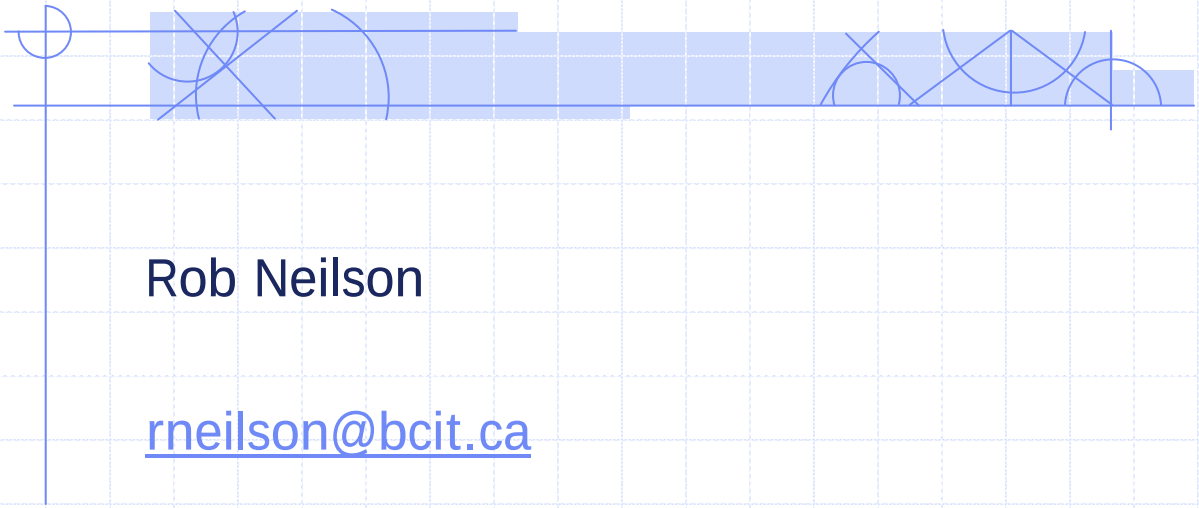
# COMP 3760: Algorithm Analysis and Design

## Lesson 5: Brute Force Algorithms



Rob Neilson

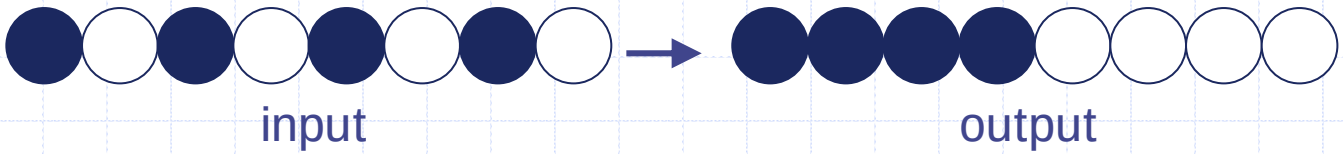
rneilson@bcit.ca

# Devising Algorithms

- Assume you have a problem to solve, for example (from pg 103 of book):

  You have 1 row of *2n* disks of 2 colors, *n* dark and *n* light. They alternate dark, light, dark, light, dark, and so on. You want to get all the dark disks on the right hand side and all the light disks on the left.

  *The only moves you are allowed to make are those that interchange the position of two neighboring disks.*




(a) Devise an algorithm that solves this problem

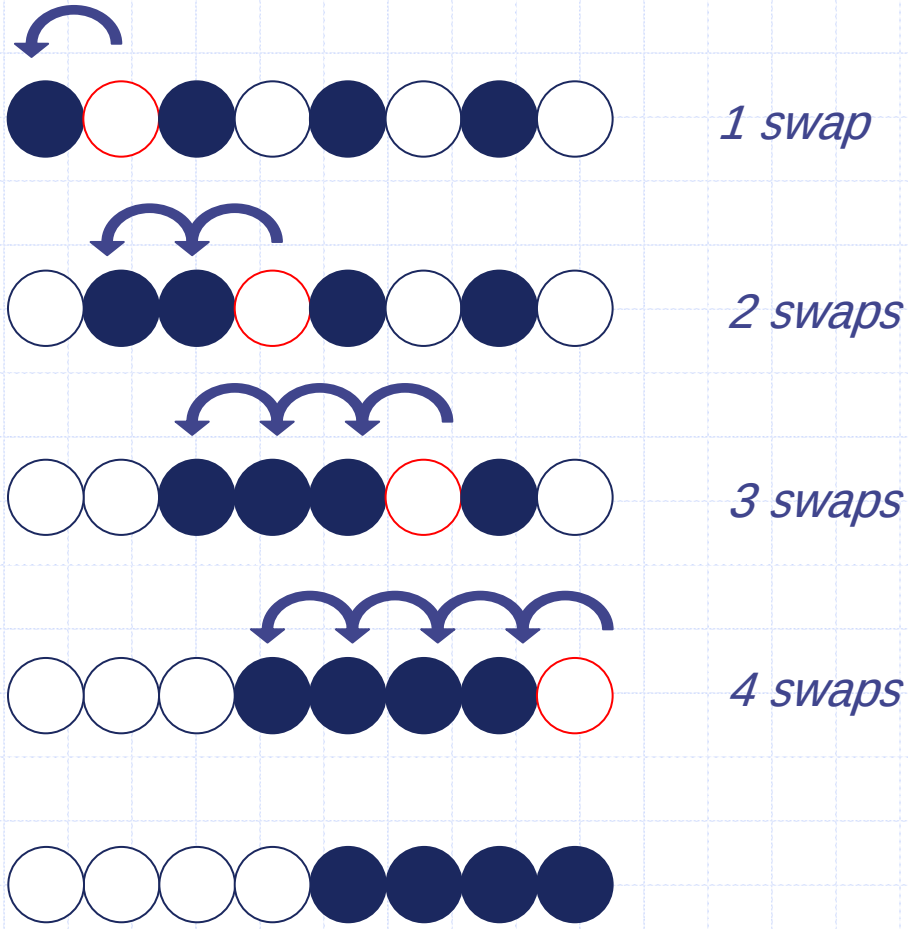
(b) Analyze the efficiency of your algorithm.

# Action Break 1

- work with a partner to write pseudo code for the alternating disks problem (pg 103 of book)
- analyze your solution and determine its worst case efficiency class

# One possible solution ...

- start with left-most and ending with the right-most white disk
- repeatedly swap it with its left neighbour until there are no dark disks to the left of it



# How to write this in pseudocode …

- let the disks be represented by an array of size 2n
- A[i] = 1 for black disks, A[i] = 0 for white disks
  - initialize as:

```
for i ← 1 to 2n                 // iterate over A
    A[i] ← i mod 2              // set color
```

  - now, the algorithm might be something like this:

```
sortdisks(A[1..2n])
   for i ← 1 to n               // iterate over half list
      w ← 2*i                   // get next white disk
      for b ← w–1 to i          //
          swap(A[w], A[b])      // move to its position
          w ← w–1               // this is to move w by 2
```

# What is the Basic Operation?

- thinking about the problem (not fixating on the pseudocode) we understand that the “swapping of two disks” is the fundamental operation
- in the pseudocode this is in fact the line “swap”
  - Note: swap() as written is a function, so we need to know it’s efficiency, as it could be an O(n!) function …
  - … luckily it isn’t; swapping is just:

    **temp ← a**

    **a ← b**

    **b ← temp**
  - … and this is 3 operations, which is constant time O(1)

# How many swaps?

- From the pictorial example: 1+2+3+…+n
- From the pseudo-code:
  - first time the inner loop executes once, second time twice, etc
- In either case we have

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in O(n^2)$$

# How did you approach this problem?

- What did you do first?
  - one way to do this is to try to relate the problem to other similar things you may have seen in the past
    - the previous example is just a variation of bubblesort
    - bubblesort works by repeatedly “bubbling” the largest element

```
bubble(A[0..n-1])
    for i ← 0 to n-2
        for j ← 0 to n-2-i
            if A[j+1] < A[j]
                swap(A[j], A[j+1])
```

# Bubble Sort

**First Pass:**

( **5 1** 4 2 8 ) ( **1 5** 4 2 8 ) ... compares the first two elements, and swap
( 1 **5 4** 2 8 ) ( 1 **4 5** 2 8 )
( 1 4 **5 2** 8 ) ( 1 4 **2 5** 8 )
( 1 4 2 **5 8** ) ( 1 4 2 **5 8** ) ... elements are in order, do not swap

**Second Pass:**

( **1 4** 2 5 8 ) ( **1 4** 2 5 8 )
( 1 **4 2** 5 8 ) ( 1 **2 4** 5 8 )
( 1 2 **4 5** 8 ) ( 1 2 **4 5** 8 )
( 1 2 4 **5 8** ) ( 1 2 4 **5 8** ) ... array is sorted, but algo does not know this.

Algorithm needs one more **whole** pass without **any** swap to know it is sorted.

**Third Pass:**

( **1 2** 4 5 8 ) ( **1 2** 4 5 8 )
( 1 **2 4** 5 8 ) ( 1 **2 4** 5 8 )
( 1 2 **4 5** 8 ) ( 1 2 **4 5** 8 )
( 1 2 4 **5 8** ) ( 1 2 4 **5 8** ) Finally, the algorithm is done.

# Brute Force Algorithm

- in the case of the white and black disks it was reasonable to look for the obvious solution
- often this “obvious” solution is considered a “brute force” algorithm

*Brute Force Algorithm:*

- **soln based on the definition of the problem**
- **not really trying to be efficient**
- **typically these are easy to implement**

# Other Brute Force Algorithms

- Bubble Sort - *repeatedly* bubble *largest element up to last posn*
- Selection Sort - *repeatedly* exchange *smallest element with first posn*

  - note that selection sort would have ordered the black/white disks in only n swaps (as opposed to n(n+1)/2)
    - *but it broke the **rules of the problem** so we couldn't use it*

- Any $O(n^2)$ **sort** algorithm is considered a brute force algorithm!

- Other brute force algorithms include
  - Linear Search
  - String Matching (the traditional way)

... any algorithm that basically just does the obvious sequence of operations without regard for efficiency should be considered a brute force algorithm.

# Brute Force String Matching

```
// find the pattern P[] in the text T[]
BruteForceStringMatch(T[0..n-1], P[0..m-1])
   for i ← 0 to n-m do
       j ← 0
       while j < m and P[j] = T[i+j] do
          j ← j+1
       if j = m return i
   return -1
```

P[] = afunnynewnegativeage  T[] = neg

# Brute Force String Matching

- what is the efficiency?
- this one is a bit more difficult to analyze because of the variables in the while loop
- we need to look at the inner while loop and ask “how many times does it iterate in the worst case?”
  - well, the worst input would cause the inner loop to compare all m characters in the target pattern before seeing that it is not a match
- so theoretically the inner loop could execute m basic instructions each time it shifts the target, ie, n-m+1 times
- the means we could have a worst case efficiency of O((n-m)(m))
  - *note we typically assume n >> m, and say that it is O(nm)*

# Action Break 2: String matching algo question (from textbook – page 106 question 5) …

- How many comparisons (basic operations) are made by the brute-force string-matching algorithm in searching for each of the following patterns in the binary text of 1000 zeros?

eg: Text[]= {0000000000000000000000000000000000...0}
Pattern[]= {00001}

a) 00001
   - it will go through the outer loop n-m+1 times (1000-5+1 = 996)
   - on each iteration it will have 4 successful comparisons, and one failure
   - therefore it requires a total of 996*5 = 4980 comparisons

b) 10000
   - in this case there will only be 1 failed compare (no successful ones)
   - so the total number of comparisons is 996

c) 01010
   - there will be one successful compare, and 1 failed compare
   - 2*996 = 1992 compares in total

# Action Break 3 (page 106 question 9)…

Consider the problem of counting the number of substrings in a given text that start with an A and end with a B.

(For example, there are four such substrings in CABAAXBYA … they are AB ABAAXB AAXB AXB)

a. Design a brute-force algorithm for this problem and determine its efficiency class.

*We observe:*

- the number of substrings that start with any specific A is given by the number of B's to the right of that position
  - **for example. the first A (after the C) has 2 B's to the right of it, so there are two substrings that use this A**

# Brute Force Solution

- therefore our algorithm involves:
  - init ABcount to 0
  - scan left to right until you find an A
  - for each A
    - count the number of B's to the right of it, and add this to ABcount
- the worst case occurs with an input text of all A's, in which case the algo makes

$n + (n-1) + (n-2) + \ldots + 2 = n(n+1)/2 \in O(n^2)$ comparisons

# The End