Dynamic Programming

Procedures:

1. **Write down** the recurrence relation or recursive relation (**write it down really makes difference**)
2. Show that the number of different parameter values taken on by your recurrence is bounded by a (hopefully small) polynomial (The time needed for filling cells)
3. Specify an order of evaluation for the recurrence so the partial results you need are always available when you need them.
4. Specify and **write down** the base case or boundary conditions, usually table [0][i] and table[i][0].(**write it down really makes difference**)
5. Reconstruction usually needs an extra table. You don't have to use only one list or table, and you can use several to record other information.
6. Running time is usually (the number of cells in the whole table, the size of state space) * (the time needed for filling each cell)

Practical Procedures:

1. Initialize table and pre_table(Sometimes you will need an extra space in table for basecase, for example ls[0]=0, in python enumerate(ls,1) will help facilitate coding in the loop)
2. loop over the table, update table and pre_table(python tuple would be used very often to record info for pre_table)
3. Make final answer correct
4. Implement reconstruct_pre_table(pre_table, index), recursively

Applied Situations:

1. Optimization problem on **left-to-right (or some order)** objects, such as characters in a string, elements of a permutation, points around a polygon, or leaves in a search tree.
2. Any problem that observes the *principle of optimality.* Roughly stated, this means that partial solutions can be optimally extended with regard to the state after the partial solution, instead of the specifics of the partial solution itself. Future decisions are made based on the *consequences* of previous decisions, not the actual decisions themselves.

Routine Functions:

1. Table Initialization (Filling Base Cases. May include row initialization and col initialization)
2. Penalty or Score (When there are several choices, what is the penalty/Score for each one)
3. Goal Cell Identification (the end point of solution, usually the last element)
4. Traceback Actions (each cell can store a parent variable)

Optimization:

1. Using only necessary space (For example, in string matching, two columns are enough. Fibonacci, two cells are enough.)

Classical Problems:

1. Fibonacci Numbers, Prefix Sum
2. Binomial Coefficients/ Calculating Combinations (Pascal triangle)
3. Approximate String Matching
    a. Edit distance
    b. Sequence Alignment
    c. Substring Matching
    d. Longest Common Subsequence (LCS)
    e. Maximum monotone Subsequence
4. Longest Increasing Sequence
5. Optimal partition without rearrangement
6. Parsing context-free grammar and parsimonious parserization

Limitations:

1. Without an inherent left-to-right ordering on the objects, dynamic programming is usually doomed to require exponential space and time.
2. Problems do not satisfy the *principle of optimality* when the specifics of the operations matter, as opposed to just the cost of the operations. Such would be the case with a form of edit distance where we are not allowed to use combinations of operations in certain particular orders

Combinatorial Search (Backtracking)

Definition:
Backtracking is a systematic way to **iterate through all the possible configurations of a search space (systematic bruteforce)**. These configurations may represent all possible arrangements of objects (permutations) or all possible ways of building a collection of them (subsets). Other situations may demand enumerating all spanning trees of a graph, all paths between two vertices, or all possible ways to partition vertices into color classes.

What these problems have in common is that we must generate each one possible configuration exactly once. **Avoiding both repetitions and missing configurations means that we must define a systematic generation order**.

Procedures:

1. Backtrack implemention framework
2. Implement construct_candidates(a,k,input,c,ncandidates), which should return c and ncandidates
3. Implement is_a_solution(a,k,input)
4. Implement process_solution(a,k,input)
5. Implement make_move(a,k,input) and unmake_move(a,k,input) depend on the situation
6. Check whether there is a early-stop or other pruning technique can be used

Applied Situations:

Routine Functions:
General implementation of backtrack:

Python version(C++ version is in the end):

```
finished = False;
def backtrack(a,k,input):
    global finished
    if is_a_solution(a,k,input):
        finished = process_solution(a,k,input)
    else:
        k += 1
        c, ncandidates = construct_candidates(a,k,input)
        for i in range(ncandidates):
            a[k] = c[i]
            make_move(a,k,input)
            backtrack(a,k,input)
```

```
            unmake_move(a,k,input)
            if finished:
                return

def Driver():
    finished = False
    a = [0] * MAXN      #Solution Vector
    k = -1
    backtrack(a,k,input)
```

1. Boolean **is_a_solution(a,k,input)** – This Boolean function tests whether the first k elements of vector a from a complete solution for the given problem. The last argument, input, allows us to pass general information into the routine. We can use it to specify n—the size of a target solution.
2. Void **process_solution(a,k,input)** – This routine prints, counts, or however processes a complete solution once it is constructed
3. **construct_candidates(a,k,input,c,ncandidates)** – This routine fills an array c with the complete set of possible candidates for the kth position of a, given the contents of the first k – 1 positions. The number of candidates returned in this array is denoted by ncandidates. Again, input may be used to pass auxiliary information. This function may need several other helper functions.
4. **make_move(a,k,input) and unmake_move(a,k,input)** – These routines enable us to modify a data structure in response to the latest move, as well as clean up this data structure if we decide to take back the move. Such a data structure could be rebuilt from scratch from the solution vector a as needed, but this is inefficient when each move involves incremental changes that can easily be undone. Typically will be used in backtracking problem like Sudoku, eight queens.

Optimization (Done in construct_candidates):

1. Search Pruning: Pruning is the technique of cutting off the search the instant we have established that a partial solution cannot be extended into a full solution.
2. Exploiting symmetry: is another avenue for reducing combinatorial searches. Pruning away partial solutions identical to those previously considered requires recognizing underlying symmetries in the search space. (for example, in TSP, every start vertex is the same, reduce to O((n-1)!) from O(n!))

Classical Problems:

1. Constructing all subsets
2. Constructing all permutations
3. Constructing all paths in a graph
4. Sudoku solver/checker
5. Eight Queens

Limitations:

1. It is essentially a brute-force method, so maybe not very efficient.

Attchment:

C++ Version:

```
bool finished = FALSE; /* found all solutions yet? */

backtrack(int a[], int k, data input){

int c[MAXCANDIDATES];  /* candidates for next position */
int ncandidates;        /* next position candidate count */
int i;    /* counter */
if (is_a_solution(a,k,input))
       process_solution(a,k,input);
else {

       k = k+1;

       construct_candidates(a,k,input,c,&ncandidates);

       for (i=0; i<ncandidates; i++) {
          a[k] = c[i];
          make_move(a,k,input);          //maybe not necessary
          backtrack(a,k,input);
          unmake_move(a,k,input);         //maybe not necessary
          if (finished) return;             /* terminate early */
} }

}
```

Heuristic Search Methods

#Motivation and description:
Combinatorial search gives us a method to find the optimal solution, but for large datasets or complicated problem, searching all configurations is too expensive. Generally, there are three different heuristic search methods: **random sampling, gradient-descent/Local greedy search, and simulated annealing**. Each method has two common components: **Solution space representation, and Cost function.**

#Random Sampling (Monte Carlo method)
It is also called the Monte Carlo method. We repeatedly construct random solutions and evaluate them, stopping as soon as we get a good enough solution, or(more likely) when we are tired of waiting. Then report the best solution found so far.

Routine code:

```
random_sampling(tsp_instance *t, int nsamples, tsp_solution *bestsol)
{
        tsp_solution s;                 /* current tsp solution */
        double best_cost;               /* best cost so far */
        double cost_now;                /* current cost */
        int i;                          /* counter */

        initialize_solution(t->n,&s);
        best_cost = solution_cost(&s,t);
        copy_solution(&s,bestsol);

        for (i=1; i<=nsamples; i++) {
                random_solution(&s);
                cost_now = solution_cost(&s,t);
                if (cost_now < best_cost) {
                        best_cost = cost_now;
                        copy_solution(&s,bestsol);
                }
        }
}
```

When to apply random sampling?
1. When there are a high proportion of acceptable solutions
2. When there is no coherence or relation in the solution space (when there is no sense of when we are getting *closer* to a solution)

#Local Greedy Search

A local search employs *local neighborhood* around every element in the solution space. Think of each element x in the solution space as a vertex, with a directed edge(x,y) to every candidate solution y that is a neighbor of x. Our search proceeds from x to the most promising candidate in x's neighborhood.
We also want a general transition mechanism that takes us to the next solution by slightly modifying the current one. Typical transition mechanisms include swapping a random pair of items or changing (inserting or deleting) a single item in the solution. Another transition is called greedy search.
Hill climbing/greedy search: starting at some arbitrary point and taking any step that leads in the direction we want to travel. (Limitation: finding local optima, but not the global optima)

Routine code:

```
hill_climbing(tsp_instance *t, tsp_solution *s)
{
     double cost;                /* best cost so far */
     double delta;               /* swap cost */
     int i,j;                    /* counters */
     bool stuck;                 /* did I get a better solution? */
     double transition();

     initialize_solution(t->n,s);
     random_solution(s);
     cost = solution_cost(s,t);

     do {
             stuck = TRUE;
             for (i=1; i<t->n; i++)
                     for (j=i+1; j<=t->n; j++) {
                             delta = transition(s,t,i,j);
                             if (delta < 0) {
                                     stuck = FALSE;
                                     cost = cost + delta;
                             }


                                else
                                        transition(s,t,j,i);
                     }
      } while (!stuck);
}
```

When do local search do well?

1. When there is great coherence in the solution space (best when solution space is convex, simple version of linear programming)

2. Whenever the cost of incremental evaluation is much cheaper than global evaluation

#Simulated Annealing
Modeling the physical system of thermodynamics. Not very practical in interview, so we skip it here.

How to design algorithms? (Problem-solving skills)

#The key to algorithm design is to ask *What if we do this? And Why can't we do it this way?*

#Have a good global strategy to the problem, and tactics to sub-problems.

#Ask youself some questions when design an algorithm or stuck on some step.

1. Do I really understand the problem?

a) What exactly does the input consist of?

b) What exactly are the desired results or output?

c) Can I construct an input example small enough to solve by hand? What happens when I try to solve it manually?

d) How important is it to my application that I always find the optimal answer? Can I settle for something close to the optimal answer? (Usually heuristic method or greedy algorithm will do the work)

e) How large is a typical instance of my problem? Will I be working on 10 items? 1000 items? 1,000,000 items?

f) How important is speed in my application? Must the problem be solved within one second? One minute? One hour? One day?

g) How much time and effort can I invest in implementation? Will I be limited to simple algorithms that can be coded up in a day, or do I have the freedom and time to experiment with a couple of approaches and see which is best?

h) Am I trying to solve a numerical problem? A graph algorithm problem? A geometric problem? A string problem? A set problem? Which formulation seems easiest?

2. Can I find a simple algorithm or heuristic for my problem?

a) Will brute force solve my problem *correctly* by searching through all subsets or arrangements and picking the best one?
    1) If so, Why am I sure that this algorithm always gives the correct answer?
    2) How do I measure the quality of a solution once I construct it?
    3) Does this simple, slow solution run in polynomial or exponential time? Is my problem small enough that this brute-force solution will suffice?
    4) Am I certain that my problem is sufficiently well defined to actually have a correct solution?

b) Can I solve my problem by repeatedly trying some simple rule, like picking the biggest item first? the smallest item first? A random item first?

    1) If so, on what types of inputs does this heuristic work well? Do these correspond to the data that might arise in my application?
    2) On what types of inputs does this heuristic work badly? If no such examples can be found, can I show that it always works well?
    3) How fast does my heuristic come up with an answer? Does it have a simple implementation?

3. Are there special cases of the problem that I know how to solve?

a) Can I solve the problem efficiently when I ignore some of the input parameters?
b) Does the problem become easier to solve when I set some of the input parameters to trivial values, such as 0 or 1?
c) Can I simplify the problem to the point where I can solve it efficient?
d) Why can't this special-case algorithm be generalized to a wider class of inputs?
e) Is my problem a special case of a more general problem ?

4. Which of the standard algorithm design paradigms are most relevant to my problem?

a) Is there a set of items that can be sorted by size or some key? Does this sorted order make it easier to find the answer?
b) Is there a way to split the problem in two smaller problems, perhaps by doing a binary search? How about partitioning the elements into big and small, or left and right? Does this suggest a divide-and-conquer algorithm?
c) Do the input objects or desired solution have a natural left-to-right order, such as characters in a string, elements of a permutation, or leaves of a tree? Can I use dynamic programming to exploit this order?
d) Are there certain operations being done repeatedly, such as searching, or finding the largest/smallest element? Can I use a data structure to speed up these queries? What about a dictionary/hash table or a heap/priority queue?
e) Can I use random sampling to select which object to pick next? what bout constructing many random configurations and picking the best one? Can I use some kind of directed randomness like simulated annealing to zoom in on the best solution?
f) Can I formulate my problem as a linear program? How about an integer program?

5. Am I still stuck?

Why don't I go back to the beginning and work through these questions again? Did any of my answers change during my latest trip through the list?

General Data Structure

In practice, it is more important to avoid using a bad data structure than to identify the single best option available.

1. Array vs Linked List

#Contiguously-allocated structures:
Composed of continuous memory, and include arrays, matrices, heaps, and hashtables.

#Array/ Dynamic Array
Array is a fundamental continuous-allocated data structure in memory. An array usually has a fixed size and each element can be efficiently located by its index or address. Dynamic array is essentially a variable-size array, whenever the size of elements equal to the capacity of array, it will open a new memory space and resize a new array with double capacity and then copy current elements to the new memory location and release old memory space.
Advantages:
1). Constant time random access given the index
2). Space efficiency (pure data, no space for link or other formatting info)
3). Memory locality and better cache performance (iterate or access to block of data in high-speed)

Disadvantages:
1). Insertion and deletion in the beginning or middle take O(n)

#Linked data structures:
Composed of distinct chunks of memory bound together by pointers, and include linked-list, trees, and graph adjacency lists.

#Linked list
Advantages:
1). Insertion and deletion are efficient, O(1)

Distadvantages:
1). Require extra space for storing pointer fields
2). Random access is not efficient O(n)

2. Stacks and Queues

#Stack:
Support retrieval by Last-in,first-out(LIFO) order. Has two operations: push and pop.

#Queue:
Support retrieval in First-in,first-out(FIFO) order. Has two operations: enqueue and dequeue

3. Dictionaries

#Unsorted linked lists or arrays implementation:
Pros:
For small data sets, an unsorted array is probably the easiest data structure to maintain.
Cons:
Once the dictionary becomes larger than 50 to 100 items, the linear search time will kill you.
#Sorted linked lists or arrays implementation:
Pros:
A sorted array will be appropriate if and only if there are not many insertions or deletions
Cons:
Usually not worth the effort
#Array implementation Running Time
Seven basic operations, can be implemented as sorted or un-sorted array.

Dictionary operation	Unsorted array	Sorted array
Search(L, k)	$O(n)$	$O(\log n)$
Insert(L, x)	$O(1)$	$O(n)$
Delete(L, x)	$O(1)^*$	$O(n)$
Successor(L, x)	$O(n)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(1)$
Minimum(L)	$O(n)$	$O(1)$
Maximum(L)	$O(n)$	$O(1)$

 # Linked list implementation Running Time

Dictionary operation	Singly unsorted	Double unsorted	Singly sorted	Doubly sorted
Search(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Delete(L, x)	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Successor(L, x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
Minimum(L)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Maximum(L)	$O(n)$	$O(n)$	$O(1)^*$	$O(1)$

#Hash tables Implementation
For applications involving a moderate-to-large number of keys, a hash table is probably the right way to go.
Pros:
A good hash table will outperform a sorted or unsorted array in most applications.
Search, insertion, deletion all in O(1)
Cons:
Several design decisions must be made:
How to deal with collision?
How big should the table be?
What hash function should I use?

#Binary search trees/ Balanced BST
Red-Black tree, AVL tree, 2/3 tree.
Splay tree, which uses rotations to move any accessed key to the root. Frequently used or recently accessed node will sit near the top of the tree, allowing faster search.
pros:
Fast insertions, deletions, and queries. O(logn)
cons:
unbalanced search trees constructed by inserting keys in sorted order are a disaster.

#B-tree/B+ tree
The idea behind a B-tree is to collapse several levels of a binary search tree into a single large node, so that we can make the equivalent of several search steps before another disk access is needed.
pros:
For data sets so large that they will not fit in main memory (more than 1 million items). It is best to store the data with B-tree.
cons:
Maybe overkill for small dataset

4. Binary Search Tree(BST)/ Balanced Binary Search Tree
Search: average, O(h)->O(logn), worst case O(n)
Find-min/max: average O(logn), worst case O(n)
Insertion/Deletion: average O(logn), worst case O(n)
Traverse: O(n)

Balanced binary search tree avoids worst case and guarantee O(logn) for most operations.

5.Priority Queue

Three primary operations: Insert, Find-min, Delete-min
They are called “priority” queues because they enable you to retrieve items not by the insertion time (as in a stack or queue), nor by a key match (as in a dictionary), but by which item has the highest priority of retrieval. It is good for mixing insertions, deletions and queries.

#Sorted array or linked list implementation

Pros:
Efficient to both identify the smallest element and delete it by decrementing the top index. O(1)
Cons:
Slow insertions. O(n), array: binary search to find it, and move all the elements. For linked list, searching the place takes O(n)
Only suitable when there will be few insertions into priority queue.

#Binary Heap Implementation

Heap is the one of the best implementation of priority queue.
Heap maintain an implicit binary tree structure in an array, such that the key of the root of any subtree is less than that of all its descendents. So, the minimum key always sits at the top of the heap.
Insertion: placing new item at an open leaf and sift up the element.
Deletion: delete root and put the last leaf item in the root and sift down.
Pros:
Insertion: O(logn)
extract-min: O(logn), find-min:O(1)
Heap construction by incremental insertion: O(nlogn)
Fast heap construction(n times bubble down): O(n)
Also good when you know the upper bound on the number of items. This could be solved by using dynamic arrays.

#Binary search trees

pros:
very effective, the smallest element is always the leftmost leaf, while the largest element is always the rightmost leaf.
Insertion, find-min, delete-min all take O(logn)

#Fibonacci heaps

Fibonacci heap is essentially a forest of binary heaps and has a pointer on the min value. It is designed to speed up decrease-key operations, where the priority of an item in the root is reduced. Insert, find-min, decrease-key, merge all take O(1).
Delete-min takes O(logn) because of consolidation.

6.Disjoint-set (Union-find) data structure

Def: A union-find data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint subsets.

It supports **Three** useful **operations**:

- **MakeSet**: makes a set containing only a given element(a singleton)
- **Find:** Determine which subset a particular element is in. Find typically returns an item from this set that serves as its 'representative'; by comparing the result of two *Find* operations, one can determine whether two elements are in the same subset
- **Union**: Join two subsets into a single subset

Techniques: Originally, union-find structure can be implemented as a linked list. But we can use two techniques to enhance this data structure:

- **Union by Rank**: happen in *Union* operation, always attach the smaller tree to the root of the larger tree. Since it is the rank(depth) of the tree that affects the running time, the tree with smaller rank gets added under the root of the deeper tree, which only increases the rank if the ranks were equal.
- **Path compression:** happen in *Find* operation, is a way of flattening the structure of the tree. The idea is that each node visited on the way to a root node may as well be attached directly to the root node; they all share the same representative.

Running Time:
If we use union by rank and path compression, the worst-case running time for *MakeSet, Union, Find* is O(1), O(logn), O(logn), respectively.

Implementation(python):

```
class Node:
        def __init__ (self, label):
                self.label = label
        def __repr__(self):
                return self.label
```

```
def MakeSet(x):
        x.parent = x
        x.rank   = 0  #a single element tree has rank 0

def Union(x, y):
        xRoot = Find(x)
        yRoot = Find(y)
        if xRoot == yRoot:
                return
        # Unless x and y are already in same set, merge them
        if xRoot.rank > yRoot.rank:          #Union by rank
                yRoot.parent = xRoot
        elif xRoot.rank < yRoot.rank:
                xRoot.parent = yRoot
        else:
```

```
                yRoot.parent = xRoot
                xRoot.rank = xRoot.rank + 1
def Find(x):
        if x.parent == x:
                return x
        else:
                x.parent = Find(x.parent)   #Path Compression
                return x.parent
```

Graph Data structure and algorithm

We assume the graph G = (V,E) contains n vertices and m edges.

#Adjacency Matrix

We can represent an n*n matrix M, where element M[i,j] = 1 if (i,j) is an edge of G, and 0 if it isn't. Space:O(V^2)

Pros:

Adjacency matrix allows fast answers to the question "is this edge in G?"(edge checking), and rapid updates for edge insertion and deletion.

Cons:

It may use excessive space for graphs with many vertices and relatively few edges(sparse graph)

#Adjacency list

We can represent sparse graphs by using linked lists to store the neighbors adjacent to each vertex. Space: O(2E)

Pros:

Easy to design for most graph algorithms

Cons:

Edge checking is slower, O(V)

Comparison	Winner
Faster to test if (x, y) is in graph?	adjacency matrices
Faster to find the degree of a vertex?	adjacency lists
Less memory on small graphs?	adjacency lists $(m + n)$ vs. (n^2)
Less memory on big graphs?	adjacency matrices (a small win)
Edge insertion or deletion?	adjacency matrices $O(1)$ vs. $O(d)$
Faster to traverse the graph?	adjacency lists $\Theta(m + n)$ vs. $\Theta(n^2)$
Better for most problems?	adjacency lists

Figure 5.5: Relative advantages of adjacency lists and matrices.

#Graph Traversal

Assume each node has three states:

- undiscovered – the vertex is in its initial, virgin state
- discovered – the vertex has been found, but we have not yet checked out all its incident edges
- processed – the vertex after we have visited all its incident edges.

Breadth-First-Search(BFS), by using queue

BFS can access both nodes and edges

If we only want to process each node, we don't need the state "discovered".

```
BFS(G, s)
    for each vertex u ∈ V[G] − {s} do
        state[u] = "undiscovered"
        p[u] = nil, i.e. no parent is in the BFS tree
    state[s] = "discovered"
    p[s] = nil
    Q = {s}
    while Q ≠ ∅ do
        u = dequeue[Q]
        process vertex u as desired
        for each v ∈ Adj[u] do
            process edge (u, v) as desired
            if state[v] = "undiscovered" then
                state[v] = "discovered"
                p[v] = u
                enqueue[Q, v]
        state[u] = "processed"
```

Depth-First-Search(DFS), by using stack

A recursive implementation of DFS:[5]

```
1  procedure DFS(G,v):
2      label v as discovered
3      for all edges from v to w in G.adjacentEdges(v) do
4          if vertex w is not labeled as discovered then
5              recursively call DFS(G,w)
```

A non-recursive implementation of DFS:[6]

```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5            v = S.pop()
6            if v is not labeled as discovered:
7                label v as discovered
8                for all edges from v to w in G.adjacentEdges(v) do
9                    S.push(w)
```

Catalog of interesting problems

#Constrained or unconstrained optimization

Input description: A function $f(x_1, \ldots, x_n)$.

Problem description: What point $p = (p_1, \ldots, p_n)$ maximizes (or minimizes) the function f?

Unconstrained optimization:
The most efficient algorithms for unconstrained optimization use derivatives and partial derivatives to find local optima, to point out the direction in which moving from the current point does the most to increase or decrease the function. Such derivatives can sometimes be computed analytically, or they can be estimated numerically by taking the difference between the values of nearby points. A method called ***steepest descent*** can be used to find local optima.

Constrained optimization:
One approach for constrained optimization is to use a method for unconstrained optimization, but add a penalty according to how many constraints are violated. Penalty function is problem-specific, but it often makes sense to vary the penalties as optimization proceeds. At the end, the penalties should be very high to ensure that all constraints are satisfied.

#Linear Programming

Input description: A set S of n linear inequalities on m variables

$$S_i := \sum_{j=1}^{m} c_{ij} \cdot x_j \geq b_i, \ 1 \leq i \leq n$$

and a linear optimization function $f(X) = \sum_{j=1}^{m} c_j \cdot x_j$.

Problem description: Which variable assignment X' maximizes the objective function f while satisfying all inequalities S?

#Job Scheduling

Processor scheduling, people-to-jobs, meetings-to-rooms, or courses-to-exam periods are all different examples of scheduling problems.

Input description: A directed acyclic graph $G = (V, E)$, where vertices represent jobs and edge (u, v) implies that task u must be completed before task v.

Problem description: What schedule of tasks completes the job using the minimum amount of time or processors?

//Topological sorting can construct a schedule consistent with precedence constraints. The graph must be a DAG(detect cycle first)

The conceptually simplest linear-time algorithm for topological sorting performs a depth-first search of the **DAG** to identify the complete set of source vertices, where source vertices are vertices of in-degree zero. At least one such source must exist in any DAG. Source vertices can appear at the start of any schedule without violating any constraints. Deleting all the outgoing edges of these source vertices will create new source vertices, which can then sit comfortably to the immediate right of the first set. We repeat until all vertices are accounted for. A modest amount of care with data structures (adjacency lists and queues) is sufficient to make this run in $O(n + m)$ time.

For problem of constructing all topological sorting, we can use backtracking, building all possible orderings from left to right.

#Multi-objective optimization
Use a cost or profit function, and give different variables a weight, then optimize the function.

Specialized Data Structure

1.String data structure:

Character strings are typically represented by arrays of characters, perhaps with a special character to mark the end of the string. **Suffix trees/arrays** are special data structures that preprocess strings to make pattern-matching operations faster.

2. Geometric data structures:

Geometric data typically consists of collections of data points and regions. Regions in the plane can be described by polygons, where a chain of line segments gives the boundary of the polygon. Polygons can be represented using an array of points (v1,…,vn,v1), such that (vi,vi+1) is a segment of the boundary. Spatial data structures such as **kd-trees** organize points and regions by geometric location to support fast search.

3.Graph data structure:

Graphs are typically represented using either **adjacency matrices** or **adjacency lists**. The choice of representation can have a substantial impact on the design of the resulting graph algorithms.(See previous section)

4.Set data structures:

Subsets of items are typically represented using a **dictionary** to support fast membership queries. Alternatively, **bit vectors** are Boolean arrays such the ith bit represents true if i is in the subset.