Data Structure and Algorithm

Class 1

Seongjin Lee

Updated: 2021-02-01
DSA_2017_01

insight@gnu.ac.kr
http://resourceful.github.io
Systems Research Lab.
GNU

Table of contents

Miscellanea

Text Book

Fundamentals of Data Structure in C, 2nd Ed.

by Horowitz, Sahni, and Anderson-Freed

http://www.cise.ufl.edu/~sahni/fdsc2ed/

Presentations are uploaded in

- https://github.com/resourceful/lecture_dsa2017-1

Contact

E-mail: insight@gnu.ac.kr

Room: 407-314

Visiting Hour: Monday and Wednesday 14:00 - 16:00

Evaluation

- Midterm - 20%
- Final - 30%
- Assignments - 40%
- Attendance - 10%

Basic Concepts

Overview: System Life Cycle

Requirements

- describe informations(input, output, initial)

Analysis

- bottom-up, top-down

Design

- data objects and operations performed on them

Coding

- choose representations for data objects and write algorithms for each operation

Overview: System Life Cycle Cnt'd

Verification

- **correctness proofs**: select algorithms that have been proven correct
- **testing**: working code and sets of test data
- **error removal**: If done properly, the correctness proofs and system test indicate erroneous code

Algorithm Specification

Algorithm Specification

Definition

- a finite set of instructions - accomplish a particular task

Criteria

- zero or more inputs
- at least one output
- definiteness(clear, unambiguous)
- finiteness(terminates after a finite number of steps)

Algorithm Specification: Selection Sort

Ex Selection Sort: Sort n($geq1$) integers

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

```
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];
}
```

Algorithm Specification: Selection Sort

finding the smallest integer

- assume that minimum is list[i]
- compare current minimum with list[i+1] to list[n-1] and find smaller number and make it the new minimum

interchanging minimum with list[i]

- **function**: swap(&a,&b)
- **macro**: swap(x,y,t)
- The function's code is easier to read than that of the macro but the macro works with any data type

Algorithm Specification: Selection Sort

- **function**: swap(&a,&b)

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

   *x = *y;

   *y = temp;
}
```

- **macro**: swap(x,y,t)

```
#define SWAP(x,y,t) ((t) = (x), (x) = (y), (y) = (t))
```

Algorithm Specification: Binary Search

assumption

- sorted n(1) distinct integers stored in the array list

return

- index i (if i, list[i] = searchnum)
- or -1 (otherwise)

Algorithm Specification: Binary Search

denote left and right

- left and right ends of the list to be searched
- initially, left=0 and right=n-1

let middle=(left+right)/2 middle position in the list

compare list[middle] with the searchnum and adjust left or right

value	1	5	7	8	13	19	20	23	29
index	0	1	2	3	4	5	6	7	8
variable	left				middle				right

assume searchnum is 23

Algorithm Specification: Binary Search

compare list[middle] with searchnum

1. searchnum < list[middle] set right to middle-1
2. searchnum = list[middle] return middle
3. searchnum > list[middle] set left to middle+1

value	1	5	7	8	13	19	20	23	29
index	0	1	2	3	4	5	6	7	8
variable						left			right

Algorithm Specification: Binary Search

if searchnum has not been found and there are more integers to check

- recalculate middle and continue search
- determining if there are any elements left to check

value	1	5	7	8	13	19	20	23	29
index	0	1	2	3	4	5	6	7	8
variable						left	middle		right

- handling the comparison (through a function or a macro)

value	1	5	7	8	13	19	20	23	29
index	0	1	2	3	4	5	6	7	8
variable								left	right

Algorithm Specification: Binary Search

- **function:** compare(int x, int y)

```
int compare(int x, int y){
   if (x < y) return -1;
   else if (x == y) return 0;
      else return 1;
}
```

- **macro:** COMPARE(x, y)

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

Algorithm Specification: Binary Search

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

Recursive Algorithms

Recursive Algorithms

direct recursion

- call themselves

indirect recursion

- call other function that invoke the calling function again

recursive mechanism

- extremely powerful
- allows us to express a complex process in very clear terms

any function that we can write using assignment, if-else, and while statements can be written recursively

Recursive Algorithms: Binary Search

transform iterative version of a binary search into a recursive one

- establish boundary condition that terminate the recursive call
  1. success: list[middle]=searchnum
  2. failure: left & right indices cross
- implement the recursive calls so that each call brings us one step closer to a solution

Recursive Algorithms: Binary Search

```
int binsearch(int list[],int searchnum,int left,int right) {
   int middle;
   if(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;
}
```

Recursive Algorithms: Permutations

given a set of n(1) elements

- print out all possible permutations of this set

eg) if set a,b,c is given,

- then set of permutations is
  (a,b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a)

Recursive Algorithms: Permutations

if look at the set a,b,c,d, the set of permutations are

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)

"**followed by all permutations**" : clue to the recursive solution

Recursive Algorithms: Permutations

```
void perm(char *list,int i,int n) {
   int j,temp;
   if(i==n) {
      for(j=0;j<=n;j++)
      printf("%c", list[j]);
      printf(" ");
   }
   else {
      for(j=i;j<=n;j++) {
         SWAP(list[i],list[j],temp);
         perm(list,i+1,n);
         SWAP(list[i],list[j],temp);
      }
   }
}
```

initial function call is **perm(list,0,n-1);**

recursively generates permutations **until i=n**

Data Abstraction

Data Abstraction: Data Type

definition

- a collection of objects and
- a set of operations that act on those objects

- basic data type
  - char, int, float, double
- composite data type
  - array, structure
- user-defined data type
- pointer data type

Data Abstraction: Abstract Data Type (ADT)

definition

- **data type** that is organized in such a way that
- **the specification** of the objects and **the specification** of the operations on the objects is separated from
- **the representation** of the objects and **the implementation** of the operations

Data Abstraction

specification

- names of every function
- type of its arguments
- type of its result
- description of what the function does

Data Abstraction

classify the function of data type

- **creator/constructor**: These functions create a new instance of the designated type.
- **transformers**: These functions also create an instance of the designated type, generally by using one or more other instance.
- **observers/reporters**: These functions provide information about an instance of the type, but they do not change the instance.

Data Abstraction: Abstract Data Type

```
structure Natural_Number(Nat_No) is
  objects: an ordered subrange of the integers
           starting at zero and ending at the max.
           integer on the computer
  functions: for all x, y in Natural_Number;
           TRUE, FALSE in Boolean,
           and where +, -, <, and == are
           the usual integer operations,

  Nat_No Zero() ::= 0
  Nat_No Add(x,y) ::= if ((x+y)<=INT_MAX) return x+y
    else return INT_MAX
  Nat_No Subtract(x,y) ::= if (x<y) return 0
    else return x-y
  Boolean Equal(x,y) ::= if (x==y) return TRUE
    else return FALSE
  Nat_No Successor(x) ::= if (x==INT_MAX) return x
    else return x+1
  Boolean Is_Zero(x) ::= if (x) return FALSE
    else return TRUE
end Natural_Number
```

Data Abstraction

objects and **functions** are two main sections in the definition

function Zero is a **constructor**

function Add, Substractor, Successor are **transformers**

function Is_Zero and Equal are **reporters**

Performance Analysis

Performance Analysis

Performance evaluation

- performance analysis: machine independent complexity theory
- performance measurement: machine dependent

space complexity

- the amount of memory that it needs to run to completion

time complexity

- the amount of computer time that it needs to run to completion

Performance Analysis: Space Complexity

fixed space requirements

- don't depend on the number and size of the program's inputs and outputs
- eg) instruction space

variable space requirement

- the space needed by *structured variable* whose size depends on the particular instance, I, of the problem being solved

Performance Analysis: Space Complexity

total space requirement $S(Program)$

```
    S(Program) = c + Sp(I)
```

c:

- constant representing the fixed space requirements

$S_p(I)$:

- function of some characteristics of the instance I
- variable space requirements for program 'p'

Performance Analysis: Space Complexity

```
float calculation(float a, float b, float c) {
   return a+b+b*c+(a+b-c)/(a+b)+4.00;
}
```

- input - three simple variables
- ouput - a simple variable
- There is no variable space requirement, fixed space requirements only
- $S_{calculation}(\mathrm{I}) = 0$

Performance Analysis: Space Complexity

Iterative Version

```
float sum(float list[], int n) {
   float tempsum = 0;
   int i;
   for(i = 0; i < n; i++)
      tempsum += list[i];
      return tempsum;
}
```

- input - an array variable
- output - a simple variable

Performance Analysis: Space Complexity

Pascal pass arrays **by value**

- entire array is copied into temporary storage before the function is executed
- $S_{sum}(I) = S_{sum}(n) = n$

C pass arrays **by pointer**

- passing *the address of the first element* of the array
- $S_{sum}(n) = 0$

Performance Analysis: Space Complexity

Recursive Version

```
float rsum(float list[],int n) {
   if(n) return rsum(list,n-1) + list[n-1];
   return 0;
}
```

handled recursively

- compiler must save
  - the parameters
  - the local variables
  - the return address
- for each recursive call

Although Recursive version allows to express very clear, it has a greater overhead than its iterative counterpart

Performance Analysis: Space Complexity

space needed for one recursive call

- number of bytes required for the two parameters and the return address
- 12 bytes needed on Intel-i7 (depends on the architecture)
  - 4 bytes for pointer list[]
  - 4 bytes for integer n
  - 4 bytes for return address

assume array has n=MAX_SIZE numbers,

total variable space S_{rsum}(MAX_SIZE)

- S_{rsum}(MAX_SIZE) = 12 * MAX_SIZE

Performance Analysis : Time Complexity

Performance Analysis: Time Complexity

The time T(P),taken by a program P,

- is the sum of its compile time and its run(or execution) time
- We really concerned only with the program's execution time, Tp

count the number of operations the program performs

- give a machine-independent estimation

Performance Analysis: Time Complexity

Iterative summing of a list of numbers

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

Performance Analysis: Time Complexity

eliminate most of the program statements from Program to obtain a simpler program that **computes the same value for count**

```
float sum(float list[], int n) {
   float tempsum=0;
   int i;
   for(i = 0; i < n; i++)
      count += 2;
   count += 3;
   return tempsum;
}
```

for one time execution **sum** function

- `count += 2` in for loop n time : $2n$
- `count += 3` : 3
- **total** $2n + 3$ **steps**

Performance Analysis: Time Complexity

Recursive summing of a list of numbers

```
float rsum(float list[], int n) {
   count++;
   if(n) {
      count++;
      return rsum(list,n-1)+list[n-1];
   }
   count++;
   return 0;
}
```

Performance Analysis: Time Complexity

when n=0 only the if conditional and the second return statement are executed (termination condition)

- step count for $n = 0$: 2
- each step count for $n > 0$: 2

total step count for function : $2n + 2$

- less step count than iterative version, but
- take more time than those of the iterative version

Performance Analysis: Time Complexity

Matrix Addition determine the step count for a function that adds two-dimensional arrays(rows and cols)

```
void add(int a[][M_SIZE],int b[][M_SIZE],int c[][M_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];
}
```

Performance Analysis: Time Complexity

apply step counts to add function

```
void add(int a[][M_SIZE],int b[][M_SIZE], int c[][M_SIZE],
         int rows,int cols) {
   int i,j;
   for(i = 0; i < rows; i++) {
      count++;
      for(j = 0; j < cols; j++) {
         count++;
         c[i][j] = a[i][j] + b[i][j];
         count++;
      }
      count++;
   }
   count++;
}
```

Performance Analysis: Time Complexity

combine counts

```
void add(int a[][M_SIZE],int b[][M_SIZE],int c[][M_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++;
}
```

- initially count = 0;
- total step count on termination : $2 \cdot \text{rows} \cdot \text{cols} + 2 \cdot \text{rows} + 1$;
- In this case, If the number of rows is more than the number of columns, swaping rows and columns will take fewer steps

Performance Analysis: Time Complexity

Tabular Method

construct a step count table

1. first determine the step count for each statement
   - steps/execution(s/e)
2. next figure out the number of times that each statement is executed
   - frequency
3. total steps for each statement
   - (total steps)=(s/e)* frequency)

Iterative function to sum a list of numbers

Statement	s/e	Frequency	Total steps
`float sum(float list[],int n) {`	0	0	0
`  float tempsum=0;`	1	1	1
`  int i;`	0	0	0
`  for(i=0;i<n;i++)`	1	n+1	n+1
`    tempsum+=list[i];`	1	n	n
`  return tempsum;`	1	1	1
`}`	0	0	0
total			2n+3

Figure: step count table

Recursive function to sum a list of numbers

Statement	s/e	Frequency	Total steps
`float rsum(float list[],int n) {`	0	0	0
`if(n)`	1	n+1	n+1
`return rsum(list,n-1)+list[n-1];`	1	n	n
`return 0;`	1	1	1
`}`	0	0	0
total			2n+2

Figure: step count table for recursive summing function

Performance Analysis: Time Complexity

Matrix addition

Statement	s/e	Frequency	Total steps
void add(int a[][M_SIZE] ···) {	0	0	0
int i,j;	0	0	0
for(i=0;i<rows;i++)	1	rows+1	rows+1
for(j=0;j<cols;j++)	1	rows· (cols+1)	rows·cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows·cols	rows·cols
}	0	0	0
total			2·rows·cols+2·rows+1

Figure: step count table for matrix addition

Performance Analysis: Time Complexity

factors: time complexity

1. input size
   - depends on size of input(n): T(n) = ?
2. input form
   - depends on different possible input formats
     - average case: A(n) = ?
     - worst case: W(n) = ?
   - concerns mostly for "worst case"
   - worst case gives "upper bound"
     - exist different algorithm for the same task
     - which one is faster ?

- The "worst case" means case that has maxium number of steps

Performance Analysis : Asymptotic Notation

Performance Analysis: Asymptotic Notation

comparing time complexities

- exist different algorithms for the same task
- which one is faster ?

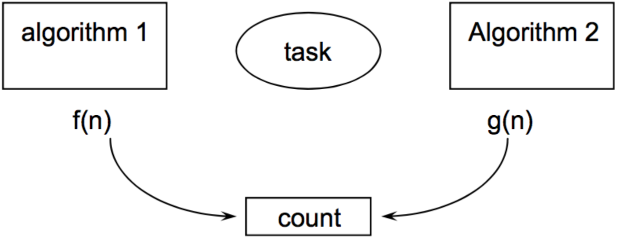

Figure: Which one is faster?

Big "OH"

- **def** f(n) = O(g(n))
  - iff(if and only if) there exist positive constants c and no such that
  - f(n) ≤ c·g(n) for all n, n ≥ no(no is the break even point)

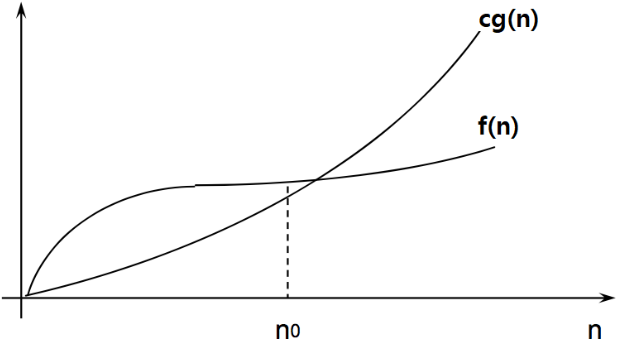

Figure: Which one is faster?

Performance Analysis: Asymptotic Notation

$f(n) = 25 \cdot n$, $g(n) = 1/3 \cdot n^2$

- $25 \cdot n = O(n^2/3)$ if let $c = 1$
- 'f of n' is 'big-oh of g of n'

n	$f(n) = 25 \cdot n$	$g(n) = n^2 / 3$
1	25	1/3
2	50	4/3
.	.	.
.	.	.
.	.	.
75	1875	1875

Figure: Which one is faster?

$|25 \cdot n| \le 1 \cdot |n^2/3|$ for all $n \ge 75$

Performance Analysis: Asymptotic Notation

$f(n) = O(g(n))$

- $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n \geq n_0$
- but, doesn't say anything about how good this bound is
  - $n = O(n^2)$, $n = O(n^{2.5})$
  - $n = O(n^3)$, $n = O(2^n)$
- $g(n)$ should be as small a function of n as one can come up with for which $f(n) = O(g(n))$

$\mathbf{f(n) = O(g(n)) \neq O(g(n)) = f(n)}$

(It is meaningless to say that $O(g(n)) = f(n)$)

theorem) if $f(n) = a_m n^m + ... + a_1 n + a_0$, then $f(n) = O(n^m)$

proof)

$$
\begin{aligned}
f(n) &\le |a_k| \cdot n^k + |a_{k-1}| \cdot n^{k-1} + ... + |a_1| \cdot n + |a_0| \\
&= |a_k| + |a_{k-1}|/n + ... + |a_1|/n^{k-1} + |a_0|/n^k \cdot n^k \\
&\le |a_k| + |a_{k-1}| + ... + |a_1| + |a_0| \cdot n^k \\
&= c \cdot n^k (c = |a_k| + |a_{k-1}| + ... + |a_1| + |a_0|) = O(n^k)
\end{aligned}
$$

Performance Analysis: Asymptotic Notation

Omega def) f(n) = Ω (g(n))

- iff there exist positive constants c and no such that f(n) c· g(n) for all n, n≥ n^0

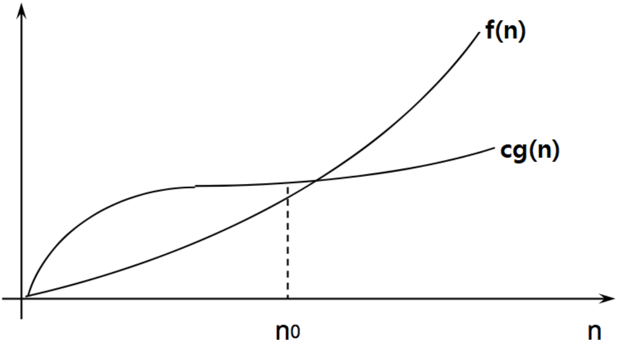

Figure: Which one is faster?

Omega

- g(n) is a lower bound on the value of f(n) for all n, n ≥ no
- should be as large a function of n as possible

theorem) if $f(n) = a_m n^m + ... + a_1 n + a_0$ and $am > 0$, then $f(n) = \Omega(n^m)$

Theta def) f(n) = Θ(g(n))

- iff there exist positive constants c^1, c^2, and n^0 such that
- $c^1 \cdot g(n) \leq f(n) \leq c^2 \cdot g(n)$ for all n, $n \geq n^0$

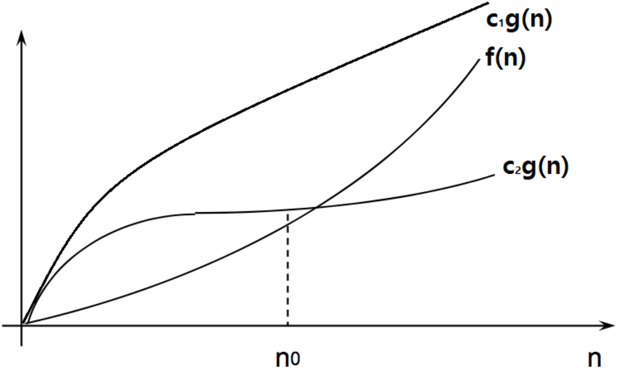

Figure: Which one is faster?

Theta

- more precise than both the “big oh” and omega notations
- g(n) is both an upper and lower bound on f(n)

Complexity of matrix addition

Statement	Asymptotic complexity
void add(int a[][M_SIZE] ⋯) {	0
int i, j;	0
for(i = 0; i < rows; i++)	$\Theta(rows)$
for(j = 0; j < cols; j++)	$\Theta(rows \cdot cols)$
c[i][j] = a[i][j] + b[i][j];	$\Theta(rows \cdot cols)$
}	0
Total	$\Theta(rows \cdot cols)$

Figure: time complexity of matrix addition

Performance Analysis : Practical Complexities

Performance Analysis: Practical Complexities

$O(1)$: constant

$O(\log_2 n)$: logarithmic

$O(n)$: linear

$O(n \cdot \log_2 n)$: log-linear

$O(n^2)$: quadratic

$O(n^3)$: cubic

polynomial time

$O(2^n)$: exponential

$O(n!)$: factorial

exponential time

Figure: Class of time complexities

Performance Analysis: Practical Complexities

polynomial time

- tractable problem exponential time
- intractable (hard) problem

eg)

- sequential search
- binary search
- insertion sort
- heap sort
- satisfiablity problem
- testing serializable scheduling

instance characteristic n							
time	name	1	2	4	8	16	32
1	constant	1	1	1	1	1	1
log n	logarithmic	0	1	2	3	4	5
n	linear	1	2	4	8	16	32
n log n	log linear	0	2	8	24	64	160
n^2	quadratic	1	4	16	64	256	1024
n^3	cubic	1	8	64	512	4096	32768
2n	exponential	2	4	16	256	655536	4294967296
n!	factorial	1	2	24	40326	20922789888000	26313×10^{33}

Figure: function value

Performance Analysis: Practical Complexities

If a program needs 2^n steps for execution

- n=40: number of steps = 1.1*1012 in computer systems
  - 1 billion steps/sec — 18.3 min
- n=50 — 13 days
- n=60 — 310.56 years
- n=100 — 4*1013 years

If a program needs n^{10} steps for execution

- n=10 — 10 sec
- n=100 — 3171 years