Fundamentals of Data Structures

TABLE OF CONTENTS

1. BASIC CONCEPTS
2. ARRAYS AND STRUCTURES
3. STACKS AND QUEUES
4. LISTS
5. TREES
6. GRAPHS
7. SORTING

Chap1 : BASIC CONCEPTS

TABLE OF CONTENTS

1. OVERVIEW: SYSTEM LIFE CYCLE
2. ALGORITHM SPECIFICATION
3. DATA ABSTRACTION
4. PERFORMANCE ANALYSIS
5. PERFORMANCE MEASUREMENT
1. OVERVIEW: SYSTEM LIFE CYCLE

- **Requirements**
  - Specify Problem & Result Information

- **Analysis**
  - Bottom-Up Analysis
    - Place an early emphasis on coding fine points.
    - Resulting program: loosely connected segments
    - Akin to constructing a building from a generic blueprint
  - Top-down Analysis
    - Divide program into manageable segments.
    - Domino Effect

- **Design**
  - Approach with perspectives of data object & operation
    - Perspective of data object → Abstract Data Type
    - Perspective of operation → Algorithm
    - Example: Scheduling System for a University
  - Language Independent

- **Refinement and Coding**
  - Choose representations for data objects.
  - Write algorithms for each operation of data objects.
  - Major issue: Performance
2. ALGORITHM SPECIFICATION

2.1 Introduction

Definition of Algorithm

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

- Input: ∃ (0 or more quantities that are externally supplied)
- Output: At least 1 quantity is produced.
- Definiteness: Each instruction is clear and unambiguous.
- Finiteness: It should terminate after a finite number of steps.
- Effectiveness: Every instruction must be basic.
Example 1 [Selection Sort]:

Suppose we must devise a program that sorts a set of \( n \geq 1 \) integers.

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

  - It is not an algorithm. (ambiguous instruction)
  - Where and how the integers are initially stored?
  - Where we should place the result?

- **Refined Solution**

  ```c
  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];
  }
  ```

  Implementation of *Interchange*
  - swap(&a, &b)  ⇐  Program 1.2
  - #define SWAP( x, y, t ) ( (t) = (x), (x) = (y), (y) = (t) )

  **Macro vs. Function**
  - Macro: efficient
  - Function: easy debugging

- **Final Solution: Program 1.3**
Program 1.2: Swap function

```c
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

Program 1.3: Selection sort

```c
#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, 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]);
    }
}
```
Program 1.3: Selection sort

```c
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);
    }
}
```

Example 2 [Binary Search]:

Assume that we have \( n \geq 1 \) distinct integers that are already sorted and sorted in the array \( list \). Given \( searchnum \), return an index \( i \), such that \( list[i] = searchnum \).

**Step 1.** Let left = 0 and right = n-1. middle = (left+right)/2

**Step 2.** Compare \( list[middle] \) with \( searchnum \).

- Initial Solution: Program 1.4
- Final Solution: Program 1.6
Program 1.4: Searching a sorted list

while ( there are more integers to check ) {
    middle = ( left + right ) / 2;
    if ( searchnum < list [ middle ] )
        right = middle - 1;
    else if ( searchnum = = list [middle ] )
        return middle;
    else left = middle + 1;
}

Program 1.6: Searching an ordered list

```c
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;
    while (left <= right) {
        middle = (left + right)/2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1: left = middle + 1;
                      break;
            case 0: return middle;
            case 1: right = middle - 1;
        }
    }
    return -1;
}
```

Program 1.4: Searching a sorted list

Program 1.6: Searching an ordered list
Program 1.5: Comparison of two integers

```c
#define COMPARE(x, y) (((x)<(y)) ? -1 : ((x)==(y) ? 0 : 1)

int compare(int x, int y)
{
    /* compare x and y, return -1 for less than, 0 for equal, 1 for greater */
    if ( x < y ) return -1;
    else if ( x == y ) return 0;
    else return 1;
}
```

Chap 1: BASIC CONCEPTS (Page 14)

2.2 Recursive Algorithms

- Characteristics
  - Recursive: *easier to understand* but *inefficient*
  - Example: "cycle checking" - recursion vs. iteration

- How can we write a recursive function?
  - Establish boundary conditions that terminate recursive calls.
  - Implement the recursive calls so that each call brings us one step closer to a solution.

- Two Examples
  - Binary Search
  - Permutations
int binsearch(int list[], int searchnum, int left, int right)
{
    /* search list[0] <= list[1] <= ... <= list[n-1] for searchnum*/
    int middle;
    while (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] :

Given a set of \( n \geq 1 \) elements, print out all possible permutations of this set.
For example, if the set is \{ \( a, b, c \) \}, then the set of permutations is
\{ \( ( a, b, c ) , ( a, c, b ) , ( b, a, c ) , ( b, c, a ) , ( c, a, b ) , ( c, b, a ) \) \}. It is
easy to see that, given \( n \) elements, there are \( n! \) permutations. We can
obtain a simple algorithm for generating the permutations if we look at
the set \{ \( a, b, c \) \}. We can construct the set of permutations by
printing :

\( 1) \ a \) followed by all permutations of \( ( b, c, d ) \)
\( 2) \ b \) followed by all permutations of \( ( a, c, d ) \)
\( 3) \ c \) followed by all permutations of \( ( a, b, d ) \)
\( 4) \ d \) followed by all permutations of \( ( a, b, c ) \)
```c
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

---

### 3. DATA ABSTRACTION

- **Data Types of C**
  - Basic: char, int, float, double
  - Extended: short, long, unsigned
  - Grouping: Array and Struct
  - Pointer

- **Definition of Data Type**
  A collection of objects and a set of operations that act on those objects.

- **Example: int**
  - objects: \{0, 1, -1, 2, -2, … , INT_MAX, INT_MIN\}
  - operations: \{+, -, *, /, %, …\}
Question: Is it useful to know the representation of object?

Answer: Yes but Dangerous

- Definition of Abstract Data Type (ADT)
  - A data type that separates specification of objects & operations from representation of objects & implementation of operations

- Example: Ada - package, C++ - class

- Specification of operation in ADT:
  - name of function, type of argument, and type of results
  - description of what the function do
  - Except internal representation or implementation details

⇒ Information Hiding

- Classifications of Functions of Data Types
  - Creator/Constructor: create a new instance
  - Transformer: create a new instance with other instances
  - Observer/Reporter: provide information about instance

- Example: Abstract Data Type - Natural Number
Structure \textit{Natural\_Number} is

\textbf{Objects:} an ordered subrange of the integers starting at zero and ending at the maximum integer (\textit{INT\_MAX}) on the computer

\textbf{function:}

for all \(x, y \in \text{Nat\_Number}\), \(\text{TRUE, FALSE} \in \text{Boolean}\)

and where +, -, <, and = = are the usual integer operations

\texttt{Nat\_No\ Zero( ) := 0}

\texttt{Boolean Is\_Zero(x) := if (x) return FALSE}

\texttt{else return TRUE}

\texttt{Nat\_No\ Add(x, y) := if ((x + y) \leq \text{INT\_MAX}) return x + y}

\texttt{else return \text{INT\_MAX}}

\texttt{Boolean Equal(x, y) := if (x = y) return TRUE}

\texttt{else return FALSE}

\texttt{Nat\_No\ Successor(x) := if (x = \text{INT\_MAX}) return x}

\texttt{else return x + 1}

\texttt{Nat\_No\ Subtract(x, y) := if (x < y) return 0}

\texttt{else return x - y}

end \textit{Natural\_Number}

---

4. PERFORMANCE ANALYSIS

- Criteria for Judgment of a Program
  - Meet original specification
  - Correctness
  - Documentation \Rightarrow \textit{Good Programming Style}
  - Modularity
  - Readability
  - Space Efficiency \Rightarrow \textit{Performance}
  - Time Efficiency

- Performance Analysis (complexity theory & simulation) vs. Performance Measurement (benchmark)

- Definition of \textit{Complexity}
  
  \textit{Space Complexity}: amount of memory to run program
  
  \textit{Time Complexity}: amount of computer time
4.1 Space Complexity

- Fixed Space Requirements
  - Space independent on the size of input & output
  - instruction space, space for simple variable, constant, ...
- Variable Space Requirements: $S_v(I)$
  - Space dependent on number, size, and values of I/O associated with I

Total Space Requirement $S(P) = c + S_v(I)$

- Example of Simple Arithmetic Function: $S_{abc}(I) = 0$

  ```c
  float abc(float a, float b, float c) {
    return a+b+b*c + (a+b-c)/(a+b) + 4.0;
  }
  ```

- Example of Summing a List of Numbers: Program 1.10

  ```c
  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

  ```c
  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
4.2 Time Complexity

- \( T_P = \text{compile time} + \text{execution time} \)
  - Compile time is fixed and is required only once.
  - Question: How can we evaluate execution time \( T_P \)?
  - \( T_P \) may be changed by compiler option & H/W.
  - One Idea: use Program Step

- Definition of **Program Step**
  A syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

- To calculate the program step: Use count
- Example: Program 1.12 ~ Program 1.16

```c
float sum(float list[], int n)
{
    float tempsum = 0; count++; /* for assignment */
    int i;
    for (i = 0; i < n; i++) {
        count++; /* for the for loop */
        tempsum += list[i]; count++; /* for assignment */
    }
    count++; /* last execution of for */
    count++; /* for return */ return tempsum;
}
```

Program 1.12 : Program 1.10 with count statements
float sum(float list[], int n)
{
    float tempsum = 0;
    int i;
    for (i = 0; i < n; i++)
        count += 2;
    count += 3;
    return 0;
}

Program 1.13: Simplified version of Program 1.12

float rsum(float list[], int n)
{
    count++; /* for if conditional */
    if (n) {
        count++; /* for return and rsum invocation */
        return rsum(list, n-1) + list[n-1];
    }
    count++;
    return list[0];
}

Program 1.14: Program 1.11 with count statements added
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
        int c[][MAX_SIZE], int rows, int cols)
{
    int i, j;
    for (i = 0; i < rows; i++)
    {
        for (j = 0; j < cols; j++)
        {
            c[i][j] = a[i][j] + b[i][j];
        }
    }
}

Program 1.15 : Matrix addition

void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
        int c[][MAX_SIZE], int rows, int cols)
{
    int i, j;
    for (i = 0; i < rows; i++)  {
        count++; /* for i for loop */
        for (j = 0; j < cols; j++)  {
            count++; /* for j for loop */
            c[i][j] = a[i][j] + b[i][j];
            count++;
            /* for assignment statement */
        }
        count++; /* last time of j for loop */
    }
    count++; /* last time of i for loop */
}

Program 1.16 : Matrix addition with count statements
void add(int a[][MAX_SIZE], int b[][MAX_SIZE], int c[][MAX_SIZE], int rows, int cols)
{
    int i, j;
    for (i = 0; i < rows; i++)
        for (j = 0; j < cols; j++)
            count += 2;
    count += 2;
}
count++;
}

Program 1.17 : Simplification of Program 1.16

Chap 1: BASIC CONCEPTS (Page 32)

<table>
<thead>
<tr>
<th>Statement</th>
<th>s/e</th>
<th>Frequency</th>
<th>Total step</th>
</tr>
</thead>
<tbody>
<tr>
<td>float sum(float list[], int n)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>n+1</td>
<td>n+1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>2n+3</td>
</tr>
</tbody>
</table>

Figure 1.2 : Step count table for Program 1.10

Chap 1: BASIC CONCEPTS (Page 33)
### Figure 1.3: Step count table for recursive summing function

<table>
<thead>
<tr>
<th>Statement</th>
<th>s/e</th>
<th>Frequency</th>
<th>Total step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float rsum(float list[], int n) { if (n) return rsum(list, n-1) + list[n-1]; return list[0]; }</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>n+1</td>
<td>n+1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>2n+2</td>
</tr>
</tbody>
</table>

### Figure 1.4: Step count table for matrix addition

<table>
<thead>
<tr>
<th>Statement</th>
<th>s/e</th>
<th>Frequency</th>
<th>Total step</th>
</tr>
</thead>
<tbody>
<tr>
<td>void add(int a[][MAX_SIZE] ...) { int i, j; for (i = 0; i &lt; rows; i++) for (j = 0; j &lt; cols; j++) c[i][j] = a[i][j] + b[i][j];</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rows+1</td>
<td>rows+1</td>
<td>rows+1</td>
</tr>
<tr>
<td></td>
<td>rows · cols+1</td>
<td>rows · cols + rows</td>
<td>rows · cols + rows</td>
</tr>
<tr>
<td></td>
<td>rows · cols</td>
<td>rows · cols</td>
<td>rows · cols</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>2rows · cols + 2rows + 1</td>
</tr>
</tbody>
</table>
4.3 Asymptotic Notation (Ο, Ω, Θ)

- **Motivation**
  - Determining Exact Step Count is Difficult.
  - The Notion of Step is Itself Inexact.
  - \(100n + 10\) vs. \(30n + 30\)

- **Approach**
  - Try: \(c_1 n^2 \leq T_p(n) \leq c_2 n^2\) or \(T_Q(n, m) = c_1 n + c_2 m\),
  - where \(c_1\) and \(c_2\) are nonnegative constants
  - \((c_1 n^2 + c_2 n) > c_3 n\) for sufficiently large value of \(n\)

- **Definition of** \(f(n) = \mathcal{O}(g(n))\)
  - \(f(n) = \mathcal{O}(g(n))\) iff \(\exists (c\) and \(n_0 > 0) such that \(f(n) \leq cg(n)\)
    - for all \(n, n \geq n_0\)

- **Example**
  - \(3n + 2 = \mathcal{O}(n)\) as \(3n + 2 \leq 4n\) for all \(n \geq 2\)
  - \(3n + 3 = \mathcal{O}(n)\) as \(3n + 3 \leq 4n\) for all \(n \geq 3\)
  - \(100n + 6 = \mathcal{O}(n)\) as \(100n + 6 \leq 101n\) for all \(n \geq 6\)
  - \(10n^2 + 4n + 2 = \mathcal{O}(n^2)\) as \(10n^2 + 4n + 2 \leq 11n^2\) for all \(n \geq 5\)
  - \(6*2^n + n^2 = \mathcal{O}(2^n)\) as \(6*2^n + n^2 \leq 7*2^n\) for all \(n \geq 4\)
  - \(3n + 2 \neq \mathcal{O}(1)\) & \(10n^2 + 4n + 2 \neq \mathcal{O}(n)\)

- \(f(n) = \mathcal{O}(g(n))\) means that \(g(n)\) is an upper bound of \(f(n)\).
  - \(10n^2 + 4n + 2 = \mathcal{O}(n^4)\) as \(10n^2 + 4n + 2 \leq 10n^4\) for all \(n \geq 2\)

- **Theorem:** If \(f(n) = a_m n^m + ... + a_1 n + a_0\), then \(f(n) = \mathcal{O}(n^m)\).

**Proof:**
\[
\sum_{i=0}^{m} |a_i| n^i \leq n^m \sum_{i=0}^{m} |a_i| n^{i-m} \leq n^m \sum_{i=0}^{m} |a_i|, \text{ for all } n \geq 1
\]
Definition of $f(n) = \Omega(g(n))$

$f(n) = \Omega(g(n))$ iff $\exists (c \text{ and } n_0 > 0)$ such that $f(n) \geq cg(n)$ for all $n, n \geq n_0$.

$\Leftarrow g(n)$ is a lower bound of $f(n)$.

Example

- $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for all $n \geq 1$
- $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq 10n^2 \forall n \geq 1$
- $6*2^n + n^2 = \Omega(2^n)$ as $6*2^n + n^2 \geq 6*2^n \forall n \geq 1$
- $3n + 2 = \Omega(1), 10n^2 + 4n + 2 = \Omega(n), 6*2^n + n^2 = \Omega(n^2)$

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

Definition of $f(n) = \Theta(g(n))$

$f(n) = \Theta(g(n))$ iff $\exists (c_1, c_2, \text{ and } n_0 > 0)$ such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n, n \geq n_0$.

$\Leftarrow g(n)$ is both an upper and lower bound of $f(n)$.

Example

- $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$
- $10n^2 + 4n + 2 = \Theta(n^2), 6*2^n + n^2 = \Theta(2^n)$
- $3n + 2 \neq \Theta(1), 6*2^n + n^2 \neq \Theta(n^2)$

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

- $T_{\text{sum}}(n) = 2n + 3 = \Theta(n)$
- $T_{\text{rsum}}(n) = 2n + 2 = \Theta(n)$
- $T_{\text{add}}(\text{row, col}) = 2\text{row.col} + 2\text{row} + 1 = \Theta(\text{row.col})$
- Binary Search: $\Theta(\log_2 n)$ ← see Program 1.6
- Permutation: $\Theta(n^n!)$ ← see Program 1.8
  \[ T_{\text{perm}}(i, n) = \Theta((n - i + 1) \times T_{\text{perm}}(i+1, n)) \]
- Magic Square (Figure 1.6 & Program 1.22)

---

Example 1.21 [Magic square] :

As our last example of complexity analysis, we use a problem from recreational mathematics, the creation of a magic square.

A magic square is an $n \times n$ matrix of the integers from 1 to $n^2$ such that the sum of each row and column and the two major diagonals is the same.

Figure 1.6 shows a magic square for case $n = 5$.

In this example, the common sum is 65.
Figure 1.6: Magic square for $n = 5$

Program 1.22: Magic square program

```
#include <stdio.h>
#define MAX_SIZE 15 /* maximum size of square */

void main(void)
/* construct a magic square, iteratively */
{
    static int square[MAX_SIZE][MAX_SIZE];
    int i, j, row, column; /* indices */
    int count; /* counter */
    int size; /* Square size */

    printf("Enter the size of the square: ");
    scanf("%d", &size);
    /* check for input errors */
    if (size < 1 || size > MAX_SIZE + 1) {
        fprintf(stderr, "Error! Size is out of range\n");
        exit(1);
    }
```
if (!(size % 2)) {
    print(stderr, "Error! Size is even\n");
    exit(1);
}
for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
        square[i][j] = 0;
square[0][(size-1) / 2] = 1; /* middle of first row */
/* i and j are current position */
i = 0;
j = (size - 1) / 2;
for (count = 2; count <= size * size; count++) {
    row = (i - 1 < 0) ? (size - 1) : (i - 1); /* up */
    column = (j - 1 < 0) ? (size - 1) : (j - 1); /* left */
    if (square[row][column]) { /* down */
        i = (++i) % size;
    } else {
        i = row;
        j = column;
    }
square[i][j] = count;
} /* output the magic square */
printf("Magic Square of size %d : \n\n", size);
for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
        printf("%5d", square[i][j]);
        printf("\n");
    }
    printf("\n\n");
}
5. PERFORMANCE MEASUREMENT

Plot of Function Values

Timing Program for Selection Sort

Program 1.8: Plot of function values
#include <stdio.h>
#include <time.h>
#define MAX_SIZE 1601
#define ITERATIONS 26
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void main(void)
{
    int i, j, position;
    int list[MAX_SIZE];
    int sizelist[MAX_SIZE] = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600};
    clock_t start, stop;
    double duration;
    printf("time\n");
    for (i = 0; i < ITERATIONS; i++) {
        for (j = 0; j < sizelist[i]; j++) {
            list[j] = sizelist[i] - j;
            start = clock();
            sort(list, sizelist[i]);
            stop = clock();
            /* CLK_TCK = number of clock ticks per second */
            duration = ((double) (stop - start)) / CLK_TCK;
            printf("%6d %f
", sizelist[i], duration);
        }
    }
}

Program 1.23: Timing program for the selection sort function
Figure 1.11: Worst case performance of selection sort (in seconds)

<table>
<thead>
<tr>
<th>n</th>
<th>Time</th>
<th>n</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>.00</td>
<td>900</td>
<td>1.86</td>
</tr>
<tr>
<td>200</td>
<td>.11</td>
<td>1000</td>
<td>2.31</td>
</tr>
<tr>
<td>300</td>
<td>.22</td>
<td>1100</td>
<td>2.80</td>
</tr>
<tr>
<td>400</td>
<td>.38</td>
<td>1200</td>
<td>3.35</td>
</tr>
<tr>
<td>500</td>
<td>.60</td>
<td>1300</td>
<td>3.90</td>
</tr>
<tr>
<td>600</td>
<td>.82</td>
<td>1400</td>
<td>4.54</td>
</tr>
<tr>
<td>700</td>
<td>1.15</td>
<td>1500</td>
<td>5.22</td>
</tr>
<tr>
<td>800</td>
<td>1.48</td>
<td>1600</td>
<td>5.93</td>
</tr>
</tbody>
</table>

Figure 1.12: Graph of worst case performance for selection sort