# CSC263 Tutorial 1

Big-O, Ω, Θ

Trevor Brown
(tabrown@cs.utoronto.ca)

# Big-O and Big-Ω

- Many students think Big-O is "worst case" and Big-Ω is "best case." They are WRONG! *Both O(x) and Ω(x) describe running time for the* ***worst possible input****!*
- What do O(x) and Ω(x) mean?

| Algorithm is O(x) | Algorithm is Ω(x) |
|---|---|
| The algorithm takes **at most** c*x steps to run on the worst possible input. | The algorithm takes **at least** c*x steps to run on the worst possible input. |

- How do we show an algorithm is O(x) or Ω(x)?

| Algorithm is O(x) | Algorithm is Ω(x) |
|---|---|
| Show: **for every input,** the algorithm takes **at most** c*x steps. | Show: **there is an input** that makes the algorithm take **at least** c*x steps. |

# Analyzing algorithms using O, Ω, Θ

- First, we analyze some easy algorithms.
- We can easily find upper and lower bounds on the running times of these algorithms by using basic arithmetic.

# Example 1

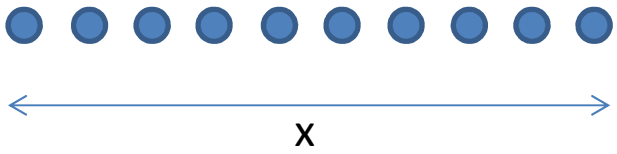
```
for i = 1..100
        print *
```

- Running time: 100*c, which is Θ(1) (why?)
- O(1), Ω(1) => Θ(1)

# Example 2

```
for i = 1..x
        print *
```

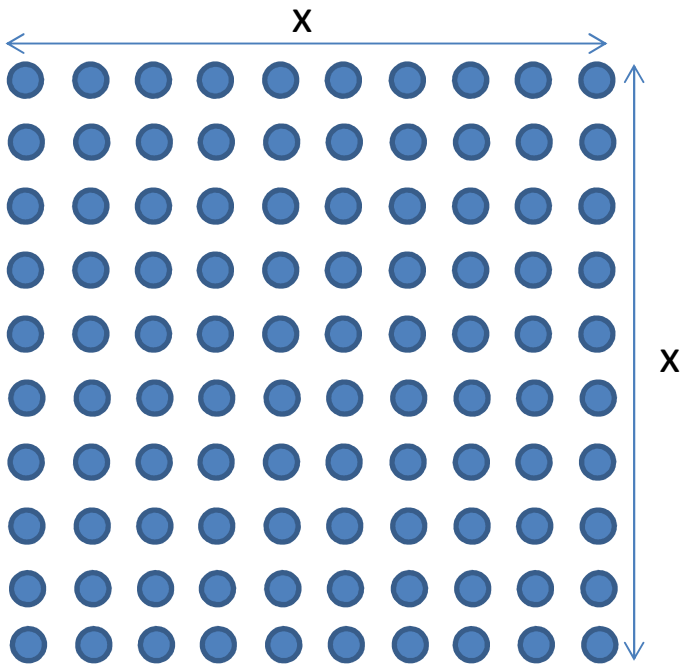
- Running time: x*c
- O(x), Ω(x) => Θ(x)
- In fact, this is also Ω(1) and O(n^2), but these are very weak statements.



# Example 3

```
for i = 1..x
        for j = 1..x
                print *
```

- O(x^2), Ω(x^2) => Θ(x^2)



# Example 4a

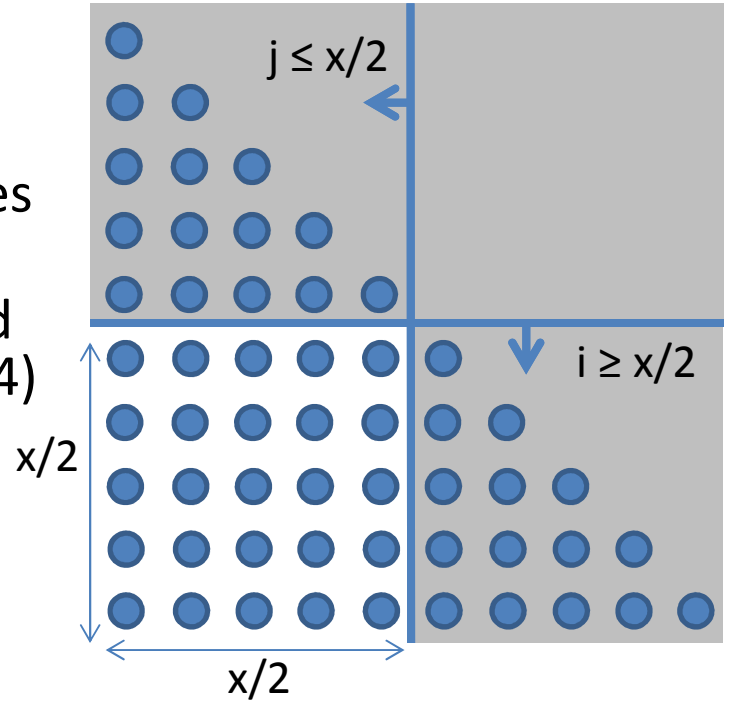
```
for i = 1..x
        for j = 1..i
                print *
```

- Big-O: i is always ≤ x, so j always iterates up to at most x, so this at most x*x steps, which is O(x^2).
- Big-Ω:
  - When i=1, the loop over j performs “print *” once.
  - When i=2, the loop over j performs “print *” twice. […]
  - So, “print *” is performed 1+2+3+…+x times.
  - Easy summation formula: 1+2+3+…+x = x(x+1)/2
  - x(x+1)/2 = x^2/2+x/2 ≥ (1/2) x^2, which is Ω(x^2)

# Example 4b

```
for i = 1..x                    >=    for i = x/2..x
      for j = 1..i                          for j = 1..x/2
            print *                               print *
```

- Big-Ω:
  - Useful trick: consider the iterations of the first loop for which i >= x/2.
  - In these iterations, the second loop iterates from j=1 to at least j=x/2.
  - Therefore, the number of steps performed in these iterations is at least x/2*x/2 = (1/4) x^2, which is Ω(x^2).
  - The time to perform ALL iterations is even more than this so, of course, the whole algorithm must take Ω(x^2) time.
  - Therefore, this function is Θ(x^2).



# Example 5

```
for i = 1..x
    for j = 1..i
        for k = 1..j
            print *
```

```
for i = x/2..x
    for j = x/4..x/2
        for k = 1..x/4
            print *
```

- Big-O: i, j, k ≤ x, so we know total number of steps is ≤ x*x*x = O(x^3).
- Big-Ω:
  - Consider only the iterations of the first loop from x/2 up to x.
  - For these values of i, the second loop always iterates up to at least x/2.
  - Consider only the iterations of the second loop from x/4 up to x/2.
  - For these values of j, the third loop always iterates up to at least x/4.
  - For these values of i, j and k, the function performs “print *” at least: x/2 * x/4 * x/4 = x^3/32 times, which is Ω(x^3).
  - Therefore, this function is Θ(x^3).

# Analyzing algorithms using O, Ω, Θ

- Now, we are going to analyze algorithms whose running times depend on their inputs.
- For some inputs, they terminate very quickly.
- For other inputs, they can be slow.

# Example 1

```
LinearSearch(A[1..n], key)
    for i = 1..n
        if A[i] = key then return true
    return false
```

- Might take 1 iteration… might take many...
- Big-O: at most n iterations, constant work per iteration, so O(n)
- Big-Ω: can you find a bad input that makes the algorithm take Ω(n) time?

# Example 2

```
int A[n]
for i = 1..n
        binarySearch(A, i)
```

- Remember: binarySearch is O(lg n).
- O(n lg n)
- How about Ω?
  - Maybe it's Ω(n lg n), but it's hard to tell.
  - **Not** enough to know binary search is Ω(lg n), because the worst-case input might make only **one** invocation of binary search take c*lg n steps (and the rest might finish in 1 step).
  - Would need to find a particular input that causes the algorithm to take a total of c(n lg n) steps.
  - In fact, binarySearch(A, i) takes c*lg n steps when i is not in A, so this function takes c(n lg n) steps if A[1..n] doesn't contain any number in {1,2,…,n}.

# Example 3

- Your boss tells you that your group needs to develop a sorting algorithm for the company's web server that runs in O(n lg n) time.
- If you can't prove that it can sort any input of length n in O(n lg n) time, it's no good to him. He doesn't want to lose orders because of a slow server.
- Your colleague tells you that he's been working on this piece of code, which he believes should fit the bill. He says he thinks it can sort any input of length n in O(n lg n) time, but he doesn't know how to prove this.

# Example 3 continued

```
WeirdSort(A[1..n])
  last := n
  sorted := false
  while not sorted do
     sorted := true
     for j := 1 to last-1 do
        if A[j] > A[j+1] then
           swap A[j] and A[j+1]
           sorted := false
     last := last-1
```