Overview

• Asymptotic notation
• Insertion sort
• Divide and conquer: merge sort
Objective

- Understand that efficiency is important
- Learn how to determine algorithm efficiency
- Get familiar with sorting algorithms

Performance is Important

- Algorithm might run on very large data set
- Be efficient in terms of CPU and memory usage

1. Look at sorting algorithms of different efficiency
2. Learn how efficiency of algorithm can be determined
Example

- Imagine you would have to sort an arbitrary set of numbers, e.g., student IDs
- How would you go about this?
- Are there more or less efficient approaches?

Asymptotic Analysis of Algorithms

- Algorithm complexity
- Asymptotic analysis
- Practical use
- Code examples
Algorithm Complexity

• Need general method for describing complexity

Asymptotic Analysis

• “Big O” notation extracts essence of algorithm performance
  • Defines an upper boundary on complexity growth
  • Definition: \( f(x) = O(g(x)) \) for \( x \rightarrow \infty \)
    if and only if there is a positive real number \( m \) and a real number \( x_0 \) such that
    \( f(x) \leq m \cdot g(x) \) for all \( x > x_0 \)
      • For all \( x \) beyond \( x_0 \), \( f(x) \) is bounded by \( m \cdot g(x) \)
Asymptotic analysis example

• What are \( g(x) \), \( m \) and \( x_0 \) for our linear search?

\[ m = 0.35, \ x_0 = 5000 \] (one of many solution)
Asymptotic analysis example

• What are g(x), m and x₀ for our binary search?

• m = 26, x₀ = 1000 (one of many solutions)
Practical use of asymptotic analysis

- Constants and lower degrees are ignored
  - Example: $n/2$ is $O(n)$; $3n^2 + 15n$ is $O(n^2)$

- Typical classes of complexity
  - $O(1)$: constant
  - $O(\log n)$: logarithmic growth
  - $O(n)$: linear growth
  - $O(n \log n)$: linearithmic (or loglinear) growth
  - $O(n^2)$: quadratic growth
  - $O(2^n)$: exponential growth

Comparison of complexity classes

- Significant differences in trends
  - Calculate different example values for $n=10$

![Graph showing comparison of complexity classes](image)
Examples

• What are complexity bounds for these functions?
  • Find tightest upper bound

• Examples
  • $0.000001 \times n^2 + 15000 \times n$ = $O(n^2)$
  • $n^2 \times n + 10 \times n^2 \log n$ = $O(n^3)$
  • $12345 + \log 54321$ = $O(1)$
  • $(n + \log n)^2$ = $O(n^2)$
  • $n(5 + \log n)$ = $O(n \log n)$
  • $1 + 2 + 3 + \ldots + n$ = $O(n)$
• What is the running time complexity of the following code example?

```python
for i in range(n):
    //do something
```

• What is the running time complexity of the following code example?

```python
for i in range(n):
    for j in range(n):
        //do something
```
Code examples

- What is the running time complexity of the following code example?

```python
for i in range(n):
    for j in range(i, n):
        #do something
```

- What is the running time complexity of the following code example?

```python
for i in range(n/10):
    for j in range(i):
        #do something
```
Code examples

- What is the running time complexity of the following code example?

```python
for i in range(0, n, i=i*2):
    for j in range(i):
        //do something
```

Insertion Sort

- Sorting is required in many applications
- Examples?
- Idea of insertion sort:
  - Insert next element into partially sorted array
  - Iterate
  - Insertion requires shifting of elements
Insertion Sort

Assume 54 is a sorted list of 1 item

inserted 26

inserted 93

inserted 17

inserted 77

inserted 31

inserted 44

inserted 55

inserted 20

Need to insert 31 back into the sorted list

54>31 so shift it to the right

77>31 so shift it to the right

54<31 so shift it to the right

26<31 so insert 31 in this position
Insertion Sort

- Write your own code

```python
def insertionSort(alist):
    for index in range(1, len(alist)):
        //do actual sorting

alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
insertionSort(alist)
print(alist)
```

```python
def insertionSort(alist):
    for index in range(1, len(alist)):
        currentvalue = alist[index]
        position = index

        while position>0 and alist[position-1]>currentvalue:
            alist[position]=alist[position-1]
            position = position-1

        alist[position]=currentvalue

alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
insertionSort(alist)
print(alist)
```
Insertion Sort – Analysis

- $n-1$ passes to sort $n$ item => $O(n^2)$
- In the best case (already sorted list), only one comparison needed
- In general, shift operation requires $3^{rd}$ of the exchange operation

Growth of processing time

- Algorithms with $O(n^2)$ complexity
  - 2x problem size, 4x running time

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>n=5000</th>
<th>n=10000</th>
<th>factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble sort</td>
<td>22.08</td>
<td>104.39</td>
<td>4.73</td>
</tr>
<tr>
<td>selection sort</td>
<td>32.35</td>
<td>128.70</td>
<td>3.98</td>
</tr>
<tr>
<td>insertion sort</td>
<td>3.76</td>
<td>14.28</td>
<td>3.80</td>
</tr>
</tbody>
</table>
**Comparison**

![Comparison Graph]

**Merge Sort**

- Divide and conquer to improve performance
- Recursive algorithm
  - Continually splits list in half
    - a) List is empty or has one item => sorted by definition
    - b) List has more than one item => split and recursively involve merge sort
- **Merge**: taking two smaller lists and combining them together
Merge Sort: Split

![Merge Sort: Split Diagram]

Merge Sort: Merge

![Merge Sort: Merge Diagram]
def mergeSort(alist):
    print("Splitting ", alist)
    if len(alist)>1:
        mid = len(alist)/2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]
        mergeSort(lefthalf)
        mergeSort(righthalf)
    i=0
    j=0
    k=0
    while i < len(lefthalf) and j < len(righthalf):
        if lefthalf[i] < righthalf[j]:
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1
        else:
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    while i < len(lefthalf):
        alist[k]=lefthalf[i]
        i=i+1
        k=k+1
    while j < len(righthalf):
        alist[k]=righthalf[j]
        j=j+1
        k=k+1
    print("Merging ",alist)

alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
mergeSort(alist)
print(alist)
Merge Sort: Merge

```
i=0
j=0
k=0
while i < len(lefthalf) and j < len(righthalf):
    if lefthalf[i] < righthalf[j]:
        alist[k]=lefthalf[i]
        i=i+1
    else:
        alist[k]=righthalf[j]
        j=j+1
    k=k+1
while i < len(lefthalf):
    alist[k]=lefthalf[i]
    i=i+1
    k=k+1
while j < len(righthalf):
    alist[k]=righthalf[j]
    j=j+1
    k=k+1
print("Merging ",alist)
```

Merge Sort – Analysis

- Split: divide a list in half log $n$ times ($n =$ length of list)
- Merge: Each item processed and placed on sorted list => $n$ operations.
- $O(n \log n)$
- NOTE:
  - function requires extra space to hold the two halves
  - additional space a critical factor if list is large (e.g., working on large data sets)
Next Steps

• Next lecture Thursday
• HW1 due on Thursday