Principles of Computer Science II

Recursive Algorithms

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 6

Recursion Coding Style

Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition fulfils the condition of recursion, we call this function a recursive function.

Termination condition:

- A recursive function has to terminate to be used in a program.
- A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case.
- A base case is a case, where the problem can be solved without further recursion.

Factorial Computation: Using Iteration

```
def iterative_factorial(n):
    result = 1
    for i in range(2,n+1):
        result *= i
    return result
```

Factorial Computation: Using Recursion

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

Factorial Computation

```
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("intermediate result for ", n, " * factorial(" ,n-1, "):
        return res

print(factorial(5))
```

Fibonacci Numbers

The Fibonacci numbers are defined by:
$F_n = F_{n-1} + F_{n-2}$
where $F_0 = 0$ and $F_1 = 1$

- 0,1,1,2,3,5,8,13,21,34,55,89, ...

Factorial Computation: Using Recursion

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Factorial Computation: Using Iteration

```
def fibi(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

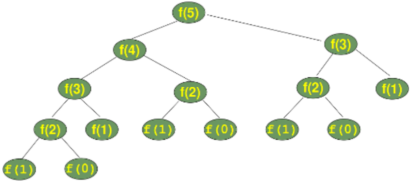
Measure Performance

```
from timeit import Timer
from fibo import fib

t1 = Timer("fib(10)","from fibo import fib")

for i in range(1,41):
        s = "fib(" + str(i) + ")"
        t1 = Timer(s,"from fibo import fib")
        time1 = t1.timeit(3)
        s = "fibi(" + str(i) + ")"
        t2 = Timer(s,"from fibo import fibi")
        time2 = t2.timeit(3)
        print("n=%2d, fib: %8.6f, fibi:  %7.6f, percent: %10.2f" % (i,
```

Fibonacci Numbers



Factorial Computation: Using Recursion and Memory

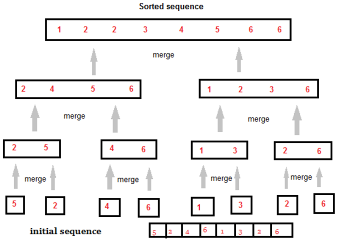
```
memo = {0:0, 1:1}
def fibm(n):
    if not n in memo:
        memo[n] = fibm(n-1) + fibm(n-2)
    return memo[n]
```

Merge Sort Algorithm

In Merge Sort the unsorted list is divided into N sublists, each having one element, because a list consisting of one element is always sorted. Then, it repeatedly merges these sublists, to produce new sorted sublists, and in the end, only one sorted list is produced.

- Divide and Conquer algorithm
- Performance always same for Worst, Average, Best case

Merge Sort: Example



Merge Sort Code

```
a = [25, 52, 37, 63, 14, 17, 8, 6]

def mergesort(list):
    if len(list) == 1:
        return list

    left = list[0: len(list) // 2]
    right = list[len(list) // 2:]

    left = mergesort(left)
    right = mergesort(right)

    return merge(left, right)
```

Merge Sort Code

```
def merge(left, right):
    result = []
    while len(left) > 0 and len(right) > 0:
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))

    while len(left) > 0:
        result.append(left.pop(0))

    while len(right) > 0:
        result.append(right.pop(0))

    return result

print("Before: ", a)
r = mergesort(a)
print("After: ", r)
```

How good is Merge Sort?

- How many comparisons are required until the list is sorted?
  - 1^{st} loop: two lists $\frac{n}{2}$ each
  - 2^{nd} loop: four lists $\frac{n}{4}$ each
  - ...
  - $\log n$ steps
  - For each partition we do n comparisons
  - In total $n \log n$ comparisons
- How much memory is needed ?
  - 1 additional slot.

Quick Sort Algorithm

Quick sort is very fast and requires very less additional space. It is based on the rule of Divide and Conquer. This algorithm divides the list into three main parts :

- Elements less than the Pivot element
- Pivot element(Central element)
- Elements greater than the pivot element

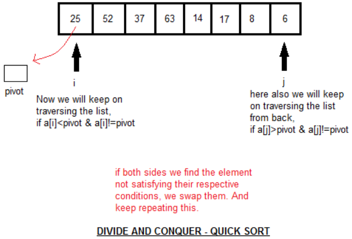
- Sorts any list very quickly
- Performance depends on the selection of the Pivot element

Quick Sort: Example

List: 25 52 37 63 14 17 8 6

- We pick 25 as the pivot.
- All the elements smaller to it on its left,
- All the elements larger than to its right.
- After the first pass the list looks like:
  6 8 17 14 25 63 37 52
- Now we sort two separate lists:
  6 8 17 14 and 63 37 52
- We apply the same logic, and we keep doing this until the complete list is sorted.

Quick Sort: Example



Quick Sort Code

```python
a = [25, 52, 37, 63, 14, 17, 8, 6]

def partition(list, p, r):
    pivot = list[p]
    i = p
    j = r
    while(1):
        while(list[i] < pivot and list[i] != pivot):
            i += 1

        while(list[j] > pivot and list[j] != pivot):
            j -= 1

        if(i < j):
            temp = list[i]
            list[i] = list[j]
            list[j] = temp
        else:
            return j
```

Quick Sort Code

```
def quicksort(list, p, r):
    if (p < r):
        q = partition(list, p, r)
        quicksort(list, p, q);
        quicksort(list, q + 1, r);

print("Before: ", a)
quicksort(a, 0, len(a) - 1)
print("After: ", a)
```

How good is Quick Sort?

- How many comparisons are required until the list is sorted?

How good is Quick Sort?

- How many comparisons are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?

How good is Quick Sort?

- How many comparisons are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 comparisons are required
  - $\Sigma \frac{n(n-1)}{2}$ comparisons are required

How good is Quick Sort?

- How many comparisons are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 comparisons are required
  - $\Sigma\frac{n(n-1)}{2}$ comparisons are required
- What if we choose the median item as pivot?

How good is Quick Sort?

- How many comparisons are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 comparisons are required
  - $\Sigma\frac{n(n-1)}{2}$ comparisons are required
- What if we choose the median item as pivot?
  - 1^{st} loop: two lists $\frac{n}{2}$ each
  - 2^{nd} loop: four lists $\frac{n}{4}$ each
  - ...
  - $\log n$ steps
  - For each partition we do n comparisons
  - In total $n \log n$ comparisons

How good is Quick Sort?

- How many comparisons are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 comparisons are required
  - $\Sigma\frac{n(n-1)}{2}$ comparisons are required
- What if we choose the median item as pivot?
  - 1^{st} loop: two lists $\frac{n}{2}$ each
  - 2^{nd} loop: four lists $\frac{n}{4}$ each
  - ...
  - $\log n$ steps
  - For each partition we do n comparisons
  - In total $n \log n$ comparisons
- How much memory is needed ?

How good is Quick Sort?

- How many comparisons are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 comparisons are required
  - $\Sigma\frac{n(n-1)}{2}$ comparisons are required
- What if we choose the median item as pivot?
  - 1^{st} loop: two lists $\frac{n}{2}$ each
  - 2^{nd} loop: four lists $\frac{n}{4}$ each
  - ...
  - $\log n$ steps
  - For each partition we do n comparisons
  - In total $n \log n$ comparisons
- How much memory is needed ?
  - 1 additional slot.