COMP1927 17x1
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 “good” software?

- **Correct**: returns expected result for all valid inputs guaranteed through formal specification
- **Reliable**: behaves "sensibly" for non-valid inputs/errors and handled gracefully
  Correctness/Reliability ensured through robust testing
- **Maintainable**: clear, well-structure code
  Coding style, recommended conventions
- **Efficient**: produces results quickly (even for large inputs)
  Efficiency determined through algorithm efficiency

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

• Algorithm efficiency determined through **algorithm analysis**, can save factors of thousands or millions in the running time

• Small speed ups in terms of operating systems, compilers, computers and implementation details are irrelevant. They may give small speed ups but usually only by a small constant factor
Algorithm Analysis

Branch of computer science to determine choice of the best algorithm for a particular task.

- **Mathematical Analysis**
  - Analyse asymptotic time complexity – the limiting behaviour of the execution time of an algorithm when the size of the problem goes to infinity
  - Usually denoted in big-O notation.
  - Can be done at design-stage (pseudo-code)

- **Empirical Analysis**
  - Post-implementation stage
  - Once it is implemented and correct, evaluate which algorithm takes longer e.g., 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
Time Complexity Analysis

- Enables us to understand the performance of algorithms
- Define a function to characterize execution cost ($\approx \text{time}$)
  - Identify the core operation in the algorithm
  - Identify the value to measure the size of the input ($N$) (e.g. #items in data structure, length of input file, no of chars in string etc)
  - Express cost in terms of #operations = $f(n)$, which is the time-complexity as a function of input size
- Shows how the cost increases with increase in input size
- Is the algorithm feasible for 100, 10000, 100000?
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 $f(n) < c \times g(n)$ for all $n > N_0$.
Informal Definition of Big-O Notation

• **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 \( O(n^2) \)
Exercise: Time Complexity

**Example:** finding max value in an *unsorted array*

```c
int findMax(int a[], int N) {
    int i, max = a[0];
    for (i = 1; i < N; i++)
        if (a[i] > max) max = a[i]; return max;
}
```

Core operation? ... compare \( a[i] \) to \( \text{max} \)
How many times? ... clearly \( N-1 \) ... \( O(n) \)
Execution cost grows *linearly* (i.e. \( 2 \times \#\text{elements} \Rightarrow 2 \times \text{cost} \))
Exercise: Time Complexity

**Example:** finding max value in a sorted array

```c
int findMax(int a[], int N) {
    return a[N-1];
}
```

No iteration needed; max is always last.
Core operation? ... index into array
How many times? ... once ... $O(1)$
Execution cost is constant (same regardless of #elements)
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?
Empirical Analysis

- 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>1000000</td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td></td>
</tr>
<tr>
<td>1000000000</td>
<td></td>
</tr>
<tr>
<td>10000000000</td>
<td></td>
</tr>
</tbody>
</table>
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 $O(\log(n))$
  • $O(\log_2(n)) = O(\log_3(n)) = \ldots$
    • (since $\log_b(a) \times \log_a(n) = \log_b(n)$)
• 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 $O(N)$
  • finding 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>1000000</td>
<td>REALLY BIG</td>
</tr>
<tr>
<td>10000</td>
<td>14</td>
<td>140000</td>
<td>100000000</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
Searching

An extremely common application in computing

- given a (large) collection of **items** and a **key** value
- find the item(s) in the collection containing that key
- an item is defined as \{key, val_1, val_2\} (i.e. a struct)
- key = value used to distinguish items (e.g. student ID)

Keys may be ...

- primary ... key value uniquely identifies one item
- secondary ... many items may have the same key value

Applications: Google, databases, .....
If we are dealing largely with primary keys, then search problem can be encapsulated as:

```c
Item search(Collection c, Key k) { ... }
```

Possible return values are:

- an `Item`
- “NOT FOUND” value

For secondary keys return an array (possibly empty) of matching items

```c
Item *search(Collection c, key k, int *nmatches) {...}
```
Searching in Linear Structures

Search in an unsorted array or list

```c
Item searchArray(Key k, Item a[], int n) {
    int i;
    for (i = 0; i < n; i++)
        { if (a[i].key == k)
            return a[i];
        }
    return NOT_FOUND;
}

Item searchList(Key k, List L) {
    List n;
    for (n = L; n != NULL; n = n->next)
        { if (n->key) == k)
            return n->data;
        }
    return NOT_FOUND;
}
```
Linear Search Cost

- Core operation? ... compare \(a[i].key\) to \(k\)
- What is the worst cast cost?
- How many comparisons between data instances were made?
- 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
Searching in a Sorted Array

Given an array a of N elements, with a[i] <= a[j] for any pair of indices i,j, with i <= j < N,

- search for a key k in the array

```c
Item searchSortedArray(Key k, Item a[], int n) {
    int found = 0;
    int finished = 0;
    int i = 0;
    while ((i < N) && (!found) && (!finished)){
        found = (a[i].key == k);
        finished = (k < a[i].key);
        i++;
    }
}
```

-exploit the fact that a is sorted
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 in a sorted array

• 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 (cont)

• 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:
    - $T_1 = 1$
    - $T_N = T_{N/2} + 1$
    - $T_N = \log_2 N + 1$
    - $O(\log n)$

Binary search is a logarithmic algorithm.