Chapter 1.

Algorithms: Efficiency, Analysis, and Order

Foundations of Algorithms, 5th Ed. Richard E. Neapolitan

Contents

- 1.1 Algorithms
- 1.2 The Importance of Developing Efficient Algorithms
- 1.3 Analysis of Algorithms
- 1.4 Order





- 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
 - by describing the *complete set of instances* it must work on
 - and what *properties* the *output* must have as a *result* of running on.



- 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:
 - S = [10, 7, 11, 5, 13, 8], and n = 6.
- 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 the location of *x* if it is in *S*, or 0 if it is not in *S*.
- An *instance* of the problem:
 - S = [10, 7, 11, 5, 13, 8], n = 6, and x = 5.
- The *solution* to this instance of the problem:
 - *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, return the location of x,
 - if *x* is not found, return 0.





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

ALGORITHM 1.1: Sequential Search

```
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;
}
```





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

```
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:

```
#include <iostream>
using namespace std;
#define MAX 256
int main() {
    int n, S[MAX], x, location;
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> S[i];
    cin >> x;
    seqsearch(n, S, x, location);
    cout << location << endl;</pre>
```



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





ALGORITHM 1.2: Adding Array Members

```
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;
```





- 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.



ALGORITHM 1.3: Exchange Sort

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



- 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

```
#define MAX 256
void matrixmult(int n,
                const number A[][MAX],
                const number B[][MAX],
                      number C[][MAX])
    index i, j, k;
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++) {
            C[i][j] = 0;
            for (k = 1; k \le n; k++)
                C[i][j] += A[i][k] * B[k][j];
```





```
int main() {
    int n, A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
    cin >> n;
    for (int i = 1; i <= n; i++)
                                                     [Input]
                                                                      [Output]
        for (int j = 1; j <= n; j++)
                                                                       28 38
            cin >> A[i][j];
                                                     2 3
                                                                       26 36
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
                                                     5 7
            cin >> B[i][j];
                                                     6 8
    matrixmult(n, A, B, C);
                                                     [Input]
                                                                      [Output]
    for (int i = 1; i <= n; i++) {
                                                                            53
                                                                                 54
                                                                                      37
        for (int j = 1; j <= n; j++)
                                                     1 2 3 4
                                                                       123 149
                                                                                 130
                                                                                      93
            cout << C[i][j] << "\t";</pre>
                                                     5 6 7 8
                                                                      95
                                                                            110
                                                                                 44
                                                                                      41
        cout << endl;</pre>
                                                     9 1 2 3
                                                                       103
                                                                            125
                                                                                 111 79
                                                     4 5 6 7
                                                     8 9 1 2
                                                     3 4 5 6
                                                     7 8 9 1
                                                     2 3 4 5
```



- 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.





- 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, 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])</pre>
            high = mid - 1;
        else // x > S[mid]
            low = mid + 1;
```





- Sequential Search .vs. Binary Search
 - Compare the number of comparisons
 - done by Algorithm 1.1 and Algorithm 1.5.
 - 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.



- 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





Problem:

- Compute the *n*th term of the **Fibonacci sequence**.
 - **-** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ···

The Fibonacci Sequence

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



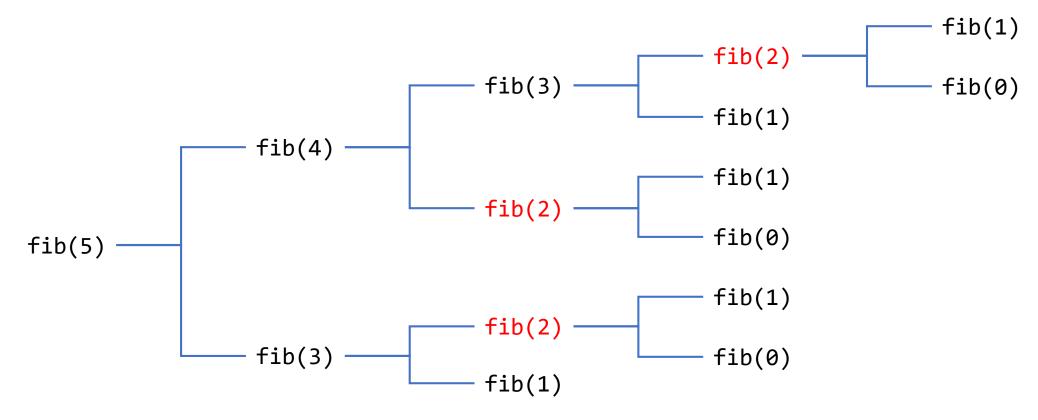
ALGORITHM 1.6: (Recursive) *n*th Fibonacci Term

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





- 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: $T(n) > 2^{n/2}$.





- 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

```
int fib2(int n) {
    index i;
    int f[MAX + 1];
   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];
```





- Two Types of Algorithm Analysis:
 - The *correctness* of an algorithm
 - develops a *proof* that the algorithm actually does
 - what it is supposed to do.
 - The **efficiency** of an algorithm
 - determines *how efficiently* the algorithm *solves* a problem
 - in terms of either *time* or *space*.





- Prove the correctness of Exchange Sort
 - Note that an algorithm should solve all instances of a problem.
 - Hence, we should show that Algorithm 1.3 *always finds*
 - a *correct solution* for *all instances* of the problem.

ALGORITHM 1.3: Exchange Sort

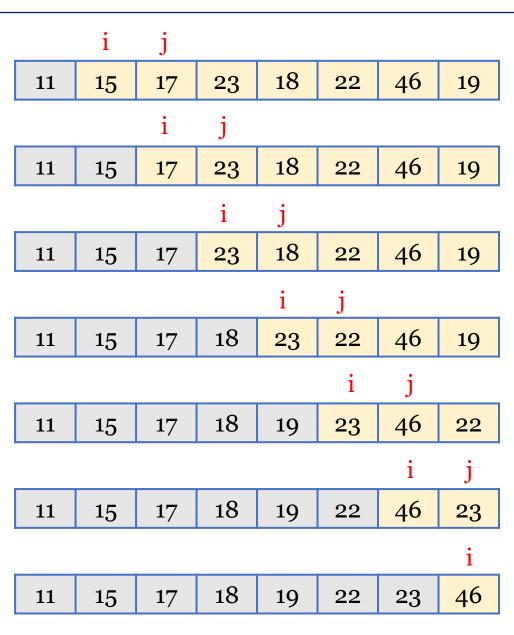




i	j						
17	15	11	23	18	22	46	19
i		j					
15	17	11	23	18	22	46	19
i			j				
11	17	15	23	18	22	46	19
i				j			
11	17	15	23	18	22	46	19
i					j		
11	17	15	23	18	22	46	19
i						j	
11	17	15	23	18	22	46	19
i							j
11	17	15	23	18	22	46	19









- Proof for the Correctness of the Algorithm 1.3:
 - Let *n* be the length of *A*.
 - After the outer for-loop completes its iteration for i = t,
 - we have a list of integers A, such as $A[t] \le A[k]$, t < k < n.
 - On subsequent iterations, for i > t,
 - there is *no change* in the values from A[0] through A[t].
 - Hence, following the last iteration of the outer for-loop,
 - we have a list of integers A, such as $A[0] \le A[1] \le \cdots \le A[n-1]$.





- Analyzing the **Efficiency** of an Algorithm
 - Two important considerations:
 - How fast does an algorithm run according to the input size?
 - *How much memories* does it require to run?
 - We need to determine the efficiency
 - independent to a particular computer on which the algorithm runs.
 - independent to a particular programming language.
 - *independent to* all the *complex details* of the algorithm.





Complexity Analysis:

- a standard technique for analyzing the efficiency of an algorithm.
- In general, analyze the efficiency of an algorithm
 - by determining the number of times some *basic operation* is done
 - as a function of the *input size*.



- Complexity Analysis of the Algorithm 1.3:
 - Finding a reasonable measure of the *input size*:
 - n: the number of elements in A.
 - Choosing a reasonable set of the *basic operation*:
 - *comparison*: a good candidate for the basic operation.
 - The total work done by the algorithm is
 - roughly *proportional to* the number of times
 - that the basic operation is done.
 - The number of comparisons with the input size n:

$$-(n-1) \times (n-2) \times \dots \times 1 = \frac{(n-1)n}{2} = \frac{1}{2}(n^2 - n)$$



1.3 Analysis of Algorithms

• Time Complexity Analysis is

- the determination of *how many times* the basic operation is done
 - for each value of the *input size*.
- Let T(n) be the number of times for an input size n.
 - T(n) is called the *every-case time complexity* of the algorithm.
- However, in some cases,
 - T(n) depends not only on the *input size*, but also on the *input's value*.



1.3 Analysis of Algorithms

- Three kinds of time complexities:
 - The **best-case** time complexity
 - the *minimum* number of times for an input size *n*.
 - The **worst-case** time complexity
 - the *maximum* number of times for an input size *n*.
 - The *average-case* time complexity
 - the *average* (*expected*) number of times for an input size *n*.





1.3 Analysis of Algorithms

- *Time Complexity* of the Algorithm 1.1 (Sequential Search):
 - Best-case: B(n) = 1.
 - Worst-case: W(n) = n.
 - Average-case time complexity:
 - for the case in which it is known that x is in A,

$$A(n) = \sum_{k=1}^{n} \left(k \times \frac{1}{n}\right) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{2}(n+1)$$

- for the case in which *x* may *not* be in *A*.
 - refer to the textbook.



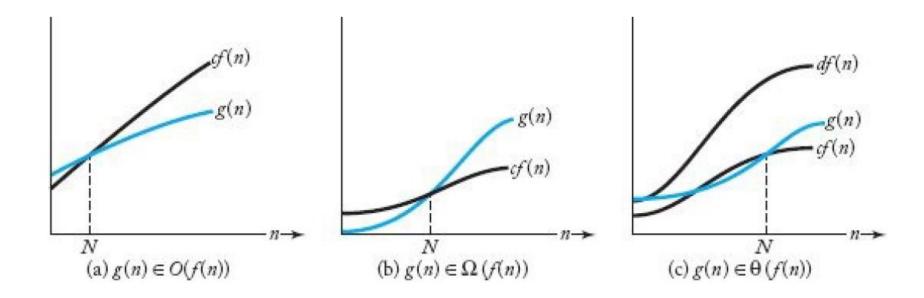
- Comparing the Efficiency of Different Algorithms
 - Suppose that we have two algorithms for the same problem:
 - Algorithm A.1 with the time complexity T(n) = n.
 - Algorithm A.2 with the time complexity $T(n) = n^2$.
 - Which one is *faster* than the other?
 - Now, two algorithms are:
 - Algorithm A.3 with T(n) = 100n.
 - Algorithm A.4 with $T(n) = 0.01n^2$.
 - Then, which one is *faster* than the other, *eventually*?





■ Asymptotic Notations: $0, \Omega, \Theta$

- **O**(*Big O*) puts an *asymptotic upper bound* on a function.
- $\Omega(Omega)$ puts an asymptotic lower bound on a function.
- $\Theta(Theta)$ puts the **order** on a function.





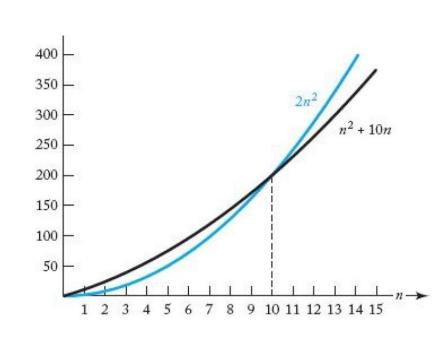


Definition of O(Big O)

For a given complexity function f(n), O(f(n)) is the set of complexity function g(n) for which there exists some positive real constant c and some nonnegative integer N such that for all $n \ge N$,

$$g(n) \le c \times f(n)$$
.

- Although g(n) starts out above cf(n),
 - eventually it falls beneath cf(n).
- For example,
 - $n^2 + 10n$ is initially above $2n^2$,
 - however, it goes below, for $n \ge 10$.





• Definition of Ω (Omega)

For a given complexity function f(n), $\Omega(f(n))$ is the set of complexity function g(n) for which there exists some positive real constant c and some nonnegative integer N such that for all $n \ge N$,

$$g(n) \ge c \times f(n)$$
.





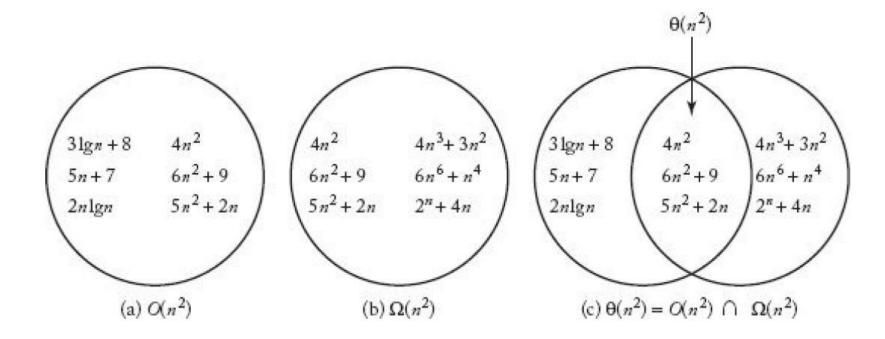
■ Definition of <mark>©(Theta)</mark>

```
For a given complexity function f(n), \Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).
```

- If $g(n) \in \Theta(f(n))$,
 - we say that g(n) is the **order** of f(n).



- Some exemplary members of $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$.
 - Note that any logarithmic or linear complexity function is in $O(n^2)$.



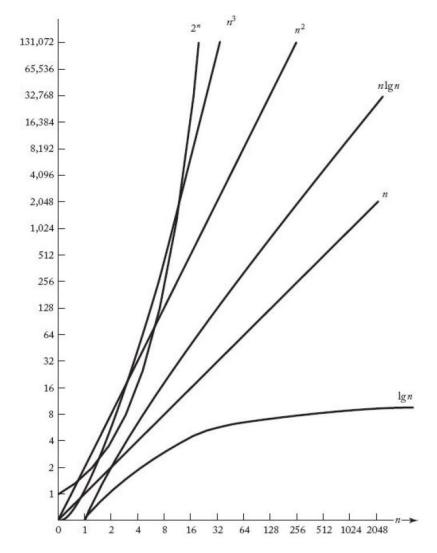


- Categorical Classification of Complexities:
 - A set of algorithms can be classified
 - by the *order* of *complexity functions*, such as $\Theta(n)$, $\Theta(n^2)$, and $\Theta(2^n)$.
 - The most common complexity categories are:
 - $\Theta(1)$: *constant* time complexity
 - $\Theta(\lg n)$: *logarithmic* time complexity
 - $\Theta(n)$: *linear* time complexity
 - $\Theta(n \lg n)$: linear logarithmic time complexity
 - $\Theta(n^2)$: quadratic time complexity
 - $\Theta(n^3)$: *cubic* time complexity
 - $\Theta(2^n)$: *exponential* time complexity
 - $\Theta(n!)$: *factorial* time complexity





Growth rates of some common complexity functions



- **Polynomial-time** complexity:
 - regarded as an *efficient* algorithm.
- *Exponential-time* complexity:
 - regarded as an *inefficient* algorithm.

- Examples of time complexity analysis:
 - Algorithm 1.1 (Sequential Search)
 - In average-case: $T(n) = \frac{1}{2}(n+1) \in \Theta(n) \in O(n)$.
 - Algorithm 1.3 (Exchange Sort)
 - $T(n) = \frac{1}{2}(n^2 n) \in \Theta(n^2) \in O(n^2).$
 - Algorithm 1.5 (Binary Search)
 - In worst case: $T(n) = \lg n + 1 \in \Theta(\lg n) \in O(\lg n)$.
 - Algorithm 1.7 (*Recursive nth Fibonacci Term*)
 - T(n) = T(n-1) + T(n-2) + 1 $> 2^{(n-1)/2} + 2^{(n-1)/2} + 1 = 2^{n/2} \in O(2^n).$

Any Questions?

