What is Brute Force?

_force of the computer, not of your intellect_

= simple & stupid

just do it!
Why study them?

✦ Simple to implement
  suppose you need to solve only one instance?
✦ Often “good enough”, especially when $n$ is small
✦ Widely applicable
✦ Actually OK for some problems, e.g., Matrix Multiplication
✦ Can be the starting point for an improved algorithm
✦ “Baseline” against which we can compare better algorithms
✦ Can be a “gold standard” of correctness
Sequential Search

`searchFor: needle``

"sequential search for needle. Returns true if found."

```plaintext
self do: [:each |
    (each == needle) ifTrue: [ ^ true ].
].

^ false
```
Sequential Search

**searchFor**: needle

"sequential search for needle. Returns true if found."

```ruby
self do: [:each |
  (each == needle) ifTrue: [ ^true ].
].
^false
```

**searchUsingSentinal**: needle

"sequential search for needle. Returns true if found."

```ruby
| i |
i ← 1.
[(self at: i) == needle ] whileFalse: [ i ← i + 1 ].
^ (i ~< self size)
```
Sequential Search

**searchFor: needle**
"sequential search for needle. Returns true if found."

self do: [:each |
  (each == needle) ifTrue: [ true ].
].
↑ false

**searchUsingAt: needle**
"sequential search for needle. Returns true if found."

| i sz |
sz ← self size.
i ← 1.
[((self at: i) == needle) | (i = sz)] whileFalse: [ i ← i + 1 ].
↑ (i ~< sz)
Sequential Search

**searchUsingAt**: `needle`

"sequential search for needle. Returns true if found."

```
<table>
<thead>
<tr>
<th>i sz</th>
</tr>
</thead>
</table>
sz ← self size.
i ← 1.
[((self at: i) == needle) | (i = sz)] whileFalse: [ i ← i + 1 ].
↑ (i ~= sz)
```

**searchUsingSentinal**: `needle`

"sequential search for needle. Returns true if found."

```
<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
</table>
i ← 1.
[(self at: i) == needle ] whileFalse: [ i ← i + 1 ].
↑ (i ~= self size)
```
Timing Sequential Search

testSequentialSearch

| A B N M res t1 t2 t3 |
N ← 100000.
M ← 5000000. "bigger than the array to be searched, and any value in it"
A ← self randomArrayOfSize: N.
t1 ← Time millisecondsToRun: [1000 timesRepeat: [res ← A searchFor: M]].
self deny: res.
B ← A copyWith: M.
t2 ← Time millisecondsToRun: [1000 timesRepeat: [res ← B searchUsingSentinel: M]].
self deny: res.
A ← A copyWith: M.
t3 ← Time millisecondsToRun: [1000 timesRepeat: [res ← A searchUsingAt: M]].
self deny: res.
Transcript show: 'Sequential search, size: ';
show: N; cr;
show: 'sequential: '; show: t1; show: 'μs'; cr;
show: 'with sentinel: '; show: t2; show: 'μs'; cr;
show: 'without sentinel: '; show: t3; show: 'μs'; cr; cr.
Timing Results

Sequential search, size: 100000
  sequential: 1668µs
  with sentinel: 878µs
  without sentinel: 1452µs

Sequential search, size: 100000
  sequential: 1515µs
  with sentinel: 802µs
  without sentinel: 1409µs
Timing Results

Sequential search, size: 100000
sequential: 1668μs
with sentinel: 878μs
without sentinel: 1452μs

Sequential search, size: 100000
sequential: 1515μs
with sentinel: 802μs
without sentinel: 1409μs

Coding details *can* make a difference!
Timing Results

Sequential search, size: 100000
  sequential: 1668µs
  with sentinel: 878µs
  without sentinel: 1452µs

Sequential search, size: 100000
  sequential: 1515µs
  with sentinel: 802µs
  without sentinel: 1409µs

Coding details *can* make a difference!

But *not* to the asymptotic complexity.
ALGORITHM SelectionSort(A[0..n − 1])

//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
for i ← 0 to n − 2 do
  min ← i
  for j ← i + 1 to n − 1 do
      min ← j
  swap A[i] and A[min]
selectionSort
"Sort me using selection sort. Levitin §3.1"

| indexOfMin n A |
A ← self.
n ← self size.
1 to: n - 1 do: [ i |
   indexOfMin ← i.
i + 1 to: n do: [ j |
       (A at: j) < (A at: indexOfMin) ifTrue: [ |
           indexOfMin ← j]. |
   A swap: i with: indexOfMin ]

ALGORITHM  SelectionSort(A[0..n – 1])
//Sorts a given array by selection sort
//Input: An array A[0..n – 1] of orderable elements
//Output: Array A[0..n – 1] sorted in ascending order
for i ← 0 to n – 2 do
   min ← i
   for j ← i + 1 to n – 1 do
   swap A[i] and A[min]
Ex 3.1, Problem 4

a. Design a brute-force algorithm for computing the value of a polynomial

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \]

at a given point \( x_0 \) and determine its worst-case efficiency class.
Ex 3.1, Problem 4

a. Design a brute-force algorithm for computing the value of a polynomial

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \]

at a given point \( x_0 \) and determine its worst-case efficiency class.

Write it down clearly, so I can project it with the document camera
Ex 3.1, Problem 4

a. Design a brute-force algorithm for computing the value of a polynomial

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0 \]

at a given point \( x_0 \) and determine its worst-case efficiency class.

b. If the algorithm you designed is in \( \Theta(n^2) \), design a linear algorithm for this problem.
Solution to Problem 4

a. Design a brute-force algorithm for computing the value of a polynomial

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \]

at a given point \( x_0 \) and determine its worst-case efficiency class.

**Algorithm** *BruteForcePolynomialEvaluation*(\( P[0..n] \), \( x \))

//The algorithm computes the value of polynomial \( P \) at a given point \( x \)
//by the “highest-to-lowest term” brute-force algorithm
//Input: Array \( P[0..n] \) of the coefficients of a polynomial of degree \( n \),
//stored from the lowest to the highest and a number \( x \)
//Output: The value of the polynomial at the point \( x \)
\( p \leftarrow 0.0 \)
for \( i \leftarrow n \) downto 0 do
    \( power \leftarrow 1 \)
    for \( j \leftarrow 1 \) to \( i \) do
        \( power \leftarrow power \times x \)
        \( p \leftarrow p + P[i] \times power \)
return \( p \)
Solution to Problem 4

Algorithm \textit{BruteForcePolynomialEvaluation}(P[0..n], x)
//The algorithm computes the value of polynomial \( P \) at a given point \( x \)
//by the “highest-to-lowest term” brute-force algorithm
//Input: Array \( P[0..n] \) of the coefficients of a polynomial of degree \( n \),
//stored from the lowest to the highest and a number \( x \)
//Output: The value of the polynomial at the point \( x \)
\( p \leftarrow 0.0 \)
for \( i \leftarrow n \) downto 0 do
  \( p \leftarrow 0.0 \)
  \( \text{power} \leftarrow 1 \)
  for \( j \leftarrow 1 \) to \( i \) do
    \( \text{power} \leftarrow \text{power} \times x \)
    \( p \leftarrow p + P[i] \times \text{power} \)
return \( p \)

- size of input is degree of polynomial, \( n \)
- number of multiplications depends only on \( n \)
- number of multiplications, \( M(n) \in \ ? \)

A. \( \Theta(n) \)  
B. \( \Theta(n^2) \)  
C. \( \Theta(n \lg n) \)  
D. \( \Theta(n^3) \)
Ex 3.1, Problem 4

a. Design a brute-force algorithm for computing the value of a polynomial

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0 \]

at a given point \( x_0 \) and determine its worst-case efficiency class.

b. If the algorithm you designed is in \( \Theta(n^2) \), design a linear algorithm for this problem.
Solution to Problem 4

Algorithm BetterBruteForcePolynomialEvaluation($P[0..n]$, $x$)
// The algorithm computes the value of polynomial $P$ at a given point $x$
// by the “lowest-to-highest term” algorithm
// Input: Array $P[0..n]$ of the coefficients of a polynomial of degree $n$,
// from the lowest to the highest, and a number $x$
// Output: The value of the polynomial at the point $x$
$p ← P[0]$;  $\text{power} ← 1$
for $i ← 1$ to $n$ do
    $\text{power} ← \text{power} \times x$
    $p ← p + P[i] \times \text{power}$
return $p$
True or False?

✧ It is possible to design an algorithm with better-than-linear efficiency to calculate the value of a polynomial.

A. True
B. False
Ex 3.1, Problem 9

iad Is selection sort stable?

- The definition of a stable sort was given in Levitin §1.3

A. Yes, it is stable

B. No, it is not stable
Ex 3.1, Problem 10

Is it possible to implement selection sort for a linked-list with the same $\Theta(n^2)$ efficiency as for an array?

A. Yes, it is possible

B. No, it is not possible
BubbleSort

**ALGORITHM**  \textit{BubbleSort}(A[0..n - 1])

//Sorts a given array by bubble sort  
//Input: An array A[0..n – 1] of orderable elements  
//Output: Array A[0..n – 1] sorted in ascending order

\bf{for} i \leftarrow 0 \bf{to} n - 2 \bf{do}

\bf{for} j \leftarrow 0 \bf{to} n - 2 - i \bf{do}

BubbleSort

**ALGORITHM** \( \text{BubbleSort}(A[0..n - 1]) \)

//Sorts a given array by bubble sort
//Input: An array \( A[0..n - 1] \) of orderable elements
//Output: Array \( A[0..n - 1] \) sorted in ascending order

for \( i \leftarrow 0 \) to \( n - 2 \) do
    for \( j \leftarrow 0 \) to \( n - 2 - i \) do

• Is BubbleSort stable?
BubbleSort

**ALGORITHM**  \( \text{BubbleSort}(A[0..n - 1]) \)

//Sorts a given array by bubble sort
//Input: An array \( A[0..n - 1] \) of orderable elements
//Output: Array \( A[0..n - 1] \) sorted in ascending order

\[
\text{for } i \leftarrow 0 \text{ to } n - 2 \text{ do } \\
\quad \text{for } j \leftarrow 0 \text{ to } n - 2 - i \text{ do } \\
\quad \quad \text{if } A[j + 1] < A[j] \text{ swap } A[j] \text{ and } A[j + 1]
\]

- Is BubbleSort stable?

- **Prove** that, if BubbleSort makes no exchanges on a pass through the array, then the array is sorted.
String Matching
Applications:

- Find all occurrences of a particular word in a given text
  - Searching for text in an editor
  - ...

- Compare two strings to see how similar they are to one another ...
  - Code diff-ing
  - DNA sequencing
  - ...

- ...
Notation

Let A be a set of characters (the alphabet)

The set of strings that consist of finite sequences of characters in A is written $A^*$ (the Kleene Star)

For a string $s$, we’ll write:

- $s[i]$ for the $i^{\text{th}}$ character in $s$
- $|s|$ for the length of $s$
- $s[i..j]$ for the substring of $s$ from $s[i]$ to $s[j]$
- $s[..n]$ for the prefix $s[1..n]$, and $s[m..]$ for $s[m.|s|]$
- $\varepsilon$ for the empty string (example: $s[1..0] = \varepsilon$)
- $st$ for the concatenation of $s$ with another string $t$
Simple Complexities:

Assume that string is represented by an array of consecutive characters.

What's the worst case running time for brute-force testing to determine:

- whether $s = t$
Simple Complexities:

Assume that string is represented by an array of consecutive characters.

What's the worst case running time for brute-force testing to determine:

- whether $s = t$

A. $\Theta(1)$

B. $\Theta(|s|)$

C. $\Theta(|\min(s, t)|)$

D. $\Theta(|s|^2)$

E. None of the above
Simple Complexities:

Assume that string $s$ is represented by an array of consecutive characters

- Worst case running time for computing $s[i]$ ?
Simple Complexities:

Assume that string $s$ is represented by an array of consecutive characters

- Worst case running time for computing $s[i]$?

  A. $\Theta(1)$
  B. $\Theta(|s|)$
  C. $\Theta(|\min(|s|, i)|)$
  D. $\Theta(i)$
  E. None of the above
Simple Complexities:

Assume that strings are represented by arrays of consecutive characters

- Worst case running time for computing $st$
Simple Complexities:

Assume that strings are represented by arrays of consecutive characters

Worst case running time for computing $st$?

A. $\Theta(1)$
B. $\Theta(|s|)$
C. $\Theta(|\min(s, t)|)$
D. $\Theta(|\min(s, t)|^2)$
E. None of the above
Simple Complexities:

Assume that string is represented by an array of consecutive characters

- Worst case running times for computing $s[i..j]$
Simple Complexities:

Assume that string is represented by an array of consecutive characters

Worst case running times for computing s[i..j]

A. $\Theta(1)$
B. $\Theta(|s[i..j]|)$
C. $\Theta(j-i)$
D. $\Theta((j-i)^2)$
E. None of the above
String Matching

- Find all occurrences of a pattern string $p$ in a text string $t$

For example:

```
abraca[da]bracalama[azo][oo]
[rac] [rac]
```
String Matching, formally

- Given a text string, \( t \), and a pattern string, \( p \), of length \( m = |p| \), find the set of all shifts \( s \) such that \( p = t[s+1..s+m] \)

```
  a b r a c a d a b r a c a l a m a z o o
```

- \( s=2 \)

```
  a b r a c a d a b r a c a l a m a z o o
```

```
  r a c
```

- \( s=9 \)

```
  a b r a c a d a b r a c a l a m a z o o
```

```
  r a c
```
Brute-force Matching Algorithm

abracadabraocalamazoo

rac
Brute-force Matching Algorithm

abracadabraclamazoo

race

abracadabraclamazoo

race
Brute-force Matching Algorithm
Brute-force Matching Algorithm

...
Brute-force Matching Algorithm

abracadabraclamazoo

drac

abracadabraclamazoo

drac

drac

abracadabraclamazoo

drac

abracadabraclamazoo

drac
# Brute-force Matching Algorithm

| a | b | r | a | c | a | d | a | b | r | a | c | a | l | a | m | a | z | o | o |
| r | a | c |

| a | b | r | a | c | a | d | a | b | r | a | c | a | l | a | m | a | z | o | o |
| r | a | c |
Brute-force Matching Algorithm

What's the asymptotic complexity of brute-force matching?:

```
abraca
dabracalama
```

```
abraca
dabracalama
```
What's the asymptotic complexity of brute-force matching?:

A. $\Theta(1)$
B. $\Theta(|t|)$
C. $\Theta(|p|)$
D. $\Theta(|p|(|t|-|p|+1))$
E. None of the above
Brute-force Matching Algorithm

\[
\text{match}(t, p) \\
m \leftarrow |p| \\
n \leftarrow |t| \\
\text{results} \leftarrow \{\} \\
\text{for } s \leftarrow 0..n-m \text{ do} \\
\hspace{1em} \text{if } p = t[s+1 .. s+m] \text{ then} \\
\hspace{2em} \text{results} \leftarrow \text{results} \cup \{s\} \\
\text{return results}
\]
Brute-force Matching Algorithm

\[
\text{match}(t, p) \\
\quad m \leftarrow |p| \\
\quad n \leftarrow |t| \\
\quad \text{results} \leftarrow \{\} \\
\quad \text{for } s \leftarrow 0..n-m \text{ do} \\
\quad \quad \text{if } p == t[s+1 .. s+m] \text{ then} \\
\quad \quad \quad \text{results} \leftarrow \text{results} \cup \{s\} \\
\quad \text{return results}
\]

Asymptotic Complexity: 
\(\Theta(m(n-m+1))\)
Can we do better?

- Perhaps surprisingly: yes!
- Key insight: when a match fails, we learned something
  - Better algorithms in Chapter 7