SWINBURNE UNIVERSITY OF TECHNOLOGY

Algorithm Efficiency and Pattern

The “Best” Algorithm

- There are usually multiple algorithms to solve any particular problem.
- The notion of the “best” algorithm may depend on many different criteria:
  - Structure, composition, and readability
  - Time required to implement
  - Extensibility
  - Space requirements
  - Time requirements

Time/Space Analysis

- Example:
  - Algorithm A runs 2 minutes and algorithm B takes 1 minutes and 45 second to complete for the same input.
- Is B "better" than A? ➠ Not necessarily!:
  - We have tested A and B only on one (fixed) input set. Another input set might result in a different runtime behavior.
  - Algorithm A might have been interrupted by another process.
  - Algorithm B might have been run on a different computer.
- A reasonable time and space approximation should be machine-independent.

Big-O notation

- Wiki : *Big O notation* is a mathematical *notation* that describes the limiting behaviour of a function when the argument tends towards a particular value or infinity
- It is used to describe the worse case, or ceiling of growth for an algorithm or a function.
- It is a simplified analysis of an algorithm's efficiency/ complexity.
- **Asymptotic** analysis
- It focuses on basic computing steps

Big-O notation : General rules

- Ignores the constant coefficient
  - n, 2n, 5n, ... O(n)
- Functions/algorithms with “higher growth rate” dominate others
  - $O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(2^n)<O(n!)$

SWINBURNE UNIVERSITY OF TECHNOLOGY

Big-O notation : General rules

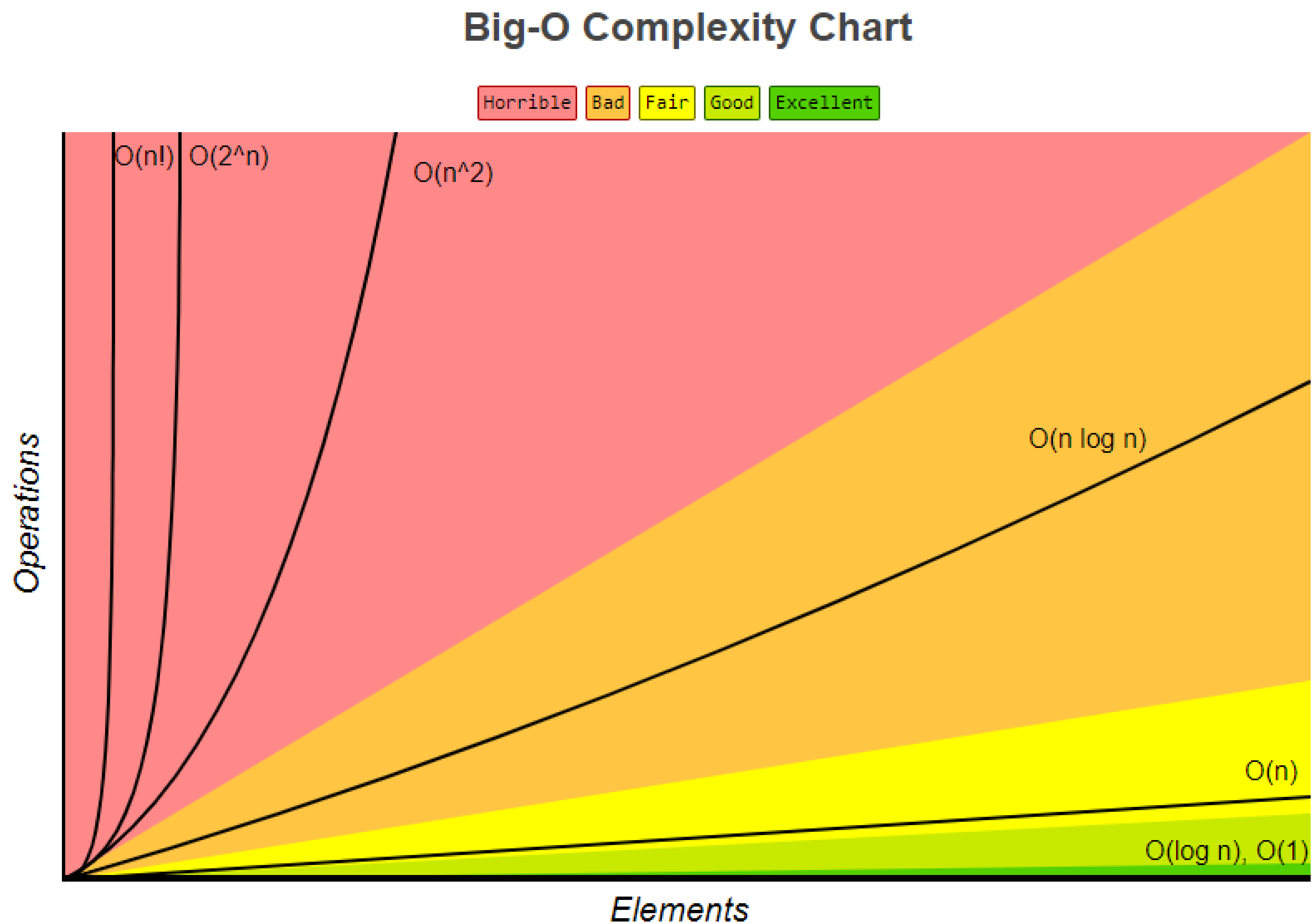


Figure from bigocheatsheet.com

Big-O : Constant time, O(1)

- Notation: **O(1)**
- `x = 3*(5+12)`
- x is independent from any input or additional variables
- We call such complexity, constant time

```
x=3*(5+12);
y=15-2;
print x-y;
```

  - What is big-O for this code?

    Total = O(1)+O(1)+O(1) = 3*O(1) => O(1)

    Since constants are ignored.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Big-O : Linear time, O(n)

- Notation: **O(n)**

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

- O(1) statement repeat n times. n is not a constant coefficient but it is possibly a variable.
- So it is n*O(1) = O(n)

SWIN BUR NE
SWINBURNE UNIVERSITY OF TECHNOLOGY

Big-O notation : Example

```
m = 3*(2+13);
for(i =0; i<n; ++i)
cout<<i<<endl;
```

- The total time = $O(1)+O(n)... = O(n)$

SWINBURNE UNIVERSITY OF TECHNOLOGY

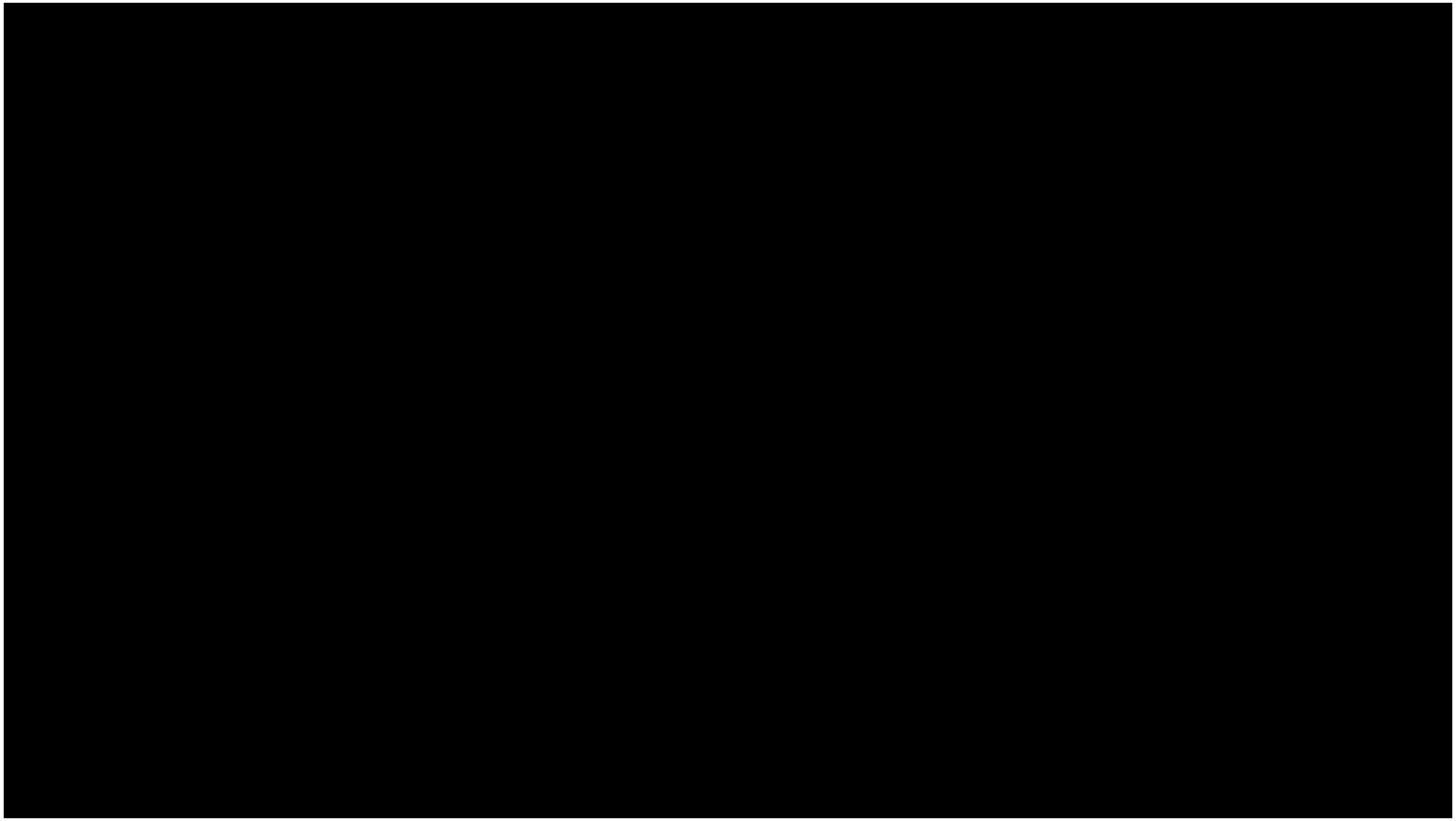
Big-O: Quadratic time, $O(n^2)$

- Two for loops

```
for(i =0; i<n; ++i)
    for(j =0; j<n; ++j)
        cout<<i*j<<endl;
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

Big O Notation in 5 minutes



URL: https://www.youtube.com/watch?v=__vX2sjlpXU

Big-O notation: exercise 1

- What is the Big-O for this?

```
x = 5 + (15 * 20);
for x in range (0, n):
    print x;
for x in range (0, n):
    for y in range (0, n):
        print x * y;
```

O(1)

O(n)

O(n^2)

The largest run time will be the Big-O notation: O(n^2)

Big-O notation: exercise 2

```
a=5
b=6
c=10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b
d = 33
```

$O(n^2)$

$O(n)$

The largest run time will be the Big-O notation: $O(n^2)$

SWINBURNE UNIVERSITY OF TECHNOLOGY

What is computable?

- Computation is usually modeled as a mapping from inputs to outputs, carried out by a “formal machine”, or program, which processes its input in a sequence of steps.
- An “effectively computable” function is one that can be computed in a finite amount of time using finite resources.

Halting Problem

- A problem that cannot be solved by any machine in finite time (or any equivalent formalism) is called uncomputable.
- An uncomputable problem cannot be solved by any real computer.
- **The Halting Problem:**
  - Given a program and its input, determine whether the program will complete or run forever.
  - The Halting Problem is provably uncomputable – which means that it cannot be solved in practice.

```
green = ON
red = amber = OFF
while(true){
   amber = ON; green =
   OFF;
   wait 10 seconds;
   red = ON; amber = OFF;
   wait 40 seconds;
   green = ON; red = OFF;
}
```

Turing and the Halting Problem

URL: https://www.youtube.com/watch?v=macM_MtS_w4

SWINBURNE UNIVERSITY OF TECHNOLOGY

Algorithmic Patterns

- Direct solution strategies:
  - Brute force and greedy algorithms
- Backtracking strategies:
  - Simple backtracking and branch-and-bound algorithms
- Top-down solution strategies:
  - Divide-and-conquer algorithms
- Bottom-up solution strategies:
  - Dynamic programming
- Randomized strategies:
  - Monte Carlo algorithms

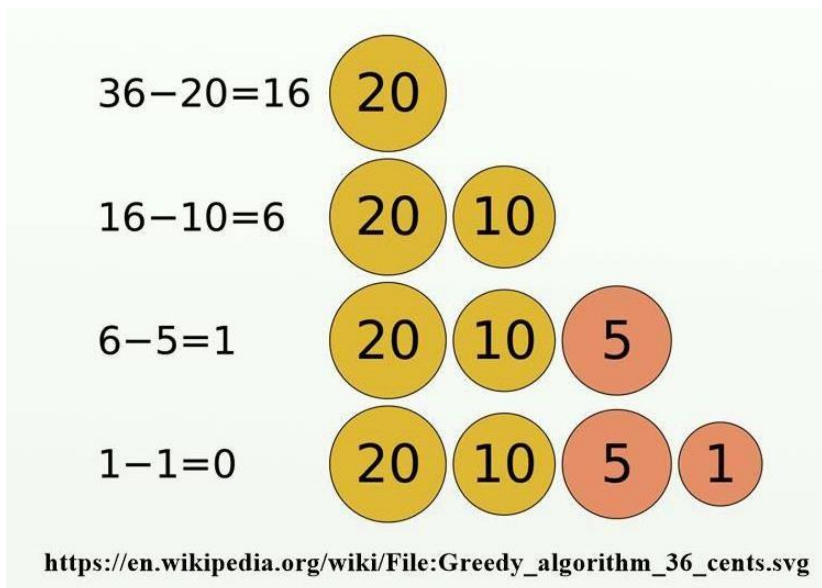
SWINBURNE UNIVERSITY OF TECHNOLOGY

Brute-force Algorithms

- Brute-force algorithms are not distinguished by their structure.
- Brute-force algorithms are separated by their way of solving problems.
- A problem is viewed as a sequence of decisions to be made. Typically, brute-force algorithms solve problems by exhaustively enumerating all the possibilities

Greedy Algorithms

- Greedy algorithms do not really explore all possibilities. They are optimized for a specific attribute.
- It **always makes the choice that seems to be the best at that moment**.
  - It makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.



Divide-and-Conquer

- Top-down algorithms use recursion to divide-and-conquer the problem space.
- This class of algorithms has the advantage that not all possibilities have to be explored.
- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
- **Solution**: Combine the solutions to the subproblems into the solution for the original problem.

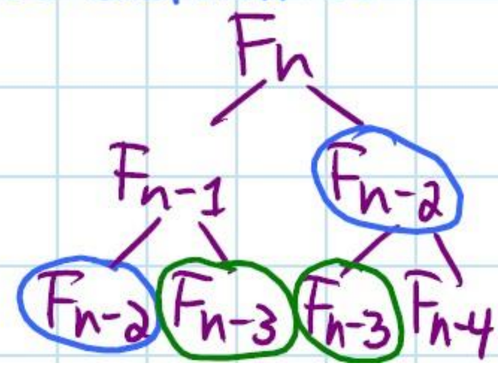
Example: Binary Search, Merge Sort, Quick Sort

Divide-and-Conquer

<u>Fibonacci numbers:</u> $F_1 = F_2 = 1$; $F_n = F_{n-1} + F_{n-2}$

– <u>naive algorithm:</u> follow recursive definition

```
fib(n):
    if n ≤ 2: return 1
    else: return fib(n-1)
                 + fib(n-2)
```



Dynamic programming

- Bottom-up algorithms employ dynamic programming.
- Bottom-up algorithms solve a problem by solving a series of subproblems.
- These subproblems are carefully devised in such a way that each subsequent solution is obtained by combining the solutions to one or more of the subproblems that have already been solved.

Dynamic programming

- simple idea: memoize

```
memo = {}
fib(n):
    if n in memo: return memo[n]
    else: if n≤2: f=1
          else: f=fib(n-1)+fib(n-2)
          memo[n] = f
          return f
```

$$\underbrace{fib(n-2)}_{\text{free}}$$

$$\Rightarrow T(n) = T(n-1) + O(1) = O(n)$$

Difference D&C and Dynamic Programming

- Divide-&-conquer works best when all subproblems are independent. So, pick partition that makes algorithm most efficient & simply combine solutions to solve entire problem.

- Dynamic programming is needed when subproblems are dependent; we don't know where to partition the problem.

- Divide-&-conquer is best suited for the case when no "overlapping subproblems" are encountered.

- In dynamic programming algorithms, we typically solve each subproblem only once and store their solutions. But this is at the cost of space.

Randomized Algorithms

- Randomized algorithms behave randomly.
- Randomized algorithms select elements in an random order to solve a given problem.
- Eventually, all possibilities are explored, but different runs can produce results faster or slower, if a solution exists.
  - Example: Monte Carlo Methods, Simulation

SWINBURNE UNIVERSITY OF TECHNOLOGY

Introduction to Big O Notation and Time Complexity

URL: https://www.youtube.com/watch?v=D6xkbGLQesk

End of Algorithm Efficiency and Pattern

Big O Cheat Sheet:
http://bigocheatsheet.com/