Analysis of Algorithm Efficiency

- Compare ***efficiency*** of algorithm
  - o Example: Searching an ordered array of integers

- How to compare?
  - o algorithm's ***execution time increases with the input size***
    - \- Example: *array size* for Searching
  - o difficult to get a precise measure
  - ***o approximate***
    - \- largest contributing factor

$$2n^2 + 5n - 7$$

Big-O notation: $O\left(n^2\right)$

`log n`	`n`	`2n`	`n*n`
1	2	4	4
2	4	8	16
3	8	16	64
4	16	32	256
5	32	64	1024
6	64	128	4096
7	128	256	16384
8	256	512	65536

The Complexity of the Search Problem

- We can describe the size of a searching problem by the number of comparison operations needed to find the one we want.
- This is also called the complexity of the searching algorithm.
- There are three cases to be considered:
  - Best-case
  - Worst-case
  - Average-case

Linear Search

Problem Definition:

Search an *ordered* array of integers for a value and return its index if the value is found. Otherwise, return -1.

```
// Searches an ordered array of integers
int lsearch(int data[], // input: array
            int size,   // input: array size
            int value   // input: value to find
            )    // output: index if found
                 //         otherwise return -1
{
     int index = 0;    // initialize index to 0

     while ((data[index] < value) &&
            (index < size))
          index++;

     if (data[index] != value)
          return -1;
     else
          return index;
} // search
```

- Given an array of size n, what is the *average* number of elements the algorithm has to examine?

Binary Search

Same Problem Definition:

Search an *ordered* array of integers for a value and return its index if the value is found. Otherwise, return -1.

0	1	2	3	4	5	6	7
7	9	10	17	30	37	46	60

0	1	2	3	4	5	6	7
7	9	10	17	30	37	46	60

Binary Search (Cont'd)

- Binary search is based on the divide and conquer concept and works as follows:
  - Start by looking at the middle component.
    - If the value it holds is too high, go to the middle of the bottom half.
    - If the value it holds is too low, go to the middle of the upper half.
  - Repeat this 'split the remainder' until you find the value you want, or decide that it does not exist.

Solution Outline (Algorithm):

```
Set lower and upper bound;

While value not found and more to search do begin

  Compute subscript of middle element
    with repect to lower and uppper bound;

  If middle element equals value (found) then
    return middle element position

  Else if lower is greater than upper bound then
    (no more to search)
    return position = -1

  Else if value is less than middle element then
    set upper bound to middle element position-1
  Else
    set lower bound to middle element position+1

End;
```

Binary Search (Cont'd)

```
// Searches an ordered array of integers
int binary(int data[], // input: array
           int size,   // input: array size
           int value   // input: value to find
           )    // output: index if found
                //         otherwise return -1
{
     enum state {searching,found,absent} status;
     int lower, middle, upper, index;

     status = searching;
     lower = 0;
     upper = size - 1;
     index = -1;

     while (status == searching) {
          middle = (lower + upper) / 2;
          if (data[middle] == value) {
               status = found;
               index = middle;
          } else if (lower > upper) {
               status = absent;
          } else if (value < data[middle]) {
               upper = middle - 1;
          } else {
               lower = middle + 1;
          }
     }
     return index;
} // binary
```

- Given an array of size n, what is the *average* number of elements the algorithm has to examine?

The Struct Type

- ☛ Scientific problems tend to revolve around just two data types: real and integer. For such application arrays work well because they carry instants of <u>homogeneous</u> (same) type of data.

- ☛ For non-scientific problems, special kinds of structure are required that use <u>heterogeneous</u> (different types) types of data. These different types of data can be related and are packed together.

- ☛ Very often, arrays of structs are used together to take advantage of both arrays and structs.

What is a *struct* Type

- *struct* is a data type that can have different data types in it.

- The values in a *struct* type can be of different type - basic or enumerated. Therefore, a struct is heterogeneous in that it can be composed of different kinds of values. In contrast, array is homogeneous since it can contain only one type of values.

- A struct type is often called record in database applications.

- Individual components of a struct type are called members.

- members can be of different types (simple, array or struct)

- A struct is named as a whole and then individual members are named using identifiers.

Syntax of *struct* type

```
struct <struct-type> {
    <type> <id-list>;
          •••
    <type> <id-list>;
};
<struct-type> <struct-variable>
```

Example:

```
struct StudentRecord {
    char Name[15];
    int Id;
    char Department[5];
    char sex;
};  // StudentRecord

StudentRecord Student1, Student2;
```

Two variables, *Student1* and *Student2*, of *StudentRecord* type are defined; each has four fields: *Name, Id, Department and sex*.

These fields are being referred to as:

Student1.Name, Student2.Name,
Student1.Id, Student2.Id
Student1.Department, Student2.Department
Student1.sex, Student2.sex

Access to struct members

- Using the member access operator , a period
  - Syntax

```
<struct-variable>.<member name>
```

  - Example

```
strcpy (Student.Name, "CHAN Tai Man");
Student.Id = 12345;
strcpy (Student.Department, "COMP");
Student.sex = ’M’;


cout << "The student is ";
switch (Student.sex) {
    case ’F’: case ’f’:
      cout << "Ms. ";
      break;
    case ’M’: case ’m’:
      cout << "Mr. ";
      break;
}
cout << Student.Name << endl;
```

struct-to-struct Assignment

- both structs must be of the same type
  - Example

```
StudentRecord Student1, Student2;

strcpy (Student1.Name, "CHAN Tai Man");
Student1.Id = 12345;
strcpy (Student1.Department, "COMP");
Student1.sex = 'M';

Student2 = Student1;
```

  - More Example

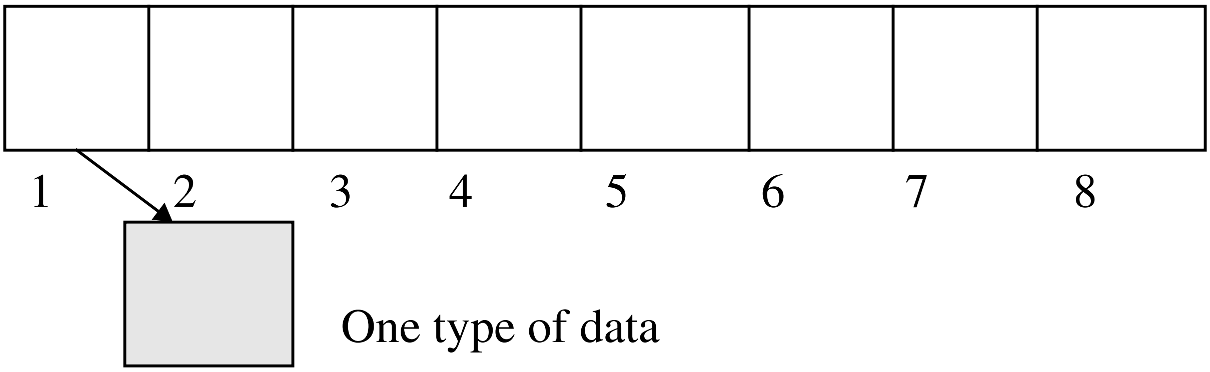
```
StudentRecord Class[200];

strcpy (Class[10].Name, "CHAN Tai Man");
Class[10].Id = 12345;
strcpy (Class[10].Department, "COMP");
Class[10].sex = 'M';

Class[0] = Class[10];
```

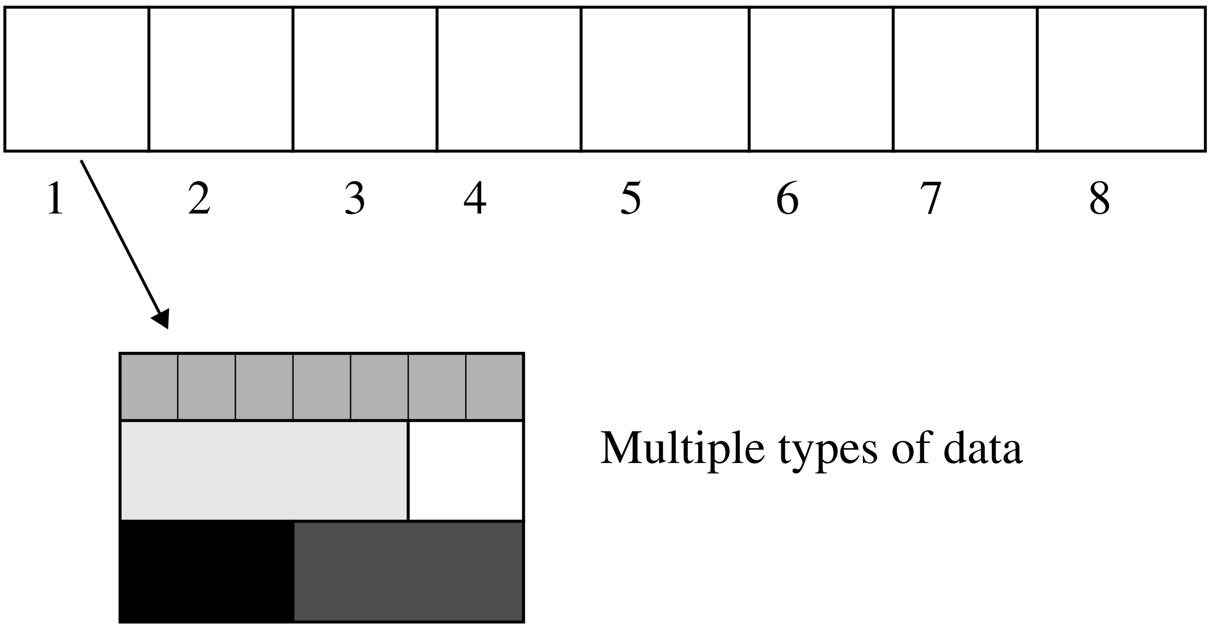
The difference between an ordinary array and array of structs

An ordinary array



One type of data

An array of structs



Multiple types of data

Nested Structure

- More Example

```
struct point {
    float X,Y;
};

// a line is defined as two points
struct line {
    point p1,p2;
}

// a triangle is defined by three points
struct triangle {
    point p1,p2,p3;
};

// a square is defined by
// an array of four points
struct square {
    point vertex[4];
};

point P;
line L;
triangle polygon3;
square polygon4;
```

- Access to members of nested structure

```
L.p1.x = 1.0;   L.p1.y = 2.0;

polygon4.vertex[0].x = 1.0;
polygon4.vertex[0].y = 2.0;
```

Given the following definitions for **Date, Student, Classlist** and **Class102**:

```
struct Date {
    int day;            // day of a Date     ASSUME: 1 <= day <= 31
    int month;          // month of a Date   ASSUME: 1 <= month <= 12
    int year;           // year of a Date    ASSUME: 1 <= year <= 2000
} ;

struct Student {
    char name[20];      // Student’s name
    Date birthdate;     // Date of a student’s birthday
    int score[5]        // Student’s test scores
};

struct Classlist {
    Student clist[50];  // Current list of students in the class
    int no_tests;       // Number of tests students have taken
    int size;           // Number of students currently enrolled
};

Classlist Class102;
```

```
// The function in_class reads the student records from the input stream ins
// and stores them in the Classlist Class.

void in_class (ifstream& ins, Classlist& Class) {
    char temp, j;

    for (int i=0; i<Class.size; i++) {          // Read student records into Class
        ins.get(temp);                          // Read student names
        j = 0;
        while (temp != '/' && j < 19) {         // Student names ends with '/'
            Class.clist[i].name[j] = temp;
            ins.get(temp);
            j++;
        }
        Class.clist[i].name[j] = '\0';          // End the string with null char

        ins >> Class.clist[i].birthdate.day;    // Read day of birth
        ins >> Class.clist[i].birthdate.month;  // Read month of birth
        ins >> Class.clist[i].birthdate.year;   // Read year of birth

        for (int k=0; k < Class.no_tests; k++)  // Read in scores
            ins >> Class.clist[i].score[k];

        ins.get(temp);                          // Read newline character
    }
}
```

Input file:

```
Chan Tai Man/ 10 9 1975 60 70 80
Cheng Hong Chung/ 5 8 1976 70 84 65
Chung Lok Hang/ 12 7 1973 81 87 98
Fung Kwok Kay/ 26 12 1972 100 100 100
Lam Yuk Yee/ 8 5 1971 91 86 83
Poon Wai Wah/ 7 8 1975 56 77 74
Pang Wing Sum/ 30 8 1973 84 59 74
Tam Ka Hung/ 19 11 1975 100 95 99
Wong Tak Sun/ 20 3 1974 83 54 71
Wong Siu Wing/ 4 5 1972 75 85 95
```