Algorithms examples
Correctness and testing

Chapter 2

2016

Computing x^n – recursive solution

- Computing x^n takes $\Theta(n)$ execution time if we are using the naive algorithm. We can devise a $\Theta(\lg n)$ algorithm using divide and conquer. The idea is to use the decomposition:

$$x^n = \begin{cases} (x^{n/2})^2 & \text{if } n \text{ is even} \\ x(x^{(n-1)/2})^2 & \text{if } n \text{ is odd} \end{cases}$$

```
POWER1(x, n)
1. if n = 0 then
2.     return 1
3. else if n is even then
4.         t ← POWER1(x, n/2)
5.         return t * t
6. else
7.       t ← POWER1(x, (n − 1)/2)
8.       return x * t * t
```

$$T(n) = \begin{cases} \Theta(1) & \text{if n=0} \\ T(\lfloor n/2 \rfloor) + \Theta(1) & \text{if } n \geq 1 \end{cases}$$

so the execution time is $T(n) = \Theta(\lg n)$ in all the situations.

Computing x^n – iterative solution

- Sometimes it is easy to derive an iterative solution from the recursive one. Iterative solutions are in general more efficient than the recursive ones because the recursive calls are avoided.

```
POWER2(x, n)
1. p ← 1
2. while n > 0 do
3.         if n is even then
4.             x ← x * x
5.             n ← n/2
6.         else
7.                 n ← n − 1
8.                 p ← p * x
9. return p
```

Note that divisibility tests and divisions by 2 can be implemented using bit operations. *n* is even if its least significant bit is 0, otherwise *n* is odd. Division by 2 is shift one position right.

Computing x^n – iterative solution example

- Let us compute 2^{26}.
  Initially $(n,p,x) = (26,1,2)$
  $(26,1,2) \rightarrow$
  $(13,1,2^2) \rightarrow$
  $(12, 2^2,2^2) \rightarrow$
  $(6, 2^2,2^4) \rightarrow$
  $(3, 2^2,2^8) \rightarrow$
  $(2, 2^{10}, 2^8) \rightarrow$
  $(1, 2^{10}, 2^{16}) \rightarrow$
  $(0, \mathbf{2^{26}}, 2^{16})$
- The algorithm can be better understood if we consider the *binary representation* of n.
- $n = 26 = 2^{16} + 2^8 + 2^2$.

Binary search

- Let a_i, $1 \leq i \leq n$, be an array of elements sorted in non-decreasing order. Consider the problem of determining whether a given element x is present in the array. In case is present we must determine a value j such that $a[j] = x$. Otherwise j is set to 0.

```
BINSEARCH(a, x)
1.  low ← 1
2.  high ← length[a]
3.  while low ≤ high do
4.        mid ← ⌊(low + high)/2⌋
4.        if x < a[mid] then
5.           high ← mid − 1
5.        else if x > a[mid] then
6.                 low ← mid + 1
7.        else
8.              j ← mid
9.              return j
10. return 0
```

We can implement a recursive version of binary search, but it will be less efficient than the iterative one because of the recursive calls. The execution time is $\Theta(\lg n)$ for unsuccessful searches in all the situations (worst, average, best). However, for successful searches The average and worst execution time is $\Theta(\lg n)$ and the best is $\Theta(1)$.

Merging

- Let's illustrate how can we implement the *merge* operation in the merge sort algorithm.

```
MERGE(a, p, q, r)
1.  i ← k ← p
2.  j ← q + 1
3.  while i ≤ q and j ≤ r do
4.          if a[i] ≤ a[j] then
5.             b[k] ← a[i]
6.             i ← i + 1
7.          else
8                 b[k] ← a[j]
9.                j ← j + 1
10.         k ← k + 1
11. while (i ≤ q) do
12.         b[k] ← a[i]
13.         i ← i + 1
14.         k ← k + 1
15. while (j ≤ r) do
16.         b[k] ← a[j]
17.         j ← j + 1
18.         k ← k + 1
19. for i = p, r do
20.       a[i] ← b[i]
```

- The time complexity of MERGE is $\Theta(r\text{-}p+1)$.

Problem 1.3-7 from the textbook

- Design a $\Theta(n \lg n)$ algorithm which, given a sequence s of n real numbers and a real number x, checks if s contains two elements with sum x.

```
CHECK-SUM(s)
1.  i ← 1
2.  j ← length[s]
3.  while i ≤ j do
4.        if s[i] < x − s[j] then
5.           i ← i + 1
6.        else if s[i] > x − s[j] then
7.                 j ← j − 1
11.       else
9.              return TRUE
10. return FALSE
```

Let n be the length of s. We can assume the elements of s are sorted (i.e. $s[1] < s[2] < \ldots < s[n]$) because sorting takes $\Theta(n \lg n)$. With this hypothesis, algorithm CHECK-SUM takes $O(n)$ time.

Iterative merge sort

- Analyzing merge sort more carefully you can notice that the process is executed as a sequence of merge stages. In the first stage we merge 1 element sub-arrays, in the second stage we merge 2 element sub-arrays, in the third stage we merge 4 element sub-arrays, ..., in the *i*-th stage we merge 2^{i-1} element sub-arrays.

```
MERGE-SORT2(a)
1. l ← 1
2. while l ≤ n do
3.         MERGE-STAGE(a, b, l)
4.         l ← 2 * l
5.         MERGE-STAGE(b, a, l)
6.         l ← 2 * l
```

The MERGE algorithm is similar with the one used by the recursive version excepting that lines 19 and 20 are discarded and *b* is added to the list of parameters.

```
MERGE-STAGE(a, b, l)
1. i ← 1
2. ▷ Merge adjacent sub-arrays of length l
3. while i ≤ n − 2l + 1 do
4.         MERGE(a, b, i, i + l − 1, i + 2l − 1)
5.         i ← i + 2 * l
6. ▷ Process the last two sub-arrays
7. if i + l − 1 < n then
8.    MERGE(a, b, i, i + l − 1, n)
9. else
10.        for j ← i, n do
11.            b[j] = a[j]
```

Selection sort

- Find the smallest element in $A[1...n]$, exchange it with the element of the first position, find the first element in $A[2...n]$, exchange it with the element of the second position, ...

The execution time is: $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1}(n-i+1) + c_5 \sum_{i=1}^{n-1}(n-j) + c_6 \sum_{i=1}^{n-1} t_i + c_7 \sum_{i=1}^{n-1} t_i + c_8(n-1) + c_9(n-1)$. This sum contains a dominant term of the form cn^2 with $c > 0$, independently of the values of t_i, and thus the best, average and worst execution times of selection sort are $\Theta(n^2)$.

SELECTION-SORT(a)

```
1. for i ← 1, length[a] − 1 do
2.     k ← i
3.     x ← a[i]
4.     for j ← i + 1, n do
5.         if a[j] < x then
6.             k ← j
7.             x ← a[j])
8.     a[k] = a[i]
9.     a[i] = x
```

Find a loop invariant !

Algorithm correctness

- A *well-defined computational problem* is a pair $P = (I,O,R)$ such that I is a specification of the set of allowed inputs, O is a specification of the set of outputs and R is a specification of the desired relation between an input and an output. A *problem instance* is given by a specific input $i \in I$. Note also that R is a functional relation.
- An algorithm for solving a problem P is ***totally correct*** iff for all problem instances $i \in I$ *it terminates* $\wedge$ *it produces the correct output* $o \in O$ (i.e. the pair $(i,o) \in R$).
- An algorithm for solving a problem P is ***partially correct*** iff for all problem instances $i \in I$ if *it terminates* $\rightarrow$ *it produces the correct output* $o \in O$ (i.e. the pair $(i,o) \in R$).

Multiplication of two integers

MULTIPLY(a, b)

1. $x \leftarrow 0$
2. $y \leftarrow b$
3. **while** $y \neq 0$ **do**
4. $\quad x \leftarrow x + a$
5. $\quad y \leftarrow y - 1$
6. **return** x

- For the multiplication problem the set *I* of inputs is a set of pairs (*a*,*b*) such that *a* and *b* are integers.
- Note that if $b < 0$ then this algorithm does not terminate.
- So, if $I = Z \times Z$ (*Z* is the set of integers) then the algorithm is only partially correct.
- However, if we restrict *I* to $Z \times N$ then the algorithm is totally correct (*N* is the set of natural numbers)

Euclid's Algorithm

Consider the well-known Euclid's algorithm for computing the greatest common divisor:

gcd(*m*,*n*) = *EUCLID*(*m*,*n*)

```
EUCLID(m, n)
1. while TRUE do
2.         r ← m mod n
3.         if r = 0 then
4.            return n
5.         m ← n
6.         n ← r
```

Correctness of Euclid's Algorithm

• Let m_0 and n_0 be the initial values of m and n. We shall prove that each time the algorithm enters the while loop the following condition holds:

$gcd(m,n) = gcd(m_0,n_0)$.

• This condition is called a *loop invariant*.

• The proof is made by induction on the number i of executions of the body of the while loop.

• If i=0 then the result is trivial because $m = m_0$ and $n = n_0$.

• For the induction step we assume that at the end of the i^{th} execution of the while loop we have $gcd(m,n) = gcd(m_0,n_0)$. It is enough to show that $gcd(m,n) = gcd(n,r)$. This follows trivially from the fact that r is computed as m modulo n.

• When the algorithm terminates m is divisible with n (because r is 0 !), so $gcd(m,n) = n$. But $gcd(m,n) = gcd(m_0,n_0)$, so the returned value n is $gcd(m_0,n_0)$.

To show that the algorithm terminates it is sufficient to notice that after each pass through the loop, the value of n decreases, so we shall have at most n_0 passes through the loop.

Euclid's Algorithm Complexity

- The number of division steps of Euclid's algorithm for $a > b > 0$ is at most $5 \log_{10} b$
- It follows that the number of division steps is O(h) where h is the number of digits of n. WHY ?
- See http://en.wikipedia.org/wiki/Euclidean_algorithm
- Theorem: Let $T(a,b)$ be the number of division steps of Euclid's algorithm, $a > b > 0$. Let n be a given natural number. The smallest numbers a and b for which $T(a,b) = n$ are the Fibonacci numbers F_{n+2} and F_{n+1}.
- According to this theorem: $b \geq F_{n+1} \geq \varphi^{n-1}$ where φ is the *golden ratio*. So $n\text{-}1 \leq \log_{\varphi} b = \log_{10} b \,/\, \log_{10} \varphi$.
- So $(n\text{-}1) \log_{10} \varphi \leq \log_{\varphi} b$. But $\varphi = (1+ \sqrt{5})/2 = 1.6180339887...$
- $\log_{10} \varphi > 1/5$ so $n\text{-}1 < 5 \log_{10} b$ i.e. $n < 5 \log_{10} b$.

Theorem Proof

- We use the induction on n.
- If $n = 1$ the smallest numbers are $a = 2$ and $b = 1$, i.e. $F_3 = 2$ and $F_2 = 1$.
- Now let us assume that the result holds for n-1 and let us prove it for n.
- Let $a = q_0 b + r_0$. $T(a, b) = 1 + T(b, r_0)$. So $T(b, r_0) = n$-1.
- According to the induction hypothesis, the minimal values for b and r_0 are F_{n+1} and F_n.
- The minimum value of a is obtained when $q_0 = 1$ for which we get $a = q_0 b + r_0 = F_{n+1} + F_n = F_{n+2}$.

Correctness of computing x^n

Let x_0 and n_0 be the initial values of x and n. Each time the algorithm enters the while loop we have: $p \times x^n = {x_0}^{n_0}$. The proof is made by induction on the number i of executions of the body of the while loop.

If $i = 0$ then $p = 1$, $x = x_0$ and $n = n_0$ so the result is obvious.

For the induction step we assume that at the end of the i-th execution we have $p \times x^n = {x_0}^{n_0}$.

If $n = 2k$ then the new values for p, x and n are p, x^2 and k so the invariant is preserved because $p \times x^{2k} = p \times (x^2)^k$. If $n = 2k+1$ the new values for p, x and n are $p \times x$, x and $2k$. The invariant is again preserved because $p \times x^{2k+1} = (p \times x) \times x^{2k}$.

When the algorithm stops $n = 0$ and so $p = {x_0}^{n_0}$. The algorithm always stops because n is decreasing after each pass through the loop.

```
POWER2(x, n)
1. p ← 1
2. while n > 0 do
3.         if n is even then
4.             x ← x * x
5.             n ← n/2
6.         else
7.             n ← n − 1
8.             p ← p * x
9. return p
```

Correctness of evaluating polynomials

Given a polynomial $p(x) = p_0 + p_1 x + \ldots + p_n x^n = \sum_{i=0}^{n} p_i x^i$ and a value x the problem is to compute $p(x)$.

EVALUATE-POLY(p, n, x)
1. $y \leftarrow 0$
2. **for** $i = n, 0$ **do**
3. $\quad y \leftarrow y * x + p[i]$
4. **return** y

We prove by induction on $i = n, \ldots, 0$ that after each execution of the for loop we have: $y = \sum_{j=0}^{n-i} p_{i+j} x^j$.
For the base case, $i = n$, we have $y = p_n$ so the result is obvious.
For the induction case, we assume that after the execution of for i $y = \sum_{j=0}^{n-i} p_{i+j} x^j$ we have. After the next pass through the loop we obtain $y = x(\sum_{j=0}^{n-i} p_{i+j} x^j) + p_{i-1} = \sum_{j=0}^{n-i+1} p_{i+j-1} x^j$ and the proof is finished.
Applying this result for $i = 0$ we obtain that the algorithm evaluates correctly $p(x)$.

Testing your program

- It is important to run your program on multiple tests.
- Write your program such that it accepts multiple input tests. This assumes preparing an input format for your tests and coding the program accordingly. It will read the tests from the standard input and write the results to the standard output.
- Then prepare a set of tests into an input test file.
- Run your tests by redirecting the standard input and (eventually) the standard output to capture the results. If the input test file is `input.txt` and the results file is `output.txt` you can run your program from the command line as follows:

```
prog.exe <input.txt >output.txt
```

- In Visual Studio go to *Project* | *Settings* | *Program arguments* and write `<input.txt` or `<input.txt >output.txt`. Also be sure that the input test file `input.txt` is located in the root directory of your project.

Compiling and linking with GCC

- Compile and link a single-file program:
  - *gcc -Wall* <<name>>.c -*o* <<name>>
    - -*Wall* turns on all the most commonly-used compile warnings. It is recommended that you always use this option!
    - -*o* specifies the file with the machine code (.exe extension by default)
- Compile and link a multiple-file program:
  - *gcc -Wall* <<name1>>.c <<name2>>.c... -*o* <<name>>
- Compile a single-file program
  - *gcc -Wall -c* <<name>>.c
- Link a single- or multiple-file program
  - *gcc* <<name1>>.o <<name2>>.o -*o* <<name>>

Example – Euclid's algorithm

```
#include <stdio.h>
int euclid(int m,int n);
int m,n;
main() {
    scanf("%d%d",&m,&n);
    while (m>0 && n>0) {
        printf("gcd(%3d,%3d)=%3d\n",m,n,euclid(m,n));
        scanf("%d%d",&m,&n);
    }
}
int euclid(int m,int n) {
    int r;
    while (1) {
        r = m % n;
        if (r == 0) return n;
        m = n; n = r;
    }
}
```

Example – binary search

```
#include <stdio.h>
int bsearch(int a[],
  int n,int x);
int n;
int a[100],x;
main() {
  scanf("%d",&n);
  while (n>0) {
    int i;
    for (i=1;i<=n;i++)
      scanf("%d",&a[i]);
    scanf("%d",&x);
    printf("%3d\n",
      bsearch(a,n,x));
    scanf("%d",&n);
  }
}
```

```
int bsearch(int a[],
  int n,int x) {
  int l = 1,r = n,m;
  while (l<=r) {
    m = (l+r) / 2;
    if (x<a[m]) {
      r = m-1;
    }
    else if (x>a[m]) {
      l = m+1;
    }
    else {
      return m;
    }
  }
  return 0;
}
```

Experimenting with algorithms in Python

- Algorithms are implemented using Python functions.
- The main program is a Python script that calls the function that implements the algorithm. The call is placed in a loop.
- The main script and the function implementing the algorithm can be placed in the same script file or they can be placed in separate files.
- The main script, let us suppose that it is called main.py, can be invoked from the command line as follows:

  ```
  python main.py
  ```

  or it can be run directly:

  ```
  main.py
  ```
- Input and output redirection work similarly with running C executable files from the command line.

Euclid's algorithm in Python

```
def gcd(m,n):
    while True:
        r = m % n
        if r == 0:
            return n
        m, n = n, r
```

Block structure is marked with 4 spaces indentation

Multiple assignments are allowed on the same text line

Reading from standard input in Python

- The `input()` function can be used to read a line of text.
- A string can be chopped into substrings separated by given delimiters (implicitly whitespace, for example blanks) using function `split()`.
- Example: `"abc 1x23 et 234"` → `["abc","1x23","et","234"]`
- Reading and printing two numbers, an integer and a float, can be achieved as follows:

```
l = input()
s = l.split()
n = int(s[0])
f = float(s[1])
print(n,f)
```

Function call notation from object oriented programming.
`l` is an object of class `str`, while `split()` is a method of class `str`.

Main script for experimenting with Euclid's algorithm

```
s = input().split()
m = int(s[0])
n = int(s[1])
while m>0 and n>0:
    d = gcd(m,n)
    print (m,n,d)
    s = input().split()
    m = int(s[0])
    n = int(s[1])
```

- Homework: Use the same method to experiment with binary search in Python.

Simplifying the main script

```
m, n = [int(x) for x in input().split()]
while m>0 and n>0:
    d = gcd(m,n)
    print (m,n,d)
    m, n = [int(x) for x in input().split()]
```

Uses “list comprehension” from functional programming.

A list can be defined by collecting the results of evaluating an expression for each element of an “iterable” (that can be also a list).

Problems

1. Let us consider a sequence of natural numbers: 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, …. For example, the 14^{th} element of this sequence is 4. Design an algorithm for computing the *n*-th element of this sequence.
2. Let us consider a book with numbered pages. The numbering requires the use of *n* digits. Design an algorithm to determine the number of book pages. For example, for a book with 500 pages the number of digits is: 9 (1..9) + 2 × 90 (10..99) + 3 × 401 (100..500) = 9 + 180 + 1203 = 1392 digits
3. Let *n* be a natural number. Design an algorithm to determine all the possible representations of *n* as a sum of consecutive natural numbers. For example, 30 = 9+10+11 = 6+7+8+9 = 4+5+6+7+8.
4. Let $1 < q \leq 9$ be a digit (different from 0 and 1) and let *n* be a natural number. Design an algorithm to determine the largest natural number $x \leq n$ which written in base *q* is made of only 0s and 1s. For example, if $n = 17$ and $q = 3$ then $x = 13 = 111_3$. But if $n = 10$ then $x = 10 = 101_3$.