Lecture Notes on
Data Structures

M1522.000900

© 2014 - 2022 by Bongki Moon

Seoul National University

Fall 2022

Part I

Analysis

Data Structure

- An organization of data.
- Provides a way to implement operations as well as storage structures.

Example 1

A subset of $U = \{0, 1, \ldots, 7\}$ can be stored in an array of eight integers. How can you perform set intersection and set union operations?

Set intersection $C = A \cap B$ can be performed by:

```
for(i=0; i < 8 ;i++) C[i] = A[i] & B[i];
```

Set union $C = A \cup B$ can be performed by:

```
for(i=0; i < 8 ;i++) C[i] = A[i] | B[i];
```

Example 2

A subset of $U = \{0, 1, \ldots, 7\}$ can be stored in a single 8-bit integer. How can you perform set intersection and set union operations?

Set intersection $C = A \cap B$ can be performed by:

```
C = A & B;
```

Set union $C = A \cup B$ can be performed by:

```
C = A | B;
```

Example 3

How about subsets of $U = \{0, 1, \ldots, 255\}$?

- Use 256-bit integers.
- Use arrays of 256 integers.
- Use arrays of eight 32-bit integers?

Example 4

You can store a subset of $U = \{0, 1, \ldots, 255\}$ in a variable-length array to consume less memory. How can you perform set intersection and set union operations?

Basic Data Types

- integer
- real
- character
- Boolean
- . . .

Abstract Data Type

What if the basic data types are not enough? For example, 256-bit integers, complex numbers, sets, queues, stacks, trees, graphs, maps, hash tables, etc.

Create a new ADT.

- A specification of a set of values, and a set of operations that can be performed on those values.
- ADT does not specify how these operations are implemented. The specification is isolated from the implementation details. It can be implemented by a particular data structure of your choice.
- Encapsulation of values and operations.
- ADT is similar to Class, but there is no inheritance in ADT.

Example 5

An ADT for complex numbers can be designed with two floating-point numbers.

1. values: a pair of floats (r, i).
2. operations: ADD, SUBTRACT, etc.

$$ADD : (r_1, i_1) \times (r_2, i_2) \rightarrow (r_1 + r_2, i_1 + i_2)$$

Operations may be implemented in different languages (*e.g.*, Java or C). Many different implementations are possible (*e.g.*. doubles instead of floats).

Problem

- A specification of a desired *output* given some *input*.
- This is a good way of describing and/or understanding a problem.
- What do we do to solve a problem?

☞ Design a data structure and an algorithm.

Algorithm

- Method or process to follow to solve a problem.
- Expressed (in pseudo-codes) as a sequence of "simple steps" or "commands" (like a recipe).
- Implemented in a computer language like Java, C, C++, producing a program.

Example 6

Problem: print the sum of integers $1, 2, \ldots, N$.

- input: N
- output: $\sum_{i=1}^{N} i$

```
Sol 1:
   sum = 0;
   for(i=1; i <= N ;i++) sum += i;
   print sum;

Sol 2:
   sum = N * (N+1) / 2;
   print sum;
```

What happens if a negative integer is given as an input?

Example 7

Print the Fibonacci numbers $f_0, f_1, f_2, \ldots, f_N$. Fibonacci numbers are: $f_0 = 0, f_1 = 1, f_i = f_{i-1} + f_{i-2}$ $(i \geq 2)$.

- input: N
- output: $f_0, f_1, f_2, f_3, \ldots, f_N$

Sol 1:

```
f[0] = 0;
f[1] = 1;
for(i=2; i <= N ;i++) f[i] = f[i-1] + f[i-2];
print f[0], f[1], ...  , f[N];
```

Sol 2:

```
f = 0;
g = 1;
print f, g;
for(i=2; i <= N ;i++) {
   if (i==even) { f += g; print f; }
   else { g += f; print g; }
}
```

Example 8

Multiply each of N integers (A[0:N-1]) by 80.

- input: $A[0 : N - 1]$
- output: $80 \times A[0 : N - 1]$

```
Sol 1:
   for(i=0; i < N ;i++) A[i] *= 80;

Sol 2:
   for(i=0; i < N ;i++) {
      A[i] = A[i] << 4;
      tmp = A[i];
      A[i] += tmp << 2;
   }
```

Algorithm Evaluation

- Time: number of operations performed. Also, need to consider the types of operations (*e.g.*, multiplication vs. addition).
- Space

Algorithm Evaluation: How?

- Empirical evaluation: implement and test-drive an algorithm on a computer; measure the running time.
- Analysis: best case, worst case, average case.
  - Iterative algorithms : a simple counting will do.
  - Recursive algorithms : a recurrence relation may be necessary.

Recursive Algorithms

- A recursive algorithm calls itself.
- Typically, consists of
  1. a few base cases, and
  2. recursion for a few smaller problems.

Example 9

Write a recursive algorithm that computes a factorial $N!$.

Algorithm 1

```
Fact(N):
   if (N <= 1) return 1;
   else return N * Fact(N-1);
```

Let $T(n)$ be the cost of Fact(N) (*i.e.*, the number of multiplications required to compute Fact(N)). Then, the recurrence relation for $T(N)$ is:

$$\begin{aligned} T(1) &= 0, \\ T(N) &= 1 + T(N-1). \end{aligned}$$

What is the closed-form solution of $T(N)$ in terms of N?

Recurrence Relation

- Used to model the cost of recursive algorithms.
- To obtain a closed-form solution,
    1. direct derivation by expanding,
    2. mathematical induction.
- Knowing some summation techniques will help.

Summation Techniques

Example 10

Show $\sum_{i=1}^{N}(2i-1) = N^2$, knowing $\sum_{i=1}^{N} i = N(N+1)/2$.

$$\begin{aligned}\sum_{i=1}^{N}(2i-1) &= 2\sum_{i=1}^{N} i - \sum_{i=1}^{N} 1 \\ &= 2 \times \frac{N(N+1)}{2} - N \\ &= N^2.\end{aligned}$$

Remark 1 (Proof by Induction)

To prove $f(n)$ is true for any integer $n \geq 1$, show that

1. $f(1)$ is true, and
2. if $f(k)$ is true for some $k \geq 1$, so is $f(k+1)$.

Example 11

Prove $\sum_{i=1}^{N}(2i-1) = N^2$ by induction.

Base: if $N = 1$, then LHS $= 1$ and RHS $= 1$.
Induction: Assume $\sum_{i=1}^{k}(2i-1) = k^2$ for for $k \geq 1$. Then, show it is true that $\sum_{i=1}^{k+1}(2i-1) = (k+1)^2$.

$$\begin{aligned}\sum_{i=1}^{k+1}(2i-1) &= (2(k+1)-1) + \sum_{i=1}^{k}(2i-1) \\ &= (2k+1) + k^2 \\ &= (k+1)^2.\end{aligned}$$

Example 12

Derive the closed-form solution of $\sum_{i=1}^{N} i$.

$$\begin{aligned}\sum_{i=1}^{N} i &= 1 + 2 + \cdots + N, \\ \sum_{i=1}^{N} i &= N + (N-1) + \cdots + 1.\end{aligned}$$

By adding the both sides of the two equations,

$$\begin{aligned}2\sum_{i=1}^{N} i &= \underbrace{(N+1) + (N+1) + \cdots + (N+1)}_{N \ times} \\ &= N(N+1).\end{aligned}$$

Example 13

Derive the closed-form solution of $\sum_{i=1}^{N} i^2$ by perturbation.

Add the $(N+1)^{th}$ term to the summation of i^3.

$$\begin{aligned} \sum_{i=1}^{N} i^3 + (N+1)^3 &= \sum_{i=0}^{N} (i+1)^3 \\ &= \sum_{i=0}^{N} i^3 + 3\sum_{i=0}^{N} i^2 + 3\sum_{i=0}^{N} i + \sum_{i=0}^{N} 1. \end{aligned}$$

By canceling $\sum_{i=1}^{N} i^3$ and $\sum_{i=0}^{N} i^3$ from the both sides of the equation,

$$\begin{aligned} 3\sum_{i=0}^{N} i^2 &= (N+1)^3 - 3\sum_{i=0}^{N} i - \sum_{i=0}^{N} 1 \\ &= (N+1)^3 - \frac{3N(N+1)}{2} - (N+1) \\ &= \frac{N(N+1)(2N+1)}{2}. \end{aligned}$$

Example 14

Derive the closed-form solution of $\sum_{i=1}^{N} i$ by "Guess and Test" technique.

Since $\sum_{i=1}^{N} i \leq \sum_{i=1}^{N} N = N^2$, we can guess

$$\sum_{i=1}^{N} i = aN^2 + bN + c$$

for some constants a, b and c. By substituting $1, 2, 3$ for N,

$$\begin{aligned} 1 &= a + b + c, \\ 3 &= 4a + 2b + c, \\ 6 &= 9a + 3b + c. \end{aligned}$$

Then, we obtain $a = 1/2, b = 1/2$ and $c = 0$.

Example 15

Derive the closed-form solution of $\sum_{i=0}^{N} ar^i$ by "Shifting" technique.

Multiply by r.

$$\begin{aligned}\sum_{i=0}^{N} ar^i &= a + ar + ar^2 + \cdots + ar^N,\\ r\sum_{i=0}^{N} ar^i &= \qquad ar + ar^2 + \cdots + ar^N + ar^{N+1}.\end{aligned}$$

By subtracting the second equation from the first, side by side,

$$(1-r)\sum_{i=0}^{N} ar^i = a - ar^{N+1}.$$

Example 16

Derive the closed-form solution of $\sum_{i=1}^{N} i2^i$ by "Shifting" technique.

Multiply by 2.

$$\begin{aligned}\sum_{i=1}^{N} i2^i &= 1\cdot 2^1 + 2\cdot 2^2 + 3\cdot 2^3 + \cdots + n2^n,\\ 2\sum_{i=1}^{N} i2^i &= \qquad 1\cdot 2^2 + 2\cdot 2^3 + \cdots + (n-1)2^n + n2^{n+1}.\end{aligned}$$

By subtracting the second equation from the first, side by side,

$$\begin{aligned}(-1)\sum_{i=1}^{N} i2^i &= 1\cdot 2^1 + 1\cdot 2^2 + 1\cdot 2^3 + \cdots + 1\cdot 2^n - n2^{n+1}\\ &= 2(2^n - 1) - n2^{n+1}.\end{aligned}$$

As a rule of thumb, you can use

- the “Guess and Test” technique for summations with polynomials and
- the “Shifting” technique for summations with exponential functions.

Recurrence Relations

Example 17

Derive the closed-form solution of the recurrence relation

$$T(1) = 0, T(n) = T(n-1) + 1 \ (n \geq 2).$$

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= T(n-2) + 1 + 1 \\ &\vdots \\ &= T(1) + n - 1 \\ &= n - 1. \end{aligned}$$

Example 18

Derive the closed-form solution of the recurrence relation

$$T(1) = 1,\ T(n) = T(n-1) + n\ (n \geq 2).$$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\vdots \\ &= T(1) + 2 + 3 + \cdots + n \\ &= 1 + 2 + 3 + \cdots + n. \end{aligned}$$

Example 19

Derive the closed-form solution of a recurrence relation

$$T(2) = 1,\ T(n) = 2T(n/2) + n\ (n \text{ is a power of two} \geq 2).$$

Let $n = 2^k$. Then,

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/2^2) + n/2) + n = 2^2T(n/2^2) + 2n \\ &= 2^2(2T(n/2^3) + n/2^2) + 2n = 2^3T(n/2^3) + 3n \\ &\vdots \\ &= 2^{k-1}T(n/2^{k-1}) + (k-1)n \\ &= 2^{k-1} + (k-1)n \\ &= n/2 + (\log_2 n - 1)n. \end{aligned}$$

Example 20

For the recurrence relation

$$T(2) = 1,\ T(n) = 2T(n/2) + n\ (n \text{ is a power of two} \geq 2),$$

Show that

$$\begin{array}{rcll} T(n) & \leq & c_1 n \log n & \text{for } n \geq n_1 \text{ and} \\ T(n) & \geq & c_2 n \log n & \text{for } n \geq n_2 \end{array}$$

where c_1, c_2, n_1 and n_2 are positive constants.
(HINT: Use $c_1 = 1, c_2 = 1/2, n_1 = 2$ and $n_2 = 2$.)

First, to show $T(n) \leq n \log n$ for $n \geq 2$,
Base: if $n = 2$, then LHS $= 1$ and RHS $= 2$.
Induction: if $T(k) \leq k \log k$ for $k \geq 2$,

$$\begin{array}{rcl} T(2k) & = & 2T(k) + 2k \\ & \leq & 2k \log k + 2k = 2k(\log k + 1) = 2k \log 2k. \end{array}$$

Second, to show $T(n) \geq (1/2) n \log n$ for $n \geq 2$,
Base: if $n = 2$, then LHS $= 1$ and RHS $= 1$.
Induction: if $T(k) \geq (1/2) k \log k$ for $k \geq 2$,

$$\begin{array}{rcl} T(2k) & = & 2T(k) + 2k \\ & \geq & k \log k + 2k \\ & \geq & k \log k + k = k(\log k + 1) = k \log 2k = (1/2)(2k) \log 2k. \end{array}$$

Algorithm Analysis

Example 21 (Selection Sort)

input: A[0:N-1] an array of N integers,
output: $A[0] \leq A[1] \leq \cdots \leq A[N-1]$.

```
for(i=0; i < N-1 ;i++) {                             // N-1
   low = i;                                          // N-1
   for(j=N-1; j > i ;j--)       // (N-1) + (N-2) + ...  + 1
      if (A[j] < A[low])        // (N-1) + (N-2) + ...  + 1
         low = j;                                   // x(?)
   swap(A[i],A[low]);                                // N-1
}
```

The total number of steps is

$$\begin{aligned} & 2(N-1) + 2\sum_{i=1}^{N-1} i + x + (N-1) \\ = \;& 3(N-1) + N(N-1) + x \\ \leq \;& (N-1)(N+3) + \sum_{i=1}^{N-1} i \qquad (\because x \leq \sum_{i=1}^{N-1} i)) \\ = \;& (N-1)(3N/2+3) \end{aligned}$$

However, more accurate analysis would need to take into account the relative costs of the steps in the algorithm. Thus, the accurate cost would be

$$c_1(N-1) + c_2(N-1) + c_3\sum_{i=1}^{N-1} i + c_4\sum_{i=1}^{N-1} i + c_5 x + c_6(N-1)$$

for some constants $c_1, \ldots, c_6$. But, then, we are not really concerned much about the constant factors, especially when the input size (N) becomes large.

Best, Worst, Average Cases

- Not all inputs of the same size take the same amount of time.
- Worst-case analysis is important for certain applications such as real-time systems and air-traffic control systems.
- Average-case analysis is the most desirable, but difficult to determine, particularly when the input does not follow the uniform distribution. Examples of non-uniform distributions are Gaussian (normal) distribution and Zipf distribution.

Example 22 (Linear Search)

Find x in an array A[0:N-1]. Return its index if found, -1 otherwise.

```
for(i=0; i < N ;i++)
   if (A[i] == x) return i;
return -1;
```

Factors to consider:

- The values in A[] are mutually distinct? Value distribution?
- The search key x is always found in the array A[] or not?

Case 1 $N = 5, x = 1, A[] = \{1, 2, 3, 4, 5\}$ The total number of steps is 2×1. [Best case]

Case 2 $N = 5, x = 1, A[] = \{5, 4, 3, 2, 1\}$ The total number of steps is 2×5. [Worst case]

Case 3 $N = 5, x = 1, A[] = \{3, 2, 1, 4, 5\}$ The total number of steps is 2×3.

Problem 1

What is the average-case running time of the linear search? Assume the following: $Prob[x \notin A[0:n-1]] = \frac{1}{2}$, and $Prob[x = A[i]] = \frac{1}{2n}$.

Let k be how many times the condition `A[i]==x` is tested.

Then,

$$\begin{aligned} k &= 1 \times \frac{1}{2N} + 2 \times \frac{1}{2N} + \ldots + N \times \frac{1}{2N} + N \times \frac{1}{2} \\ &= \sum_{i=1}^{N} (i \times \frac{1}{2N}) + N \times \frac{1}{2} \\ &= \frac{1}{2N} \times \frac{N(N+1)}{2} + \frac{N}{2} \\ &= \frac{3N+1}{4} \end{aligned}$$

Problem 2 (**Rank by Counting**)

Rank N integers in the increasing order.
input: A[0:N-1], an array of N mutually distinct integers,
output: Rank[0:N-1], Rank[i] is the number of integers in A < A[i].

```
for(i=0; i < N ;i++) Rank[i] = 0;
for(i=0; i < N ;i++)
   for(j=0; j < N ;j++)
      if (A[i] > A[j]) Rank[i]++;
```

How many times is `Rank[i]++` executed?

Consider the following cases.

1. The values of `A[0:N-1]` are in sorted order,
2. The values of `A[0:N-1]` are in inversely sorted order.

In which case will the ranking algorithm finish sooner?

Asymptotic Analysis

In asymptotic analysis, we are interested in the properties of a function $f(n)$ as n becomes very large (*i.e.*, the limiting behavior).

- If $f(n) = n^2 + 3n$, then as n becomes very large, the term $3n$ becomes insignificant compared with n^2.
- $f(n)$ is said to be "asymptotically equivalent to n^2, as $n \to \infty$."

In computer science, asymptotic analysis refers to the study of an algorithm as the input size gets big or reaches a limit.

- It attempts to estimate the resource consumption of an algorithm.
- It allows us to compare the relative costs of two or more algorithms for solving the same problem.

Asymptotic Analysis: $\mathcal{O}/\Omega/\Theta$-Notations

Definition 1

$$\boxed{T(n) \in \mathcal{O}(f(n)) \text{ iff } \exists\ c > 0, n_0 > 0 \text{ such that } T(n) \leq cf(n) \text{ for } n > n_0.}$$

The implication is that for a large input data set ($n > n_0$), the algorithm takes no more than $cf(n)$ steps. That is, the big-oh notation provides an upper-bound of running time.

Using the Big-Oh Notation

For $T(n)$ such that $T(n) \in \mathcal{O}(f(n))$,

- We can say "$T(n)$ is $\mathcal{O}(f(n))$," "$T(n)$ is *big-oh* of $f(n)$" or "$T(n)$ is *order* of $f(n)$."
- Of course, it is also correct to say "$T(n) \in \mathcal{O}(f(n))$."
- It is considered poor taste to say "$T(n) \leq \mathcal{O}(f(n))$."
- If $f(n) - g(n) \in \mathcal{O}(h(n))$, then we can say "$f(n)$ is $g(n) + \mathcal{O}(h(n))$."

Another definition of $\mathcal{O}(f(n))$:

$$\mathcal{O}(f(n)) = \{g(n) \mid \exists\, c > 0, n_0 > 0 \text{ such that } g(n) \leq cf(n) \text{ for } n > n_0\}$$

Example 23

$T(n) = 3n^2$. Then, $T(n) \in \mathcal{O}(n^3)$, $T(n) \in \mathcal{O}(n^2)$, but $T(n) \notin \mathcal{O}(n)$.

$T(n) \notin \mathcal{O}(n)$ because $\nexists\, c > 0, n_0 > 0$ such that $T(n) = 3n^2 \leq cn$ for all $n > n_0$.

Note that $\mathcal{O}(n^2)$ is the tightest (*i.e.*, best) upper-bound of $T(n)$.

Example 24

$T(n) = 15n + 3$. Show that $T(n) \in \mathcal{O}(n)$.

$$T(n) = 15n + 3 \leq 16n \text{ for } n > 2.$$

That is, $c = 16$, $n_0 = 2$ and $f(n) = n$.

Example 25

$T(n) = 10$. Show that $T(n) \in \mathcal{O}(1)$.

Problem 3

What is the big-oh upper-bound of $T(n) = log(n!)$?

$$\begin{aligned} T(n) &= \log\left(n \times (n-1) \times (n-2) \ldots 2 \times 1\right) \\ &= \sum_{i=1}^{n} \log i \\ &\leq \sum_{i=1}^{n} \log n \\ &= n \log n. \end{aligned}$$

Thus, $T(n) \in \mathcal{O}(n \log n)$.

Definition 2

$T(n) \in \Omega(f(n))$ iff $\exists\ c > 0, n_0 > 0$ such that $T(n) \geq cf(n)$ for $n > n_0$.

The big-omega notation provides a lower-bound of running time.

Example 26

$T(n) = 3n^2 + 4n$. Then, $T(n) \in \Omega(1)$, $T(n) \in \Omega(n)$, $T(n) \in \Omega(n^2)$, but $T(n) \notin \Omega(n^3)$.

Note that $\Omega(n^2)$ is the tightest (*i.e.*, best) lower-bound of $T(n)$.

Definition 3

$T(n) \in \Theta(f(n))$ iff $T(n) \in \mathcal{O}(f(n))$ and $T(n) \in \Omega(f(n))$.

Example 27

For an algorithm with a cost function $T(n)$ such that

$$T(2) = 1, T(n) = 2T(n/2) + n\ (n \geq 2),$$

show the algorithm is in $\Theta(n \log n)$. (Hint: See Example 19 and Example 20.)

1. By expanding, $T(n) = n \log n - n/2$. Thus, $T(n) \in \Theta(n \log n)$.
2. By induction, $T(n) \leq n \log n$ for $n > 1$. Thus, $T(n) \in \mathcal{O}(n \log n)$. By induction, $T(n) \geq \frac{1}{2} n \log n$ for $n > 1$. Thus, $T(n) \in \Omega(n \log n)$. Therefore, $T(n) \in \Theta(n \log n)$.

Problem 4

What is the running time of the factorial algorithm given in Example 9 in the big-Oh notation?

(HINT: Since it is a recursive algorithm, establish a recurrence relation, and derive a closed-form formula.)

Simplifying Rules

1. If $T(n) \in \mathcal{O}(f(n))$ and $f(n) \in \mathcal{O}(g(n))$, then $T(n) \in \mathcal{O}(g(n))$.
2. If $T(n) \in \mathcal{O}(kf(n))$ for a constant k, then $T(n) \in \mathcal{O}(f(n))$.
3. If $T_1(n) \in \mathcal{O}(f_1(n))$ and $T_2(n) \in \mathcal{O}(f_2(n))$, then $T_1(n) + T_2(n) \in \mathcal{O}(\max\{f_1(n), f_2(n)\})$.
4. If $T_1(n) \in \mathcal{O}(f_1(n))$ and $T_2(n) \in \mathcal{O}(f_2(n))$, then $T_1(n) \times T_2(n) \in \mathcal{O}(f_1(n) \times f_2(n))$.

When do we use these rules?

We can write the similar set of rules for Ω and Θ notations by replacing $\mathcal{O}$ with Ω or Θ.

Question 1

If $T_1(n) \in \Omega(f_1(n))$ and $T_2(n) \in \Omega(f_2(n))$, then which is correct

$$\begin{aligned} & T_1(n) + T_2(n) \in \Omega(\max\{f_1(n), f_2(n)\}) \\ \textit{or} \quad & T_1(n) + T_2(n) \in \Omega(\min\{f_1(n), f_2(n)\})? \end{aligned}$$

Both are correct, but max is a tighter lower-bound.

Running Time of Program Segments

Example 28

What is the running time of the following code?

```
sum = 0;
for(i=0; i < n ;i++)
   for(j=0; j < n ;j++)
      sum++;
```

$T(n) \in \Theta(n^2)$.

Example 29

What is the running time of the following code?

```
sum = 0;
for(i=0; i < n ;i++)
   for(j=0; j <= i ;j++)
      sum++;
for(k=0; k < n ;k++)
   A[k] = sum;
```

$\}\ T_1(n) \in \Theta(n^2)$

$\}\ T_2(n) \in \Theta(n)$

$T(n) = T_1(n) + T_2(n)$ and $T_1(n) \in \Theta(n^2)$, $T_2(n) \in \Theta(n)$.
$\therefore\ T(n) \in \Theta(\max\{n^2, n\})$.

Example 30

What is the running time of the following code? Assume n is a power of 2.

```
sum = 0;
for(i=1; i <= n ;i*=2)
   for(j=1; j <= n ;j++)
      sum++;
```

$\}\ T_2(n) \in \Theta(n)$

$\}\ T_1(n) \in \Theta(\log n)$

$T(n) = T_1(n) \times T_2(n)$ and $T_1(n) \in \Theta(\log n)$, $T_2(n) \in \Theta(n)$.
$\therefore\ T(n) \in \Theta(\log n \times n)$.

Example 31

What is the running time of the following code? Assume n is a power of 2.

```
sum = 0;
for(i=1; i <= n ;i*=2)
   for(j=1; j <= i ;j++)
      sum++;
```

There is a dependency between the loops.
Assume $k = \log n$ (or $n = 2^k$).
$T(n) = 1 + \log n + \sum_{m=0}^{k} 2^m + \sum_{m=0}^{k} 2^m$. Thus, $T(n) \in \Theta(n)$.
(Recall $\sum_{m=0}^{k} 2^m \in \Theta(2^k)$.)

Example 32

Matrix addition for $n \times m$ matrices A, B and C.

```
for(i=0; i < n ;i++)
   for(j=0; j < m ;j++)
      C[i,j] = A[i,j] + B[i,j];
```

The running time is $T(n, m) \in \Theta(n \times m)$, and we cannot make it simpler because the variables n and m are independent.

Example 33

Binary search: Find x in a sorted array A[0:N-1] (*i.e.*, $A[i] < A[j]$ for $i < j$). Return its index if found, -1 otherwise.

```
low = 0; high = n-1;
while(low <= high) {
   mid = (low + high) / 2;
   if (A[mid] < x) low = mid + 1;
   else if (A[mid] > x) high = mid - 1;
   else return mid;
}
return -1;
```

Note that the search space (*i.e.*, the subarray to look at) is cut in half after each iteration.

Case 1 $N = 7, x = 5, A[] = \{1, 3, 4, 5, 7, 8, 9\}$
The total number of loop iterations is 1. [Best case]

Case 2 $N = 7, x = 1, A[] = \{1, 3, 4, 5, 7, 8, 9\}$
The total number of loop iterations is 3. [Worst case]

Problem 5

What is the average-case analysis of the binary search? Assume the following:

1. $x \in A[0 : n-1]$ (*i.e.*, the key x is always in the array),
2. $A[i] \neq A[j]$ for $i \neq j$ (*i.e.*, no multiple occurrences),
3. $Prob[A[i] = x] = \frac{1}{n}$ (*i.e.*, uniform distribution),
4. $n = 2^k - 1 = \sum_{i=0}^{k-1} 2^i$ (only for simpler math).

the no. of loop iterations performed until x is found	the no. of positions in A[] where x can be found	probability of the case
1	1	$1/n$
2	2	$2/n$
3	4	$4/n$
4	8	$8/n$
⋮	⋮	⋮
k	2^{k-1}	$2^{k-1}/n$

The expected number of loop iterations is given by

$$\begin{aligned}
\sum_{i=1}^{k} i \times \frac{2^{i-1}}{n} &= \frac{1}{2n}\sum_{i=1}^{k} i2^i \\
&= \frac{1}{2n} \times ((k-1)2^{k+1} + 2) \\
&= \frac{1}{n}((\log(n+1) - 1)(n+1) + 1) \\
&= \log(n+1) - 1 + \frac{\log(n+1)}{n} \\
&\in \Theta(\log n)
\end{aligned}$$