Basic Concepts (chapter 1)

Our focuses in this chapter:

- Specification of **Abstract Data Types (ADTs)**
- Building algorithms
  - Goal → Idea → Design
  - The issue of **Recursion** vs. **Loop**
  - Performance analysis (**complexity**)

Abstract Data Types (ADTs)

- Data Type Specification
  - Objects
  - Operations
- **Abstract Data Types**: Data types specified in a way that is independent of the *representation* of the objects and the *implementation* of the operations.

Representation is how the actual "data" are stored or organized in the implementation of the data type.

Example:

Abstraction: object x is an integer (in common sense)

Representation: object x is of type **`int`** (as in C)

Abstract Data Types (ADTs)

Example: Let us consider the similarities and differences between a standard HDD and a USB flash drive, etc.:

	ADT		**HDD**	**USB**
Objects		Representations		
Operations		Implementations		

Specifying Operations of ADTs

- Names of functions (operations)
- For each function, we need to specify:
  - Types of arguments
  - Types of results
  - Descriptions of what the functions do (without implementation details)

ADT Example: Natural Numbers

ADT *NaturalNumber* ***is***

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (MAXINT) on the computer

functions:

for all $x, y \in$ *NaturalNumber*; TRUE, FALSE $\in$ *Boolean*

and where +, −, <, ==, and = are the usual integer operations

Zero(): *NaturalNumber* ::= 0

IsZero(x): *Boolean* ::= if (x==0) return TRUE else return FALSE

Add(x,y): *NaturalNumber* ::= if (x+y <= MAXINT) return x+y else return MAXINT

Equal(x,y): *Boolean* ::= if (x==y) return TRUE else return FALSE

Successor(x): *NaturalNumber* ::= if (x == MAXINT) return x else return x+1

Subtract(x,y): *NaturalNumber* ::= if (x<y) return 0 else return $x-y$

end *NaturalNumber*

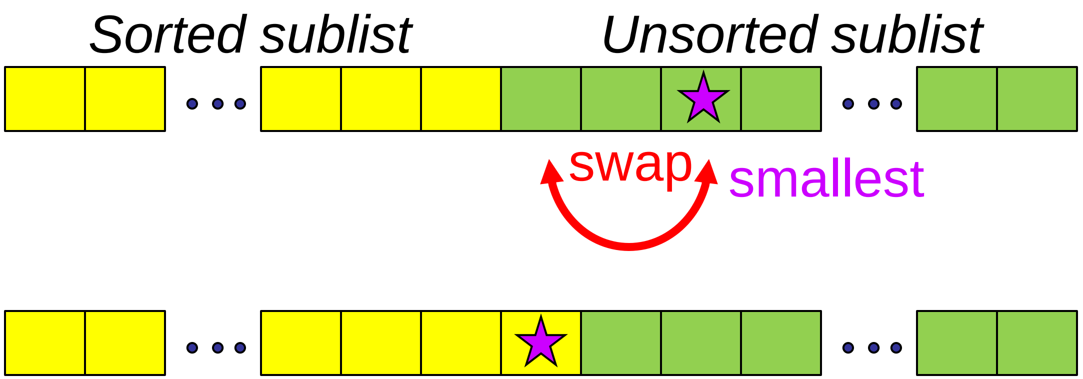
Algorithm Criteria

- Input
- Output
- Definiteness
- Finiteness
- Effectiveness

Programs ≠ Algorithms
A program doesn’t have to be finite (e.g. OS scheduling).

Example Algorithm: Selection Sort

- Goal: To sort a list of integers, small to large
- Idea: From those integers that are unsorted, find the smallest one and place it right after the current sorter sublist.



- Design
  - Integers in an array **`a[0]`** to **`a[n-1]`**
  - In each iteration (**`i`** from **`0`** to **`n-1`**), find the smallest element in **`a[i]`** to **`a[n-1]`** and swap it with **`a[i]`**.

Example Algorithm: Selection Sort

A C-like form of the algorithm so far

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

C++ function for the algorithm

```
void SelectionSort(int a[ ], int n)
{
   for (int i = 0; i < n; i++) {
      int j=i;
      for (k = i+1; k < n; k++) {
         if (a[k] < a[j])
            j = k;
      }
      swap(a[i], a[j]);
   }
}
```

Example Algorithm: Binary Search

- Goal: To determine the existence and location of a particular integer within a sorted list of distinct integers.
  - If it's present, return the index.
  - Otherwise, return -1.
- Idea:
  - Let `left` and `right` be the range of indices to be searched.
  - Each time compare `x` (the number to be searched) with the entry at `middle`, which is half-way between `left` and `right`.
  - If `x == a[middle]`, return `middle`
  - If `x < a[middle]`, set `right` to `middle-1`
  - If `x > a[middle]`, set `left` to `middle+1`

Example Algorithm: Binary Search

A C-like form of the algorithm so far

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

C++ function for the algorithm

```
int BinarySearch(int a[], int x, int n)
{
  int left=0, right=n-1;
  while (left <= right)  {
    int middle = (left + right) / 2;
    if (x < a[middle]) right = middle -1;
    else if (x > a[middle]) left = middle+1;
    else return middle;
  }
  return -1;
}
```

Example Algorithm: Binary Search

OK, now let's go through an example here:

0 2 4 6 8 10 12 14 16 18 20 22 24 26 28

(1) Find 6; (2) find 25

Recursive Algorithms

- A recursive algorithm involves a function calling itself, either directly or indirectly.
- When to use recursion: We can obtain the result of a task using results from subtasks that are both ***similar*** and ***simpler***.
- **Boundary condition**: When the task becomes so simple, the function generates the result directly so that the recursion terminates.
- Any task implemented using loops can be implemented using recursion. You just need to choose the method based on efficiency and clarity.

Example Recursive Algorithms

- Recursive addition:

```
int RecursiveAdd(int a[ ], int n)
{
  if (n == 0) return a[0];
  else return a[n] + RecursiveAdd(a, n-1);
}
```

- Recursive multiplication:

```
int RecursiveMultiply(int a[ ], int n)
{
  if (n == 0) return a[0];
  else return a[n] * RecursiveMultiply(a, n-1);
}
```

Recursive Binary Search

```
int BinarySearch(int a[], int x, int left, int right)
{
  if (left <= right)  {
    int middle = (left + right) / 2;
    if (x < a[middle])
      return BinarySearch(a, x, left, middle-1);
    else if (x > a[middle])
      return BinarySearch(a, x, middle+1, right);
    else return middle;
  }
  return -1;
}
```

Q: Identify the "boundary conditions" in this code.

Recursive Permutation Generator

To generate all the permutations of the string **ABCD**:

ABCD
ABDC
ACBD
ACDB
… (24 in total)

We can do

A followed by all the permutations of **BCD**
B followed by all the permutations of **ACD**
C followed by all the permutations of **ABD**
D followed by all the permutations of **ABC**

Recursive Permutation Generator

```
void perm(char *a, int i, int n)
{
  // generate all the permutations of a[i] to a[n]
  if ( i == n) {
    for (j=0; j <= n; j++) cout << a[j];
  }
  else {
    for (j = i; j <= n; j++) {
      swap(a[i], a[j]);
      perm(a, i+1, n);
      swap(a[i], a[j]);
    }
  }
}
```

Q: Identify the "boundary conditions" in this code.

Q: Why do we need the second **swap**?

Recursive Permutation Generator

Now let's generate all the permutations of the string "ABC":

Perm("ABC",0,2)

j=0: A ↔ A
perm("ABC",1,2)
A ↔ A
j=1: A ↔ B
perm("BAC",1,2)
B ↔ A
j=2: A ↔ C
perm("CBA",1,2)
C ↔ A

j=1: B ↔ B
perm("ABC",2,2)
B ↔ B
j=2: B ↔ C
perm("ACB",2,2)
C ↔ B

j=1: A ↔ A
perm("BAC",2,2)
A ↔ A
j=2: C ↔ A
perm(" BCA",2,2)
A ↔ C

j=1: B ↔ B
perm("CBA",2,2)
B ↔ B
j=2: A ↔ B
perm("CAB",2,2)
B ↔ A

print out "ABC"

print out "ACB"

print out "BAC"

print out "BCA"

print out "CBA"

print out "CAB"

Recursion vs. Loop

- Disadvantages of using recursion:
  - Recursion has higher runtime overhead than loops (extra space and time for each function call).
  - It usually takes more work to debug recursive codes.
- As a result, recursions are mostly used when
  - The subtask is ***much simpler*** than the original task (so there will not be too many levels of recursion).
  - The task is ***more straightforward*** for recursion than for loops (e.g., permutation generation).

Recursion vs. Loop

- `RecursiveAdd` and `RecursiveMultiply`:
  - It's easier to understand using loops, and the subtask is only slightly simpler.
  - The overhead makes these recursive functions impractical. (The program will likely crash if you have, say, a billion items.)
- Binary search
  - The subtask is *much simpler* (half of the original size on average).
  - The task is more straightforward using recursion.

Performance Evaluation

- **Performance Analysis**
  - Estimation (from the algorithm, code)
  - Machine-independent
- **Performance Measurement**
  - Measurement (actual testing)
  - Machine-dependent

Performance Analysis

- **Space complexity**: amount of memory used
  - $S(P) = c + S_P(I)$
  - c: fixed space
  - $S_P(I)$: depends on "instance characteristics"
- **Time complexity**: amount of computer time
  - $T(P) = c + T_P(I)$
  - c: fixed time
  - $T_P(I)$: depends on "instance characteristics"
- **Instance characteristics**: Amounts and other properties of inputs and outputs that may affect the space or time requirements.

Program Step Count

- Program step (a loose definition): A meaningful segment in a program, with its execution time independent of the instance characteristics.

- Example:

```
float sum(float list[ ], int n)
{
  float tempsum = 0;            <- 1 step
  int i;
  for (i = 0; i < n; i++) {     <- n+1 steps
    tempsum += list[i];         <- n steps
  }
  return tempsum;               <- 1 step
}
```

Total: 2n+3 steps

Step Count: One More Example

The step count table for matrix addition:

steps/execution

Statement	s/e	Frequency	#Steps
void add (int a[ ][MAX_SIZE]•••)	**0**	**0**	**0**
{	**0**	**0**	**0**
int i, j;	**0**	**0**	**0**
for (i = 0; i < row; i++)	**1**	**rows+1**	**rows+1**
for (j=0; j< cols; j++)	**1**	**rows•(cols+1)**	**rows•cols+rows**
c[i][j] = a[i][j] + b[i][j];	**1**	**rows•cols**	**rows•cols**
}	**0**	**0**	**0**
Total	2rows * cols + 2 rows + 1		

Beyond Step Counts

- Step counts are of limited practical use because
  - A "program step" is often ambiguous.
  - Different steps take different amounts of execution time.
- **Asymptotic complexity**: How the amount of computation grows with the amount of data
  - **Big-O notation** (upper bound)
  - **Omega notation** (lower bound)
  - **Theta notation** (both upper and lower bounds)
- Three types of cases
  - Worst case
  - Best case
  - Average case

Big-O Notation

$$f(n) = O(g(n)) \Leftrightarrow \exists c > 0, n_0 > 0, s.t. \forall n \geq n_0, \ f(n) \leq cg(n)$$

Here $f(n)$ is the step count.

Examples:

- $f(n) = 3n+2$
- $f(n) = 3n^3+2n^2+100$
- $f(n) = 5\log(n)+n$

Big-O notation is the "upper bound" of complexity.

We should always use the "most strict" one, i.e., the "least upper bound".

Omega Notation

$$f(n) = \Omega(g(n)) \Leftrightarrow \exists c > 0, n_0 > 0, s.t. \forall n \geq n_0, \; f(n) \geq cg(n)$$

Here $f(n)$ is the step count.

Examples:

- $f(n) = 3n+2$
- $f(n) = 3n^3+2n^2+100$
- $f(n) = 5\log(n)+n$

Omega notation is the “lower bound” of complexity.

We should always use the "most strict" one, i.e., the "most lower bound".

Theta Notation

$$f(n) = \Theta(g(n)) \Leftrightarrow \exists c_1 > 0, c_2 > 0, n_0 > 0,$$
$$s.t.\ \forall n \geq n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n)$$

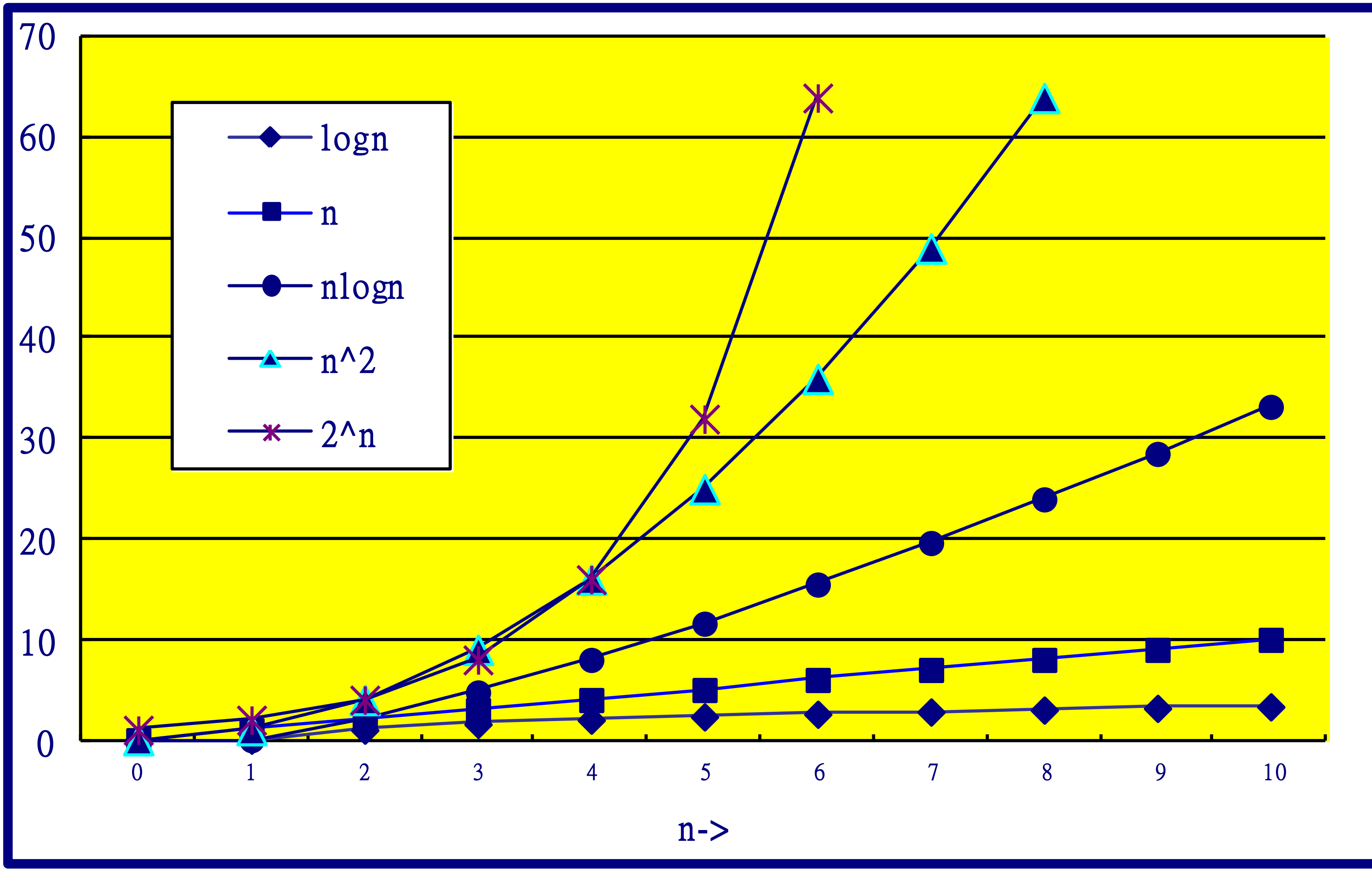
Here $f(n)$ is the step count.

Examples:

- $f(n) = 3n+2$
- $f(n) = 3n^3+2n^2+100$
- $f(n) = 5\log(n)+n$

Theta notation is the most precise form, being both upper and lower bounds of complexity.

Growth Rate



Notes on Big-O Notations

- Although the theta notation is the precise one, in practice about everyone uses the big-O notation in its most strict sense. (Nobody cares about the lower bound?)
- With this in mind, here are some useful rules for combining big-O notations:

$$f_1(n) = O(g_1(n)) \quad and \quad f_2(n) = O(g_2(n))$$

$$\Leftrightarrow$$

$$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$$

$$f_1(n) f_2(n) = O(g_1(n) g_2(n))$$

Complexities of Example Algorithms

- Summing an array:
- Matrix addition:
- Matrix multiplication:
- Selection sort:
- Binary search:
- Permutation:
- Recursive Fibonacci series:

```
int Fibonacci(int n)
{
  if (n==1 || n==2) return 1;
  else return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Extra Reading Assignments

- From the textbook: Sections 1.5.2, 1.6, 1.7.2.