Data Structures and Algorithms

UBCO Master of Data Science – DATA 532

Some administration first

before we really get started …

The Essence of the Course

The overall goal of this course is for you to:

How to choose and use appropriate algorithms and data structures to help solve data science problems

This course covers basic algorithms and data structures that will enable you to develop your own algorithms that are faster, efficient, and scalable in real-world.

Admin stuff…

Course syllabus

Weekly Lab Assignment

Weekly lab assignments are worth 30% of your overall grade.

Lab assignments may take more than the two hours lab time.

- No late labs will be accepted.
- A lab may be submitted in any time before the due date.

Lab assignments are done individually.

The lab assignments are critical to learning the material and are designed both to prepare you for the exams and build up your skills!

Academic Dishonesty

Cheating is strictly prohibited and is taken very seriously by UBC.

A guideline to what constitutes cheating:

- Labs
  - Submitting code produced by others.
  - Working in groups to solve questions and/or comparing answers to questions once they have been solved (except for group assignments).
  - Discussing detailed HOW to solve a particular question instead of WHAT the question involves.
- Quiz
  - Only materials permitted by instructor should be in the exam.

Academic dishonesty may result in a "F" for the course and removal from the MDS program.

How to Excel in This Course

Attend ***every*** class:

- Read notes ***before*** class as preparation and try the questions if there any.
- Participate in class exercises and questions.

Attend and complete all labs:

- Labs practice the fundamental employable skills as well as being for marks.

Practice on your own. Practice makes perfect.

- Do more questions than in the labs.
- Read the additional reference material and perform practice questions.

Systems and Tools

Course material is on GitHub.

https://github.com/ubco-mds-2021/data532

Marks are distributed on Canvas.

Labs are submitted on Canvas.

Exams are online

Announcements are on Canvas

The course: Objectives

For any computational task on data you need an algorithm to solve it, and you need to store the data in a suitable data structure to access the data. If the data are large, you need these algorithms and data structures to be efficient.

At the end of this course, you should be able:

- Select a suitable basic algorithm and data structure for a given task
- Design efficient algorithms for simple computational tasks

Diverse Background Experience

This is a diverse class with different background and experience.

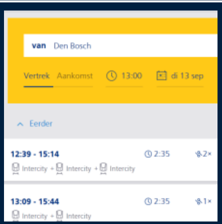
- Some material you may already know. Help others!! The best way to learn is to teach. Maintain Academic honesty – Do not pass your solutions

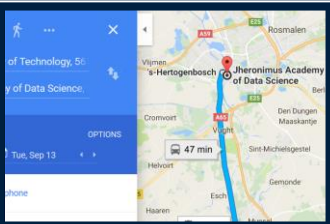
Today's Lecture Objectives

- Importance of algorithms and their applications
- Learn about complexity of algorithms
- Understand the idea of searching
  - Linear Search
  - Binary Search

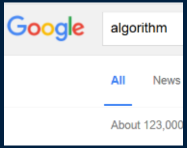
Why is this course important ?

Route planning
shortest-path algorithms



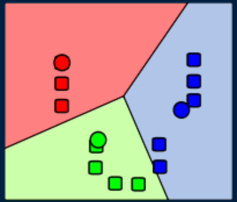


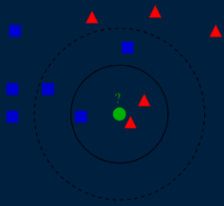
Search engines
matching and ranking algorithms



Data Analysis
e.g., k-means clustering algorithm,

k-nearest neighbor algorithm, …





Algorithms run everywhere: cars, smartphones, laptops, servers, climate-control systems, elevators …

Algorithms on computers

How do you “teach” a computational device to perform an algorithm?

Computers do not have intuition or spatial insight

“An algorithm is a set of steps to accomplish a task that is described *precisely* enough that a computer can run it.”

Algorithms

Algorithm

a well-defined computational procedure that takes some value, or a set of values, as input and produces some value, or a set of values, as output.

Algorithm

sequence of computational steps that transform the input into the output.

Algorithms & Data Structures

fast algorithms require the data to be stored in a suitable way.

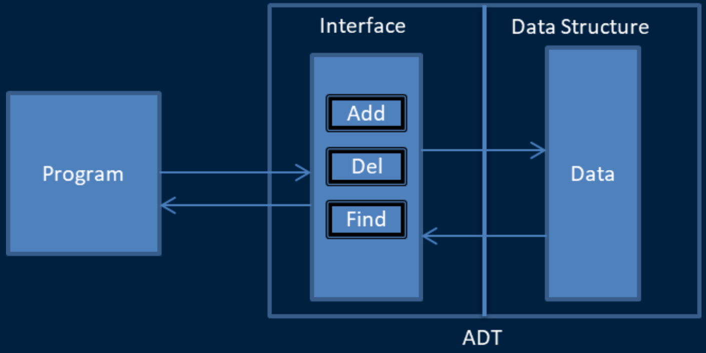
Data structures

Data Structure

a way to store and organize data to facilitate access and modifications.

Abstract data type

describes external functionality (which operations are supported) to operate on a data structure



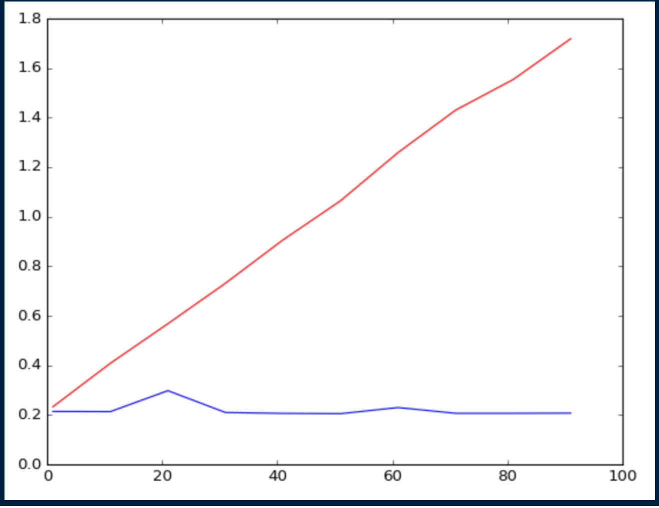
Implementation

a way to realize the desired functionality

- how is the data stored (array, linked list, …)

Example: Select a data structure

If you often need to search your data, simply storing it in an array/list will considerably slow down the computations

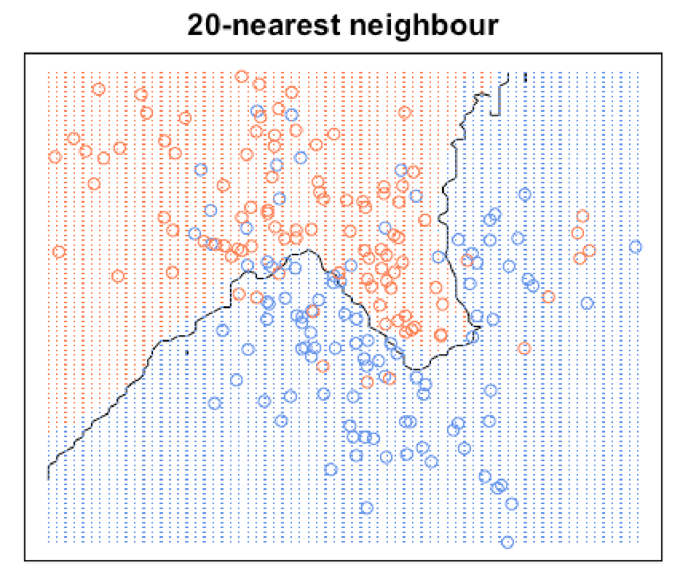



```
In [5]: timeit.timeit(stmt='1 in A', setup='A = list(range(2, 300))')
Out[5]: 4.774117301917343

In [6]: timeit.timeit(stmt='1 in A', setup='A = set(range(2, 300))')
Out[6]: 0.04995419831658 41
```

Example: Select an algorithm

Algorithms for finding the k nearest neighbors are used for analysis tasks like classification



Example: The KNN Algorithm

1. Load the data
2. chose K value
3. For each example in the data

3.1 Calculate the distance between the query example and the current example from the data.

3.2 Add the distance and the index of the example to an ordered collection

4. Sort the ordered collection of distances and indices from smallest to largest (in ascending order) by the distances
5. Pick the first K entries from the sorted collection
6. Get the labels of the selected K entries
7. If regression, return the mean of the K labels
8. If classification, return the mode of the K labels

Simplest version of KNN is brute-force algorithm

Which algorithm/implementation is suitable for your data?

Example: Select an algorithm

http://scikit-learn.org/stable/modules/neighbors.html

1.6.4. Nearest Neighbor Algorithms

1.6.4.1. Brute Force

Fast computation of nearest neighbors is an active area of research in machine learning. The most naive neighbor search implementation involves the brute-force computation of distances between all pairs of points in the dataset: for N samples in D dimensions, this approach scales as $O[DN^2]$. Efficient brute-force neighbors searches can be very competitive for small data samples. However, as the number of samples N grows, the brute-force approach quickly becomes infeasible. In the classes within `sklearn.neighbors`, brute-force neighbors searches are specified using the keyword `algorithm = 'brute'`, and are computed using the routines available in `sklearn.metrics.pairwise`.

1.6.4.2. K-D Tree

To address the computational inefficiencies of the brute-force approach, a variety of tree-based data structures have been invented. In general, these structures attempt to reduce the required number of distance calculations by efficiently encoding aggregate distance information for the sample. The basic idea is that if point A is very distant from point B, and point B is very close to point C, then we know that points A and C are very distant, *without having to explicitly calculate their distance*. In this way, the computational cost of a nearest neighbors search can be reduced to $O[DN\log(N)]$ or better. This is a significant improvement over brute-force for large N.

An early approach to taking advantage of this aggregate information was the *KD tree* data structure (short for *K-dimensional*

Which of the algorithms would you choose ?

A) Brute-force

B) K-D Tree

C) A combination of the two algorithms depending on your probelm

D) Doesn’t matter both will solve the problem

K-D tree performs well enough when D <20. With larger D, it again takes longer time. This is known as "***curse of dimensionality***"

Unless you do the brute force on a GPU, overall, the KD-Tree should be faster

What do we expect from an algorithm?

What do we expect from an algorithm?

Computer algorithms solve computational problems

Computational problems have well-specified input and output

- Question: Is the following problem well-specified?

  ~~*“Given a collection of values, find a certain value x.”*~~

array: sequential collection of elements, which allows constant-time access to an element by its index. (*Note: We (and Python) use as first index 0, in some textbooks, they start at 1.*)

35	30	19	30	8	12	11	17	2	5
0	1	2	3						

“Given an array A of elements and another element x, output either an index i for which A[i] = x, or Not-Found”

There are 2 requirements on the algorithm:

1. Given an input the algorithm should produce the correct output
2. The algorithm should use resources efficiently

Correctness

Given an input the algorithm should produce the correct output

What is a correct solution?
For example, the shortest-path … but given traffic, constructions … input might be incorrect

Not all problems have a well-specified correct solution

➡ we focus on problems with a clear correct solution

Randomized algorithms and approximation algorithms
special cases with alternative definition of correctness

Efficiency

The algorithm should use resources efficiently

The algorithm should be reasonably fast (elapsed time)

The algorithm should not use too much memory

Other resources: network bandwidth, random bits, disk operations …

➡ we focus on time

How do you measure time?

How do we measure time of an algorithm ?

A) Use my wrist watch

B) Use a sand clock

C) Use the clock on the computer

D) Implement the algorithm and measure time

E) Theoretical analysis of the algorithm

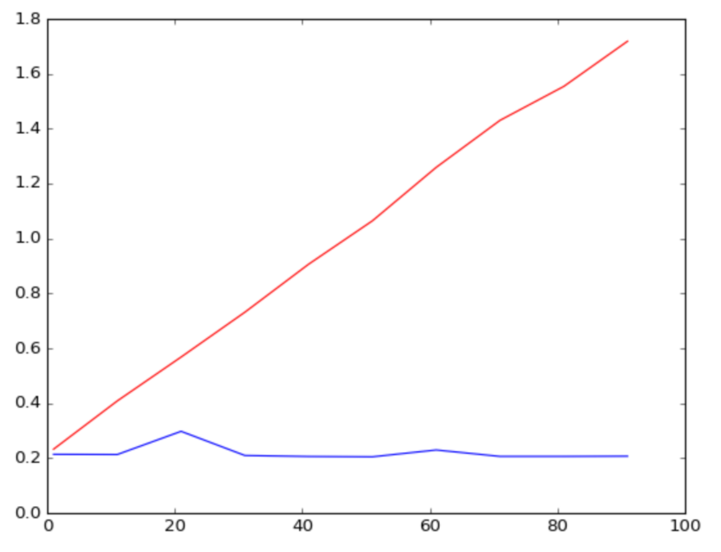
Experiments?

```
In [5]: timeit.timeit(stmt='1 in A', setup='A = list(range(2, 300))')

Out[5]: 4.774117301917343

In [6]: timeit.timeit(stmt='1 in A', setup='A = set(range(2, 300))')

Out[6]: 0.0499541983165841
```



Efficiency

The algorithm should use resources efficiently

How do you measure time?

Extrinsic factors: computer system, programming language, compiler, skill of programmer, other programs …

➡ implementing an algorithm, running it on a particular machine and input, and measuring time gives very little information

Efficiency analysis

Two components:

1. Determine running time as function $T(n)$ of input size n
2. Characterize rate of growth of $T(n)$

Focus on the order of growth *ignore all but the most dominant terms*

Examples

Algorithm A takes $50n + 125$ machine cycles to search a list

- $50n$ dominates 125 if $n \geq 3$, even factor 50 is not significant
  ➡ the running time of algorithm A grows linearly in n

Algorithm B takes $20n^3 + 100n^2 + 300\,n + 200$ machine cycles

➡ the running time of algorithm B grows as n^3

Recall (maybe): Logarithms

$\log n$ denotes $\log_2 n$

We have for $a, b, c > 0$:

1. $\log_c (ab) = \log_c a + \log_c b$

2. $\log_c (a^b) = b \log_c a$

3. $\log_a b = \log_c b \,/\, \log_c a$

Comparing orders of growth

$\log^{35} n$ vs. $\sqrt{n}$?

- logarithmic functions grow slower than polynomial functions
- $\lg^a n$ grows slower than n^b for all constants $a > 0$ and $b > 0$

n^{100} vs. 2^n ?

- polynomial functions grow slower than exponential functions
- n^a grows slower than b^n for all constants $a > 0$ and $b > 1$

The Big-O…

An algorithm runs in $O(g)$ time if the number of steps it executes on an input of size n is at most proportional to $g(n)$ when n is large.

Notes about the use of O notation:

- Constant factors are not important
  - So $O(n^2)$ is the same as $O(5n^2)$ or $O(n^2/2)$.
- Added/subtracted lower order terms can also be ignored
  - So $O(n^2 + 5n + 4)$ and $O(n^2 + n\log n - 2)$ are the same as $O(n^2)$.

Time Complexity

Some typical time complexities (ordered from slower to larger growth):

- $O(1)$: constant
- $O(\log n)$: logarithmic
- $O(n)$: linear
- $O(n \log n)$
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(n^k)$: polynomial
- $O(2^n)$, $O(3^n)$, $O(k^n)$: exponential
- $O(n!)$: factorial

In general: Order of growth of some common functions

$O(1) < O(\log n) < O(n) < O(n * \log n) < O(n^2) < O(n^k) < O(2^n) < O(n!)$

Exercise

Compare growth rates

Practice Exercises 1

Exercise 1

Rank the following functions of n by order of growth (starting with the slowest growing). Functions with the same order of growth should be ranked equal.

$\log n^3$, n, $n \log n$, 4^n , $\log \sqrt{n}$, $n + \log n^4$, $2^{\log 16}$, n^{-1}, 16, $n^{\log 4}$

Some typical time complexities:

- $O(1)$: constant
- $O(\log n)$: logarithmic
- $O(n)$: linear
- $O(n \log n)$
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(n^k)$: polynomial
- $O(2^n)$, $O(3^n)$, $O(k^n)$: exponential

What is the run-time complexity of an algorithm with following g(n) ?

$$g(n) = n^2 + (n\log n)^4 + 2^n + (n!)$$

A) $\mathbf{n^2}$

B) $\mathbf{2^n}$

C) $\mathbf{n!}$

D) $\mathbf{n^2 + 2^n}$

Describing algorithms

A complete description of an algorithm consists of three parts:

1. the algorithm
   - *expressed in whatever way is clearest and most concise,*
   - *can be English and / or "readable code",*
   - *readable: pseudo-code or python code*
   - *code will nearly always need a short high-level description in words*
2. a proof of the algorithm's correctness
3. a derivation of the algorithm's running time

Searching

Linear Search – Pseudo-Code

Linear-Search(A, n, x)

Input and Output specification

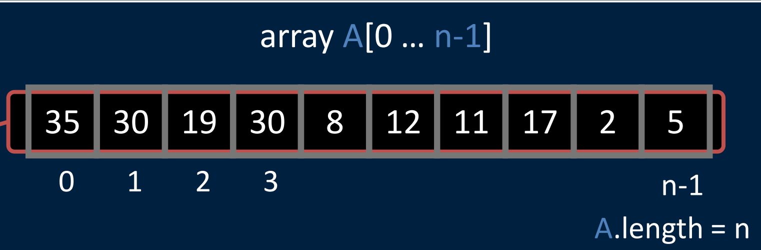
Input:
- *A: an array*
- *n: the number of elements in A to search through*
- *x: the value to be searched for*

Output: Either an index i for which A[i] = x, or Not-Found

1. Set answer to Not-Found
2. For each index i, going from 0 to n-1, in order:
   A. If A[i] = x, then set answer to the value of i
3. Return the value of answer as the output

Linear Search

Linear-Search(A, n, x)



Input:

- *A: an array*
- *n: the number of elements in A to search through*
- *x: the value to be searched for*

Output: Either an index i for which A[i] = x, or Not-Found

1. Set answer to Not-Found
2. For each index i, going from 0 to n-1, in order:
   A. If A[i] = x, then set answer to the value of i
3. Return the value of answer as the output

Linear Search in Python

```
In [6]: def linear_search(A, x):
            answer = -1
            for i in range(0, len(A)):
                if A[i] == x: answer = i
            return answer

In [7]: linear_search([10, 5, 9, 9], 10)

Out[7]: 0

In [8]: linear_search([10, 5, 9, 9], 9)

Out[8]: 3

In [9]: linear_search([10, 5, 9, 9], 8)

Out[9]: -1
```

What do you notice?

Linear Search

Linear-Search(A, n, x)

Input:
- *A: an array*
- *n: the number of elements in A to search through*
- *x: the value to be searched for*

Output: Either an index i for which A[i] = x, or Not-Found

1. Set answer to Not-Found
2. For each index i, going from 0 to n-1, in order:
   A. If A[i] = x, then set answer to the value of i
3. Return the value of answer as the output

Loop with variable i

Body of the loop

- *This loop always runs until n-1. Is that necessary?*
- *What happens if there is more than one cell with the same value we are searching for?*

Linear Search

Better-Linear-Search(A, n, x)

35	30	19	30	8	12	11	17	2	5
0	1	2	3						n-1

Input:

- *A: an array*
- *n: the number of elements in A to search through*
- *x: the value to be searched for*

Output: Either an index i for which A[i] = x, or Not-Found

1. For i = 0 to n-1:
   A. If A[i] = x, then return the value of i as the output
2. Return Not-Found as the output

Binary Search

Search for the number 51 ?

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

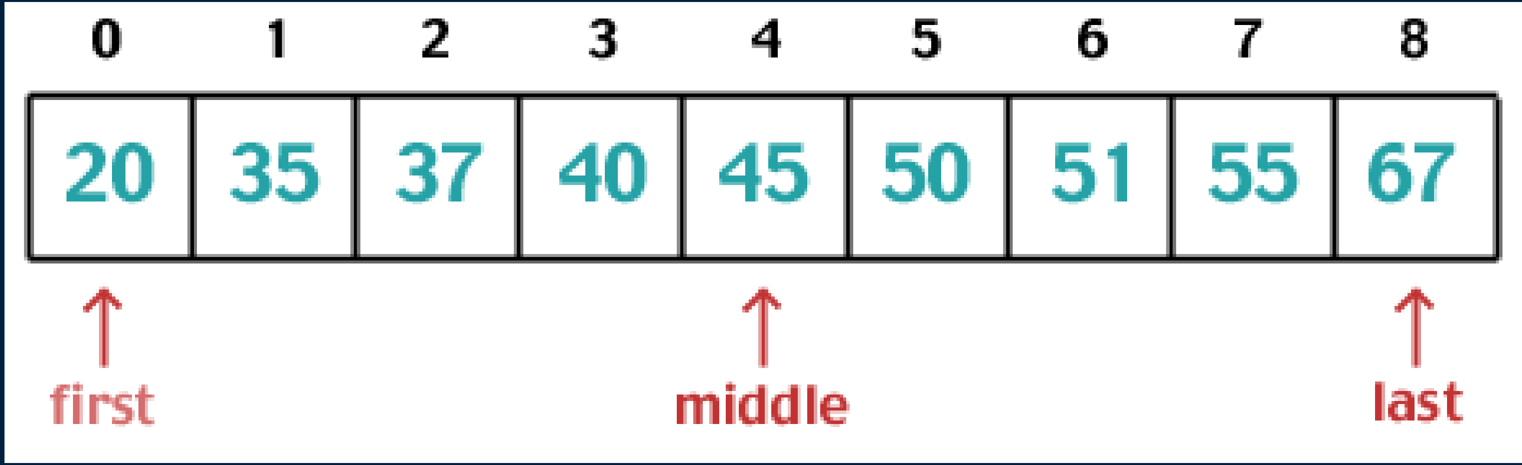
Binary Search Contd…

Compare 51 with middle

if 51 == middle then “Hurray”

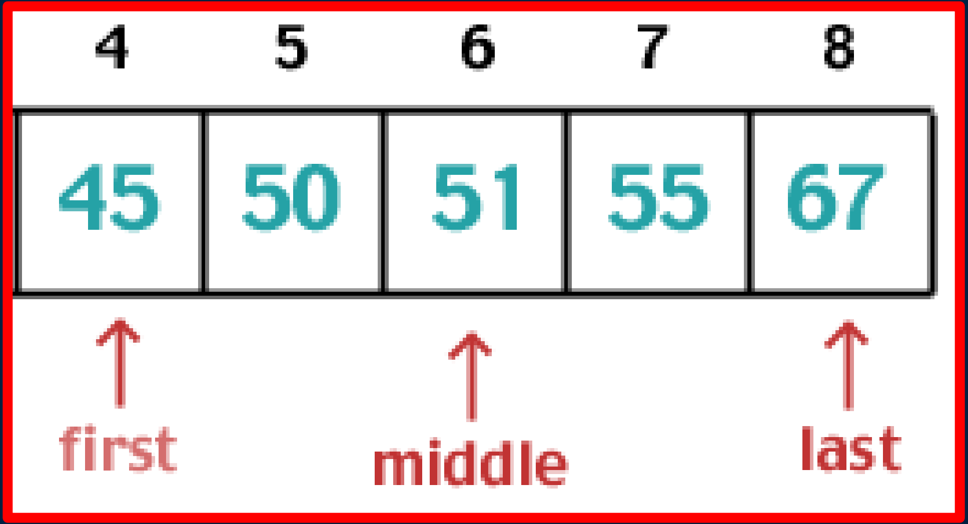
if 51 < middle then between first and middle

if 51 > middle then between middle and last



Binary Search Contd..

Compare 51 with middle
if 51 == middle then "Hurray"



Guess the O( ) for Binary search ?

A) O(n)

B) O(log n)

C) O($\sqrt{n}$)

D) O(1)

Take home messages…

- Understand notion of algorithm and data structure
- Analyze algorithm for time complexity, correctness, and efficiency
- Compare algorithms on the basis on Big O
- Design a linear search algorithm
- Improve the linear search algorithm
- Learn the Binary Search algorithm


UBC
THE UNIVERSITY OF BRITISH COLUMBIA