Divide* Step: the input is partitioned into p parts.
- *Conquer* Step: q recursive call(s) of the smaller parts.
- *Combine* Step: combine the solved parts to obtain the desired output.
- In binary search:
- Divide into two parts sized n/2, one recursive call of the smaller part.

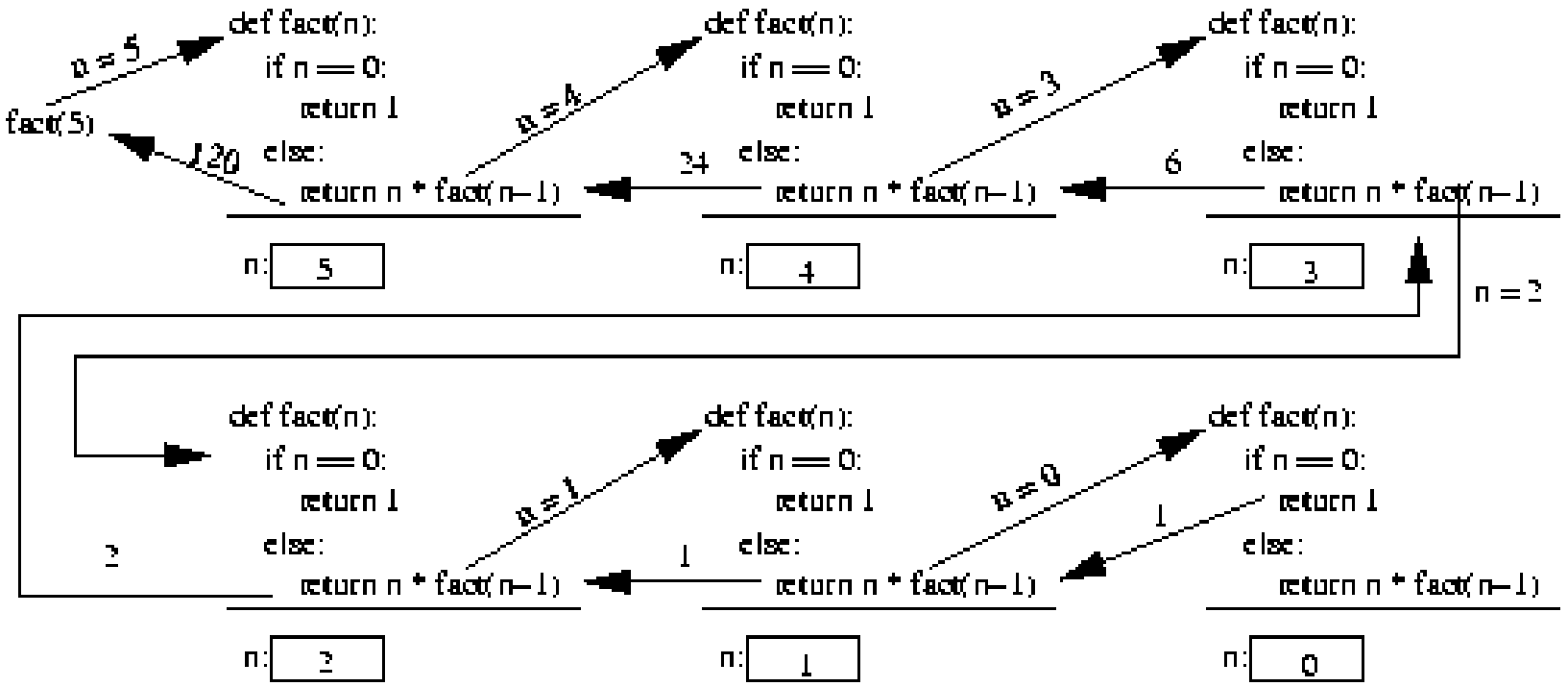
Recursion

- In mathematics, recursion is frequently used:
- n! = n* (n-1)!, 0! = 1
- fib(n) = fib(n-1) + fib(n-2), fib(1) = 1, fib(2) = 1
- Two key characteristics:
  - There are one or more base cases for which no recursion is applied.
  - All chains of recursion eventually end up at one of the base cases.

Factorial function

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

Factorial calling stack



Examples

- String reversal
  - reverse(“Hello“) -> “olleH”
- Anagrams of a string
  - anagram(“abc”) -> [“abc”,”acb”,…,”cba”]
- Fast exponentiation

Binary search

- ```
  def recBinSearch(x, nums, low, high):
      if low > high:
          return -1
      mid = (low + high)/2
      item = nums[mid]
      if item == x:
          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)
  ```

- ```
  def search(x, nums):
      return recBinSearch(x, nums, 0, len(nums)-1)
  ```

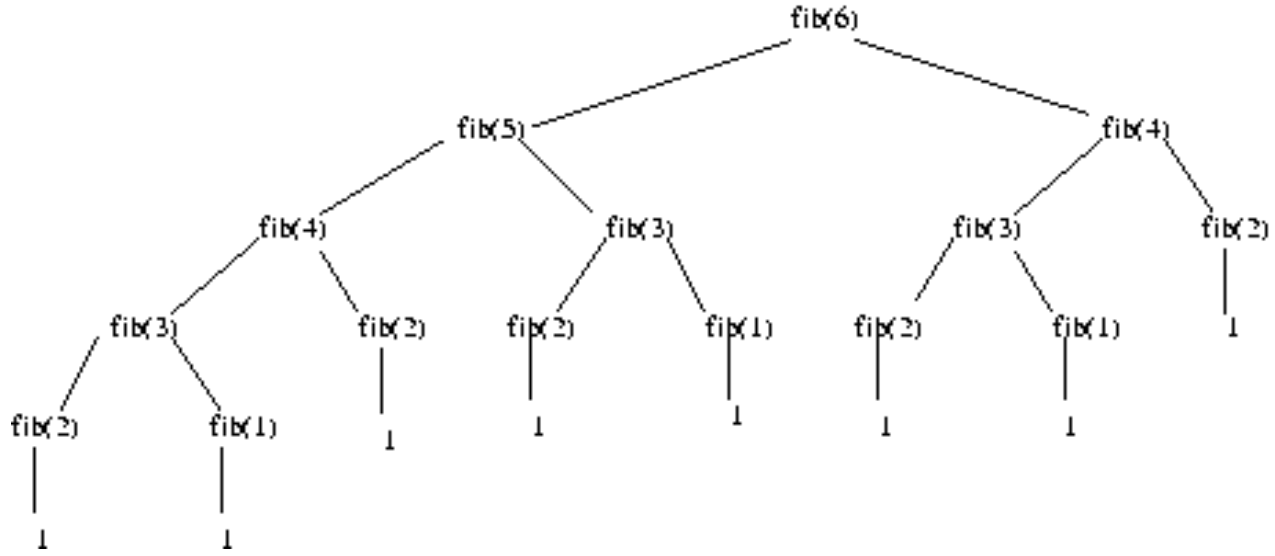
Recursion v.s. Iteration

- Anything that can be done with a loop can be done with a simple recursive function!
- Some problems that are simple to solve with recursion are quite difficult to solve with loops.
- Recursion is not always better than iterations. E.g., Fibonacci function:

$$fib(n) = \begin{cases} 1 & \text{if } n < 3 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

Recursion v.s. Iteration

- For Fibonacci number, the recursion version is very inefficient:



Lessons

- Recursion is another tool in your problem-solving toolbox.
- Sometimes recursion provides more elegant solution.
- At other times, when both algorithms are quite similar, use the looping solution on the basis of speed.
- Avoid the recursive solution if it is terribly inefficient, unless you can’t come up with an iterative solution.

Roadmap

- Searching problem
  - Linear and binary search
- Recursion and Divide-and-Conquer
- Sorting problem
  - Selection sort and merge sort
- Intractability and unsolvability

Built-in sort functions

- `sorted(lst[, key = func, reverse = True])`
  - `Key` can be anonymous functions, e.g.,

    ```
    key = lambda a: a.getName()
    ```
- `lst.sort([key= func, reverse = True])`

Naïve sorting

- Selection sort
- Loop n-1 times:
  - Find the smallest element, wap it to the correct position
- ```
  def selSort(nums):
      n = len(nums)
      for bottom in range(n-1):
          mp = bottom
          for i in range(bottom+1, n):
              if nums[i] < nums[mp]:
                  mp = i
          nums[bottom], nums[mp] = nums[mp], nums[bottom]
  ```
  ```

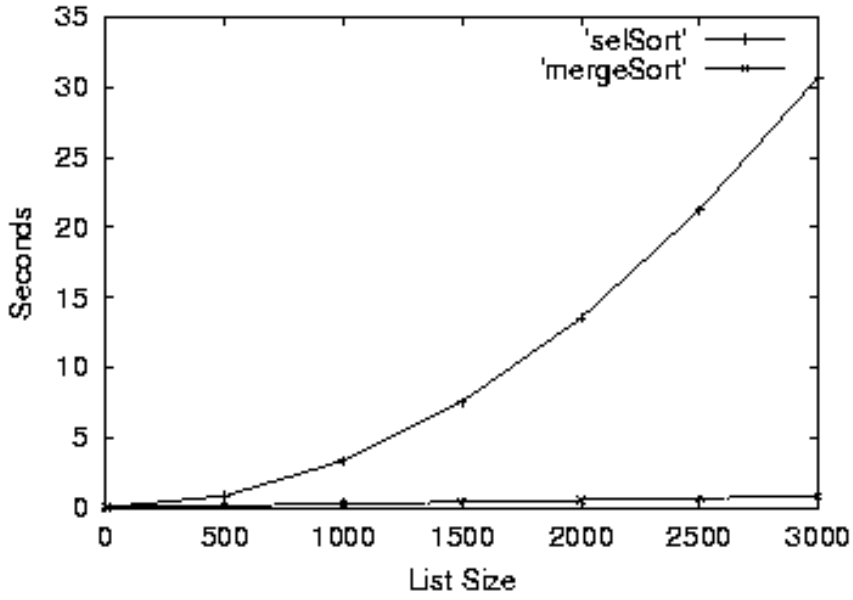
Divide-and-conquer

- Merge sort
  - Divide array into two halves.
  - *Recursively* sort each half.
  - Merge two halves.

- Assume we need to merge two sorted arrays, with M, N elements respectively. Then
  - How many comparisons in *best* case?
  - How many comparisons in *worst* case?

Comparing sorts

- http://www.sorting-algorithms.com/
- Selection sort:
  - $n + (n-1) + (n-2) + (n-3) + \ldots + 1$
- Merge sort:
  - n log n



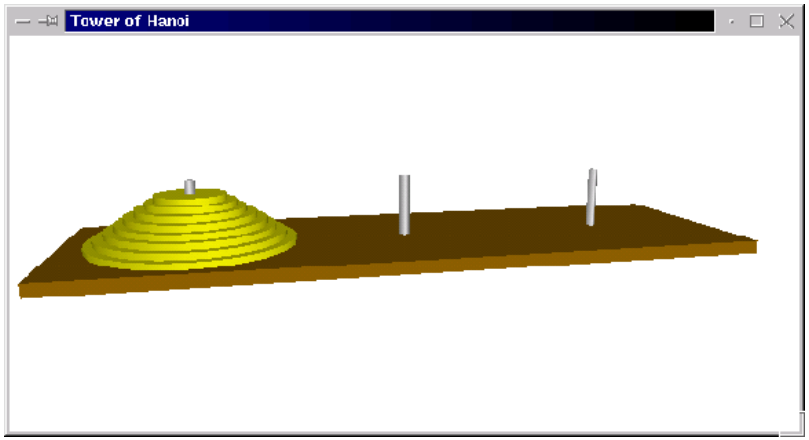
Hard Problems

- We have efficient algorithms for searching and sorting problems.
- Divide and conquer and recursion are very powerful techniques for algorithm design.
- Not all problems have efficient solutions!

Roadmap

- Searching problem
  - Linear and binary search
- Recursion and Divide-and-Conquer
- Sorting problem
  - Selection sort and merge sort
- Intractability and unsolvability

Towers of Hanoi




- ```
  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)
  ```

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

Analysis of Hanoi

- To solve a puzzle of size n will require 2^n-1 steps -- an exponential time algorithm.
- For 64 disks, moving one a second, round the clock, would require 580 *billion years* to complete (The current age of the universe is estimated to be about 15 billion years).
- It belongs to a class of problems known as *intractable* problems

Conclusion

- Recursion is a very important technique.
- Linear searching v.s. binary searching
- Naïve sort v.s. merge sort
- Good algorithms are better than super computers.
- Computer Science is more than programming!
- The most important computer for any computing professional is between their ears.