

CHAPTER 1

BASIC CONCEPTS

1.1 Overview: system life cycle

- Good programmers regard large-scale computer programs as systems that contain many complex interacting parts.
- As systems, these programs undergo a development process called the *system life cycle*.
- We consider this cycle as consisting of five phases.
 - À **Requirements**
 - Á **Analysis**
 - Â **Design**
 - Ã **Refinement and coding**
 - Ä **Verification**

1.2 Algorithm specification

- Introduction

Definition:

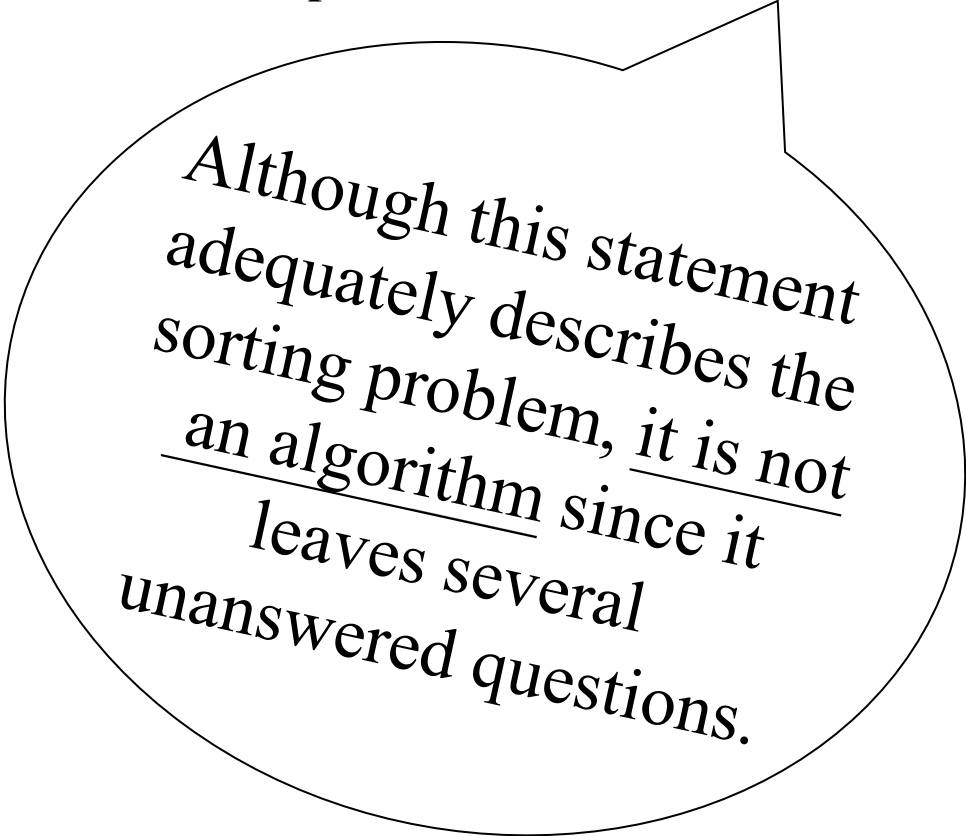
An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

- ¶ **Input.** There are zero or more quantities that externally supplied.
- **Output.** At least one quantity is produced.
- **Definiteness.** Each instruction is clear and unambiguous.
- ¹ **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- º **Effectiveness.** Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be define as in \square ; it also must be feasible.

– **Example 1.1** [*Selection sort*]:

- Suppose we must devise a program that sorts a set of $n \geq 1$ integers. A simple solution is given by the following:

From those integers that are currently unsorted, find the smallest and place it next in the sorted list.



Although this statement adequately describes the sorting problem, it is not an algorithm since it leaves several unanswered questions.

- Program 1.1 is our first attempt at deriving a solution. Notice that it is written partially in C and partially in English.

```
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];  
}
```

Program 1.1: Selection sort algorithm

- Program 1.3 contains a complete program which you may run on your computer. (next page)

```

#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [],int); /*selection sort */
void main(void)
{
    int i,n;
    int list[MAX_SIZE];
    printf("Enter the number of numbers to generate: ");
    scanf("%d",&n);
    if( n < 1 || n > MAX_SIZE) {
        fprintf(stderr, "Improper value of n\n");
        exit(1);
    }
    for (i = 0; i < n; i++) { /*randomly generate numbers*/
        list[i] = rand() % 1000;
        printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for (i = 0; i < n; i++) /* print out sorted numbers */
        printf("%d ",list[i]);
    printf("\n");
}

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

```

- Recursive algorithms

- Typically, beginning programmer view a function as something that is invoked (called) by another function. It executes its code and then returns control to the calling function.
- This perspective ignores the fact that functions can call themselves (*direct recursion*) or they may call other functions that invoke the calling function again (*indirect recursion*).
- These recursive mechanisms are not only extremely powerful, but they also frequently allow us to express an otherwise complex process in very clear term.

– Example 1.3 [*Binary search*]:

```
int binsearch(int list[], int searchnum, int left,
              int right)
{
    /* search list[0] <= list[1] <= . . . <= list[n-1] for
    searchnum. Return its position if found. Otherwise
    return -1 */
    int middle;
    if (left <= right) {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1: return
                binsearch(list, searchnum, middle + 1, right);
            case 0 : return middle;
            case 1 : return
                binsearch(list, searchnum, left, middle - 1);
        }
    }
    return -1;
}
```

Program 1.7: Recursive implementation of binary search

– Example 1.4 [*Permutations*]:

- 是指所有組合之意，例如一個 set $\{a, b, c\}$, 所有 permutations 有 $\{a, b, c\}, \{a, c, b\}, \{b, a, c\}, \{b, c, a\}, \{c, a, b\}, \{c, b, a\}$.
- 從數學上可知 n 個 elements 的 all permutations 有 $n!$ 個.
- 以 set $\{a, b, c, d\}$ 為例，找出所有的 permutations:
 - 1) a + all permutations of $\{b, c, d\}$
 - 2) b + all permutations of $\{a, c, d\}$
 - 3) c + all permutations of $\{a, b, d\}$
 - 4) d + all permutations of $\{a, b, c\}$

這部分等同於原始問題

\therefore 可用 **recursive** 方式處理

– Example 1.4 [*Permutations*]:

```
void perm(char *list, int i, int n)
/* generate all the permutations of list[i] to list[n] */
{
    int j, temp;
    if (i == n) {
        for (j = 0; j <= n; j++)
            printf("%c", list[j]);
        printf("    ");
    }
    else {
        /* list[i] to list[n] has more than one permutation,
        generate these recursively */
        for (j = i; j <= n; j++) {
            SWAP(list[i],list[j],temp);
            perm(list,i+1,n);
            SWAP(list[i],list[j],temp);
        }
    }
}
```

Program 1.8: Recursive permutation generator

1.4 Data abstraction

Definition:

A data type is a collection of *objects* and a set of *operations* that act on those objects.

Definition:

An abstract data type (ADT) is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operations.

– Example 1.5 [*Abstract data type Natural_Number*]

structure *Natural_Number* is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT-MAX*) on the computer

functions:

for all $x, y \in \text{Nat_Number}$; $TRUE, FALSE \in \text{Boolean}$
and where $+$, $-$, $<$, and $==$ are the usual integer operations

<i>Nat-No</i> Zero()	::=	0
<i>Boolean</i> Is-Zero(x)	::=	if (x) return FALSE else return TRUE
<i>Nat-No</i> Add(x, y)	::=	if ((x + y) <= INT-MAX) return x + y else return INT-MAX
<i>Boolean</i> Equal(x, y)	::=	if (x == y) return TRUE else return FALSE
<i>Nat-No</i> Successor(x)	::=	if (x == INT-MAX) return x else return x + 1
<i>Nat-No</i> Subtract(x, y)	::=	if (x < y) return 0 else return x - y

end *Natural_Number*

Structure 1.1: Abstract data type *Natural_Number*

1.5 Performance analysis

- There are many criteria upon which we can judge a program, including:
 - Does the program meet the original specifications of the task?
 - Does it work correctly?
 - ® Does the program contain documentation that show how to use it and how it works?
 - Does the program effectively use functions to create logical units?
 - ° Is the program's code readable?
 - ± Does the program efficiently use primary and secondary storage?
 - ² Is the program's running time acceptable for the task?

- These criteria focus on performance evaluation, which we can loosely divide into two distinct fields.
 - *Performance analysis*. This field focuses on obtaining estimates of time and space that are machine independent. Its subject matter is the heart of an important branch of computer science known as *complexity theory*.
 - *Performance measurement*. This field obtains machine-dependent running times. These times are used to identify inefficient code segments.

– **Definition:**

The *space complexity* of a program is the amount of memory that is needed to run to completion. The *time complexity* of a program is the amount of computer time that it needs to run to completion.

- **Space complexity**

- The space needed by a program is the sum of the following components:
 - **Fixed space requirements.** This component refers to space requirements that do not depend on the number and size of the program's input and output.
 - **Variable space requirements.** This component consists of the space needed by structured variables whose size depends on the particular instance, I , of the program being solved. It also includes the additional space required when a function uses recursions.
- We can express the total space requirement $S(P)$ of any program as:
 - $$S(P) = c + S_P(I)$$
 - where c is a constant representing the fixed space requirements, and the $S_P(I)$ is the variable space requirement of a program P working on an instance I .

– Examples:

- Example 1.6: In program 1.9, $S_{abc}(I)=0$.

```
float abc(float a, float b, float c)
{
    return a+b+b*c + (a+b-c) / (a+b)+4.00;
}
```

Program 1.9: Simple arithmetic function

- Example 1.7: In program 1.10, $S_{sum}(n)=0$.

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

Program 1.10: Iterative function for summing a list of numbers

- Example 1.8: Program 1.11 is a recursive function for addition. Figure 1.1 shows the number of bytes required for one recursive call.

```
float rsum(float list[], int n)
{
    if (n) return rsum(list,n-1) + list[n-1];
    return 0;
}
```

Program 1.11: Recursive function for summing a list of numbers

Type	Name	Number of bytes
parameter: float	<i>list[]</i>	2
parameter: integer	<i>n</i>	2
return address: (used internally)		2 (unless a far address)
TOTAL per recursive call		6

Figure 1.1: Space needed for one recursive call of Program 1.11

- Time complexity

- The time, $T(P)$, taken by a program, P , is the sum of its compile time and its run (or execution) time.
- The compile time is similar to the fixed space component since it does not depend on the instance characteristics. In addition, once we have verified that the program runs correctly, we may run it many times without recompilation.
- We are really concerned only with the program's execution time, T_P .

- Asymptotic notation (O , Ω , Θ)

Definition: [Big “oh”]

$f(n) = O(g(n))$ (read as “ f of n is big oh of g of n ”) iff (if and only if) there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n , $n \geq n_0$.

Theorem 1.2:

If $f(n) = a_m n^m + \dots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Definition: [Omega]

$f(n) = \Omega(g(n))$ (read as “ f of n is omega of g of n ”) iff there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all n , $n \geq n_0$.

Theorem 1.3:

If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Definition: [Theta]

$f(n) = \Theta(g(n))$ (read as “ f of n is theta of g of n ”) iff there exist positive constants c_1 , c_2 , and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n , $n \geq n_0$.

Theorem 1.4:

If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

- Practical complexity
 - To get a feel for how the various functions grow with n , you are advised to study Figures 1.7 and 1.8 very closely.

		Instance characteristic n					
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

Figure 1.7 Function values

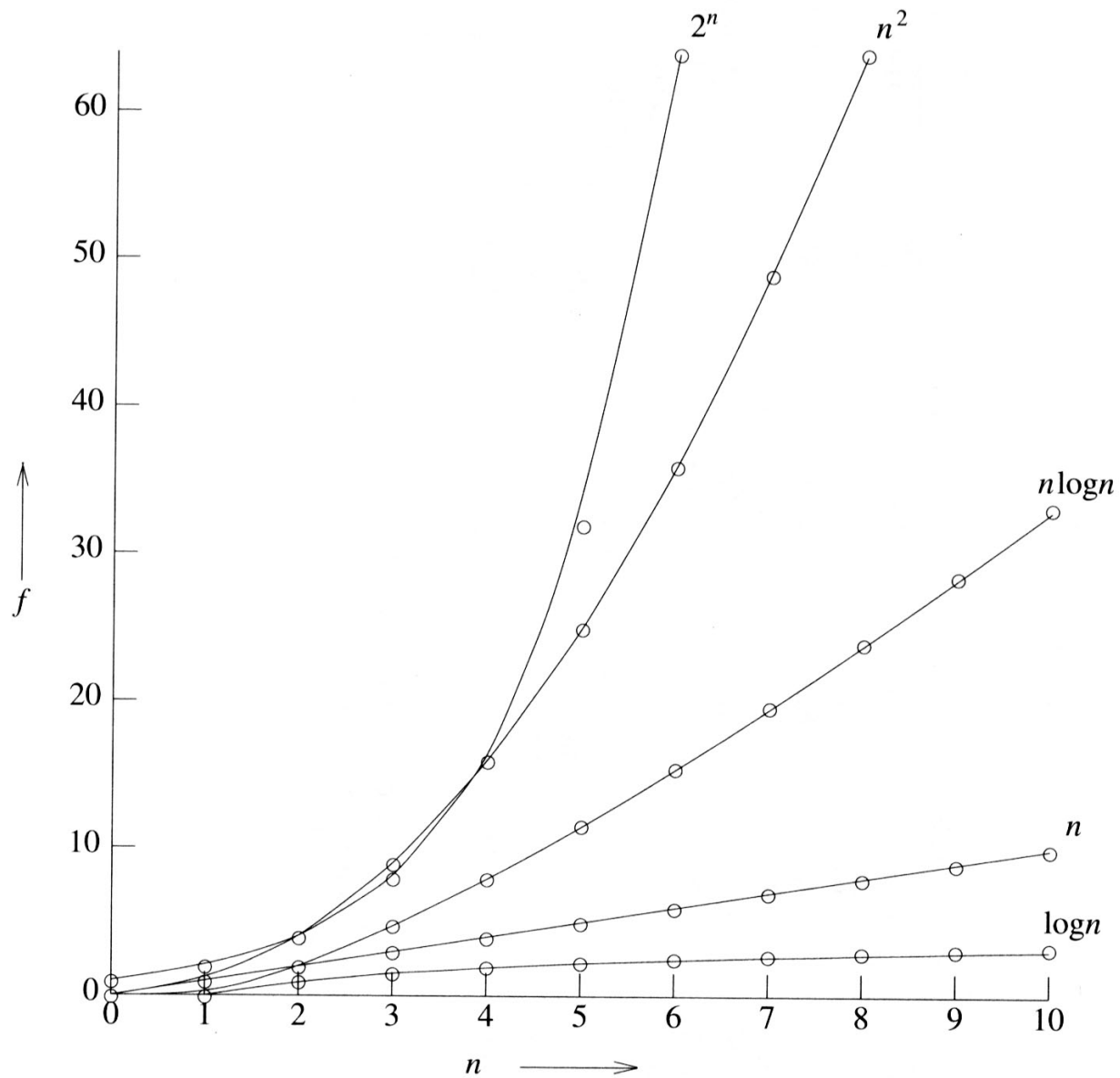


Figure 1.8 Plot of function values

- Figure 1.9 gives the time needed by a 1 billion instructions per second computer to execute a program of complexity $f(n)$ instructions.

Figure 1.9 Times on a 1 billion instruction per second computer

$n \log n$

Time for $f(n)$ instructions on a 10^9 instr/sec computer

n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01 μ s	.03 μ s	.1 μ s	1 μ s	10 μ s	10sec	1 μ s
20	.02 μ s	.09 μ s	.4 μ s	8 μ s	160 μ s	2.84hr	1ms
30	.03 μ s	.15 μ s	.9 μ s	27 μ s	810 μ s	6.83d	1sec
40	.04 μ s	.21 μ s	1.6 μ s	64 μ s	2.56ms	121.36d	18.3min
50	.05 μ s	.28 μ s	2.5 μ s	125 μ s	6.25ms	3.1yr	13d
100	.10 μ s	.66 μ s	10 μ s	1ms	100ms	3171yr	$4 \cdot 10^{13}$ yr
1,000	1.00 μ s	9.96 μ s	1ms	1sec	16.67min	$3.17 \cdot 10^{13}$ yr	$32 \cdot 10^{283}$ yr
10,000	10.00 μ s	130.03 μ s	100ms	16.67min	115.7d	$3.17 \cdot 10^{23}$ yr	
100,000	100.00 μ s	1.66ms	10sec	11.57d	3171yr	$3.17 \cdot 10^{33}$ yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17 \cdot 10^7$ yr	$3.17 \cdot 10^{43}$ yr	

μ s = microsecond = 10^{-6} seconds

ms = millisecond = 10^{-3} seconds

sec = seconds

min = minutes

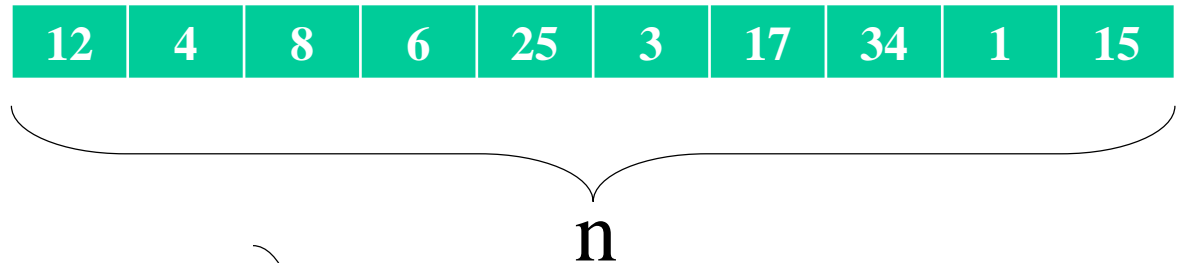
hr = hours

d = days

yr = years

How to find the complexity of an algorithm ?

- Selection sort:



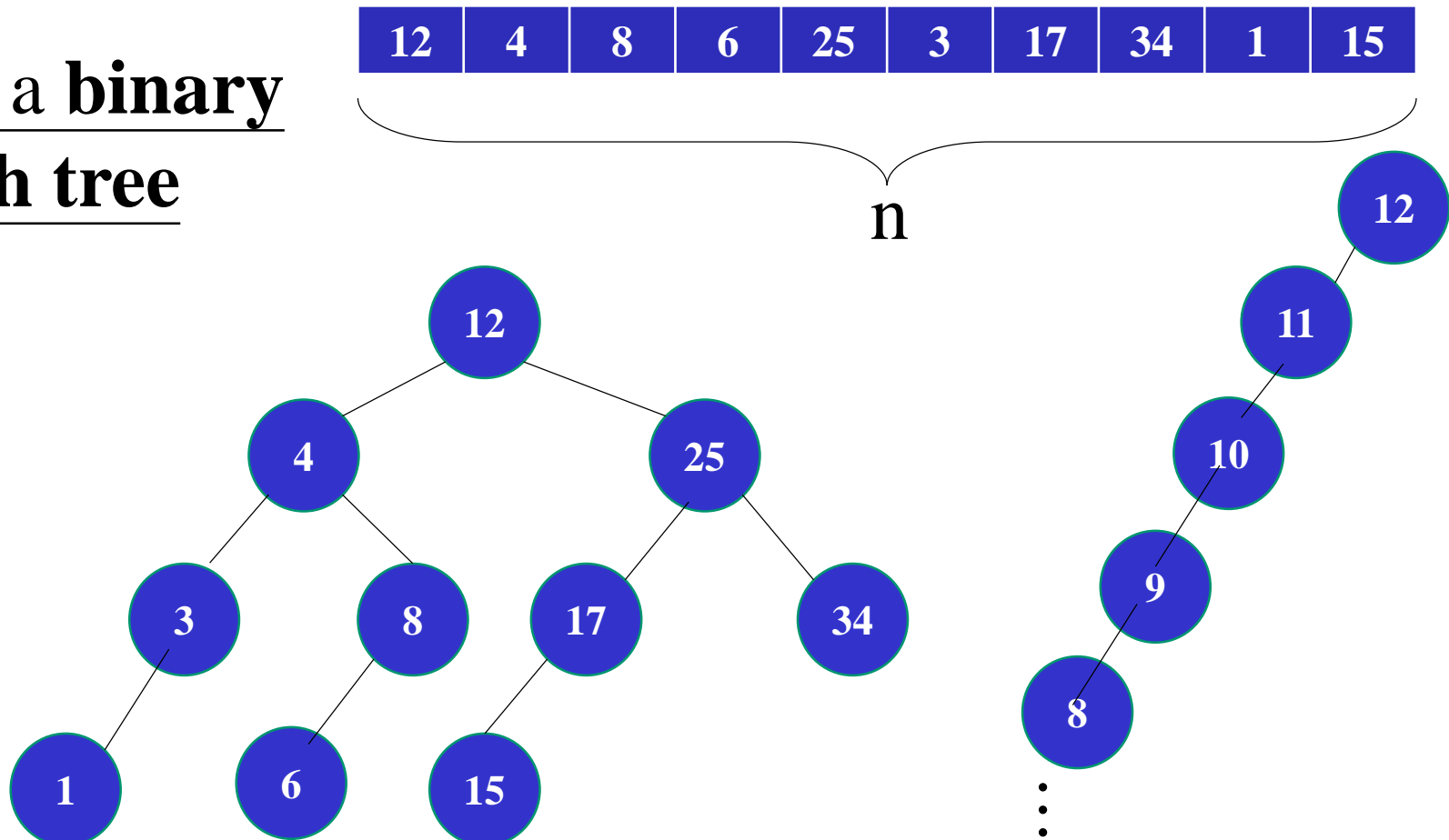
- scan ***n*** items to find the smallest
- scan ***n-1*** items
- scan ***n-2*** items
-
- scan **3** items
- scan **2** items
- scan **1** items

$$\begin{aligned} \text{Total: } & n + (n-1) + (n-2) + \dots + 2 + 1 \\ &= \frac{1}{2} n(n+1) = \frac{1}{2} (n^2 + n) \end{aligned}$$

⇒ Complexity is $O(n^2)$.

How to find the complexity of an algorithm ?

- Build a **binary search tree**



Complexity is ?

Average: $O(n \log n)$.

Worst case: $O(n^2)$.

1.5 Performance measurement

- Although performance analysis gives us a powerful tool for assessing an algorithm's space and time complexity, at some point we must also consider how the algorithm executes on our machine.
 - This consideration moves us from the realm of analysis to that of measurement.
- There are actually two different methods for timing events in C.
 - Figure 1.10 shows the major differences between these two methods.
 - Method 1 uses *clock* to time events. This function gives the amount of processor time that has elapsed since the program began running.
 - Method 2 uses *time* to time events. This function returns the time, measured in seconds, as the built-in type *time_t*.
 - The exact syntax of the timing functions varies from computer to computer and also depends on the operating system and compiler in use.

	Method 1	Method 2
Start timing	<code>start = clock();</code>	<code>start = time(NULL);</code>
Stop timing	<code>stop = clock();</code>	<code>stop = time(NULL);</code>
Type returned	<code>clock_t</code>	<code>time_t</code>
Result in seconds	<code>duration = ((double) (stop-start)) / CLK_TCK;</code>	<code>duration = (double) difftime(stop,start);</code>

Figure 1.10: Event timing in C

- **Example 1.22** [*Worst case performance of the selection function*]:
 - The tests were conducted on an IBM compatible PC with an 80386 cpu, an 80387 numeric coprocessor, and a turbo accelerator. We use Broland's Turbo C compiler.

n	Time	n	Time
30 ... 100	.00	900	1.86
200	.11	1000	2.31
300	.22	1100	2.80
400	.38	1200	3.35
500	.60	1300	3.90
600	.82	1400	4.54
700	1.15	1500	5.22
800	1.48	1600	5.93

Figure 1.11: Worst case performance of selection sort (in seconds)

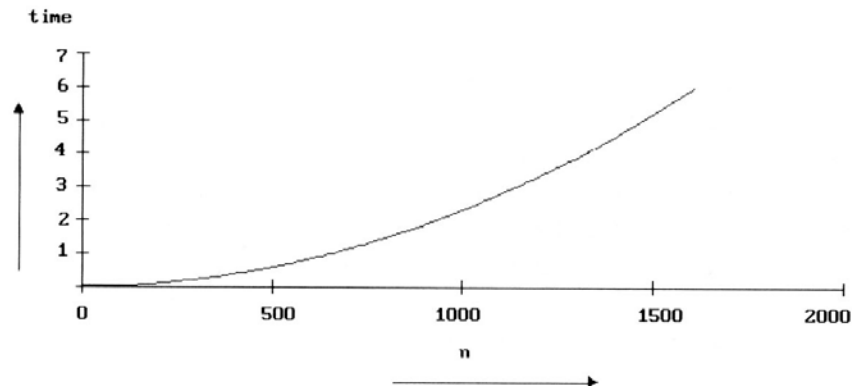


Figure 1.12: Graph of worst case performance for selection sort

n	Iterations	Ticks	Total time (sec)	Duration
0	30000	16	0.879121	0.000029
10	12000	16	0.879121	0.000073
20	6000	14	0.769231	0.000128
30	5000	16	0.879121	0.000176
40	4000	16	0.879121	0.000220
50	4000	20	1.098901	0.000275
60	4000	23	1.263736	0.000316
70	3000	20	1.098901	0.000366
80	3000	23	1.263736	0.000421
90	2000	17	0.934066	0.000467
100	2000	18	0.989011	0.000495
200	1000	18	0.989011	0.000989
400	500	18	0.989011	0.001978
600	500	27	1.483516	0.002967
800	500	35	1.923077	0.003846
1000	300	27	1.483516	0.004945

Figure 1.13: Worst case performance of sequential search