Data Structure: Introduction

chap. 1.1, 1.3-1.5

system life cycle

- programming is more than writing code
- development process → system life cycle
  - sequential, but highly interrelated

system life cycle

- requirements
  - define the purpose of the project
  - describe information including input and output
- analysis
  - break the problems into manageable pieces
  - bottom-up vs. top-down
- design
  - view the system as both data objects and operations
  - for example, scheduling system for a university
    - objects: students, courses, professors…
    - operations: inserting, removing, and searching each object…

system life cycle

- coding
  - choose representations for data objects and write algorithms for each operation
- verification
  - correctness proofs
    - can select algorithms that have been proven correct
  - testing
    - with working code and sets of test data
    - include all possible scenarios (more than syntax error)
    - running time should be considered

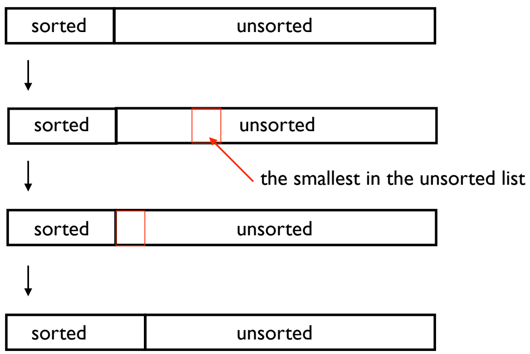
algorithm

- an algorithm is a finite set of instructions that accomplishes a particular task
- algorithms satisfy the following criteria
  - zero or more inputs
  - at least one output
  - definiteness (clear, unambiguous)
  - finiteness (terminates after a finite number of steps)
  - effectiveness

algorithm: selection sort

algorithm: selection sort

From the unsorted integers, find the smallest and place it next to the sorted list.



algorithm: selection sort

From the unsorted integers, find the smallest and place it next to the sorted list.

```
for (i = 0; i <  n-1; i++) {
      examine list[i] to list[n-1] to find the smallest integer (i.e. list[min])
      interchange list[i] and list[min];
}
```

i	[0]	[1]	[2]	[3]	[4]
	30	10	50	40	20
0	10	30	50	40	20
1	10	20	50	40	30
2	10	20	30	40	50
3	10	20	30	40	50

algorithm: selection sort

From the unsorted integers, find the smallest and place it next to the sorted list.

```
for (i = 0; i <  n; i++) {
     Examine list[i] to list[n-1] to find the smallest integer (i.e. list[min])
     interchange list[i] and list[min];
}
```

```
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);
   }
}
```

algorithm: selection sort

```
#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;
  int 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]);
  }
  sort(list,n);
  printf("\n Sorted array:\n ");
  for (i = 0; i < n; i++) /* print out sorted numbers */
    printf("%d  ",list[i]);
  printf("\n");
}
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);
  }
}
```

Program 1.3: Selection sort

algorithm specification: binary search

find query in the list

middle = (start + end) / 2;
compare list[middle] with query

1) query < list[middle]

   set right to middle-1

2) query = list[middle]

   return middle

3) query > list[middle]

   set left to middle+1

[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	14	26	30	43	50	52

left	right	middle	list[middle]	:	searchnum
0	6	3	30	<	43
4	6	5	50	>	43
4	4	4	43	==	43

left	right	middle	list[middle]	:	searchnum
0	6	3	30	>	18
0	2	1	14	<	18
2	2	2	26	>	18
2	1	-			

algorithm specification: binary search

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

```
int   binsearch  (int list[], int query, int start, int end) {

     int middle;
     while(start <= end) {
          middle = (start + end) / 2;
          switch(compare(list[middle],query)) {
                case -1: start = middle + 1;   break;
                case  0: return middle;
                case  1: end = middle - 1;
          }
     }
     return -1;
}
```

recursive algorithms

- recursion
  - direct recursion: call themselves
  - indirect recursion: call other functions 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

establish boundary condition that terminates the recursive call
1) success
list[middle]=query
2) failure
start & end indices cross

```
int   binsearch  (int list[],  int query,  int start,  int end) {
        int middle;
        if(start <= end) {
                middle=(start+end) / 2;
                switch(compare(list[middle], query)) {
                      case -1 : return binsearch(list,  query,  middle+1,  end);
                      case  0 : return middle;
                      case  1 : return binsearch(list,  query,  start,  middle-1);
            }
        }
        return -1;
}
```

recursive algorithms: permutations

given a set of n(≥1) elements, print out all possible permutations of this set

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, b, a) (c, a, b)

recursive algorithms: permutations

given a set of n(≥1) elements, print out all possible permutations of this set

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, b, a) (c, a, b)

for the set {a,b,c}, the set of permutations are
1) a followed by all permutations of (b,c)   (a, (b, c))
2) b followed by all permutations of (a,c)   (b, (a, c))
3) c followed by all permutations of (b,a)   (c, (b, a))

recursive algorithms: permutations

given a set of n(≥1) elements, print out all possible permutations of this set

if set {a,b,c} is given, then set of permutations is
(a, b, c, d) (a, b, d, c) (a, c, b, d) (a, c, d, b) (a, d, c, b) (a, d, b, c)
(b, a, c, d) :

for the set {a,b,c,d}, the set of permutations are
1) a followed by all permutations of (b,c,d) (a, (b, c, d))
2) b followed by all permutations of (a,c,d) (b, (a, c, d))
3) c followed by all permutations of (b,a,d) (c, (b, a, d))
4) d followed by all permutations of (b,c,a) (d, (b, c, a))

recursive algorithms: permutation

```
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("\n");
    else {
        for(j=i; j<=n; j++) {
            swap(list[i], list[j]);
            perm(list, i+1, n);
            swap(list[i], list[j]);
        }
    }
}
```

```
void main(){
    :
    perm(list, 0, n-1);
}
```

data abstraction

- a data type is a collection of objects and a set of operations that act on those objects
  - the data type int consists of the objects {0, +1, -1, +2, -2, …, INT_MAX, INT_MIN} and the operations {+, -, *, /, and %}
- different data types
  - basic data type: char, int, float, double
  - composite data type: array, structure
  - user-defined data type
  - pointer data type

data abstraction

- an abstract data type (ADT) is a data type that is organized in such a way that the specification of the objects and their operations is separated from the implementation of the objects and operations
- specification of operations consists of
  - function name
  - types of arguments
  - types of its results
  - description of what the function does

data abstraction: an example

ADT 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 ∈ Natural_Number; TRUE, FALSE ∈ Boolean and +, -, <, 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

performance evaluation

- performance analysis (machine independent, complexity theory)
  - 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 measurement (machine dependent)

space complexity

- fixed space requirements: C

  not depend on the number and size of the program's inputs and outputs

  eg) instruction space, simple variable, fixed-size structure variables, constant

- variable space requirement: $S_p(I)$

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

total space requirement S(P)

$S(P) = C + S_P(I)$

C : fixed space requirements

$S_p(I)$: function of some characteristics of the instance I

Example: a simple arithmetic function

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

- input - three simple variables
- output - a simple value

- variable space requirements $S_{abc}(I) = 0$
- need only fixed space requirements

Example: iterative function for summing a list of numbers

```
float  sum (float list[],  int n) {

      float temp_sum = 0;
      int i;
      for(i = 0; i < n; i++)
          temp_sum += list[i];
      return temp_sum;
}
```

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

- **C** passes arrays **by pointer**
  passing the address of the first element of the array (not copying the array)
  variable space requirements $S_{sum}(n) = 0$

Example: recursive function for summing a list of numbers

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

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

type	name	number of bytes
parameter: array pointer parameter: integer return address	list[] n	4 4 4
total per recursive call		12

- assume that array has n=MAX_SIZE numbers,
- total variable space S_{rsum}(MAX_SIZE) = 12 * MAX_SIZE

time complexity

- time T(P), taken by a program P, is the sum of its compile time and its run (or execution) time
  - compile time is similar to the fixed space component
- We really concerned only about the program's execution time, T_p
  - count the number of operations that the program performs
  - give a machine-independent estimation
- A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics

Example: iterative summing of a list of numbers

statement	steps/ execution	total steps
float sum (float list[], int n) {		
float temp_sum=0;	1	1
int i;	0	0
for(i = 0; i < n; i++)	1	n+1
temp_sum += list[i];	1	n
return temp_sum;	1	1
}		
total		2n+3

Example: recursive summing of a list of numbers

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

Example: matrix addition

statement	s/e	total steps
`void add(int a[][M_SIZE], ...) {`		
`int i, j;`	0	0
`for(i = 0; i < rows; i++)`	1	rows+1
`for(j = 0; j < cols; j++)`	1	rows*(cols+1)
`c[i][j] = a[i][j] + b[i][j];`	1	rows*cols
`}`		
total		2rows*cols+2rows+1

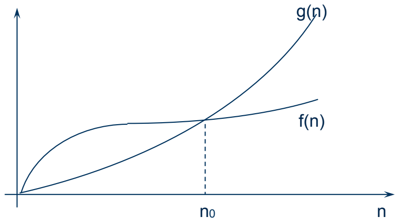
Asymptotic notation: big-O notation

Definition [big-O]

f(n) = O(g(n)) iff there exist positive constants c and n_0 such that $f(n) \leq c\, g(n)$ for all n, $n \geq n_0$

- g(n) is an upper bound on the value of f(n) for all $n \geq n_0$
- but, doesn't say anything about how good this bound is

  $n = O(n^2)$, $n = O(n^{2.5})$, $n = O(n^3)$, $n = O(2^n)$

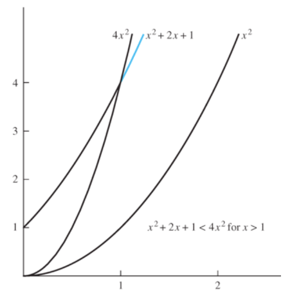


big-O notation

- show that $T(x) = x^2 + 2x + 1$ is $O(x^2)$

  $\boxed{|T(x)| \leq C\,|g(x)| \quad \text{whenever } x > k}$

  - when $x > 1$, $x < x^2$ and $1 < x^2$
  - $x^2 + 2x + 1 < x^2 + 2x^2 + x^2 = 4x^2$
  - $|T(x)| \leq 4\,|x^2|$ whenever $x > 1$
  - $T(x)$ is $O(x^2)$ when $C = 4$, $k = 1$



big-O notation

- show that $T(x) = x^2 + 2x + 1$ is $O(x^3)$

  $|T(x)| \le C\,|g(x)|$ whenever $x > k$

  - when $x > 1$, $x^2 < x^3$, $x < x^3$, and $1 < x^3$
  - $x^2 + 2x + 1 < x^3 + 2x^3 + x^3 = 4x^3$
  - $|T(x)| \le 4\,|x^3|$ whenever $x > 1$
  - $T(x)$ is $O(x^3)$ when $C = 4$, $k = 1$

Ω notation

Definition [Omega]

$f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \ge c \cdot g(n)$ for all n, $n \ge n_0$

- $g(n)$ is a lower bound on the value of $f(n)$ for all n, $n \ge n_0$
- if $f(n) = a_m n^m + \ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$

Θ notation

Definition [Theta]

$f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that $c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for all n, $n \ge n_0$

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

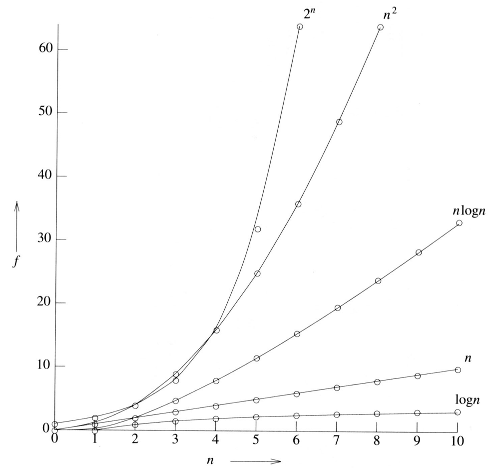
Θ notation

statement	total steps
void add(int a[][M_SIZE], ...) {	
int i, j;	0
for(i = 0; i < rows; i++)	Θ(rows)
for(j = 0; j < cols; j++)	Θ(rows*cols)
c[i][j] = a[i][j] + b[i][j];	Θ(rows*cols)
}	
total	Θ (rows*cols)

asymptotic notation

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
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

time complexity of algorithms



asymptotic notation

If a program needs 2^n steps for execution

- n=40 --- number of steps = $1.1*10^{12}$
    - in computer systems 1 billion (10^9) steps/sec --- 18.3 min
- n=50 --- 13 days
- n=60 --- 310.56 years
- n=100 --- $4*10^{13}$ years

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

- n=10 --- 10 sec
- n=100 --- 3171 years