Teacher's Sign / Remarks

Date 4/06/18
Page

Introduction to Cryptology – Bimal Roy

- crypt- hidden
- logy- systematic way of studying

- Cryptology – systematic way of studying ways of hiding information

- Cryptology → cryptography (hiding the information)
  Cryptology → Cryptanalysis (analyzing to find the hidden content)

- Cryptology is a 2-person game, and we never underestimate the opponent.

- key – chosen by the user.

- Story of "Dancing Figures" – Sherlock Holmes

- Brute Force Attack – Establishing all possible keys to break the code.

- FI-1, FFI-2 (Encryption Algorithms)

- Assume that the algorithm to be used is not secret key to be chosen by the user at runtime, and not involved in the algorithm.

- If key is a random number, it is difficult to break

- Mainly the security lies in the choice of the key.

- Public Key Cryptography – sender must be sure of the receiver and the receiver must be sure of the sender.
  [key wala suitcase]
  ↳ double keys

- Shannon's Entropy : [$x+y = z$ → known, x and y cannot be calculated separately]

• Privacy Preserving Computation - Adding secret key to amount spend and get a crypted message.

• Bank Locker Problem :-

① Digital Lock

② Divide key into two parts : → first part a number m
→ second part a number c

Compute a straight line : $y = mx + c$.

Take two random points on the line : one to bank one to customer.

③ Consider circle : if bank locker is shared.

key → radius

centre → bank

random two points on the circumference : one to one customer, other to the other customer (both customers share the locker)

• Secret Sharing of a Key. (who has the key)

• No important or secret information should be accessed by a single person. . . . (Fundamental concept of secret sharing).

• Bitcoin - Distributed and Anonymous
↳ Sender and Receiver are Anonymous, leading to concept of Hash values.
↳ then public key encryption is used.

• ECC (Eliptic Curve Cryptography)

Date 4/06/18
Page

Introductory Talks on Sorting and Searching
— Subhas Nandy

- An algorithm should be:
  - → Clever: Efficient data structure
  - → Efficient: Space and Time complexity should be minimum, algorithm should be simple.

- Input → [ALGORITHM] → Output

→ Correctness of algorithm

→ for every input n, there should be same number of steps in the algorithm, everytime it is run.

- We always consider worst case time complexity.

- Average Case Time Complexity: The input comes from a statistical distribution, where a probability is associated with each element in the distribution.

- Amortized Time Complexity is used when the inputs are known.

- Worst Case time complexity bounds the time complexity of an algorithm from above.

- Time Complexity is measured in terms of ~~input~~ a ~~s~~function of input size or output size, or both
  - Input → Input Sensitive
  - Output → Output sensitive

- Sorting: (Defn. from Cormen and RKP paper)

- Internal Sort: Data stored in Computer's main memory. we count the number of arithmatic and logical computations.

- External Sort : Sorting is done on data stored in some external memory. We compute the number of fetches.

- Inplace Sort : Amount of extra space required to sort the data is a small constant, and is not dependent of the input size. No external memory or space is required.

- Stable Sort : It ~~preserves~~ the preserves relative order of ~~rec~~ records with equal keys, key values should come in order of original input.

- Insertion Sort :-

```
(n-1)        1. for j ← 2 to n
(n-1)        2.      do key = A[j]
(n-1)        3.         /*Insert A[j] in the sorted list A[1,2,.....,j-1]*/
Σ(j=2..n) tj 4.         i ← j-1
  ,,         5.         while i > 0 and A[i] > key
  ,,         6.              do A[i+1] ← A[i]
  ,,         7.                 i ← i-1
  ,,         8.         A[i+1] ← key
```

Time Complexity : $O(n^2)$

- Bubble Sort :

```
for i ← ~~to~~ 1 to n
     do j = length (A) downto i+1
          /*fill A[i] by min{A[j], j=i,i+1,...n} */
     do if [~~A~~ A[j] < A[j-1] then exchange A[j], A[j-1] }
```

Updated:

```
FLAG = false, i=1
while FLAG = false do
     FLAG = true
     for j = length (A) down to i+1 do
          /*fill A[i] by min {A[j], j=i, i+1,.....n} */
          if A[j] < A[j-1] then
```

FLAG = false
exchange (A[j], A[j-1])
i = i+1

** Time complexity remains same. [Worst case: Ascending order given → finding descending order]

• Quicksort :

→ Quicksort (A, p, q) //In text

1. If p ≥ q EXIT.
2. Compute s ← correct position of A[p] in the sorted order of the elements of A from p-th location to q-th location.
3. Move the pivot A[p] into position A[s].
4. Move the remaining elements of A[p-q] into appropriate sides.
5. Recursively sort the segments to the left and right of the pivot.

5a. QSORT (A, p, s-1)
5b. QSORT (A, s+1, q)

→ QSORT (A, p, q) // A bit more in detail.

1. If p ≥ q EXIT.

2 and 4. Compute j ← correct position of A[p] in the sorted order of the elements of A from pth location to q-th location.

2a. pivot = A[p]; i = p+1, j = q
2b. while (i < j) do
2c. while A[i] ≤ pivot do i = i+1
2d. while A[j] > pivot do j = j-1
2e. if i < j then SWAP (A[i], A[j])
3. Move the pivot A[p] into ~~the~~ position A[s]
3a. ~~Re~~ SWAP (A[p], A[j])
5. Recursively sort the segments to the left and right of the pivot.
5a. QSORT (A, p, s-1)
5b. QSORT (A, s+1, q)

→ An inplace algorithm.

→ Worst case : $O(n^2)$ Average case: $O(n \log n)$

Scanned by CamScanner

→ Inductive proof of $T(n) = O(n \log n)$

$$T(n) = 1/n \left[T(1) + T(n) + \sum_{q=1}^{n-1} (T(q) + T(n-q)) + O(n) \right]$$

→ Scope of improvement :-

~ In each level of recursion, total number of comparisons $= O(n)$.

~ We need to reduce the number of levels of recursion. ~~to log~~

~ We split each sub-array under consideration into two equal halves using the median of that ~~set~~ subarray as the pivot.

Reason: If there are k levels than $2^k = n$

$$\Rightarrow k = \log_2 n$$

Median can be computed in $O(n)$.

Median computation needs extra workspace, which means that space complexity increases and quicksort does not remain inplace.

• Randomized Quick Sort –

→ Central Splitter

It is an index s such that the number of elements less than A[s] is at least n/4.

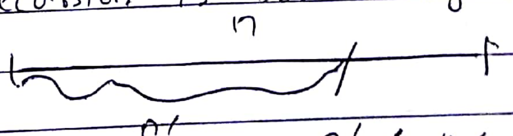
→ The randomized quicksort chooses a random number as pivot and we use concept of probability to analyze the problem.

→ $P[\text{Central Splitter}] = \dfrac{\text{No. of favourable cases}}{\text{Total no. of cases}} = \dfrac{n/2}{n} = \dfrac{1}{2}$

∴ Expected number of trials to get a central splitter = 2

∴ Expected time required to get ~~the~~ a central splitter = 2n.

→ Worst-case size of each partition in j-th level of ~~recurs~~ recursion is bounded by $n \times (3/4)^j$



$n' \leq \frac{3n}{4}$ $\qquad \frac{n'}{4} \leq n'' \leq \frac{3n'}{4} \leq \left(\frac{3}{4}\right)^2 n$

$n'' \longrightarrow$ Partition at next level.

$\therefore n_j \leq (3/4)^j n$.

Hence, we will get $T(n) = 2T\left(\frac{3n}{4}\right) + O(n) = O(n \log n)$

$$T(n) = 2T\left(\frac{3n}{4}\right) + O(n)$$

$$= 4T\left(\left(\frac{3}{4}\right)^2 n\right) + O(n) + O(n)$$

$$\vdots$$

$$= 2^k T\left(\left(\frac{3}{4}\right)^k n\right) + kO(n)$$

$$= 2^{\log_{4/3} n} T(1) + \log_{4/3} n \, O(n)$$

$$= 2^c \log n + c' n \log n$$

$$= c' \log n + c'' n \log n$$

[We use the result :
levels of recursion $= \log_{4/3} n$
$= O(\log n)$

• Merge Sort

→ We divide the list into two halves and sort each half separately

→ We merge the sorted halves into one sorted array

→ We executed k steps, where each step takes linear time.

$$2^k = n/2$$

$$\Rightarrow k = (\log_2 n - 1)$$

→ Hence, total time = $k.n = O(n\log n)$ ⇒ Not inplace sort

• Heap

→ A heap is a complete binary tree with elements from a partially ordered set, such that the element at every node is less than or equal to the elements in the subtree rooted at that node.

→ We represent the heaps in an array

Childs of i : $2i, 2i+1$

Parents of i : $\lfloor i/2 \rfloor$

→ Heaps can be implemented in priority queue due to the parent-child property ; the first priority being at the root.

→ The ~~root is at~~ root is ~~at~~ at location 1.

→ A heap of height k will have elements between 2^k and $2^{k+1} - 1$.

→ Heap with height n elements has height $\lfloor \log_2 n \rfloor$

→ The minimum of the ~~root~~ heap is at the root. findmin() operation will have worst-case $O(1)$

→

→ Heapify (A, i)

1. $l \leftarrow left(i) = 2i$; $r \leftarrow right(i) = 2i+1$
2. if $l \leq n$ and $A[l] > A[i]$ then $largest \leftarrow l$
   else $largest \leftarrow r$
3. if $r \leq n$ and $A[r] > A[largest]$ then largest
4. if $largest \neq i$ then SWAP $(A[i], A[largest])$
   Heapify (A, largest)

→ Build Heap(A)

1. for $i = \lfloor n/2 \rfloor$ downto 1
2. Heapify (A, i)

→

→ Heapify (A, i)

1. $l \leftarrow left(i) = 2i$; $r \leftarrow right(i) = 2i+1$
2. if $l \leq n$ and $A[l] > A[i]$ then largest $\leftarrow l$
   else largest $\leftarrow r$
3. if $r \leq n$ and $A[r] > A[largest]$ then largest $\leftarrow$
4. if largest $\neq i$ then SWAP $(A[i], A[largest])$;
   Heapify (A, largest)

→ Build Heap(A)

1. for $i = \lfloor n/2 \rfloor$ downto 1
2. Heapify (A, i)