Motivation
Efficiency
Empirical Analysis
Theoretical Analysis
Binary Search
Exercise

COMP2521 23T3
Analysis of Algorithms

Kevin Luxa
cs2521@cse.unsw.edu.au
• Program runtime is critical for many applications:
  • Finance, robotics, games, database systems, ...

• We may want to compare programs to decide which one to use

• We may want to determine whether a program will be "fast enough"
What determines how fast a program runs?

- The operating system?
- Compilers?
- Hardware?
  - E.g., CPU, GPU, cache
- Load on the machine?
- Most important: the data structures and algorithms used
Algorithm Efficiency

- The running time of an algorithm tends to be a function of input size
- Typically: larger input $\Rightarrow$ longer running time
  - Small inputs: fast running time, regardless of algorithm
  - Larger inputs: slower, but how much slower?

![Graph showing running time vs. input size](image)
What to Analyse?

- **Best-case performance**
  - Not very useful
  - Usually only occurs for specific types of input
- **Average-case performance**
  - Difficult; need to know how the program is used
- **Worst-case performance**
  - Most important; determines how long the program could possibly run
Efficiency of an algorithm can be investigated in two ways:

- Empirically: Measuring the time that a program implementing the algorithm takes to run
- Theoretically: Counting the number of basic operations performed by the algorithm as a function of input size
Write a program that implements the algorithm
2. Run the program with inputs of varying size and composition
3. Measure the runtime
4. Plot the results
We can measure running time of a program using the `time` command.

The `time` command produces three times:

- **real**: total elapsed time
- **user**: CPU time spent executing program code
- **sys**: CPU time spent by the operating system on behalf of the program
  - e.g., opening a file

Example:

```
$ time ./prog
real   0m0.440s
user   0m0.380s
sys    0m0.000s
```
Absolute times will differ between machines, between languages...so we’re not interested in absolute time. We are interested in the *relative* change as the input size increases.
Let’s empirically analyse the following search algorithm:

```c
// Returns the index of the given key in the array if it exists, // or -1 otherwise
int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}
```
Empirical Analysis

Sample results:

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>0.005</td>
</tr>
<tr>
<td>10,000,000</td>
<td>0.028</td>
</tr>
<tr>
<td>100,000,000</td>
<td>0.246</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>2.437</td>
</tr>
</tbody>
</table>

Conclusion: The worst-case runtime of linear search appears to grow linearly as input size increases.
Limitations of Empirical Analysis

- Requires implementation of algorithm, which may be difficult
- Different choice of input data $\Rightarrow$ different results
  - Choosing good inputs is extremely important
- Timing results affected by runtime environment
  - E.g., load on the machine
- In order to compare two algorithms...
  - Need "comparable" implementation of each algorithm
  - Must use same inputs, same hardware, same O/S, same load
Theoretical Analysis

- Uses high-level description of algorithm (pseudocode)
  - Can use the code if it is implemented already
- Characterises runtime as a function of input size
- Takes into account all possible inputs
- Allows us to evaluate the efficiency of the algorithm
  - Independent of the hardware/software environment
Pseudocode

- Pseudocode is a plain language description of the steps in an algorithm.
- Uses structural conventions of a regular programming language:
  - if statements, loops
- Omits language-specific details:
  - variable declarations
Pseudocode for linear search:

\[
\text{linearSearch}(A, \text{key}):
\]

\[
\text{Input: array } A \text{ of } n \text{ integers}
\]

\[
\text{Output: index of } \text{key} \text{ in } A \text{ if it exists, otherwise -1}
\]

\[
\text{for } i \text{ from 0 up to } n - 1 \text{ do}
\]

\[
\text{if } A[i] = \text{key} \text{ then}
\]

\[
\text{return } i
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

\[
\text{return -1}
\]
Every algorithm uses a core set of basic operations.

Examples:

- Assignment
- Indexing into an array
- Calling/returning from a function
- Evaluating an expression
- Increment/decrement

We call these operations **primitive** operations.

Assume that primitive operations take the same constant amount of time.
Counting Primitive Operations

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm as a function of the input size.

```
linearSearch(A, key):
    Input: array A of n integers
    Output: index of key in A if it exists, otherwise -1
    for i from 0 up to n - 1 do 1 + (n + 1) + n
        if A[i] = key then 2n
            return i
        end if
    end for
    return -1 1
```

Note: Assuming that the for loop is implemented as
```
for (int i = 0; i < n; i++)
```
There is 1 assignment, n increments, and (n + 1) checks of the condition.
Linear search requires $4n + 3$ primitive operations in the worst case.

If the time taken by a primitive operation is $c$, then the worst-case running time of linear search is $c(4n + 3)$.

Hence, the worst-case running time of linear search is linear in $n$. 

We are interested in how the running time of the algorithm changes as the input size is scaled:

- E.g., if we double the input size, how does the running time change?

As the input size increases, lower-order terms become less significant.

- For example, suppose the running time of an algorithm is $4n + 3$.
- As $n$ increases, the lower-order term (i.e., 3) becomes less significant (i.e., becomes a smaller proportion of the running time)
Growth rate is not affected by constant factors.

Example: Suppose the running time $T(n)$ of an algorithm is $n^2$.

- What happens when we double the input size?

$$T(2n) = (2n)^2$$

$$= 4n^2$$

$$= 4T(n)$$

When we double the input size, the running time quadruples.
Example: Now suppose the running time $T(n)$ of an algorithm is $10n^2$.

- Now what happens when we double the input size?

$$T(2n) = 10 \times (2n)^2$$
$$= 10 \times 4n^2$$
$$= 4 \times 10n^2$$
$$= 4T(n)$$

When we double the input size, the running time also quadruples!
To summarise:

- Lower-order terms become insignificant as \( n \) increases
- Growth rate is unaffected by constant factors

This means we can ignore lower-order terms and constant factors when characterising the growth rate of the running time of an algorithm.

Examples:

- If \( T(n) = 100n + 500 \), ignoring lower-order terms and constant factors gives \( n \)
- If \( T(n) = 5n^2 + 2n + 3 \), ignoring lower-order terms and constant factors gives \( n^2 \)
This also means that for sufficiently large inputs, the algorithm that has the running time with the highest-order term will always take longer.
Discarding lower-order terms and constant factors...

- Allows us to easily compare the efficiency of algorithms
  - For example, if after discarding lower-order terms and constant factors, algorithm A has a running time of $n$ and algorithm B has a running time of $n^2$, then we can say that for sufficiently large inputs, algorithm A will perform better.
Since growth rate is not affected by constant factors, instead of counting primitive operations, we can simply count line executions.

This is because each line of code contains only a constant number of primitive operations.

linearSearch(A, key):
   Input: array A of n integers
   Output: index of key in A if it exists, otherwise -1
   
   for i from 0 up to n - 1 do
       if A[i] = key then
           return i
       end if
   end for
   
   return -1

\[
\text{Estimating running times} \quad 2n + 1
\]
Big-Oh is a notation used to describe the asymptotic relationship between functions.

**Formally:**
Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if:
- There are positive constants $c$ and $n_0$ such that:
  - $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

**Informally:**
Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if for sufficiently large $n$, $f(n)$ is bounded above by some multiple of $g(n)$. 
Big-Oh Notation

Example 1

\[ f(n) = O(g(n)) \]
Consider the functions $f(n) = 2n + 10$ and $g(n) = n$. We want to show that $f(n)$ is $O(g(n))$, i.e., that $2n + 10$ is $O(n)$.

We need to find some $c$ such that for sufficiently large $n$, $2n + 10 \leq c \cdot n$.

Yes! For $c = 3$, $2n + 10 \leq 3n$ when $n \geq 10$. 
Consider the functions $f(n) = n^2$ and $g(n) = n$. We want to show that $f(n)$ is $O(g(n))$, i.e., that $n^2$ is $O(n)$.

We need to find some $c$ such that for sufficiently large $n$, $n^2 \leq c \cdot n$.

Impossible!
Time complexity is the amount of time taken by an algorithm to run, as a function of the input.

In this course, we usually express time complexity using big-Oh notation. For example, linear search is $O(n)$ in the worst case.
To determine the worst-case time complexity of an algorithm:

- Determine the number of primitive operations/line executions performed in the worst case in terms of the input size
- Discard lower-order terms and constant factors
- The worst-case time complexity is then the big-Oh of the term that remains
Commonly encountered functions in algorithm analysis:

- **Constant**: 1
- **Logarithmic**: $\log n$
- **Linear**: $n$
- **N-Log-N**: $n \log n$
- **Quadratic**: $n^2$
- **Cubic**: $n^3$
- **Exponential**: $2^n$
- **Factorial**: $n!$
Common Functions

Motivation
Efficiency
Empirical Analysis
Theoretical Analysis
Pseudocode
Primitive operations
Estimating running times
Big-Oh notation
Time complexity

Binary Search
Exercise
Relative of Big-Oh

All The Mathematics!

\[ f(n) \text{ is } O(g(n)) \]
if \( f(n) \) is asymptotically less than or equal to \( g(n) \)

\[ f(n) \text{ is } \Omega(g(n)) \]
if \( f(n) \) is asymptotically greater than or equal to \( g(n) \)

\[ f(n) \text{ is } \Theta(g(n)) \]
if \( f(n) \) is asymptotically equal to \( g(n) \)

Given \( f(n) \) and \( g(n) \), we say \( f(n) \) is \( O(g(n)) \)
if we have positive constants \( c \) and \( n_0 \) such that
\[ \forall n \geq n_0, f(n) \leq c \cdot g(n) \]
Linear search requires $4n + 3$ primitive operations in the worst case.

Therefore, linear search is $O(n)$ in the worst case.
Is there a faster algorithm for searching an array?

Yes... if the array is sorted.

Let’s start in the middle.

- If \( key == a[N/2] \), we found \( key \); we’re done!
- Otherwise, we split the array:
  - ... if \( key < a[N/2] \), we search the left half \((a[0] \text{ to } a[(N/2) - 1])\)
  - ... if \( key > a[N/2] \), we search the right half \((a[(N/2) + 1] \text{ to } a[N - 1])\)
Binary search is a more efficient search algorithm for *sorted arrays*:

```c
int binarySearch(int arr[], int size, int key) {
    int lo = 0;
    int hi = size - 1;

    while (lo <= hi) {
        int mid = (lo + hi) / 2;

        if (key < arr[mid]) {
            hi = mid - 1;
        } else if (key > arr[mid]) {
            lo = mid + 1;
        } else {
            return mid;
        }
    }

    return -1;
}
```
Successful search for 6:

\[
\begin{array}{cccccc}
2 & 3 & 5 & 6 & 8 & 9 \\
lo & mid & hi \\
\end{array}
\]

5 < 6

\[
\begin{array}{cccccc}
2 & 3 & 5 & 6 & 8 & 9 \\
lo & mid & hi \\
\end{array}
\]

8 > 6

\[
\begin{array}{cccccc}
2 & 3 & 5 & 6 & 8 & 9 \\
lo & mid & hi \\
\end{array}
\]

Exercise
Unsuccessful search for 7:

```
[0]  [1]  [2]  [3]  [4]  [5]
2    3    5    6    8    9
```

`lo` `mid` `hi`

5 < 7

```
[0]  [1]  [2]  [3]  [4]  [5]
2    3    5    6    8    9
```

`lo` `mid` `hi`

8 > 7

```
[0]  [1]  [2]  [3]  [4]  [5]
2    3    5    6    8    9
```

`lo` `mid` `hi`

6 < 7

```
[0]  [1]  [2]  [3]  [4]  [5]
2    3    5    6    8    9
```

`hi` `lo`
How many iterations of the loop?

- **Best case:** 1 iteration
  - Item is found right away
- **Worst case:** $\log_2 n$ iterations
  - Item does not exist
  - Every iteration, the size of the subarray being searched is halved

Thus, binary search is $O(\log_2 n)$ or simply $O(\log n)$
If I know my algorithm is quadratic (i.e., $O(n^2)$), and I know that for a dataset of 1000 items, it takes 1.2 seconds to run ...

- how long for 2000?
- how long for 10,000?
- how long for 100,000?
- how long for 1,000,000?
If I know my algorithm is quadratic (i.e., $O(n^2)$), and I know that for a dataset of 1000 items, it takes 1.2 seconds to run ...

- how long for 2000? 4.8 seconds
- how long for 10,000?
- how long for 100,000?
- how long for 1,000,000?
If I know my algorithm is quadratic (i.e., $O(n^2)$), and I know that for a dataset of 1000 items, it takes 1.2 seconds to run ... 

- how long for 2000? 4.8 seconds
- how long for 10,000? 120 seconds (2 mins)
- how long for 100,000?
- how long for 1,000,000?
If I know my algorithm is quadratic (i.e., $O(n^2)$), and I know that for a dataset of 1000 items, it takes 1.2 seconds to run …

- how long for 2000? 4.8 seconds
- how long for 10,000? 120 seconds (2 mins)
- how long for 100,000? 12000 seconds (3.3 hours)
- how long for 1,000,000?
If I know my algorithm is quadratic (i.e., $O(n^2)$), and I know that for a dataset of 1000 items, it takes 1.2 seconds to run ... 

- how long for 2000? 4.8 seconds
- how long for 10,000? 120 seconds (2 mins)
- how long for 100,000? 12000 seconds (3.3 hours)
- how long for 1,000,000? 1200000 seconds (13.9 days)