Sunbeam Institute of Information Technology

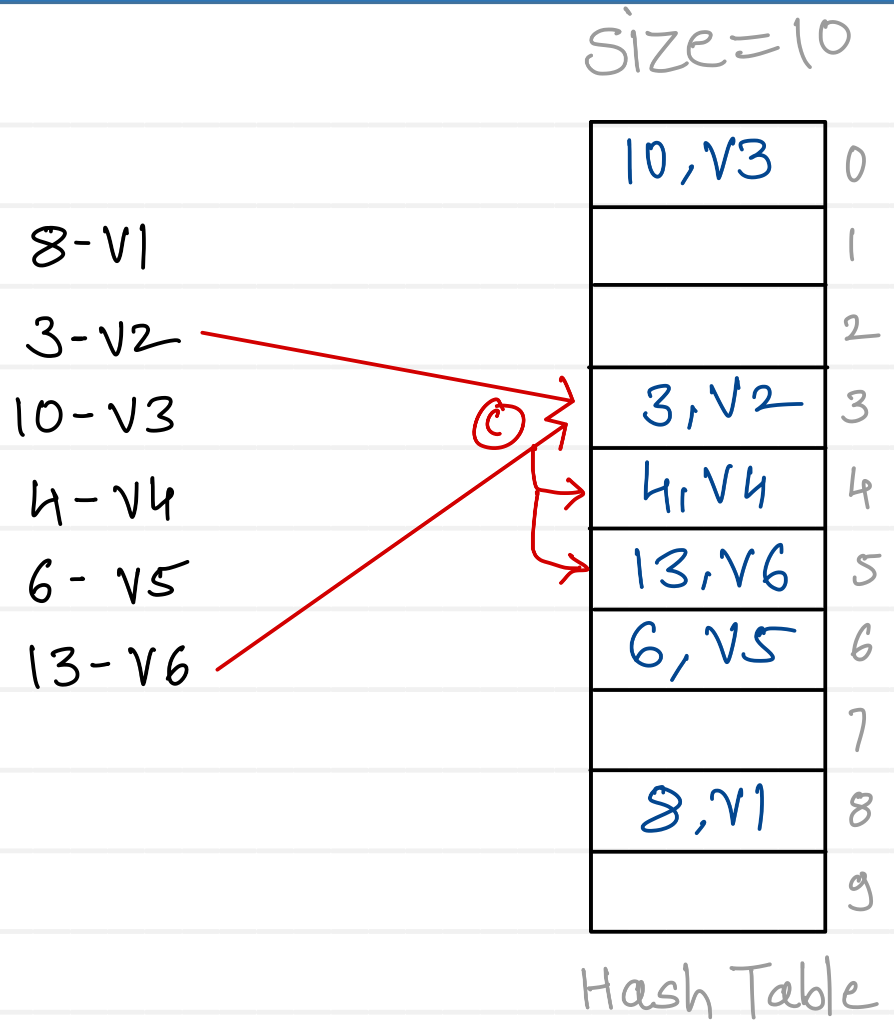
Pune and Karad

Module – Data Structures and Algorithms

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Open addressing - Linear probing



h(k) = k % size

h(k, i) = [h(k) + f(i)] % size

f(i) = i

where i = 1, 2, 3, …

h(13) = 13 % 10 = 3 c

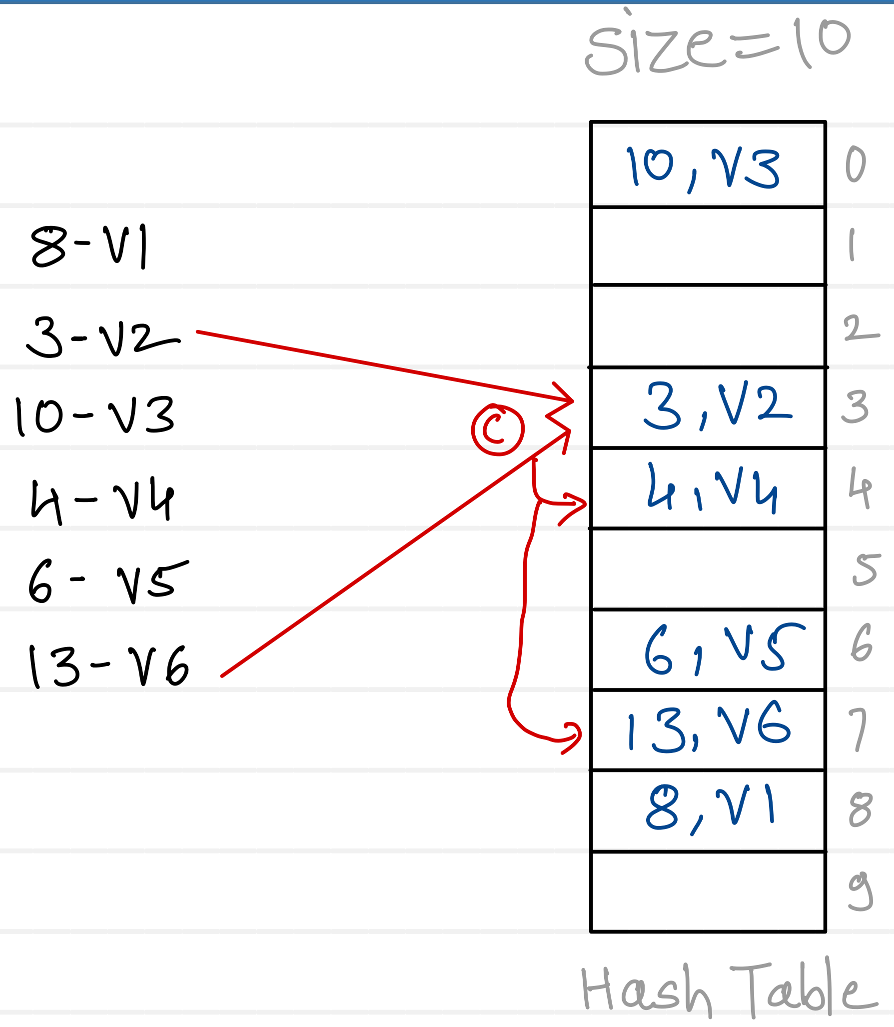
h(13,1) = [3 + 1] % 10 = 4 (1st) c

h(13,2) = [3 + 2] % 10 = 5 (2nd)

Probing : finding next empty slot to store key value pair whenever collision will occur

Primary clustering : to find next empty, need to take long run of filled slots, "near" key position.

Open addressing - Quadratic probing



h(k) = k % size

h(k, i) = [h(k) + f(i)] % size

f(i) = i^2

where i = 1, 2, 3, …

h(13) = 13 % 10 = 3 c

h(13,1) = [3 + 1] % 10 = 4 (1st) c

h(13,2) = [3 + 4] % 10 = 7 (2nd)

- in this technique, there is no guarantee of getting free slot for the key.

Open addressing - Quadratic probing

size=10

8-V1
3-V2
10-V3
4-V4
6-V5
13-V6
23, V7

10, V3	0
	1
23, V7	2
3, V2	3
4, V4	4
	5
6, V5	6
13, V6	7
8, V1	8
	9

Hash Table

$$h(k) = k \ \% \ size$$

$$h(k, i) = [\ h(k) + f(i)\ ] \ \% \ size$$

$$f(i) = i\text{^}2$$

where i = 1, 2, 3, …

$h(23) = 23 \% 10 = 3$ C

$h(23,1) = [3+1] \% 10 = 4$ (1^{st}) C

$h(23,2) = [3+4] \% 10 = 7$ (2^{nd}) C

$h(23,3) = [3+9] \% 10 = 2$ (3^{rd})

Secondary clustering : to find next empty slot, need to take long run of filled slots "away" key position.

Open addressing - Double hashing

size = 11

8 - V1
3 - V2
10 - V3
25 - V4

0	
1	
2	
3	3, V2
4	
5	
6	25, V4
7	
8	8, V1
9	
10	10, V3

Hash Table

- primary & secondary clustering is removed because for two distinct key, different path is followed to find free slot.

h1(k) = k % size

h2(k) = 7 - (k % 7)

h(k, i) = [h1(k) + i * h2(k)] % size

$h1(25) = 25 \% 11 = 3$ c

$h2(25) = 7 - (25 \% 7) = 3$

$h(25,1) = [3 + 1 * 3] \% 11$
$= 6$ (1^{st})

$h_1(36) = 3$

$h_2(36) = 6$

$h(36,1) = [3 + 1 * 6] \% 11 = 9$

Rehashing

$$\text{Load factor } (\lambda) = \frac{n}{N}$$

e.g. size = 10

n = 6 , N = 10

$$\lambda = \frac{n}{N} = \frac{6}{10} = 0.6$$

λ = 0.6 means, hash table is 60 % filled.

n - number of elements (key-value) present in hash table
N - number of total slots in hash table

- Load factor ranges from 0 to 1.
- If n < N Load factor < 1 - free slots are available
- If n = N Load factor = 1 - free slots are not available

- In rehashing, whenever hash table will be filled more than 60 or 70 % size of hash table is increased by twice
- Existing key value pairs are remapped according to new size

Algorithm Design Techniques

1) Divide & Conquor - merge & quick sort
2) Greedy - optimal solution
   Prims, Dijkstras, Kruskals

3) Dynamic programming
   Bellaman Ford, Floyd Warshal

Memoization	Tabulation
dp array → 1D or 2D array + recursion	1D or 2D array + loops
(Top-down approach)	(Bottom-up apprach)

Problem solving technique : Greedy approach

- A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.
- We can make choice that seems best at the moment and then solve the sub-problems that arise later.
- The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the sub-problem.
- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one.
- A greedy algorithm never reconsiders its choices.
- A greedy strategy may not always produce an optimal solution.

e.g. Greedy algorithm decides minimum number of coins to give while making change.

coins available : 50, 20, 10, 5, 2, 1

Recursion

- Function calling itself is called as recursive function.
- For each function call stack frame is created on the stack.
- Thus it needs more space as well as more time for execution.
- However recursive functions are easy to program.
- Typical divide and conquer problems are solved using recursion.
- For recursive functions two things are must
    - Recursive call (Explain process it terms of itself)
    - Terminating or base condition (Where to stop)

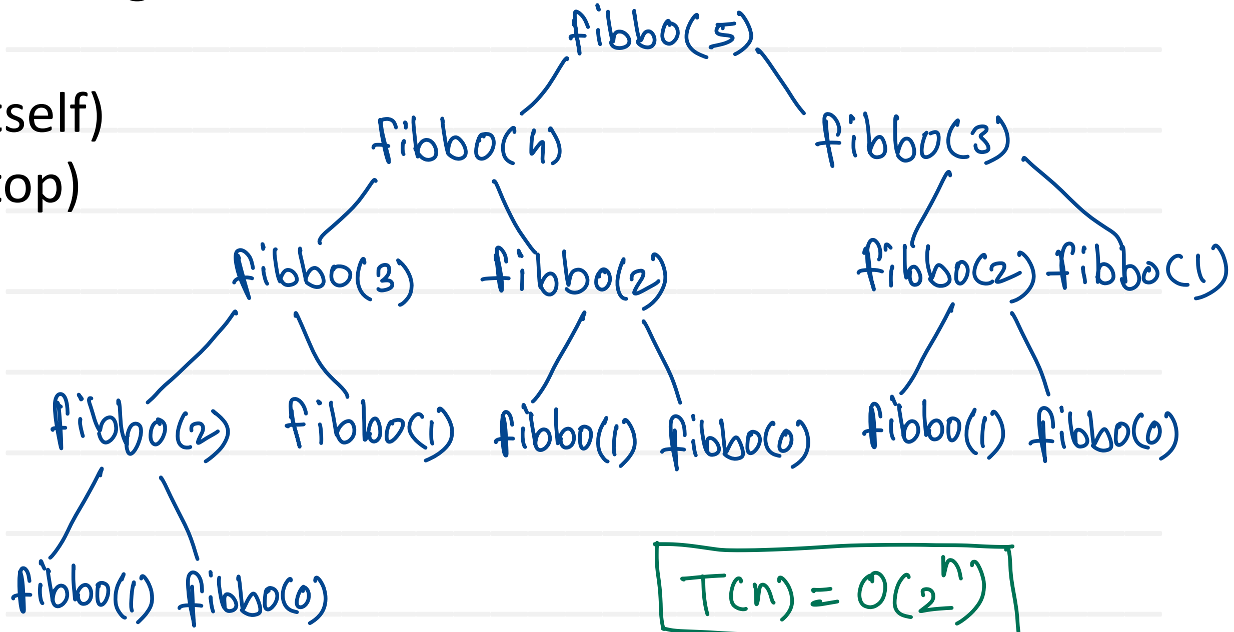
e.g. Fibonacci Series

- Recursive formula

  Tn = Tn-1 + Tn-2
- Terminating condition

  T1 = T2 = 1
- Overlapping sub-problem

```
int fibbo(int n) {
    if(n==0 || n==1)
        return n;
    return fibbo(n-1) + fibbo(n-2);
}
```



$$\boxed{T(n) = O(2^n)}$$

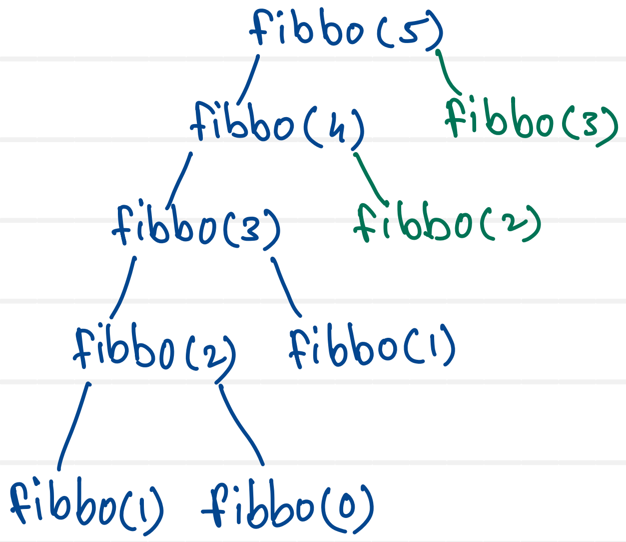
Memoization

- It's based on the Latin word memorandum, meaning "to be remembered".
- Memoization is a technique used in computing to speed up programs.
- This is accomplished by memorizing the calculation results of processed input such as the results of function calls.
- If the same input or a function call with the same parameters is used, the previously stored results can be used again and unnecessary calculation are avoided.
- Need to rewrite recursive algorithm. Using simple arrays or map/dictionary.

dp

0	1	2	3	4	5
0 ~~-1~~	1 ~~-1~~	1 ~~-1~~	2 ~~-1~~	3 ~~-1~~	5 ~~-1~~

```
int fibbo(int n){
    if(n==0 || n==1)
        return n;
    if( dp[n] != -1)
        return dp[n];
    dp[n] = fibbo(n-1) + fibbo(n-2);
    return dp[n];
}
```



Dynamic Programming

- Dynamic programming is another optimization over recursion.
- Typical DP problem give choices (to select from) and ask for optimal result (maximum or minimum).
- Technically it can be used for the problems having two properties
  - Overlapping sub-problems
  - Optimal sub-structure
- To solve problem, we need to solve its sub-problems multiple times.
- Optimal solution of problem can be obtained using optimal solutions of its sub-problems.

- Greedy algorithms pick optimal solution to local problem and never reconsider the choice done.
- DP algorithms solve the sub-problem in a iteration and improves upon it in subsequent iterations.

Dynamic programming

dp

0	1	2	3	4	5
0	1	1	2	3	5

```
int fibbo(int n) {
    dp[0] = 0;
    dp[1] = 1;
    for (int i=2; i<=n; i++)
        dp[i] = dp[i-1] + dp[i-2];

    return dp[n];
}
```

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com