Ceng 111 – Fall 2015
Week 12a

Iteration and Complexity

Credit: Some slides are from the “Invitation to Computer Science” book by G. M. Schneider, J. L. Gersting and some from the “Digital Design” book by M. M. Mano and M. D. Ciletti.
Nested Loops in Python

- You can put one loop within another one
  - No limit on nesting level

```python
for i in range(1,10):
    print i, "::",
    for j in range(1,i):
        print j, "-",
    print ""
```

1:
2: 1 –
3: 1 – 2 –
4: 1 – 2 – 3 –
5: 1 – 2 – 3 – 4 –
6: 1 – 2 – 3 – 4 – 5 –
7: 1 – 2 – 3 – 4 – 5 – 6 –
8: 1 – 2 – 3 – 4 – 5 – 6 – 7 –
9: 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 –
Break statements

- While loop:
  ```
  while <cond-1>:
    <statements-before-break>
    if <cond-2>:
      break
    <statements-after-break>
  <statements-after-while>
  ```

- For loop:
  ```
  for <var> in <list>:
    <statements-before-break>
    if <cond>:
      break
    <statements-after-break>
  <statements-after-for>
  ```
"break" example

```python
1 x = 4
2 List = [1, 4, -2, 3, 8]
3 for m in List:
4     print m
5     if m == x:
6         print "I have found a match"
7         break
```
Continue statements

while <cond-1>:
    <statements-before-continue>
    if <cond-2>:
        continue
    <statements-after-continue>
    <statements-after-while>

for <var> in <list>:
    <statements-before-continue>
    if <cond>:
        continue
    <statements-after-continue>
    <statements-after-for>

- <var> will point to the next item in the list.
Loops with “else:” parts

- The “else:” part is executed when the loop exits.
- If you use a “break” statement, the “else” part is not executed.

```python
while <cond>:
    <statements>
else:
    <else-statements>
```
Examples for Iteration

What does the following do?

def f(List):
    length = len(List)
    changed = 1
    while changed:
        changed = 0
        i = 0
        while i < length-1:
            if List[i] > List[i+1]:
                (List[i], List[i+1]) = (List[i+1], List[i])
                changed = 1
            i += 1
Counting sort

\[
\begin{align*}
&\text{for } i \leftarrow 1 \text{ to } k \\
&\quad \text{do } C[i] \leftarrow 0 \\
&\text{for } j \leftarrow 1 \text{ to } n \\
&\quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1 & \triangleright C[i] = |\{\text{key} = i\}| \\
&\text{for } i \leftarrow 2 \text{ to } k \\
&\quad \text{do } C[i] \leftarrow C[i] + C[i-1] & \triangleright C[i] = |\{\text{key} \leq i\}| \\
&\text{for } j \leftarrow n \text{ downto } 1 \\
&\quad \text{do } B[C[A[j]]] \leftarrow A[j] \\
&\quad \quad C[A[j]] \leftarrow C[A[j]] - 1
\end{align*}
\]
def csort(A):
    # Assume that the numbers are in the range 1,...,k
    k = max(A)
    C = [0] * k

    # Count the numbers in A
    for x in A:
        C[x-1] += 1

    # Accumulate the counts in C
    i = 1
    while i < k:
        C[i] += C[i-1]
        i += 1

    # Place the numbers into correct locations
    B = [0] * len(A)
    for x in A:
        B[C[x-1]-1] = x
        C[x-1] -= 1

    return B
Today

- Finish up Iteration
- Complexity
Another Example for Iteration

- What happens when the circled part is written as follows:

  \[
  \text{Result} = \text{List}
  \]
TAIL RECURSION VS. RECURSION VS. ITERATION
Consider these two implementations:

- The second implementation uses "tail recursion".
- tail recursion $\rightarrow$ the result of the called function is not used by the calling function.

```python
def fact1(n):
    if n == 0:
        return 1
    else:
        return n * fact1(n-1)

def fact2(n):
    fact_helper(n, 1)

def fact_helper(n, r)
    if n == 0:
        print r
    else:
        fact_helper(n-1, n*r)
```
Tail recursion & iteration

Then, we can implement the tail-recursion version like on the right.

```python
def fact2(n):
    fact_helper(n, 1)

def fact_helper(n, r):
    if n == 0:
        print(r)
    else:
        fact_helper(n-1, n*r)
```

```python
while n != 0:
    r = n * r
    n = n - 1
```
More properly,

\[ r = 1 \]
\[ \text{while } n \neq 0 \]
\[ r = n \times r \]
\[ n = n - 1 \]

```python
def fact2(n):
    fact_helper(n, 1)

def fact_helper(n, r):
    if n == 0:
        print(r)
    else:
        fact_helper(n-1, n*r)
```
Recursion vs. Iteration

- Any recursive algorithm can be transformed into an iterative algorithm.
- The reverse is also true.
Recursion vs. Iteration

- Which one is better?
- Better in what way?
  - Resource-wise:
    1. Iteration without stacks
    2. Iteration with stacks
    3. Recursion
  - Implementation-wise:
    1. Recursion
    2. Iteration without stack
    3. Iteration with stack
TIME ANALYSIS OF ALGORITHMS
How do you compare these two algorithms?

**Bubble sort**

```python
def f(List):
    length = len(List)
    changed = 1
    while changed:
        changed = 0
        i = 0
        while i < length - 1:
            if List[i] > List[i+1]:
                (List[i], List[i+1]) = (List[i+1], List[i])
                changed = 1
            i += 1
```

**Count sort**

```python
def csort(A):
    # Assume that the numbers are in the range 1,...,k
    k = max(A)
    C = [0] * k
    # Count the numbers in A
    for x in A:
        C[x-1] += 1
    # Accumulate the counts in C
    i = 1
    while i < k:
        C[i] += C[i-1]
        i += 1
    # Place the numbers into correct locations
    B = [0] * len(A)
    for x in A:
        B[C[x-1]-1] = x
        C[x-1] -= 1
    return B
```
Analyzing Performance of Algorithms

How do you compare the performance of algorithms?

1. Implement them and count the time they take?
2. Count the number of main steps that affect the performance and depend on the size of the data.
Analyzing Algorithms

Consider these two algorithms for searching in a list:

1. Binary search vs Sequential search

   - Binary search: \( \log N \) comparisons
   - Sequential search: \( N \) comparisons
There are several measures for complexity.

A measure for complexity is basically a bound for the running time of an algorithm.

Look at $f(n) = 2n^2$

$f(n)$ is bounded by $3n^2$
Measuring complexity

- Consider again $f(n) = 2n^2$
- There are several functions that can bound $f(n)$:
  1. $3n^2$, $4n^2$, $6n^2$, ...
  2. $n^3$, $2n^3$, $3n^3$, ...
  3. $n^4$, $2n^4$, $3n^4$, ...
  4. ...
  5. ...

- In computational complexity, we are interested in the most “suitable” bounding function.
Big-o Notation; O()

- A function f(n) is O(g(n)) iff
  - \(|f(n)| \leq M|g(n)|\) for sufficiently large n, for a constant number M > 0.

- So, for f(n) = 2n^2 (or, f(n) \(\epsilon\) 2n^2), g(n) = n^2.
  - In other words: f(n) = O(n^2)
  - But, it is also O(n^3) and O(n^4)
  - We prefer the smallest.
  - For example:
    \[ f(n) = 9\log n + 5(\log n)^3 + 3n^2 + 2n^3 \in O(n^3). \]
<table>
<thead>
<tr>
<th>Notation</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant</td>
<td>Determining if a number is even or odd; using a constant-size lookup table or hash table</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>logarithmic</td>
<td>Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap.</td>
</tr>
<tr>
<td>$O(n^c), 0 &lt; c &lt; 1$</td>
<td>fractional power</td>
<td>Searching in a kd-tree</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>linear</td>
<td>Finding an item in an unsorted list or a malformed tree (worst case) or in an unsorted array; Adding two n-bit integers by ripple carry.</td>
</tr>
<tr>
<td>$O(n \log n) = O(\log n!)$</td>
<td>linear arithmetic, loglinear, or quasilinear</td>
<td>Performing a Fast Fourier transform; heapsort, quicksort (best and average case), or merge sort</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>quadratic</td>
<td>Multiplying two $n$-digit numbers by a simple algorithm; bubble sort (worst case or naive implementation), shell sort, quicksort (worst case), selection sort or insertion sort</td>
</tr>
<tr>
<td>$O(n^c), c &gt; 1$</td>
<td>polynomial or algebraic</td>
<td>Tree-adjoining grammar parsing; maximum matching for bipartite graphs</td>
</tr>
<tr>
<td>$L_n[\alpha, c], 0 &lt; \alpha &lt; 1 = \frac{e^{(c+o(1))}(\ln n)^{\alpha} (\ln \ln n)^{1-\alpha}}{e^{c+o(1)}(\ln n)^{\alpha}}$</td>
<td>L-notation or sub-exponential</td>
<td></td>
</tr>
<tr>
<td>$O(c^n), c &gt; 1$</td>
<td>exponential</td>
<td>Finding the (exact) solution to the traveling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search</td>
</tr>
<tr>
<td>$O(n!)$</td>
<td>factorial</td>
<td>Solving the traveling salesman problem via brute-force search, finding the determinant with expansion by minors.</td>
</tr>
</tbody>
</table>
Other notations for computational complexity

- **Ω notation: f(n) = Ω(g(n))**: 
  - Lower boundary for f(n).
  - f(n) is Ω(g(n)) iff |f(n)| ≥ M|g(n)|, for sufficiently large n, for some constant number M > 0.
  - 2n = Ω(n)
  - n² = Ω(n²)
Other notations for computational complexity

- $\Theta$ notation: $f(n) = \Theta(g(n))$:
  - Lower and upper boundary for $f(n)$.
  - $f(n)$ is $\Theta(g(n))$ iff $f(n)$ is bounded by below and above by $g(n)$:
    - $M|g(n)| \geq |f(n)| \geq N|g(n)|$, for sufficiently large $n$, for some constant numbers $M, N > 0$.
  - $2n = \Theta(n)$
  - $n^2 = \Theta(n^2)$
Importance of Developing Efficient Algorithms

Sequential search vs Binary search

<table>
<thead>
<tr>
<th>Array size</th>
<th>No. of comparisons by seq. search</th>
<th>No. of comparisons by bin. search</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>128</td>
<td>8</td>
</tr>
<tr>
<td>1,048,576</td>
<td>1,048,576</td>
<td>21</td>
</tr>
<tr>
<td>~4.10⁹</td>
<td>~4.10⁹</td>
<td>33</td>
</tr>
</tbody>
</table>

Execution times for algorithms with the given time complexities:

<table>
<thead>
<tr>
<th>n</th>
<th>f(n)=n</th>
<th>nlg(n)</th>
<th>n²</th>
<th>2ⁿ</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.02 µs</td>
<td>0.086 µs</td>
<td>0.4 µs</td>
<td>1 ms</td>
</tr>
<tr>
<td>10⁶</td>
<td>1 µs</td>
<td>19.93 ms</td>
<td>16.7 min</td>
<td>31.7 years</td>
</tr>
<tr>
<td>10⁹</td>
<td>1 s</td>
<td>29.9 s</td>
<td>31.7 years</td>
<td>!!! centuries</td>
</tr>
</tbody>
</table>

Anaysis of Algorihms, A.Yazici