Course Notes for

CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- Situation:
  - You have discovered a new computational problem during your CS research
  - What should you do?
    - Try to find an efficient solution (polynomial) for the problem
    - If unsuccessful (or if you think it likely that it is not possible), try to prove the problem is NP-complete
      - This way, you can at least show that it is likely that no polynomial solution to the problem exists
      - You may also try to develop some heuristics to give an approximate solution to the problem

Proving NP-Completeness

- How to prove NP-completeness?

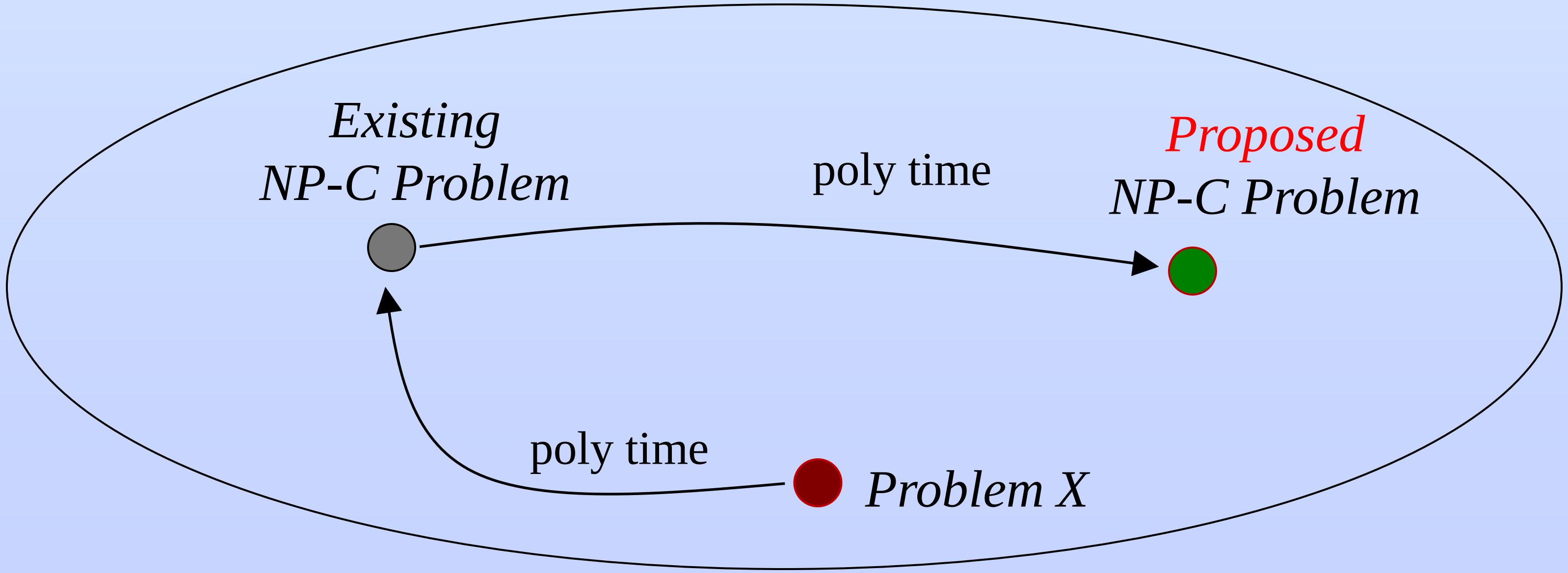
1) From scratch
   - Show the problem is in NP
   - Show that all problems in NP can be transformed to this problem in polynomial time
   - Very complicated – takes a lot of work to do
   - Luckily, we only need to do this once, thanks to method 2) below

2) Through reduction
   - Show the problem is in NP
   - Show that some problem already known to be NP-complete is reducible (transformable) to the new problem in polynomial time
   - Idea is that, since one prob. can be transformed to the other in poly time, their solution times differ by at most a polynomial amount

Proving NP-Completeness

- We know all problems in NP can be reduced to the existing NP-C problem in poly time
- We show that existing NP-C problem can be reduced to new prob. in poly time
- Thus all problems in NP can be reduced to our new problem in poly time

- There are many already known NP-C problems that can be used for the reduction

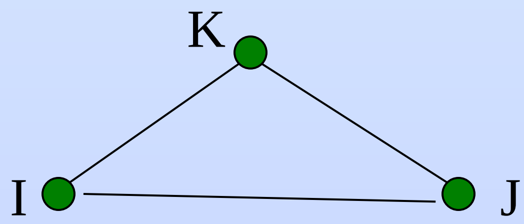


- Ok, so a problem is NP-complete
  - But we may still want to solve it
  - If solving it exactly takes too much time using a deterministic algorithm, perhaps we can come up with an approximate solution
    - Ex: Optimizing version of TSP wants the minimum tour of the graph
      - Would we be satisfied with a tour that is pretty short but not necessarily minimum?
    - Ex: Course scheduling
    - Ex: Graph coloring
      - Can one "color" a graph with K colors? NP-Complete for any K > 2

- We can use heuristics for this purpose
  - Goal: Program runs in a reasonable amount of time, yet gives an answer that is close to the optimal answer
- How to measure quality?
  - Let's look at TSP as an example
    - Let H(C) be the total length of the minimal tour of C using heuristic H
    - Let OPT(C) be the total length of the optimal tour
    - Ratio bound:
      - H(C)/OPT(C) gives us the effectiveness of H
      - How much worse is H than the optimal solution?

- For original TSP optimization problem, it can be proven that no constant ratio bound exists for any polynomial time heuristic
  - (assuming P != NP)
- But, for many practical applications, we can restrict TSP as follows:
  - For each distance in the graph:

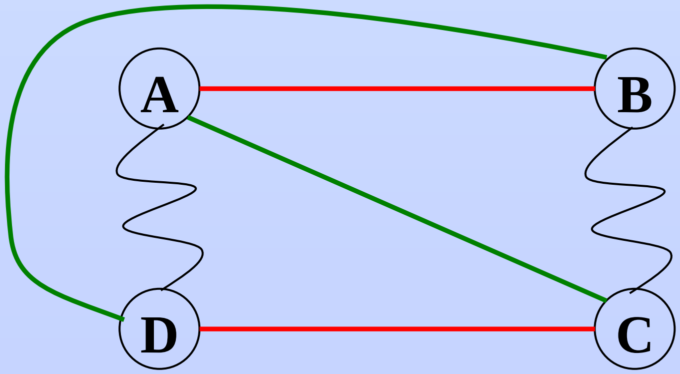
$d(c_I,c_J) <= d(c_I,c_K) + d(c_K,c_J)$



    - TRIANGLE INEQUALITY
      > A direct path between two vertices is always the shortest
  - Given this restriction, heuristics have been found that give ratio bounds of 1.5 in the worst case

- How do heuristics approach a problem?
  - Many different approaches, but we will look only at one: Local Search
    - Idea: Instead of optimizing the entire problem, optimize locally using a neighborhood of a previous sub-optimal solution
    - We are getting a local optimum rather than a global optimum
  - Let's look at an example local search heuristic for TSP (assuming triangle inequality)
    - We call this heuristic 2-OPT

- 2-OPT Idea:
  - Assume we already have a valid TSP tour
  - We define a neighborhood of the current tour to be **all possible tours** derived from the current tour by the following change:
    - Given (non-adjacent) edges (a,b) and (c,d) in the current tour, replace them with edges (a,c) and (b,d)



- Note that we still have a valid tour
- Given all neighbor tours of the current tour, we choose the one which reduces the overall tour length by the greatest amount
- This is 1 iteration of 2-OPT
- We repeat until current tour is best

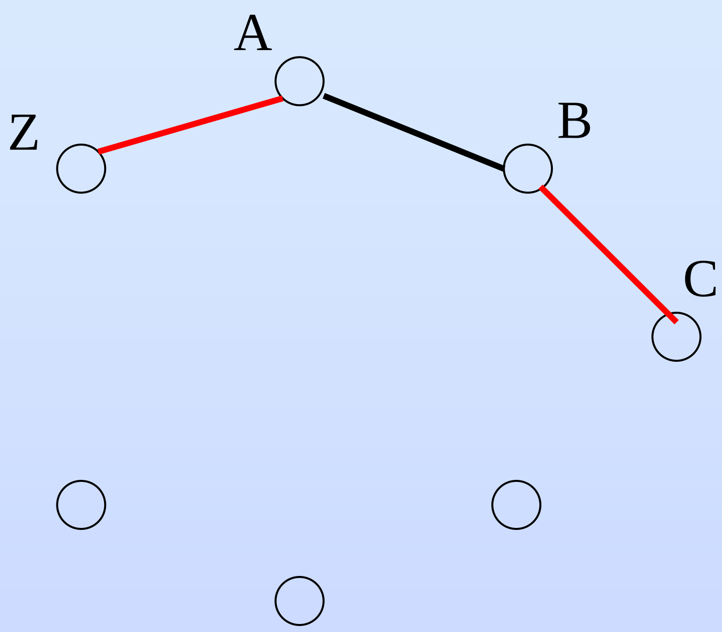
- Implementation: can be done in a fairly straightforward way using an adjacency matrix
  - Pseudocode:

```
bestlen = length of initial tour
while not done do
   improve = 0;
   for each two-opt neighbor of current tour
       diff = (C(A,B) + C(C,D)) - (C(A,C) + C(B,D))
       if (diff > improve)
              improve = diff
              store current considered neighbor
   if (improve > 0)
       update new tour
       bestlen -= improve
   else
       done = true
```

  - Look at the code if you wish – twoopt.c

- Performance of 2-opt:
  - Note nested loops in pseudocode
  - How many overall iterations will we have to perform before local optimum is found (i.e. how many iterations will we have in the outer while loop)?
  - How many neighbor tours are in a neighborhood (i.e. how many iterations will the inner foreach loop have)?
- Total run-time of 2-opt is the product of these two times

2-OPT Performance



2-opt number of neighbors of a given tour

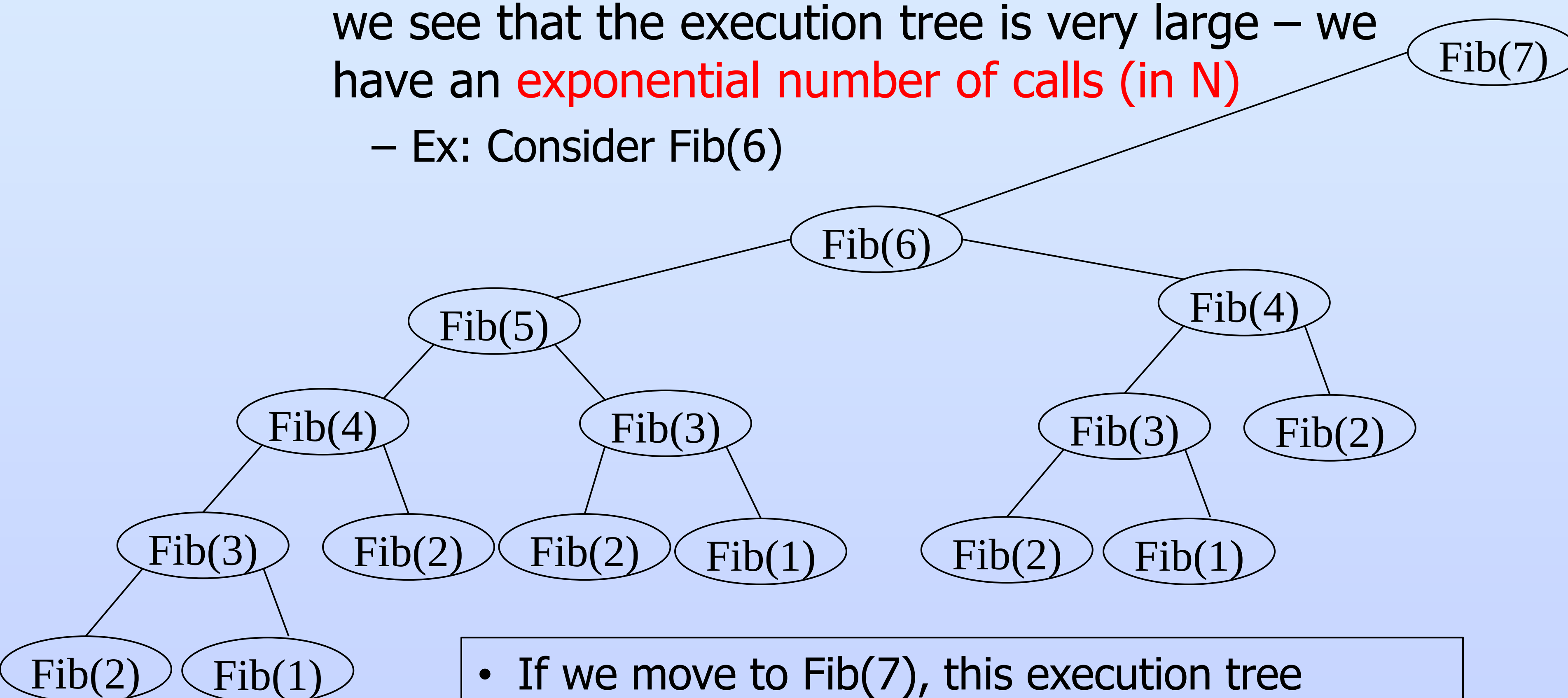
Theta(V^2)

- Consider edge (A,B)
- How many neighbor tours include this edge?
  - Edges cannot be adjacent
  - Thus second edge to make neighbor cannot be:
    - (Z,A), (A,B), (B,C)
- We have V edges in the tour
  - Each edge can be in (V-3) neighbor tours
  - However, we count each neighbor twice (ex: (A,B),(C,D) and (C,D), (A,B))
- Thus total we have (V)(V-3)/2 possible neighbors

- How many overall iterations are necessary (of outer while loop)?
  - In the worst case this can be extremely high (exponential)
  - But this is a rare (and contrived) case
  - In practice, few iterations of the outer loop are generally required
- What is the quality of the final tour?
  - Again, in the worst case it is not good
  - In normal usage, however, it does very well (a ratio bound of 1.05 on average, given a good initial tour)
- Variations on local search can improve the situation so that the worst case problems of 2-OPT go away
  - More sophisticated algorithms such as simulated annealing and genetic algorithms

- With recursive algorithms
  - We break a large problem into subproblems and use the subproblem solutions to solve the original problem
  - However, in some situations this approach is not an efficient one
    - Inefficiency occurs when subproblems must be recalculated many times over and over
  - Famous example is the Fibonacci Sequence:
  - Recursively we see:
    - FIB(N) = FIB(N-1) + FIB(N-2) for N > 2
    - FIB(2) = FIB(1) = 1

- However, if we trace the execution of this problem, we see that the execution tree is very large – we have an exponential number of calls (in N)
  - Ex: Consider Fib(6)



- If we move to Fib(7), this execution tree becomes the left side of the new execution tree

- Overall run-time is ϕ^N
  - Where ϕ is the golden ratio
    - See https://en.wikipedia.org/wiki/Golden_ratio
- This is very poor
  - It seems like we should be able to do much better
- If we approach this problem from the "bottom up", we can improve the solution efficiency
  - Start by solving FIB(1), then FIB(2), and build the solution iteratively until we get to desired value of N
    - Each new term is sum of previous two
  - Now we are calculating each smaller solution only one time
- See fibo.java

- Overall run-time is ϕ^N
  - Where ϕ is the golden ratio
    - See https://en.wikipedia.org/wiki/Golden_ratio
- This is very poor
  - It seems like we should be able to do much better
- If we approach this problem from the "bottom up", we can improve the solution efficiency
  - Start by solving FIB(1), then FIB(2), and build the solution iteratively until we get to desired value of N
    - Each new term is sum of previous two
  - Now we are calculating each smaller solution only one time
- See fibo.java

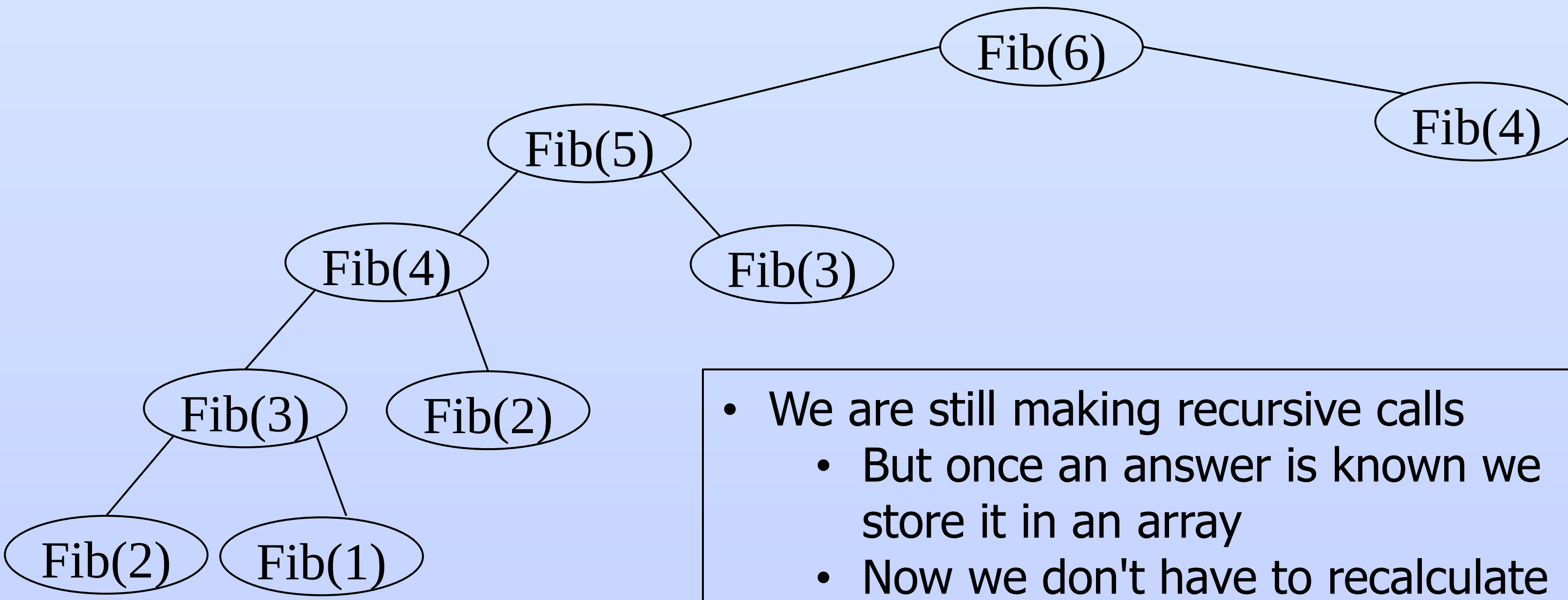
- General idea:
  - Rather than working "top down" as is done in a recursive approach we work from the "bottom up"
  - Smaller, partial solutions are used to build larger solutions, until the desired problem size is reached
    - Partial solutions often must be stored in some way
      - There were only 2 with Fibonacci (prev. two results) but for more complex problems there may be a lot
    - Many dynamic programming solutions have substantial memory requirements

- Dynamic programming can also be done "top down"
  - Often in this case we program a "normal" recursive solution, with the following changes:
    - Once a recursive subproblem has been solved, the result is stored in some fashion (ex: in a table)
    - Prior to making a recursive call, the table is consulted
      - If the solution is there use it
      - If the solution is not there, make the recursive call
  - This technique is called "memoization"
  - See fibMemo.java and next slide

Memoization Fibonacci Solution

fibSol

1	2	3	4	5	6	7	8
1	1	2	3	5	8	0	0



- We are still making recursive calls
  - But once an answer is known we store it in an array
  - Now we don't have to recalculate answers over and over

Memoization Fibonacci Solution

```
public static long [] fibSol = null;  // variable for result array

// in main()
if (fibSol == null || n >= fibSol.length)  fibSol = new long[n+1];
result = fibM(n);

static long fibM(int n)
{
    if (n <= 2)
    {
        return 1;   // base case
    }
    else if (fibSol[n] > 0)
    {
        return fibSol[n];
    }
    else
    {
        long ans = fibM(n-1) + fibM(n-2);
        fibSol[n] = ans;
        return ans;
    }
}
```

- Code has same basic recursive approach as before
- However, now we store each subproblem answer as we generate it
- Before using recursive solution we check to see if answer for given n exists in the array
    - If so use it
- Note that if we call fibM more than once from main we may not have to recurse at all for calls after the first

- Some problems that have exponential run-times (ex: NP-C problems) can be solved in pseudo-polynomial time using dynamic programming
  - Idea: Given some instances of the problem, by starting at a solution for a small size and building up to a larger size, we end up with a polynomial run-time
  - Example: Subset Sum
    - Given a set of N items, each with an integer weight and another integer M
    - Is there a subset of the set that sums exactly to M?

- Exhaustive Search
  - Try (potentially) every subset until one is found that sums to M or none are left
  - Theta(2^N) since there are that many subsets
- Pruning through Branch and Bound
  - If we do this recursively, we can stop the recursion and backtrack whenever the current sum exceeds M
    - If we are already past M we cannot add any more items to the subset so don't even try
    - Greatly reduces the number of recursive calls required
    - Still exponential in the worst case

Subset Sum Using Pruning

index	1	2	3	4	5	6	7	8	9	10
size	15	30	21	6	11	8	4	19	44	17
store	1	1	1	1	0	1	1	0	0	0

sum
50

M
50

- Consider the set of integers with sizes shown
- The store value indicates whether the number is currently in (1) or not in (0) the subset
- Assume our goal value for M is 50
  - But it can be arbitrary
- The recursive / backtracking solution will have a call for each item
  - It will recurse forward to add the "next" item
  - It will backtrack if no solution can be reached using that item
- View and listen to the presentation to see how we progress to a solution in this case
  - See next slide and subset.java for the code

Subset Sum Using Pruning

```
// in main
found = false;
for (int i = 1; i <= N; i++)  // loop in main selects first item in set
{
    if (size[i] <= M)
          find_Subset2(i, 0);
}

public void find_Subset2 (int lvl, int sum)
{
    store[lvl] = 1;          // add current item to set
    sum += size[lvl];        // increment sum
    if (sum == M)            // if solution is found, print it out
    {
          found = true;
          // print out solution
    }
    else                     // otherwise try other items in set
    {
          for (int i = lvl + 1; i <= N; i++)
          {
              if (sum + size[i] <= M)  // only try if within bound
              {
                    find_Subset2 (i, sum);
              }
          }
    }
    store[lvl] = 0;  // remove item from set to backtrack
}
```