

Basic Concepts

Overview & Algorithm Specification

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2025



Outline

- 1 System Life Cycle
- 2 Algorithm Specification
 - Recursive Algorithms
- 3 Data Abstraction



Outline

- 1 System Life Cycle
- 2 Algorithm Specification
 - Recursive Algorithms
- 3 Data Abstraction



System Life Cycle

- Requirements.
- Analysis.
 - Bottom-up vs. top-down.
- Design.
 - Data objects.
 - Operations.
- Refinement and coding.
- Verification.
 - Correctness **proofs**.
 - Testing.
 - Error removal (debugging).



Requirement

- Purpose of the project.
- We should rigorously define the input and the output.



Analysis

- Bottom-up:
 - Older and unstructured.
 - Due to not having a master plan for the project, the resulting program frequently has many loosely connected, error-ridden segments.
- Top-down:
 - Begin with the purpose.
 - Use the end (purpose) to divide the program into manageable segments.
 - Generate diagrams that are used to design the system.



Design

- The designer starts to approach the system...
- The data **objects** that the program needs and the **operations** performed on them.
- For example, consider a scheduling system for a university.
 - Data objects: students, courses, assistants, professors, etc.
 - Operations: inserting, removing, searching within each object or between them.
- We postpone the implementation decisions because the abstract data types and the algorithms specifications are **language-independent**.



Outline

- 1 System Life Cycle
- 2 Algorithm Specification
 - Recursive Algorithms
- 3 Data Abstraction



Algorithm Specification

Algorithm

An algorithm is a finite set of instructions that accomplishes a particular task.

Criteria of an algorithm

- Input.
- Output.
- Definiteness (clear & unambiguous).
- Finiteness* (terminate after a finite number of steps).
- Effectiveness (each instruction must be basic enough to be carried out).



Example: Selection Sort

- **Goal:** Sort a set of n (unsorted) integers.



Example: Selection Sort

- **Goal:** Sort a set of n (unsorted) integers.
- A solution:



Example: Selection Sort

- **Goal:** Sort a set of n (unsorted) integers.
- A solution: Find the smallest and place it next in the sorted list.

What's the issue/problem here?



Example: Selection Sort

- **Goal:** Sort a set of n (unsorted) integers.
- A solution: Find the smallest and place it next in the sorted list.

What's the issue/problem here?

- How the integers are **initially stored**?
- **Where** should we place **the result**?



Selection Sort

- Assume that the integers are stored in an **array** 'list', such that the i th integer is stored in the i th position `list[i]`.

```
for (i = 0; i < n; i++) {  
    Examine list[i] to list[n-1] and suppose that the smallest  
    integer is at list[min];  
    Interchange list[i] and list[min];  
}
```

⇒ sample code.

```

#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 101

void SWAP(int *x, int *y) {
    *x = *x^*y; *y = *x^*y; *x = *x^*y;
}

void sort(int [], int ); /* 選擇排序 */

int main() {
    int i, n;
    int list[MAX_SIZE];
    scanf("%d", &n); /* 多少輸入值? */
    for (i = 0; i < n; i++) { /* 隨機產生 */
        list[i] = rand() % 1000;
        printf("%d ", list[i]);
    }
    printf("\n");
    sort(list, n);
    for (i = 0; i < n; i++) printf("%d ", list[i]);
    return 0;
}

```

```

void sort(int list[], int n) {
    int i, j, min, temp;
    for (i = 0; i < n; i++) {
        min = i;
        for (j = i+1; j < n; j++) {
            if (list[j] < list[min])
                min = j;
        }
        if (i != min) {
            SWAP(&list[i], &list[min]);
        }
    }
}

```

Example: Binary Search

- **Goal:** Searching in a sorted list.

```
while (there are more integers to check) {  
    middle = (left + right) / 2;  
    if (searchnum < list[middle])  
        right = middle - 1;  
    else if (searchnum == list[middle])  
        return middle;  
    else  
        left = middle + 1;  
}
```



Example

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	3	11	15	20	23	29	31	35	36	43	47	49	50	53	56
	3	11	15	20	23	29	31	35	36	43	47	49	50	53	56
	3	11	15	20	23	29	31	35	36	43	47	49	50	53	56

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
	3	11	15	20	23	29	31	35	36	43	47	49	50	53
	3	11	15	20	23	29	31	35	36	43	47	49	50	53
	3	11	15	20	23	29	31	35	36	43	47	49	50	53
	3	11	15	20	23	29	31	35	36	43	47	49	50	53

sample code



```
int binSearch(int list[], int target, int left, int right) {  
    /* return its position if found. Otherwise return -1 */  
    int middle;  
    while (left <= right) {  
        middle = (left + right)/2;  
        if (list[middle] < target) {  
            left = middle + 1;  
        } else if (list[middle] == target)  
            return middle;  
        else  
            right = middle - 1;  
    }  
    return -1;  
}
```



Outline

- 1 System Life Cycle
- 2 Algorithm Specification
 - Recursive Algorithms
- 3 Data Abstraction







Recursion

- Direction recursion.
 - Functions can call themselves.
- Indirect recursion.
 - Functions may call other functions that invoke the calling function again.

```
void recurse() {  
    ...  
    recurse();  
    ...  
}  
  
int main() {  
    ...  
    recurse();  
    ...  
}
```



Benefits of using recursion

- Extremely powerful/elegant
- It allow us to express an otherwise complex process in very clear term usually.

Example:



Benefits of using recursion

- Extremely powerful/elegant
- It allow us to express an otherwise complex process in very clear term usually.

Example:
$$\binom{n}{m} = \binom{n}{m-1} + \binom{n-1}{m-1}.$$



Another Example: Fibonacci Sequence

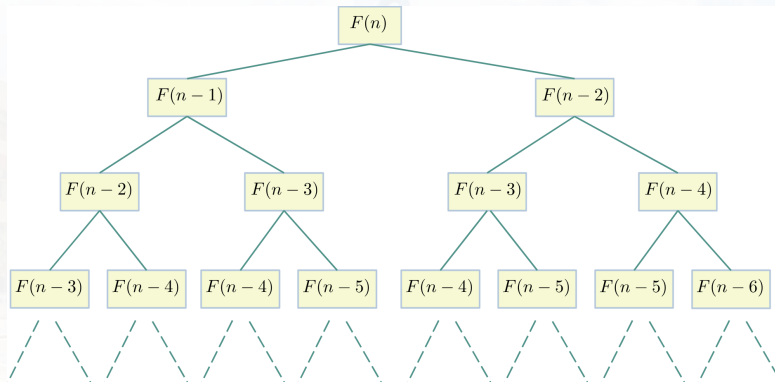
- $F(n) = F(n-1) + F(n-2)$, for $n \geq 2$.
 $F(0) = 0, F(1) = 1$: boundary conditions.



Another Example: Fibonacci Sequence

- $F(n) = F(n-1) + F(n-2)$, for $n \geq 2$.
 $F(0) = 0, F(1) = 1$: boundary conditions.
- However, a recursive algorithm for computing $F(n)$ given an arbitrary n is NOT a good idea. ☹️





Recursive Binary Search

```
int binSearch(int list[], int target, int left, int right) {  
    /* return its position if found. Otherwise return -1 */  
    int middle;  
    while (left <= right) {  
        middle = (left + right)/2;  
        if (list[middle] < target) {  
            return binSearch(list, target, middle+1, right);  
        } else if (list[middle] == target)  
            return middle;  
        else  
            return binSearch(list, target, left, middle-1);  
    }  
    return -1;  
}
```



Outline

- 1 System Life Cycle
- 2 Algorithm Specification
 - Recursive Algorithms
- 3 Data Abstraction



Data Abstraction

Data Type:

A collection of

- objects.
- a set of operations that act on the objects.



Data Abstraction

Data Type:

A collection of

- objects.
- a set of operations that act on the objects.

Recall what we have learned in C/C++ courses.

- The data types in C:
 - **Basic** types: char, int, float, double,



Data Abstraction

Data Type:

A collection of

- objects.
- a set of operations that act on the objects.

Recall what we have learned in C/C++ courses.

- The data types in C:
 - **Basic** types: char, int, float, double, ...
 - **Group** data types: array, struct, ...



Data Abstraction

Data Type:

A collection of

- objects.
- a set of operations that act on the objects.

Recall what we have learned in C/C++ courses.

- The data types in C:
 - **Basic** types: char, int, float, double,
 - **Group** data types: array, struct, ..., **Pointer** data types.
 - **User-defined** types.



Example

```
struct student {  
    char last_name;  
    int student_id;  
    float grade;  
};
```



Abstract Data Type

Abstract Data Type (ADT):

A data type that is organized in such a way that the **specification** of the objects and the **operations** on the objects is separated from the **representation** of the objects and the **implementation** of the operations.

- We know what it does, but not necessarily how it will do it.



Abstract Data Type

Abstract Data Type (ADT):

A data type that is organized in such a way that the **specification** of the objects and the **operations** on the objects is separated from the **representation** of the objects and the **implementation** of the operations.

- We know what it does, but not necessarily how it will do it.
- Example in C++:



Abstract Data Type

Abstract Data Type (ADT):

A data type that is organized in such a way that the **specification** of the objects and the **operations** on the objects is separated from the **representation** of the objects and the **implementation** of the operations.

- We know what it does, but not necessarily how it will do it.
- Example in C++: **class**.



Abstract Data Type

Abstract Data Type (ADT):

A data type that is organized in such a way that the **specification** of the objects and the **operations** on the objects is separated from the **representation** of the objects and the **implementation** of the operations.

- We know what it does, but not necessarily how it will do it.
- Example in C++: **class**.
- The nature of an ADT argues that we avoid implementation details. Therefore, we will usually use a form of structured English to explain the meaning of the functions.



ADT in C

- struct.
- the functions that operate on the ADT defined separately from the struct.

```
struct Triangle {  
    double a;  
    double b;  
    double c;  
};  
  
int main() {  
    Triangle t1 = { 3, 4, 5 };  
    Triangle t2 = { 3, 3, 3 };  
}
```



ADT in C

```
double perimeter(const Triangle *tri) {  
    return tri->a + tri->b + tri->c;  
}  
  
void scale(Triangle *tri, double s) {  
    tri->a *= s;  
    tri->b *= s;  
    tri->c *= s;  
}  
  
int main() {  
    Triangle t1 = { 3, 4, 5 };  
    scale(&t1, 2);  
    cout << perimeter(&t1) << endl;    // 6+8+10 = 24  
}
```



Discussions

