Greedy Algorithms

Greedy algorithms

- Greedy Algorithms •
- Shortest Path in Graph - Dijkstra’s algorithm •
- Minimum Spanning Trees
- (Already covered under graph theory but put here again for sake of completeness)

- Travelling Salesman Problem
- P vs NP

Greedy Algorithms

- Shortest Path in Graph - Dijkstra’s algorithm •
- Minimum Spanning Trees
- Already covered under graph theory.
- Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.
- Greedy algorithms build solutions to problems by adding the best or cheapest solution left at each step.
- For example, to decide which notes/coins are needed for change, we find the largest note/coin that is smaller than the change required and add that to the change:

Greedy Algorithms

- A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.
- The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.
- This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

Dijkstra's Algorithm

Input: weighted-directed graph `G(V,E)` and `source` vertex

Create a table `T` of "best distance" to each destination vertex

- values of `T`: `T[node] = <distance, lastStep>`
- Initialize table elements of `T` to `<`∞`, undefined>`

Create a set `Q` of vertices

Loop over all elements of set `Q`

- find cheapest vertex `u` (vertex you can get to in smallest distance)
- remove `u` from `Q`
- look at neighbors of `u` - is the path through `u` cheaper than previously known?
- if so, update their values in `T`

Calculate final paths using `lastStep`

Dijkstra’s Algorithm

```
Create table T[V] = <∞,undefined>
T[source] = <0, undefined>

Create set Q = set(V)

while Q is not empty {
        u = min q in Q of T[q].distance
        S = neighbors(u)
        Q = Q - u

        for each neighbor v of u {
                d = T[u].distance + E[u,v]
                if (d < T[v].distance) {
                        // shorter path!
                        T[v].distance = d
                        T[v].lastStep = u
                }
        }
        // done with u
}

return T
```

Dijkstra’s Algorithm

Dijkstra’s is a Greedy Algorithm

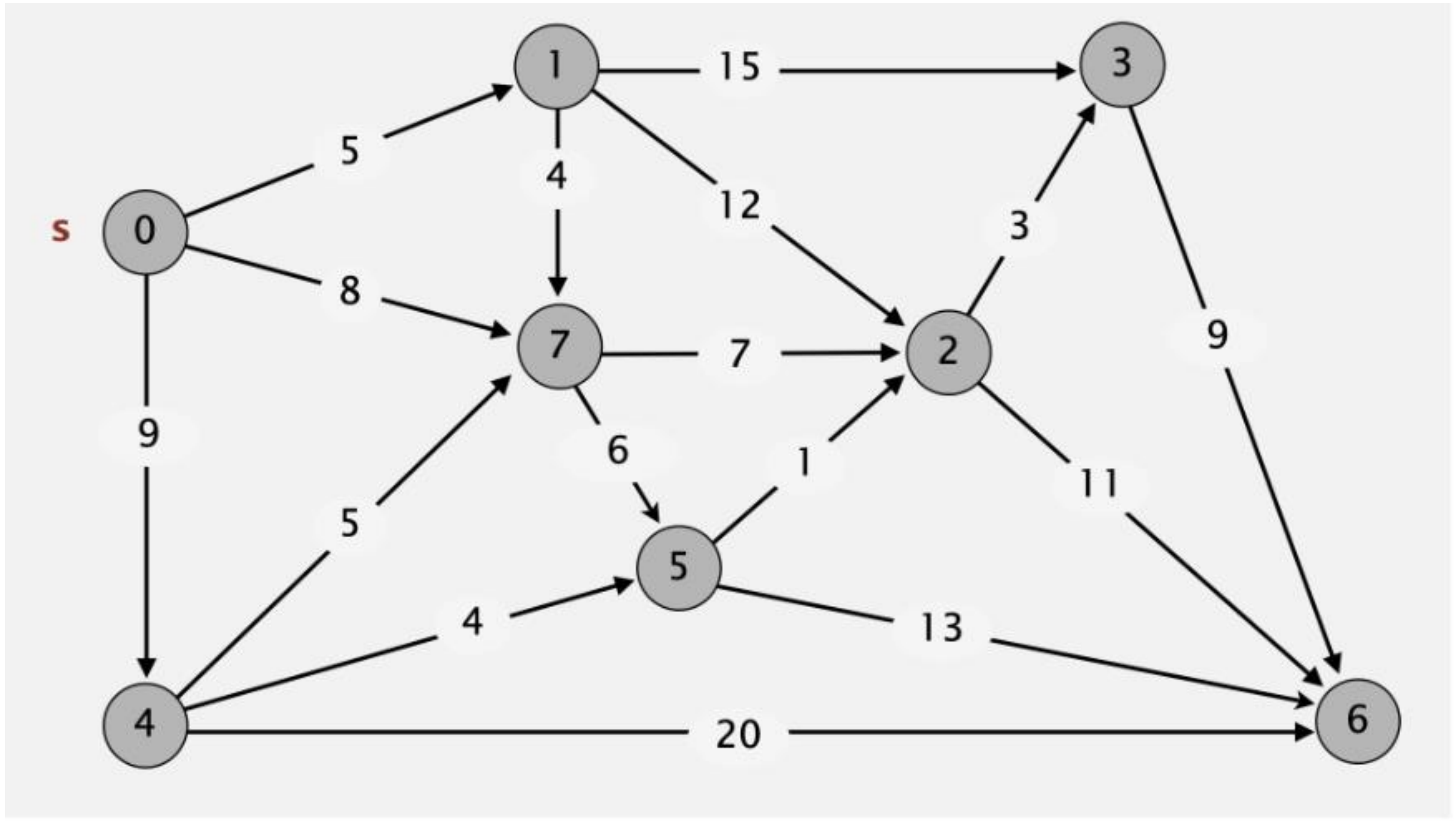
At each step, a greedy algorithm chooses the locally best choice – in this case the closest/cheapest next vertex

- In some case, a greedy approach can give you the right/best answer
- In other cases, it will not be guaranteed to give you the best one – but the greedy answer might well be close (good enough)

Some greedy algorithms have a relaxation step – where greedy choices that have been already made can be cleaned up when better information comes along

Example

http://web.archive.org/web/20170709061020/https://algs4.cs.princeton.edu/lectures/44DemoDijkstra.pdf



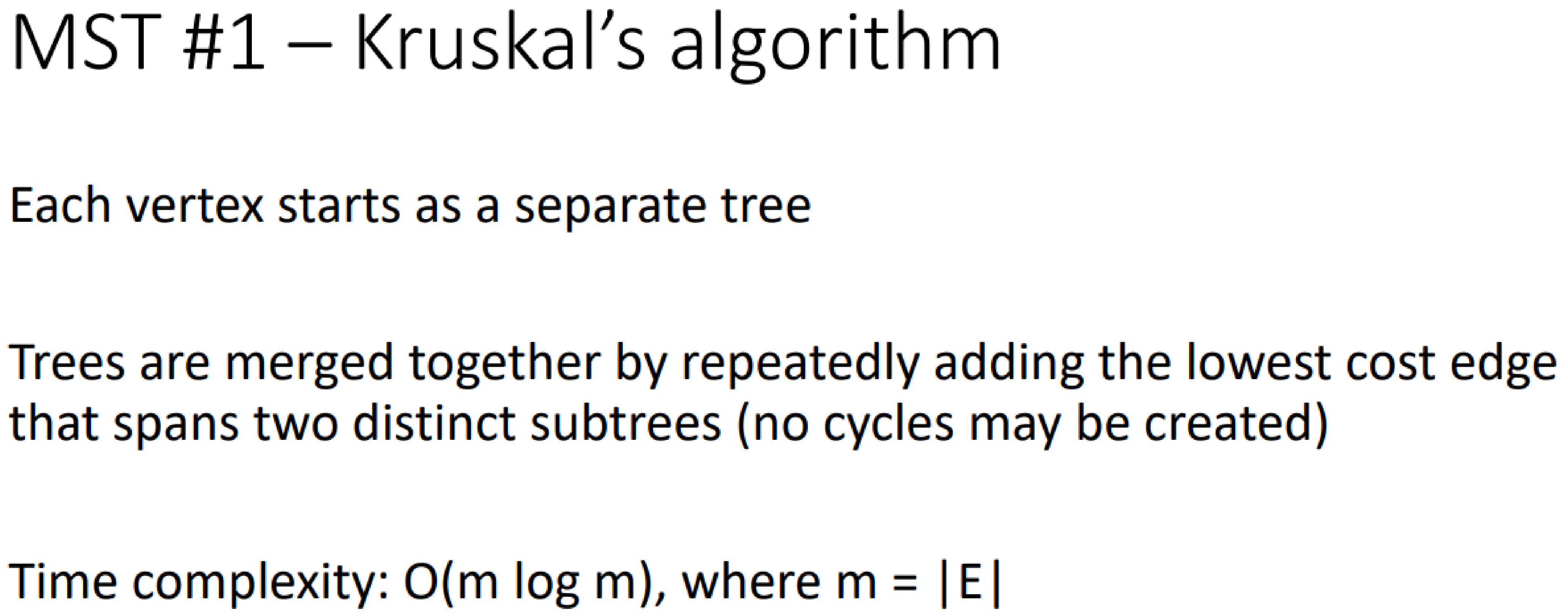
Additional videos to watch

- YouTube lecture on Dijkstra's algorithm:
  https://www.youtube.com/watch?v=_lHSawdgXpI
- Interactive demo of Dijkstra's algorithm:
  https://visualgo.net/en/sssp?slide=1

Minimum Spanning Tree

- The minimum spanning tree (MST) of a graph defines the cheapest subset of edges that keeps the graph in one connected component.
- We covered Prim’s and Kruskal’s algorithm under greedy algorithm

Kruskal’s algorithm



MST #1 – Kruskal’s algorithm

Each vertex starts as a separate tree

Trees are merged together by repeatedly adding the lowest cost edge that spans two distinct subtrees (no cycles may be created)

Time complexity: O(m log m), where m = |E|

Prim’s algorithm

MST #2 – Prim’s algorithm

Start with an arbitrary vertex v

Grow a tree from it by repeatedly finding the lowest-cost edge the links some new vertex into the tree

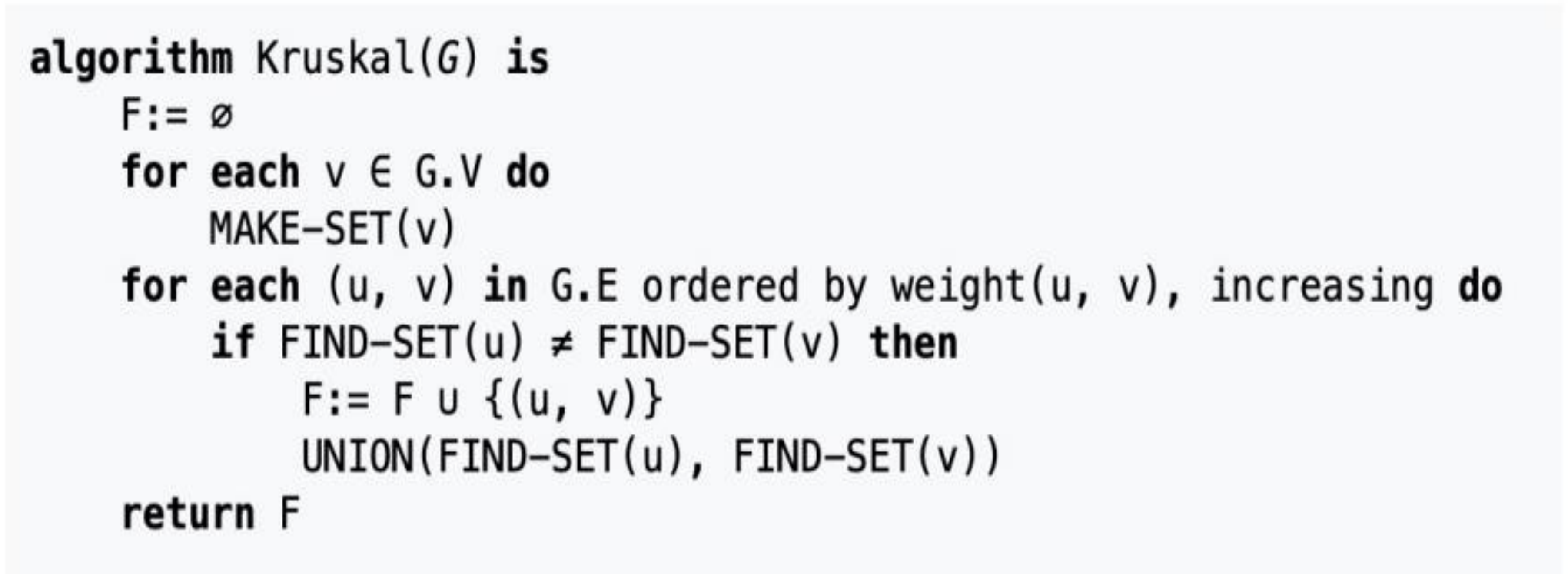
Time complexity: $O(n^2)$, where $n = |V|$

Kruskal’s Algorithm

1. create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
    a. remove an edge with minimum weight from S
    b. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree

Kruskal’s algorithm

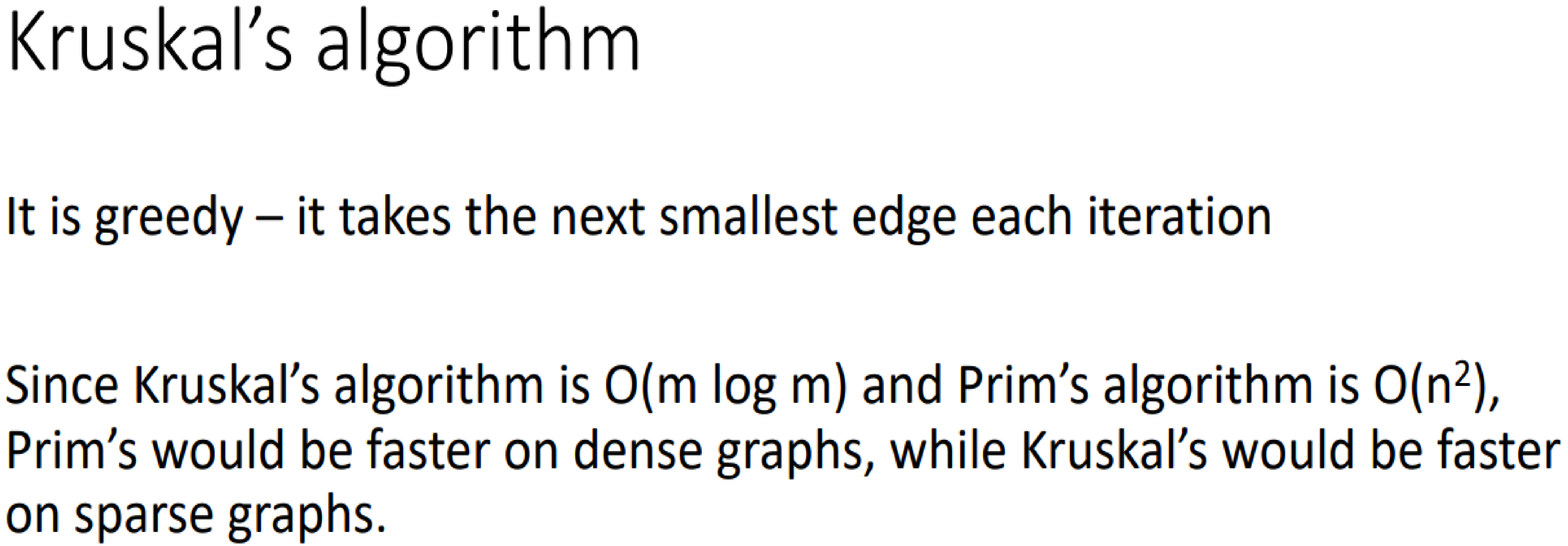


```
algorithm Kruskal(G) is
    F:= ∅
    for each v ∈ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v), increasing do
        if FIND-SET(u) ≠ FIND-SET(v) then
            F:= F ∪ {(u, v)}
            UNION(FIND-SET(u), FIND-SET(v))
    return F
```

Animation of Kruskal’s algorithm

- From https://www.cs.usfca.edu/~galles/visualization/Kruskal.html

Kruskal’s algorithm



Kruskal’s algorithm

It is greedy – it takes the next smallest edge each iteration

Since Kruskal’s algorithm is O(m log m) and Prim’s algorithm is $O(n^2)$, Prim’s would be faster on dense graphs, while Kruskal’s would be faster on sparse graphs.

Kruskal’s algorithm

It is greedy – it takes the next smallest edge each iteration

Since Kruskal’s algorithm is O(m log m) and Prim’s algorithm is O(n^2), Prim’s would be faster on dense graphs, while Kruskal’s would be faster on sparse graphs.

There are advanced data structures (called paring heaps) that makes Prim’s algorithm have time complexity O(m + n log n) and it then becomes the fastest practical choice for both sparse and dense graphs.

Travelling Salesman Problem

- Travelling Salesman Problem (TSP):
- Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.
- Example of NP complete Problem
- All scheduling problems

NP Complete Problems

- In the field of complexity, any algorithm that can be solved deterministically and is bounded by polynomial time is considered efficient or solvable.
- There are a set of problems for which no deterministic polynomial (P) solution has been found, but for which it takes polynomial time to prove that a specific set of outputs is a solution.
- This set of problems is known as the NP-complete set of problems
- NP hard problems

Dynamic Programming

- Dynamic programming involves storing solutions to be used later
- For example, consider the Fibonacci sequence of numbers, where:
  - f0 = f1 = 1
  - f2 = f0 + f1 = 2
  - f3 = f1 + f2 = 3

- The above problem has been solved using recursion

Fibonacci Sequence

```
• int FibonacciTest (int num)
• {
•         Base Case
•         if (num == 0 || num == 1)
•         {
•                 return 1;
•         }
•         else // recurring case
•         {
•                 return FibonacciTest(n-2) + FibonacciTest(n-1);
•         }
• }
```

Dynamic Programming

- The complexity of that algorithm is O(1.6n).
- Using dynamic programming, we can improve on the above. Given the declaration using a map as defined below:
- map<int, int> fib1;
- Use the following initialization
- map[0] = 1;
- map[1] = 1;
- Complexity can be improved to O(n)