COMP1927 16x1
Computing 2

Complexity
Problems, Algorithms, Programs and Processes

- **Problem**: A problem that needs to be solved
- **Algorithm**: Well defined instructions for completing the problem
- **Program**: Implementation of the algorithm in a particular programming language
- **Process**: An instance of the program as it is being executed on a particular machine
Analysis of software

• What makes software "good"?
  • returns expected result for all valid inputs
  • behaves "sensibly" for non-valid inputs
  • clear code, easy to maintain/modify
  • interface is clear and consistent (API or GUI)
  • returns results quickly (even for large inputs)
  • It is efficient

• We may sometimes also be interested in other measures
  • memory/disk space, network traffic, disk IO etc
Algorithm Efficiency

• The algorithm is by far the most important determinant of the efficiency of a program

• Small speed ups in terms of operating systems, compilers, computers and implementation details are irrelevant

• May give small speed ups but usually only by a small constant factor
Determining Algorithm Efficiency

- At the design stage
  - Theoretical approach
  - Complexity theory

- After the testing stage
  - Once it is implemented and correct you can empirically evaluate performance eg using the time command
Timing

• Note we are not interested in the absolute time it takes to run.
• We are interested in the relative time it takes as the problem increases
• Absolute times differ on different machines and with different languages
Complexity Theory Example

1. int linearSearch(int a[], int n, int key){
2.    for indexes from 0 to n-1
3.       if key equals current element array
4.          return current index
5.    return -1
6.}

• What is the worst case cost?
  ◦ When does this occur?
  ◦ How many comparisons between data instances were made?
Complexity Example

• How many times does each line run in the worst case?

C0: line 2: For loop $n+1$ times
C1: line 3: $n$ comparisons
C2: line 4: 0 times (worst case)
C3: line 5: 1 time (worst case)

Total: $C0(n+1) + C1(n) + C3 = O(n)$

• For an unsorted sequence that is the best we can do
Informal Definition of Big-O Notation

- We express complexity using **big-O notation**
- Represents the asymptotic **worst case** (unless stated otherwise) time complexity
- Big-O expressions do not have constants or low-order terms as when \( n \) gets larger these do not matter
- For example: For a problem of size \( n \), if the cost of the worst case is
  - \( 1.5n^2 + 3n + 10 \)
  - in Big-O notation would be \( O(n^2) \)
Big O-notation Formal Definition

The big O-notation is used to classify the work complexity of algorithms.

Definition: A function $f(n)$ is said to be in (the set) $O(g(n))$ if there exist constants $c$ and $N_0$ such that $f(n) < c \cdot g(n)$ for all $n > N_0$. 
Empirical Analysis Linear Search

- Use the ‘time’ command in Linux.

Run on different sized inputs

time ./prog < input > /dev/null

not interested in real-time
interested in user-time

What is the relationship between
- input size
- time

<table>
<thead>
<tr>
<th>Size of input(n)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>100000000</td>
<td></td>
</tr>
<tr>
<td>1000000000</td>
<td></td>
</tr>
</tbody>
</table>
Predicting Time

• If I know my algorithm is quadratic and I know that it takes 1.2 seconds to run on a data set of size 1000
• Approximately how long would you expect to wait for a data set of size 2000?
• What about 10000?
• What about 100000?
• What about 1000000?
• What about 10000000?
Searching in a Sorted Array

• Given an array a of N elements, with \( a[i] \leq a[j] \) for any pair of indices \( i,j \), with \( i \leq j < N \),

• search for an element e in the array

```c
int a[N];         // array with N items
int found = 0;
int i = 0;

while ((i < N) && (!found)){
    found = (a[i] == e);
    i++;
}
```
Searching in a Sorted Array

• Given an array $a$ of $N$ elements, with $a[i] \leq a[j]$ for any pair of indices $i,j$, with $i \leq j < N$,

• search for an element $e$ in the array

```c
int a[N];         // array with $N$ items
int found = 0;
int finished = 0;
int i = 0;
while ((i < N) && (!found) && (!finished)) {
    found = (a[i] == e);    // exploit the fact that $a$ is sorted
    finished = (e < a[i]);
    i++;
}
```
Searching in a Sorted Array

• How many steps are required to search an array of $N$ elements

Best case: $T_N = 1$

Worst case: $T_N = N$

Average: $T_N = N/2$

• Still a linear algorithm, like searching in a unsorted array
Binary Search

• We start in the middle of the array:
  • if \(a[N/2] == e\), we found the element and we’re done
  • and, if necessary, `split’ array in half to continue search
  • if \(a[N/2] < e\), continue search on \(a[0] \) to \(a[N/2 -1]\)
  • if \(a[N/2] > e\), continue search on \(a[N/2+1] \) to \(a[N-1]\)
• This algorithm is called binary search.
Binary Search

See binary.c for implementation

- We maintain two indices, l and r, to denote leftmost and rightmost array index of current part of the array
  - initially l=0 and r=N-1
- iteration stops when:
  - left and right index define an empty array, element not found
  - Eg l > r
  - a[(l+r)/2] holds the element we’re looking for
- if: a[(l+r)/2] is larger than element, continue search on left
  a[l]..a[(l+r)/2-1]
else continue search on right
  a[(l+r)/2+1]..a[r]
Binary Search

- How many comparisons do we need for an array of size $N$?
  - **Best case:**
    - $T_N = 1$
  - **Worst case:**
    - $T_1 = 1$
    - $T_N = 1 + T_{N/2}$
    - $T_N = \log_2 N + 1$
    - $O(\log n)$

- Binary search is a logarithmic algorithm
Big-O Notation

• All constant functions are in $O(1)$
• All linear functions are in $O(n)$
• All logarithmic function are in the same class $O(\log(n))$
  • $O(\log_2(n)) = O(\log_3(n))$ = ....
  • (since $\log_b(a) \cdot \log_a(n) = \log_b(n)$)
• We say an algorithm is $O(g(n))$ if, for an input of size $n$, the algorithm requires $T(n)$ steps, with $T(n)$ in $O(g(n))$, and $O(g(n))$ minimal
  • binary search is $O(\log(n))$
  • linear search is $O(n)$
• We say a problem is $O(g(n))$ if the best algorithm is $O(g(n))$
  • finding the maximum in an unsorted sequence is $O(n)$
Common Categories

- **$O(1)$**: constant - instructions in the program are executed a fixed number of times, independent of the size of the input
- **$O(\log N)$**: logarithmic - some divide & conquer algorithms with trivial splitting and combining operations
- **$O(N)$**: linear - every element of the input has to be processed, usually in a straightforward way
- **$O(N \log N)$**: Divide & Conquer algorithms where splitting or combining operation is proportional to the input
- **$O(N^2)$**: quadratic. Algorithms which have to compare each input value with every other input value. Problematic for large input
- **$O(N^3)$**: cubic, only feasible for very small problem sizes
- **$O(2^N)$**: exponential, of almost no practical use
# Complexity Matters

<table>
<thead>
<tr>
<th>n</th>
<th>log n</th>
<th>nlogn</th>
<th>n^2</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
<td>40</td>
<td>100</td>
<td>1024</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
<td>700</td>
<td>10000</td>
<td>1.3E+30</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>10000</td>
<td>100000</td>
<td>REALLY BIG</td>
</tr>
<tr>
<td>10000</td>
<td>14</td>
<td>140000</td>
<td>10000000</td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>17</td>
<td>1700000</td>
<td>10000000000</td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>20</td>
<td>20000000</td>
<td>10000000000000</td>
<td></td>
</tr>
</tbody>
</table>
Exercise

What would be the time complexity of inserting an element at the beginning of

• a linked list
• an array

What would be the time complexity of inserting an element at the end of

• a linked list
• an array