Chapter 1: Algorithms: Efficiency, Analysis, and Order

Contents

1.1 Algorithms

- **Problem**
  - **Ex. 1.1: *Sorting* - Sort the list *S* of *n* numbers in nondecreasing order. The answer is the numbers in sorted sequence.**
  - **Ex. 1.2: *Searching* - Determine whether the number *x* is in the list *S* of *n* numbers. The answer is yes if *x* is in *S* and no if it is not.**
  - **Ex) *Partition* – Decide whether a given multiset A= {a_1, … , a_n} of n positive integers has a partition P such that** $$\sum_{i \in P} a_i = \sum_{i \notin P} a_i$$

Parameters, Instance, and Solution

- **Problem may contain *Parameters***
  - **Ex. 1.1 - Sorting : *S*, *n***
  - **Ex. 1.2 - Searching: *S*, *n*, *x***
  - **Ex) Partition: *A*, *n***
- ***Instance* of a problem: each specific assignment of values to parameters**
  - **Ex. 1.3: Instance of *Sorting***
    - **S = [10, 7, 11, 5, 13, 8] and n = 6**
  - **Ex. 1.4: Instance of *Searching***
    - **S = [10, 7, 11, 5, 13, 8], n = 6, and x = 5**
  - **Instance of *Partition***
    - **A = {10, 7, 11, 5, 13, 8} and n = 6 → sol = ‘no’**
    - **A = {10, 7, 11, 5, 15, 8} and n = 6 → sol = ‘yes’**
- ***Solution***

Algorithm

- **Step by step procedure for producing the solution to each instance**
- **Def.(S. Sahni): An algorithm is a finite set of instructions that accomplish a particular task.**
  - **Input – zero or more**
  - **Output - zero or more**
  - **Definiteness – each instruction is clear and unambiguous**
    - **Ex) add 6 or 7 to x (X)**
  - **Finiteness: must terminate after a finite number of steps.**
    - **cf) procedure**
    - **OS (X)**

Pseudocode and Program

- **Effectiveness: each instruction must be very basic so that it can be carried out by a person using pencil and paper. It also must be feasible.**
  - **Integer arithmetic (O)**
  - **Real arithmetic (X) – decimal expansion might be infinitely long**
- ***Pseudocode*: C++ like**
- ***Program*: expression of an algorithm in a PL**

1.2 Importance of Developing Efficient Algorithms

- **1.2.1 Sequential search vs. Binary search**
  - **Algorithm 1.1 vs. Algorithm 1.5**
  - **# of comparisons (worst case)**
    - n vs. $\log_2 n + 1$
  - **See Table 1.1**

Table 1.1 The number of comparisons done by Sequential Search and Binary Search when x is larger than all the array items

Array Size	Number of Comparisons by Sequential Search	Number of Comparisons by Binary Search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

Algorithm 1.1 Sequential Search

- **Problem**: Is the key *x* in the array *S* of *n* keys?
- **Input (parameters)**: positive integer *n*, array of keys *S* indexed from 1 to *n*, and a key *x*.
- **Output**: *location*, the location of *x* in *S* (0 if *x* is not in *S*)

```
void seqsearch (        int n,
                        const keytype S[ ],
                        keytype x,
                        index& location)
{
   location = 1;
   while (location <= n && S[location] != x)
      location ++;
   if (location > n )
      location = 0;
}
```

Algorithm 1.5 Binary Search (1/2)

- **Problem: Determine whether *x* in the sorted array *S* of *n* keys.**
- **Inputs: positive integer *n*, sorted (nondecreasing order) array of keys *S* indexed from 1 to *n*, and a key *x*.**
- **Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*)**

Algorithm 1.5 Binary Search (2/2)

```
void binsearch (        int n,
                        const keytype S[ ],
                        keytype x,
                        index& location)
{
    index low, high, mid;
    low = 1; high = n;
    location = 0;
    while (low <= high && location == 0) {
        mid = ⌊(low + high) / 2⌋;
        if (x == S[mid])
            location = mid;
        else if (x < S[mid])
            high = mid – 1
        else
            low = mid + 1;
    }
}
```

1.2.2 Fibonacci Sequence

$$f_0 = 0,$$
$$f_1 = 1,$$
$$f_n = f_{n-1} + f_{n-2} \quad (n \geq 2)$$

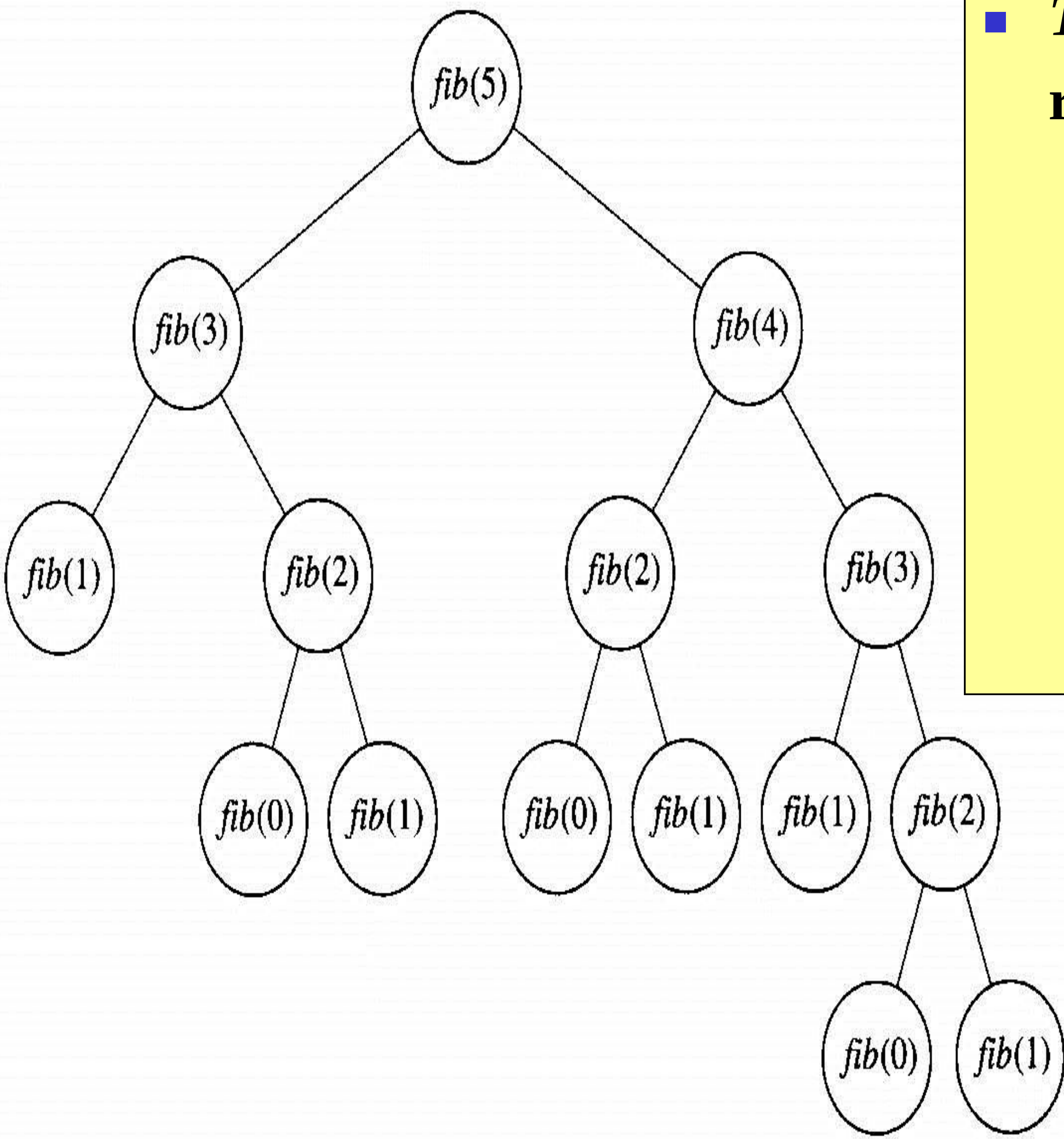
Algorithm 1.6 n–th Fibonacci Term (Recursive)

- **<u>Problem</u>: Determine the *n*-th term in the Fibonacci Sequence.**
- **<u>Inputs</u>: a nonnegative integer *n*.**
- **<u>Outputs</u>: *fib*, the *n*-th term of the Fibonacci Sequence.**

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

$$f_0 = 0, \quad f_1 = 1,$$
$$f_n = f_{n-1} + f_{n-2} \quad (n \geq 2)$$

Figure 1.2 The recursion tree corresponding to Algorithm 1.6 when computing the fifth Fibonacci term.



- ***T*(*n*): # of terms in the recursion tree for *n*.**

$$T(n) > 2 \times T(n-2)$$

$$> 2 \times 2 \times T(n-4)$$

$$> \underbrace{2 \times 2 \times \ldots \times 2}_{n/2 \text{ terms}} \times T(0)$$

$$\therefore\ T(n) > 2^{n/2}$$

(proof by induction in Th. 1.1)

Algorithm 1.7 *n*–th Fibonacci Term (Iterative)

- **<u>Problem</u>: Determine the *n*-th term in the Fibonacci Sequence.**
- **<u>Inputs</u>: a nonnegative integer *n*.**
- **<u>Outputs</u>: *fib2*, the *n*-th term in the Fibonacci Sequence.**

```
int fib2 (int n)
{
   index i;
   int f[0..n];
   f[0] = 0;
   if (n > 0) {
      f[1] = 1;
      for (i = 2; i <= n; i++)
         f[i] = f[i – 1] + f[i – 2];
   }
      return f[n];
}
```

- **dynamic programming: compute (*n*+1) terms**

Table 1.2 A comparison of Algorithms 1.6 and 1.7

n	$n+1$	$2^{n/2}$	Execution Time Using Algorithm 1.7	Lower Bound on Execution Time Using Algorithm 1.6
40	41	1,048,576	41 ns*	1048 μs†
60	61	1.1×10^9	61 ns	1 s
80	81	1.1×10^{12}	81 ns	18 min
100	101	1.1×10^{15}	101 ns	13 days
120	121	1.2×10^{18}	121 ns	36 years
160	161	1.2×10^{24}	161 ns	3.8×10^7 years
200	201	1.3×10^{30}	201 ns	4×10^{13} years

*1 ns = 10^{-9} second.

†1 μs = 10^{-6} second.

1.3 Analysis of Algorithms

- **Efficiency**
  - **Space complexity: memory**
  - **Time complexity: execution time**

1.3.1 Time Complexity Analysis

- **Want a measure independent of**
  - **Computer**
  - **Programming language**
  - **Programmer**
  - **Complex details of algorithms (pointer setting, incrementing of loop indices)**
- **Not want # of CPU cycles or instructions**
  - **Ex) binary search is more efficient than sequential search**
    - **# of comparisons: log n < n**
- **Algorithm's efficiency: # of basic operations executed as a function of input size**

Algorithm 1.2 Add Array Members

- **<u>Problem</u>: Add all the numbers in the array *S* of *n* numbers.**
- **<u>Inputs</u>: positive integer *n*, array of numbers *S* indexed from 1 to *n*.**
- **<u>Outputs</u>: *sum*, the sum of the numbers in *S*.**

```
number sum (int n, const number S[ ])
{
   index i;
   number result;
   result = 0;
   for (i = 1; i <= n ; i++)
      result = result + S[i];
   return result;
}
```

Algorithm 1.3 Exchange Sort

- Problem: Sort *n* keys in nondecreasing order.
- Inputs: positive integer *n*, array of keys *S* indexed from 1 to *n*.
- Outputs: the array *S* containing the keys in nondecreasing order.

```
void exchangesort (int n, keytype S[ ])
{
   index i, j;
   for (i = 1; i <= n -1; i++)
      for (j = i+1; j <= n; j++)
         if (S[j] < S[i])
            exchange S[i] and S[j];
}
```

Algorithm 1.4 Matrix Multiplication

- **Problem**: Determine the product of two *n* x *n* matrices.
- **Inputs**: a positive integer *n*, 2D arrays of numbers *A* and *B*, each of which has both its rows and columns indexed from 1 to *n*.
- **Outputs**: a 2D array of numbers *C*, which has both its rows and columns indexed from 1 to *n*, containing the product of *A* and *B*.

```
void matrixmult (        int n,
                         const number A[ ] [ ],
                         const number B[ ] [ ],
                              number C[ ] [ ])
{
   index i, j, k;
   for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++) {
         C[i][j] = 0;                                    /* b.o. → e.t. = a
         for (k = 1; k <= n; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];       /* b.o. → e.t. = b
   }
}
```

Input size and Basic operation

- **Input size**
  - **Sequential search, binary search, add array members, exchange sort: array *S* of *n* keys**
  - **Matrix multiplication: *n*, # of rows and columns**
  - **Graph: *n*, *e*, # of nodes and edges**
  - **Fibonacci number: $\lfloor \log n \rfloor + 1$, # of binary digits to encode *n* (*n* is input not input size)**
- **Basic operation**
  - **Single instruction or group of instructions**
  - **Execution time is independent of *n***
  - **Ex) search: comparison**

Example

```
for (i = 1; i <= n; i++)
   for (j = 1; j <= n; j++) {
      C[i][j] = 0;      /* b.o. → e.t. = a
      for (k = 1; k <= n; k++)
         C[i][j] = C[i][j] + A[i][k] * B[k][j];
                        /* b.o. → e.t. = b
      }
```

- **Ex) matrix multiplication**
  - **Execution time → see Algorithm 1.4**
  - **Discussion**

    (1) $a \cdot n^2 + b \cdot n^3$

    - **n = 10 →** $a \cdot 10^2 + b \cdot 10^3 \approx b \cdot 10^3$
    - **n = 100 →** $a \cdot 10^4 + b \cdot 10^6 \approx b \cdot 10^6$

    (2) Ignore the time for incrementing loop indices

    (3) No time difference

    - **temp = A[i][k] * B[k][j];**
    - **C[i][j] = C[i][j] + temp**

Time Complexity Analysis

- **Determination of how many times the basic operations is done for each value of the input size.**
- **In some cases, depends not only the input size but also on the *input value***
  - **Ex) sequential search**
    - **Best case** $B(n) = 1$
    - **Worst case** $W(n) = n$
    - **Average case** $A(n) = (n+1)/2 \leftarrow \sum_{k=1}^{n} (k \times \frac{1}{n})$
  - **Ex) array add**
    - **Every case** $T(n) = n$
  - **Ex) Exchange sort**
    - $T(n) = (n\text{-}1) + (n\text{-}2) + \ldots + 1 = (n\text{-}1)n/2$
  - **Ex) matrix multiplication** $T(n) = n^3$
- **If $T(n)$ exists** $T(n) = W(n) = A(n) = B(n)$

  If not $W(n), A(n)$

Space Complexity Analysis

- **Fixed part that is independent of I/O characteristics**
  - **Instruction (code) space**
  - **Simple variable $x = 3$**
  - **Constants**
  - **Fixed size component variables (A[10], …)**
- **Variable part**
  - **Variables depending on input size**

    Ex : S[n], A[n][n]
  - **Recursion stack (formal parameters, local variables, return address): space $\geq 3(n+1)$ words, n is depth of recursion**

    Ex : Fibonacci number (Alg. 1.6): proportional to n