Data Structure & Algorithms

Sunbeam Infotech

Module Pre-requisites

- Java Programming Language
  - Language Fundamentals
  - Methods
  - Class & Object
  - static members
  - Arrays
  - Collections
  - Inner classes

1. thinking
2. algorithms
3. implementation
4. interviews

Module plan

1. Time & Space complexity
2. Linear & Binary search
3. Recursion
4. Basic sorting
5. Linked list
6. Queues
7. Stacks
8. Trees
9. Heap
10. Advanced sorting
11. Hashing
12. Graphs

Problem Solving Techniques

1. Divide & Conquer
2. Greedy approach
3. Dynamic Programming

Evaluations

1. Theory - 40 marks - CCEE
2. Lab - 40 marks
3. Internal - 20 marks

Data Structure

- Data Structure
  - Organizing data in memory
  - Processing the data

  efficiently

- Basic data structures
  - Array
  - Linked List
  - Stack
  - Queue
- Advanced data structures
  - Tree
  - Heap
  - Graph

- Data structures are ADT. ← Object Oriented

Abstract Data Types

Array ADT : random access

1. get ele at index
2. set ele at index

Stack ADT : LIFO

1. push
2. pop
3. peek
4. isEmpty

Queue ADT : FIFO

1. push
2. pop
3. peek
4. isEmpty

Data Structure

- Data Structure
  - Organizing data in memory
  - Processing the data

efficient

exact analysis
* bytes
* time

asymptotic analysis
* unit space
* iterations

- Common data structures
  - Array
  - Linked List
  - Stack
  - Queue
- Advanced data structures
  - Tree
  - Heap
  - Graph

- Asymptotic analysis
  - It is not exact analysis

Big O

→ Order of

1 → const k

n → $S \propto n$ O(n)

n^2 → $S \propto n^2$ O(n^2)

- Space complexity
  - Unit space to store the data (Input space) and additional space to process the data (Auxiliary space).
  - O(1), O(n), O(n^2)
- Time complexity
  - Unit time required to complete any algorithm.
  - Approximate measure of time required to complete any algorithm. → based on some factor.

Time complexity

- Time complexity of a algorithm depends on number of iterations in the algorithm.

- Common time complexities
  - O(1) → T = k
  - O(n) → T ∝ n → single loop
  - O(n²) → T ∝ n² → nested loops - 2
  - O(n³) → T ∝ n³ → nested loops - 3
  - O(log n) → T ∝ log n → partitioning
  - O(n log n) → T ∝ n log n → partitioning repeated

- Asymptotic notations
  - Big O – O(…) – Upper bound → max time - worst case
  - Omega – Ω(…) – Lower bound → min time - best case
  - Theta ϴ – ϴ(…) – Upper & Lower bound → avg time - avg case

1 check if num is prime.

→ O(n) :

```
for(i=2 ; i<n; i++){
    if(n%i ==0)
        pf("not prime");
}
if(i==n)
    pf("prime");
```

2 print table of "n".

→ O(1) :

```
for(i=1; i<=10; i++)
    pf(num*i);
```

Linear Search

- Find a number in a list of given numbers (random order).

88	33	66	99	~~11~~ 44	77	22	55	11

```
for(i=0; i<n; i++) {
    if(a[i]==key)
        return i;     → return index if found
}
return -1;   → if not found.
```

- Time complexity
  - Worst case → max iterations → $T \propto n$ → $O(n)$
    n itr
  - Best case → min iterations → $T=k$ → $\Omega(1)$ / $O(1)$
    only 1 itr.
  - Average case → avg iterations → $T \propto \frac{n}{2}$ → $\theta(n)$ / $O(n)$
    $T \propto n$

Space Complexity

Input space: $T \propto n$
$O(n)$

Aux space: $S = k$
$O(1)$

Binary Search

40

* for sorted array only.

```
while(l <= r)
{
    m = (l+r)/2
    if(key == a[m])
        return m;
    if(key < a[m])
        r = m-1;        ← left part
    else
        l = m+1;        ← right part
}
return -1;
```

0	1	2	3	4	5	6	7	8
11	22	33	44	55	66	77	88	99
		R	L, m					

$\boxed{2^i \simeq n} \rightarrow 2^{4} = 16$

$16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

$i \log 2 = \log n$

$i = \frac{\log n}{\log 2}$

$T \propto i$

$T \propto \frac{\log n}{\log 2}$

$\boxed{T \propto \log n}$

$\rightarrow O(\log n)$

Recursion – Divide & Conquer

- Function calling itself is called as recursive function.
- To write recursive function consider
  - Explain process/formula in terms of itself
  - Decide the end/terminating condition

    base
- Examples:
  - $n! = n * (n-1)!$ $\qquad$ $0! = 1$
  - $x^y = X * x^{y-1}$ $\qquad$ $x^0 = 1$
  - $T_n = T_{n-1} + T_{n-2}$ $\qquad$ $T_1 = T_2 = 1$
  - factors(n) = prime factor of n * factors(n / prime factor)

    factors(1) = 1
- Function call in recursion
- Pros & Cons of recursion

- On each function call, function activation record or stack frame will be created on stack.

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}


res=fact(4);
```

return addr.

back-substitution

24 $\quad 4! = 4 * 3!$ $\quad$ 6

6 $\quad 3! = 3 * 2!$ $\quad$ 2

2 $\quad 2! = 2 * 1!$ $\quad$ 1

1 $\quad 1! = 1 * 0!$ $\quad$ 1

$0! = 1$

$2^3 = 2 * 2^2$

$2^2 = 2 * 2^1$

$2^1 = 2 * 2^0$

$2^0 = 1$

Recursion

24
res = fact(4)

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

function activation record / stack frame is created on stack for each fn call.

* created when fn called
* destroyed when fn return

FAR contains

1. local variables
2. arguments
3. return address
   ↳ addr of next instru to be executed when fn returns.
   = addr of next instru after fn call.

Recursion

main() {
res = fact(4)
}

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

f(4): n=4, r= ret=m()
m: res, ... ret=jvm/os

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

f(3): n=3, r= ret=f()
f(4): n=4, r= ret=m()
m: res, ... ret=jvm/os

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

f(2): n=2, r= ret=f()
f(3): n=3, r= ret=f()
f(4): n=4, r= ret=m()
m: res, ... ret=jvm/os

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

f(1): n=1, r= ret=f()
f(2): n=2, r= ret=f()
f(3): n=3, r= ret=f()
f(4): n=4, r= ret=m()
m: res, ... ret=jvm/os

```
int fact(int n) {   // n = 0
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

f(0): n=0, r= ret=f()
f(1): n=1, r= ret=f()
f(2): n=2, r= ret=f()
f(3): n=3, r= ret=f()
f(4): n=4, r= ret=m()
m: res, ... ret=jvm/os

Binary Search

0	1	2	3	4	5	6	7	8
11	22	33	44	55	66	77	88	99
L			m-1	m	m+1			R

```
m=(l+r)/2;
if( key == a[m])
    return key
if( key < a[m])
    binSearch(l, m-1, key)
else
    binSearch(m+1, r, key)
```

base cond: right < left

Sorting

- Arranging array elements in ascending or descending order.

- Popular sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Quick sort
  - Merge sort
  - Heap sort

Selection Sort

6 4 2 8 3 1

```
for(i=0; i<5; i++) {
    for(j=i+1; j<6; j++) {
        if(a[i] > a[j])
            swap(a[i], a[j]);
```

0	1	2	3	4	5
1	2	3	4	6	8

3

3

$i = (n-1) + (n-2) + (n-3) + \dots + 1$

$i = n(n-1)/2$

$T \propto \frac{n^2 - n}{2}$

$T \propto n^2 - n$

$T \propto n^2$

$O(n^2)$

Theory of Approximation

$n >> 1$

$\boxed{n^2 >>>>> n}$

all lower order terms can be neglected.

i=0	j → 1 to 5	- 5
i=1	j → 2 to 5	- 4
i=2	j → 3 to 5	- 3
i=3	j → 4 to 5	- 2
i=4	j → 5 to 5	- 1

+ 15

Selection Sort

class Person : id, name, age.

0	1	2	3	4	5
1 B 24	5 X 20	6 P 28	4 G 30	7 N 39	3 Q 22

```
for(i=0; i<5; i++) {
    for(j=i+1; j<6; j++) {
        if(a[i].age > a[j].age)
            Swap(a[i], a[j]);
    }
}
```

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>