COMP2521 24T1
Analysis of Algorithms

Kevin Luxa

cs2521@cse.unsw.edu.au

Motivation

- Program efficiency 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”

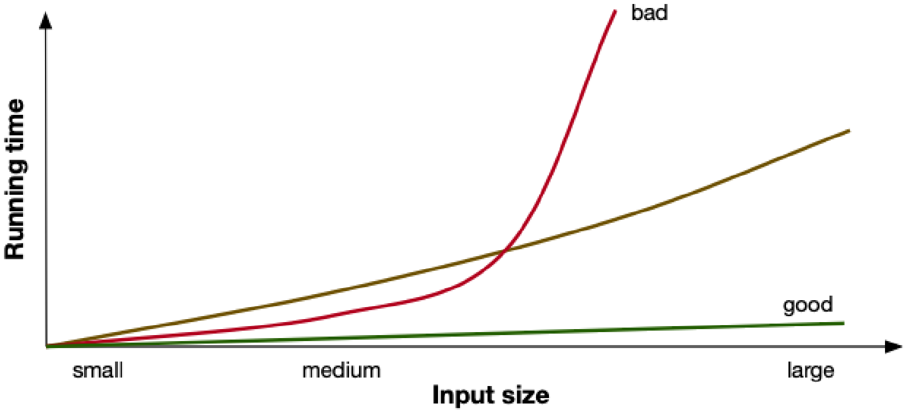
Program Efficiency

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?



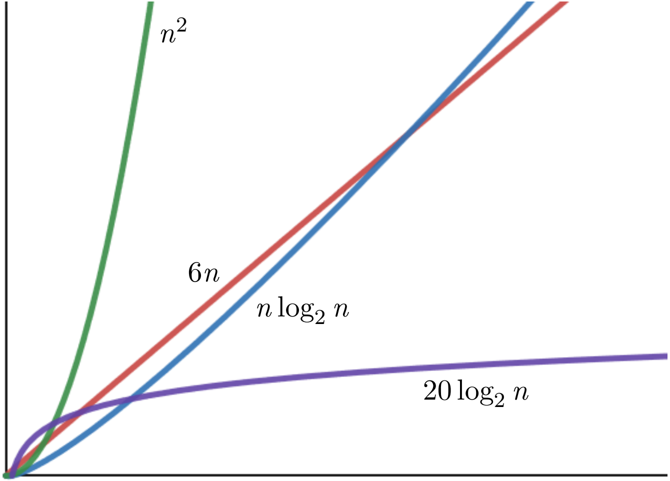
What to Analyse?

- 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

Time Complexity

Time complexity is
the amount of time it takes to run an algorithm,
as a function of the input size

Example functions:



Analysing Time Complexity

The time complexity of an algorithm can be analysed in two ways:

- Empirically: Measuring the time that a program implementing the algorithm takes to run
- Theoretically: Counting the number of operations or “steps” performed by the algorithm as a function of input size

Searching

The search problem:

Given an array of size n and a value,
return the index containing the value if it exists,
otherwise return -1.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
2	16	11	1	9	4	15

Empirical Analysis

1. Write a program that implements the algorithm
2. Run the program with inputs of varying size and composition
3. Measure the running time of the algorithm
4. Plot the results

Timing Execution

We can measure the running time of an algorithm using *clock(3)*.

- The *clock()* function determines the amount of processor time used since the start of the process.

```c
#include <time.h>

clock_t start = clock();
// algorithm code here...
clock_t end = clock();

double seconds = (double)(end - start) / CLOCKS_PER_SEC;
```

Timing Execution

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:

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

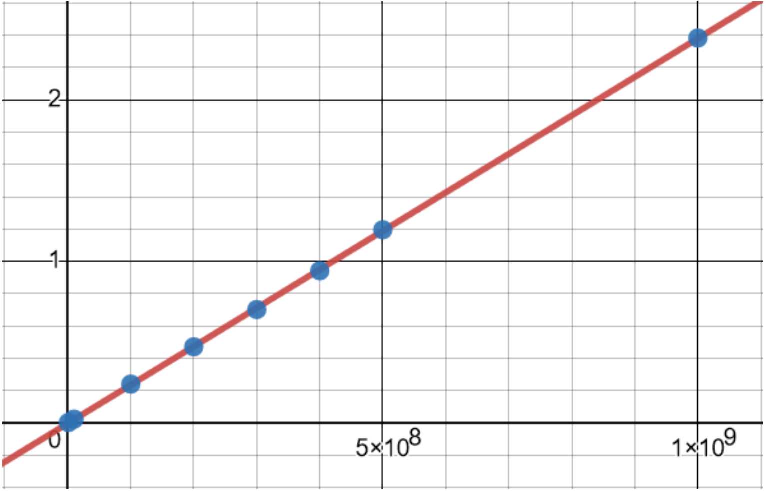
Empirical Analysis

Sample results:

Input Size	Running Time
1,000,000	0.002
10,000,000	0.023
100,000,000	0.240
200,000,000	0.471
300,000,000	0.702
400,000,000	0.942
500,000,000	1.196
1,000,000,000	2.384

The worst-case running time of linear search grows linearly as the input size increases.

Empirical Analysis



Limitations of Empirical Analysis

- Requires implementation of algorithm
- 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 running time as a function of input size
- 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
  - allocating/freeing memory

Pseudocode

Pseudocode for linear search:

```
linearSearch(A, val):
    Input:  array A of size n, value val
    Output: index of val in A if it exists
            -1 otherwise

    for i from 0 up to n-1:
        if A[i] = val:
            return i

    return -1
```

Primitive Operations

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

Example

How many primitive operations are performed by this line of code?

```
for (int i = 0; i < n; i++)
```

Counting Primitive Operations
Example

How many primitive operations are performed by this line of code?

```
for (int i = 0; i < n; i++)
```

The assignment `i = 0` occurs 1 time
The comparison `i < n` occurs $n + 1$ times
The increment `i++` occurs n times

Total: $1 + (n + 1) + n$ primitive operations

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, val):
    Input:  array A of size n, value val
    Output: index of val in A if it exists
            -1 otherwise

    for i from 0 up to n-1:           1 + (n + 1) + n
        if A[i] = val:                2n
            return i

    return -1                         1
                                      ---------------
                                      4n + 3   (total)
```

Counting Primitive Operations

Linear search requires $4n + 3$ primitive operations in the worst case.

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

We are mainly interested in
how the running time of an algorithm changes
as the input size increases.

This is called the **asymptotic behaviour** of the running time.

Asymptotic Analysis

Lower-Order Terms

Asymptotic behaviour is not affected by lower-order terms.

- For example, suppose the running time of an algorithm is $4n + 100$.
- As n increases, the lower-order term (i.e., 100) becomes less significant (i.e., becomes a smaller proportion of the running time)

Asymptotic Analysis

Constant Factors

Asymptotic behaviour 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?

$$\begin{aligned} T(2n) &= (2n)^2 \\ &= 4n^2 \\ &= 4\,T(n) \end{aligned}$$

When we double the input size, the time taken quadruples.

Example: Now suppose the running time $T(n)$ of an algorithm is $10n^2$.

- Now what happens when we double the input size?

$$\begin{aligned} T(2n) &= 10 \times (2n)^2 \\ &= 10 \times 4n^2 \\ &= 4 \times 10n^2 \\ &= 4T(n) \end{aligned}$$

When we double the input size, the time taken also quadruples!

Asymptotic Analysis

To summarise:

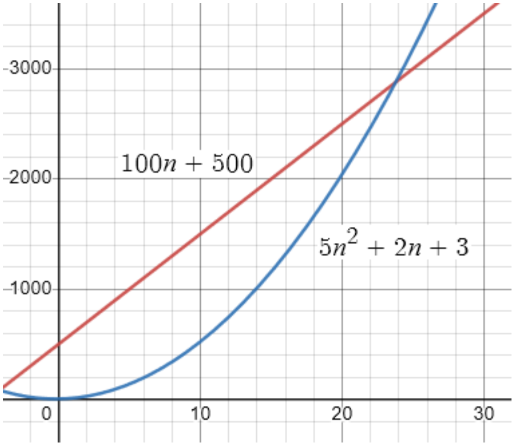
- Asymptotic behaviour is unaffected by lower-order terms
- Asymptotic behaviour is unaffected by constant factors

This means we can ignore lower-order terms and constant factors when characterising the asymptotic behaviour 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.



Big-Oh Notation

Big-Oh notation
is used to classify the asymptotic behaviour of an algorithm,
and this is how we usually express time complexity in this course.

For example, linear search is $O(n)$ in the worst case.

Big-Oh notation allows us to easily compare the efficiency of algorithms

- For example, if algorithm A has a time complexity of $O(n)$ and algorithm B has a time complexity of $O(n^2)$, then we can say that for sufficiently large inputs, algorithm A will perform better.

Big-Oh Notation

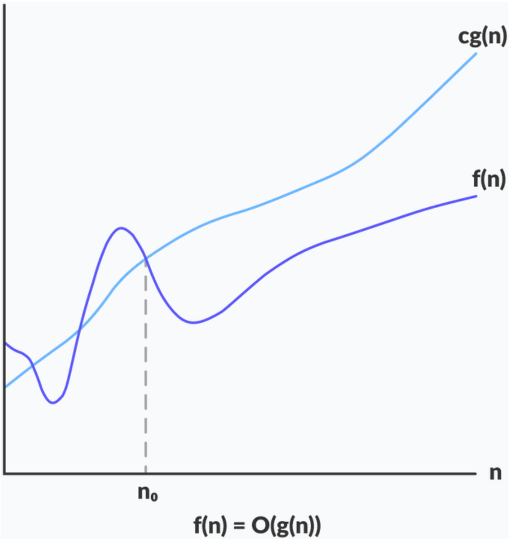
Formally, big-Oh is actually a notation used to describe the asymptotic relationship between functions.

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



Relatives of Big-Oh

All The Mathematics!

$f(n)$ is $O(g(n))$
if $f(n)$ is asymptotically less than or equal to $g(n)$

$f(n)$ is $\Omega(g(n))$
if $f(n)$ is asymptotically greater than or equal to $g(n)$

$f(n)$ is $\Theta(g(n))$
if $f(n)$ is asymptotically equal to $g(n)$

Analysing Complexity

Since time complexity is not affected by constant factors, instead of counting primitive operations, we can simply count line executions.

```
linearSearch(A, value):
    Input:  array A of size n, value
    Output: index of value in A if it exists
            -1 otherwise

    for i from 0 up to n-1:             n
        if A[i] = value:                n
            return i

    return -1                           1
                                        ---------------
                                        2n + 1  (total)
```

Worst-case time complexity: $O(n)$

Analysing Complexity

To determine the worst-case time complexity of an algorithm:

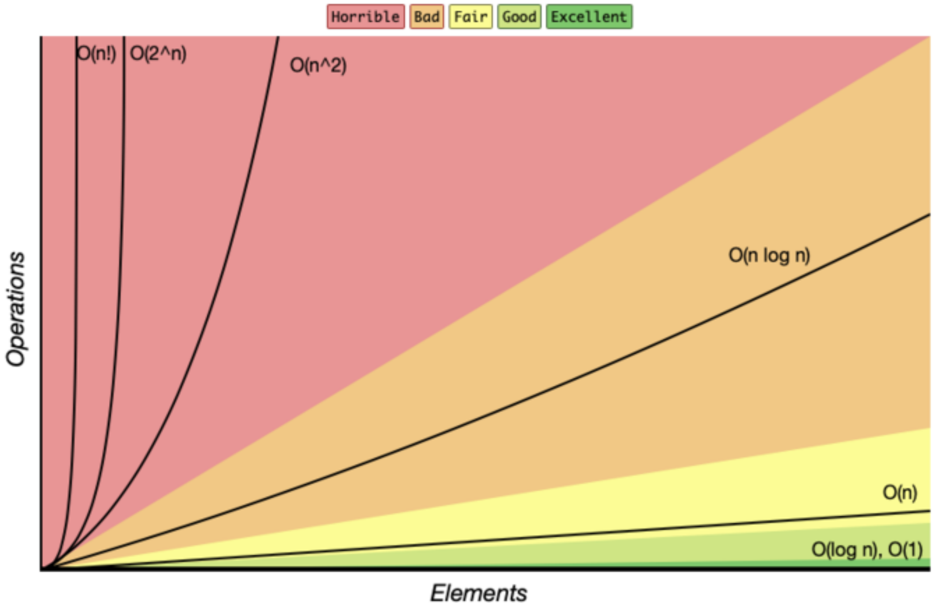
- Determine the number of line executions performed in the worst case in terms of the input size
- Discard lower-order terms and constant factors
- The worst-case time complexity is then the big-Oh of the term that remains

Common Functions

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



Back to Linear Search

Linear search requires $4n + 3$ primitive operations in the worst case.

Therefore, linear search is $O(n)$ in the worst case.

Searching in a Sorted Array

Is there a faster algorithm for searching an array?

Yes... if the array is sorted.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	2	4	9	11	15	16

Let's start in the middle.

- If $a[N/2] = val$, we found val; we're done!
- Otherwise, we split the array:
  ... if $val < a[N/2]$, we search the left half ($a[0]$ to $a[(N/2) - 1)]$)
  ... if $val > a[N/2]$, we search the right half ($a[(N/2) + 1)]$ to $a[N - 1]$)

Binary search is a more efficient search algorithm for **sorted arrays**:

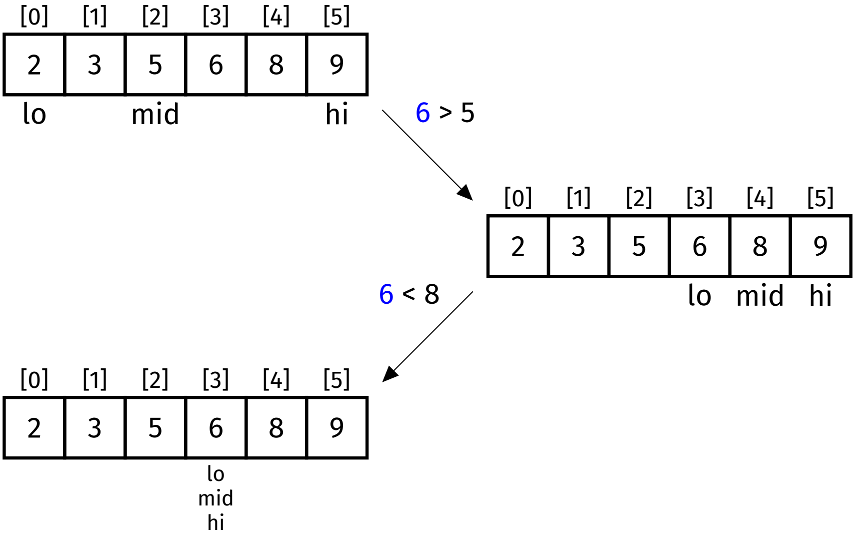
```c
int binarySearch(int arr[], int size, int val) {
    int lo = 0;
    int hi = size - 1;

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

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

    return -1;
}
```

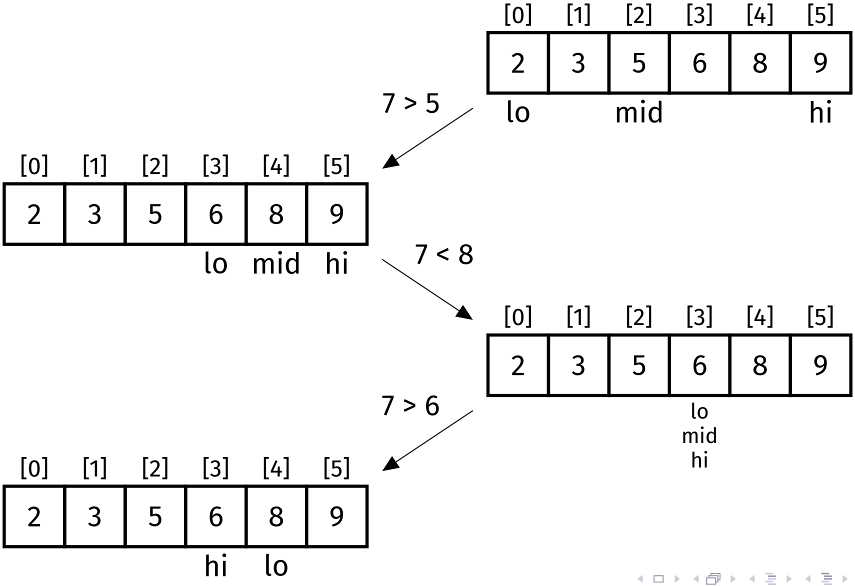
Successful search for 6:



Binary Search

Example

Unsuccessful search for 7:



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 $O(\log n)$

$$O(\log_2 n) = O(\log n)$$

Why drop the base?

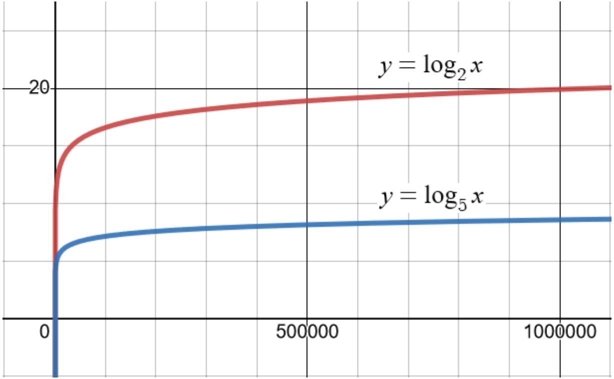
According to the change of base formula:

$$\log_a n = \frac{\log_b n}{\log_b a}$$

If a and b are constants,
$\log_a n$ and $\log_b n$ differ by a constant factor

For example:

$$\log_2 n = \frac{\log_5 n}{\log_5 2}$$
$$\approx 2.32193 \log_5 n$$



Multiple Variables

What if an algorithm takes multiple arrays as input?

If there is no constraint on the relative sizes of the arrays, their sizes would be given as two variables, usually n and m

Multiple Variables

Example time complexities with two variables:

$$O(n + m)$$

$$O(nm)$$

$$O(\max(n, m))$$

$$O(\min(n, m))$$

$$O(n \log m)$$

$$O(n \log m + m \log n)$$

Problem:

Given two arrays, where each array contains no repeats, find the number of elements in common

Multiple Variables

Example

```
numCommonElements(A, B):
    Input:  array A of size n
            array B of size m
    Output: number of elements in common

    numCommon = 0
    for i from 0 up to n-1:
        for j from 0 up to m-1:
            if A[i] = B[j]:
                numCommon = numCommon + 1

    return numCommon
```

Time complexity: $O(nm)$

Feedback

https://forms.office.com/r/5c0fb4tvMb

Appendix

Predicting Time

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?

(answers on the next slide)

Predicting Time

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)