Chapter 1. (Part 1)

Algorithms:
Efficiency, Analysis, and Order

경북대학교 배준현 교수

(joonion@knu.ac.kr)

Contents

1.1 Algorithms

- An ***algorithm*** is
  - a step-by-step *procedure* for solving a *problem*.

- A *computer algorithm* is
  - a *finite sequence* of *instructions* to solve a problem using a *computer*.

- A *problem* is
  - a question to which we seek an *answer*, to say, a *solution*.

- An *instance* of a problem:
  - A problem may contain *variables*, called *parameters*,
    - which are *not* assigned *specific values* in the statement of the problem.
  - An algorithmic problem is specified
    - by describing the *complete set of instances* it must work on and
    - what *properties* the *output* must have as a *result* of running on.

1.1 Algorithms

- An example of a *problem*:
  - Sort a list S of n numbers in *nondecreasing* order.
  - The *solution* of this problem is the numbers in sorted sequence.
- An *instance* of the problem:
  - $n = 6$, $S = [10, 7, 11, 5, 13, 8]$.
- The *solution* to this *instance* of the problem:
  - $S' = [5, 7, 8, 10, 11, 13]$

- Another example of a *problem*:
  - Determine whether the number x is in the list S of n numbers.
  - The *solution* is
    - *yes* if x is in S, and *no* 0 if it is not in S.
- An *instance* of the problem:
  - $n = 6, S = [10, 7, 11, 5, 13, 8]$, and $x = 5$.
- The *solution* to this instance of the problem:
  - *yes*, x is in S. and $location = 4$ if the $location$ starts from 1.

- An *algorithm* for solving the previous problem:
  - Starting with the *first* item in S,
    - compare x with each item in S in sequence,
      - until x is found or until S is exhausted.
  - If x is found, answer *yes* by returning the location of x,
    - if x is not found, answer *no* by returning 0.

sqential search
-> function is return x or 0
(return x === location
value)

- An *implementation* of the algorithm in C++:

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n, x, location;
    cin >> n;
    vector<int> S(n + 1);
    for (int i = 1; i <= n; i++)
        cin >> S[i];
    cin >> x;
    location = 1;
    while (location <= n && S[location] != x)
        location++;
    if (location > n)
        location = 0;
    cout << location << endl;
}
```

- Using a ***pseudo-code*** for writing an algorithm:

ALGORITHM 1.1: Sequential Search

```
typedef int keytype;
typedef int index;

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;
}
```

- *Testing* the algorithm:

```
int main() {
    int n, x, location, T;
    cin >> n;
    vector<int> S(n + 1);
    for (int i = 1; i <= n; i++)
        cin >> S[i];
    cin >> x;
    int *pS = &S[0];
    seqsearch(n, pS, x, location);
    cout << location << endl;
}
```

1.1 Algorithms

- Compile and run the program with GNU Toolchain:
  - a.exe in Windows, a.out in Linux

1.1.1.in

```
[Input]
6
10 7 11 5 13 8
5
```

```
[Output]
4
```

1.1.2.in

```
[Input]
6
10 7 11 5 13 8
15
```

```
[Output]
0
```

```
> g++ 1.1_SequentialSearch.cpp

> a.exe < 1.1.1.in
4
```

Redirection

```
> a.exe < 1.1.2.in
0

> a.exe < 1.1.3.in > 1.1.3.out
```

- **Exercise:** *Adding Array Members*
  - Problem:
    - *Add* all the numbers in the array S of n numbers.
  - Inputs:
    - positive integer n, array of numbers S indexed from 1 to n.

-> 교재에선 이렇게 하라니
까 input은 인덱스 1부터 n
이 될때까지 해보자.

  - Outputs:
    - sum, the sum of the numbers in S.

ALGORITHM 1.2: Adding Array Members

```
int sum(int n, vector<int>& S)
{
    int result = 0;
    for (int i = 1; i <= n; i++)
        result += S[i];
    return result;
}
```

```
[Input]                 [Output]
6                       54
10 7 11 5 13 8
```

- Exercise: *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 non-decreasing order.

오름차순

ALGORITHM 1.3: Exchange Sort

```
void exchangesort(int n, vector<int>& S)
{
    for (int i = 1; i <= n; i++)
        for (int j = i + 1; j <= n; j++)
            if (S[i] > S[j])
                swap(S[i], S[j]);
}
```

```
#include <utility>
```

```
[Input]
6
10 7 11 5 13 8
```

```
[Output]
5 7 8 10 11 13
```

- **Exercise:** *Matrix Multiplication*
  - Problem:
    - Determine the product of two $n \times n$ matrices.
  - Inputs:
    - a positive number n, two-dimensional arrays of numbers A and B,
      - each of which has both its *rows* and *columns* indexed from 1 to n.
  - Outputs:
    - a two-dimensional array of numbers C,
      - which has both its *rows* and *columns* indexed from 1 to n,
      - containing the *product* of A and B.

- Matrix Multiplication
  - If we have two 2×2 matrices, $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, $B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$,
    - then the product $C = A \times B$ is given by $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j}$.
  - For example,
    - $\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \times \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 28 & 38 \\ 26 & 36 \end{bmatrix}$.
  - In general,

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj} \quad \text{for } 1 \le i, j \le n.$$

ALGORITHM 1.4: Matrix Multiplication

```
typedef vector<vector<int>> matrix_t;

void matrixmult(int n, matrix_t A, matrix_t B, matrix_t& C)
{
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) {
            C[i][j] = 0;
            for (int k = 1; k <= n; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}
```

Call By Value

Call By Reference

```
int main() {
    int n;
    cin >> n;
    matrix_t A(n + 1, vector<int>(n + 1));
    matrix_t B(n + 1, vector<int>(n + 1));
    matrix_t C(n + 1, vector<int>(n + 1));
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> A[i][j];
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> B[i][j];
    matrixmult(n, A, B, C);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++)
            cout << C[i][j] << " ";
        cout << endl;
    }
}
```

```
[Input]
2
2 3
4 1
5 7
6 8
```

```
[Output]
28 38
26 36
```

```
[Input]
4
1 2 3 4
5 6 7 8
9 1 2 3
4 5 6 7
8 9 1 2
3 4 5 6
7 8 9 1
2 3 4 5
```

```
[Output]
43 53 54 37
123 149 130 93
95 110 44 41
103 125 111 79
```

1.1 Algorithms

- Five *Properties* of Algorithms
  - *Zero* or more ***inputs***.
  - *One* or more ***outputs***.
  - *Unambiguity*.
    - Each *instruction* in an algorithm must be *clear enough* to follow.
  - ***Finiteness***.
    - An algorithm *must terminate* after a finite number of steps.
  - *Feasibility*.
    - An algorithm *must be feasible* enough to be carried out.

1.2 The Importance of Developing Efficient Algorithms

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

ALGORITHM 1.5: Binary Search

```
void binsearch(int n, vector<int>& S, int x, int& location)
{
    int 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 // x > S[mid]
            low = mid + 1;
    }
}
```

1.2 The Importance of Developing Efficient Algorithms

[Input]	[Output]
6 5 7 8 10 11 13 5	1
6 5 7 8 10 11 13 8	3
6 5 7 8 10 11 13 13	6
6 5 7 8 10 11 13 15	0

- Sequential Search .vs. Binary Search
  - Compare *the number of comparisons*
    - done by **Algorithm 1.1** (Sequential) and **Algorithm 1.5** (Binary).
  - If the array S contains 32 items and x is *not in* the array.
    - Algorithm 1.1 (Sequential Search): 32 comparisons.
    - Algorithm 1.5 (Binary Search): 6 comparisons *at most*.
  - In general, if n is a power of 2,
    - Sequential Search with n keys: n comparisons.
    - Binary Search with n keys: $\lg n + 1$ comparisons *at most*.

※ $\lg n = \log_2 n$

- 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
n	n	$\lg n + 1$

- Problem:
  - Compute the nth term of the **Fibonacci sequence**.
    - $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \cdots$

- The Fibonacci Sequence
  - is defined *recursively* as follows:
    - $f_0 = 0$
    - $f_1 = 1$
    - $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$.

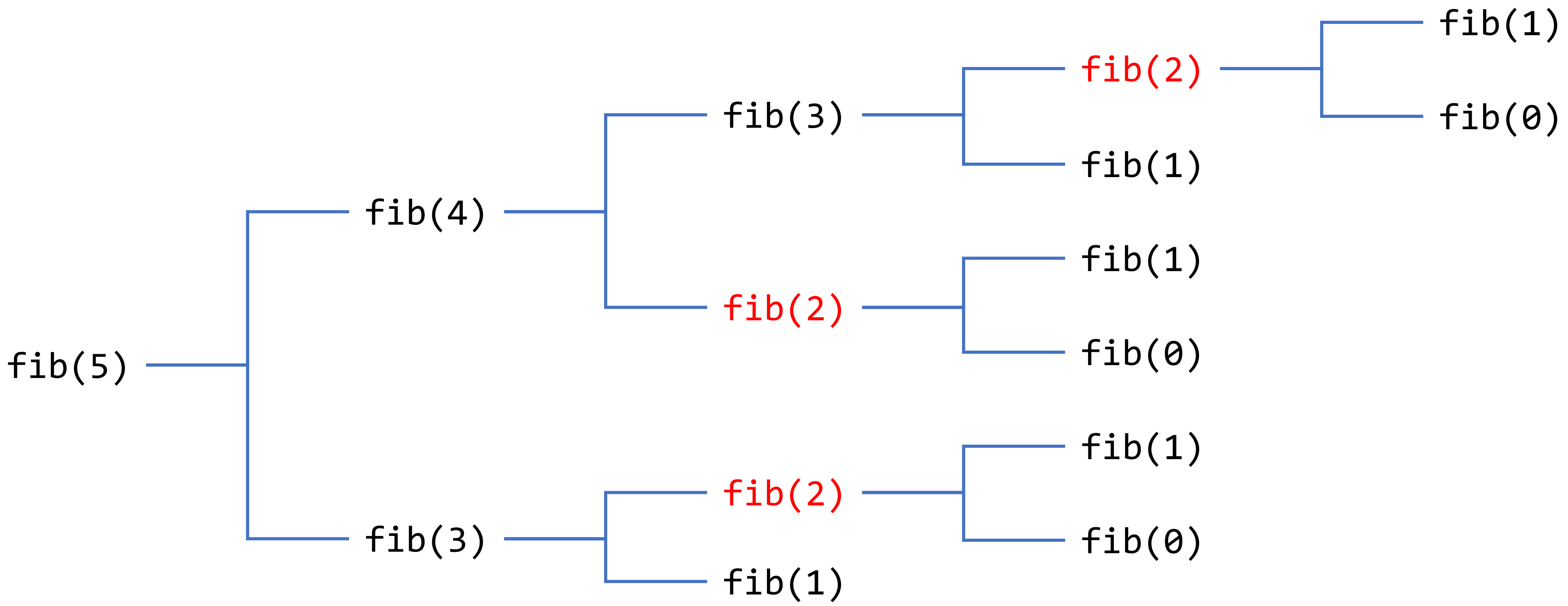
ALGORITHM 1.6: (Recursive) *n*th Fibonacci Term

```
typedef unsigned long long LongInt;

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

1.2 The Importance of Developing Efficient Algorithms

- The *inefficiency* of the *recursive* algorithm:
  - The algorithm invokes $fib(2)$ 3 times to calculate $fib(5)$.
  - Let $T(n)$ be the *number of terms* in the *recursion tree*: $T(n) > 2^{n/2}$.



1.2 The Importance of Developing Efficient Algorithms

- Developing an *efficient* algorithm:
  - We *do not* need to *recompute* the same value over and over again.
  - When a value is computed, we *save it* in an array. (*memoization*)
    - Then, we can *reuse the saved value* whenever we need it.

ALGORITHM 1.7: (Iterative) *n*th Fibonacci Term

```
typedef unsigned long long LongInt;

LongInt fib2(int n)
{
    vector<LongInt> F;
    if (n <= 1)
        return n;
    else {
        F.push_back(0);
        F.push_back(1);
        for (int i = 2; i <= n; i++)
            F.push_back(F[i - 1] + F[i - 2]);
        return F[n];
    }
}
```

Any Questions?