DS286 | 2016-11-11,16

L25,26: Classes of Algorithms

Yogesh Simmhan
simmhan@cds.iisc.ac.in

© David Matuszek, UPenn, CIT594, Programming Languages & Techniques II

Algorithm classification

- Algorithms that use a *similar problem-solving approach* can be grouped together
  - A classification scheme for algorithms
- Classification is neither exhaustive nor disjoint
- The purpose is not to be able to classify an algorithm as one type or another, but to *highlight the various ways in which a problem can be attacked*

A short list of categories

- Algorithm types we will consider include:
  1. Simple recursive algorithms
  2. Backtracking algorithms
  3. Divide and conquer algorithms
  4. Dynamic programming algorithms
  5. Greedy algorithms
  6. Branch and bound algorithms
  7. Brute force algorithms
  8. Randomized algorithms

Simple Recursive Algorithms

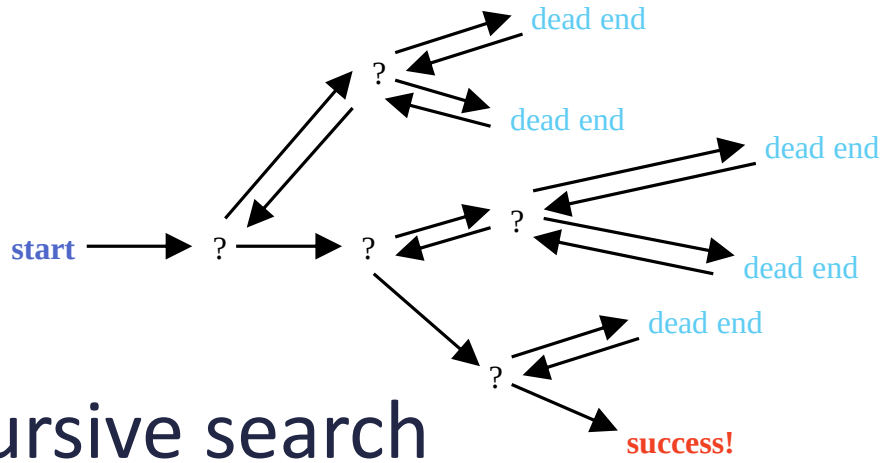
- A simple recursive algorithm:
  1. Solves the base cases directly
  2. Recurs with a simpler subproblem
  3. Does some extra work to convert the solution to the simpler subproblem into a solution to the given problem

- These are “simple” because several of the other algorithm types are inherently recursive

Sample Recursive Algorithms

- To count the number of elements in a list:
  - If the list is empty, return zero; otherwise,
  - Step past the first element, and count the remaining elements in the list
  - Add one to the result
- To test if a value occurs in a list:
  - If the list is empty, return false; otherwise,
  - If the first thing in the list is the given value, return true; otherwise
  - Step past the first element, and test whether the value occurs in the remainder of the list

Backtracking algorithms



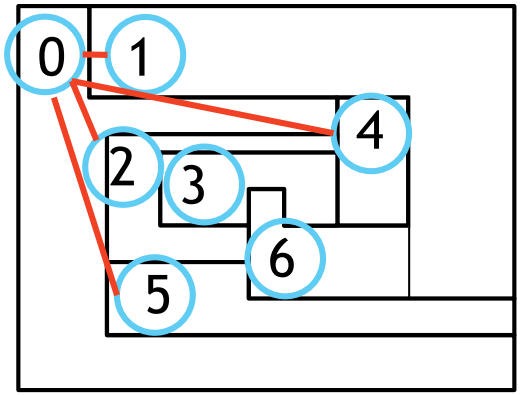
- Uses a depth-first recursive search
  - Test to see if a solution has been found, and if so, returns it; otherwise
  - For each choice that can be made at this point,
    - Make that choice
    - Recurse
    - If the recursion returns a solution, return it
  - If no choices remain, return failure

Sample backtracking algo.

- *The **Four Color Theorem** states that any map on a plane can be colored with no more than four colors, so that no two countries with a common border are the same color*
- **color(Country n)**
  - If all countries have been colored (*n* > no. of countries) return success; otherwise,
  - For each color *c* of four colors,
    - If country *n* is not adjacent to a country that has been colored *c*
      - Color country *n* with color *c*
      - recursively color country *n+1*
      - If successful, return success
  - If loop exits, return failure

Map is a graph. Countries are list of vertices. Adjacency list has neighboring countries.

```
mapColors = int[map.length];
int RED=0, GREEN=1, PINK=2, BLUE=3;
boolean explore(int country, int color) {
   if (country >= map.length) return true;
   if (okToColor(country, color)) {
      mapColors[country] = color;
      for (int i = RED; i <= BLUE; i++) {
         if (explore(country + 1, i)) return
         true;
      }
   }
   return false;
}
```



Divide and Conquer

- A divide and conquer algorithm consists of two parts:
  - *Divide* the problem into smaller subproblems of the same type, and solve these subproblems recursively
  - *Combine* the solutions to the subproblems into a solution to the original problem
- *Traditionally, an algorithm is only called "divide and conquer" if it contains at least two recursive calls*

Examples

- Quicksort:
  - Partition the array into two parts *(smaller numbers in one part, larger numbers in the other part)*
  - Quicksort each of the parts
  - No additional work is required to combine the two sorted parts
- Mergesort:
  - Cut the array in half, and mergesort each half
  - Combine the two sorted arrays into a single sorted array by merging them

Binary search tree lookup?

- Here's how we look up something in a binary search tree:
  - Compare the key to the value in the root
    - If the two values are equal, report success
    - If the key is less, search the left subtree
    - If the key is greater, search the right subtree
- This is ***not*** a divide and conquer algorithm because, although there are two recursive calls, only one is used at each level of the recursion

Fibonacci numbers

- $n_i = n_{(i-1)} + n_{(i-2)}$
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- To find the n^{th} Fibonacci number:
  - If n is zero or one, return 1; otherwise,
  - Compute fibonacci(n-1) and fibonacci(n-2)
  - Return the sum of these two numbers

- This is an expensive algorithm
  - It requires O(fibonacci(n)) time
  - This is equivalent to exponential time, that is, $O(2^n)$
    - *Binary tree of height 'n' with f(n) having two children, f(n-1), f(n-2)*

Dynamic Programming (DP)

- A dynamic programming algorithm "remembers" past results and uses them to find new results
  - *Memoization*
- Dynamic programming is generally used for optimization problems
  - Multiple solutions exist, need to find the "best" one
  - Requires "optimal substructure" and "overlapping subproblems"
    - **Optimal substructure**: Optimal solution can be constructed from optimal solutions to subproblems
    - **Overlapping subproblems**: Solutions to subproblems can be stored and reused in a bottom-up fashion
- *This differs from Divide and Conquer, where subproblems generally need not overlap*

Fibonacci numbers again

- To find the n^{th} Fibonacci number:
  - If *n* is zero or one, return one; otherwise,
  - Compute, *or look up in a table*, fibonacci(n-1) and fibonacci(n-2)
  - Find the sum of these two numbers
  - *Store the result in a table* and return it
- Since finding the n^{th} Fibonacci number involves finding all smaller Fibonacci numbers, the second recursive call has little work to do
- The table may be preserved and used again later
- Other examples: *Floyd–Warshall All-Pairs Shortest Path (APSP)* algorithm, *Towers of Hanoi*, …

Greedy algorithms

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases: At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Example: Counting money

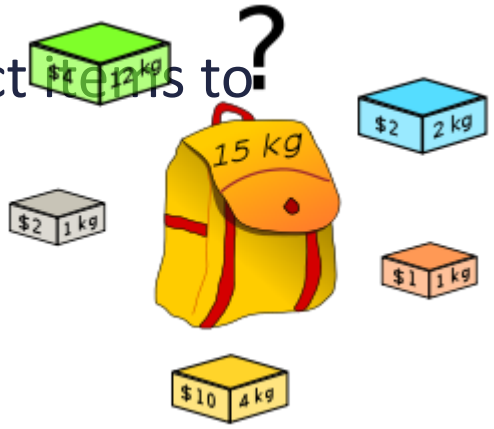
- *Suppose you want to count out a certain amount of money, using the fewest possible currency notes and coins*
- A greedy algorithm would do this would be:
  At each step, take the largest possible note or coin that does not overshoot
  - Example: To make ₹639, you can choose:
    - a ₹ 500 note
    - a ₹ 100 note, to make ₹ 600
    - a ₹ 20 note, to make ₹ 620
    - A ₹ 10 note, to make ₹ 630
    - A ₹ 5 coin, to make ₹ 635
    - four ₹ 1 coins, to make ₹ 639
- For INR and US money, the greedy algorithm gives the optimum solution

A failure of the greedy algorithm

- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

Other applications of Greedy Algorithms

1. Shortest path problem
   - A simple greedy strategy, Dijikstra's greedy algorithm
   - Greedily pick the shortest among the vertices touched so far
2. 0/1 Knapsack problem on combinatorial optimization
   - Pack a knapsack of weight capacity *c*
   - Given *n* items with weight and profit, select items to Maximize **sum($p_i x_i$)**
   - Subject to constraints *sum($w_i x_i$) <= c*



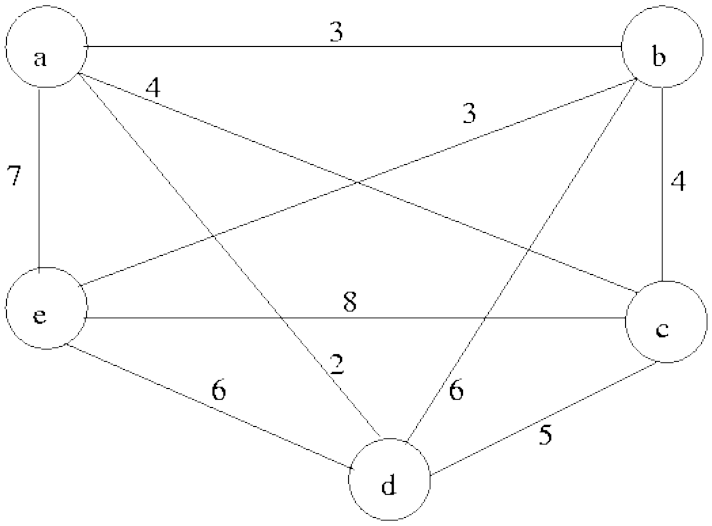
Branch & Bound algorithms

- Branch and bound algorithms are generally used for optimization problems. *Similar to backtracking.*
  - As the algorithm progresses, a tree of subproblems is formed
  - The original problem is considered the “root problem”
  - A method is used to construct an upper and lower bound for a given problem
  - At each node, apply the bounding methods
    - If the bounds match, it is deemed a feasible solution to that particular subproblem
    - If bounds do *not* match, partition the problem represented by that node, and make the two subproblems into children nodes
  - Continue, using the best known feasible solution to trim sections of the tree, until all nodes have been solved or trimmed

Example branch and bound algorithm

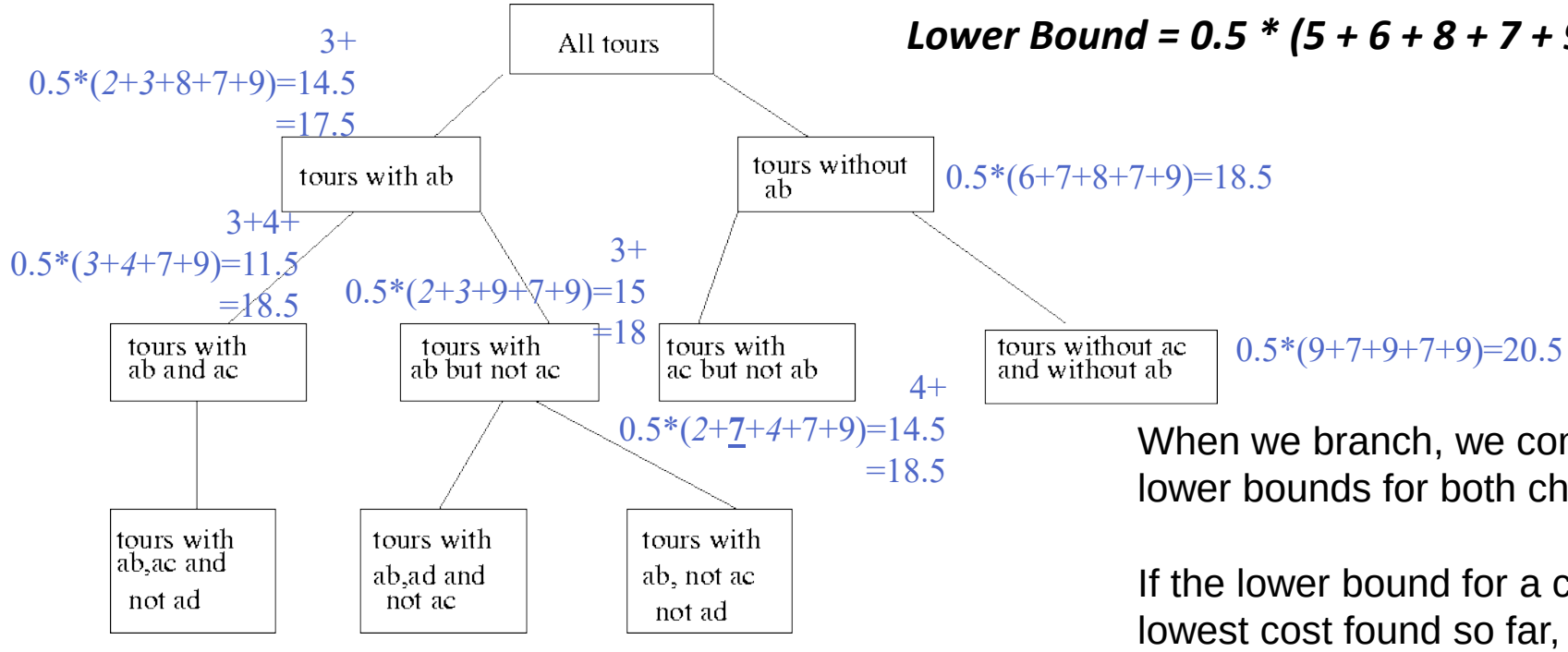
- *"Suppose it is required to minimize an objective function. Suppose that we have a method for getting a lower bound on the cost of any solution among those in the set of solutions represented by some subset. If the best solution found so far costs less than the lower bound for this subset, we need not explore this subset at all."*
- **Traveling salesman problem**: A salesman has to visit each of *n* cities once each and return to the original city, while minimize total distance traveled
  - Split into two subproblems, whether to take an out edge from a vertex or not.
  - If current best solution smaller than the lower bound of a subset, do not explore.
  - Lower bound given by *0.5*(sum of tours on two edges, for all vertices)*

http://lcm.csa.iisc.ernet.in/dsa/node187.html



Node	Least cost edges	Total cost
a	(a, d), (a, b)	2+3=5
b	(a, b), (b, e)	3+3=6
c	(c, b), (c, a)	4+4=8
d	(d, a), (d, c)	2+5=7
e	(e, b), (e, d)	3+6=9

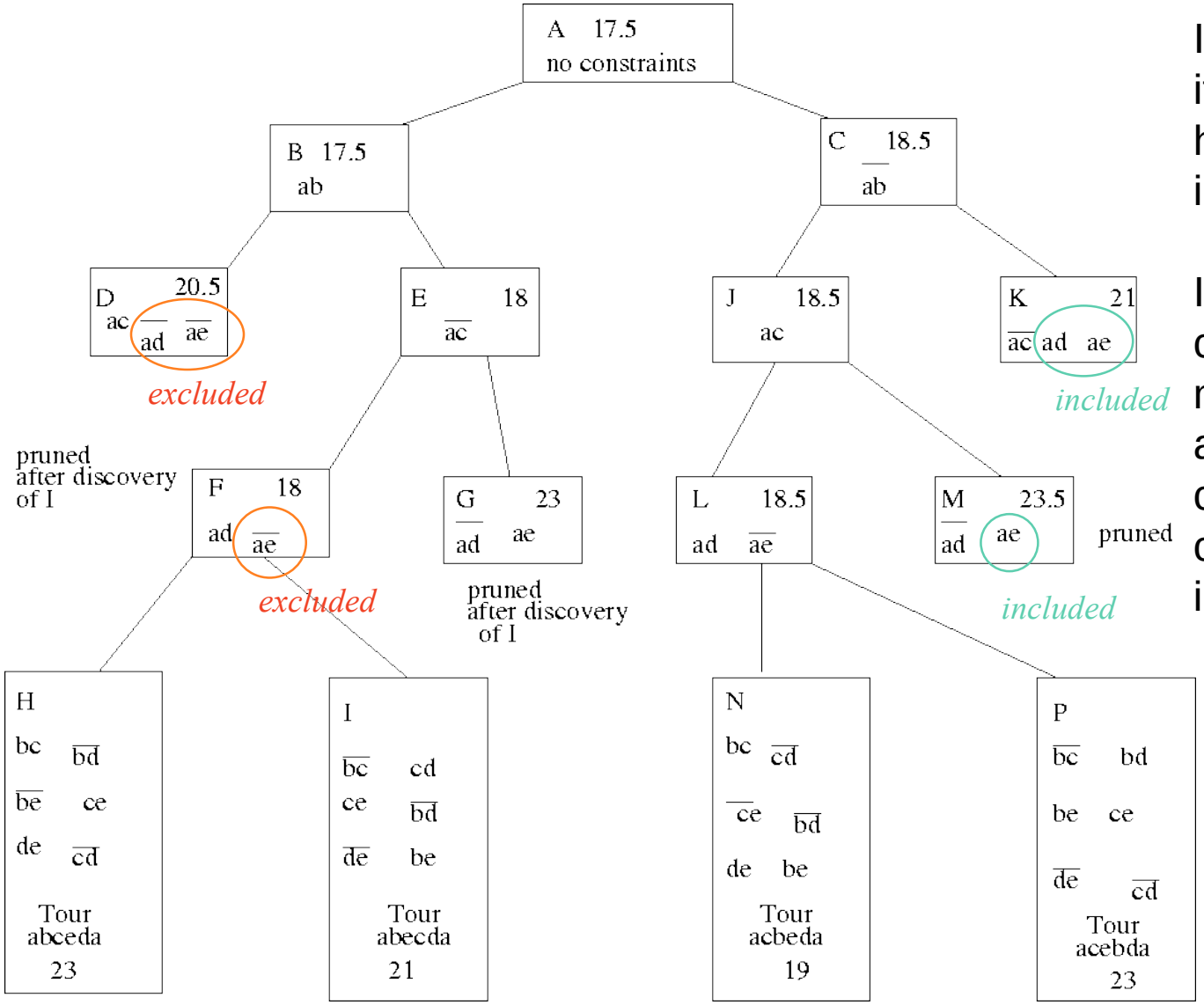
Lower Bound = 0.5 * (5 + 6 + 8 + 7 + 9) = 17.5



When we branch, we compute lower bounds for both children.

If the lower bound for a child is >= lowest cost found so far, we prune that child.

http://lcm.csa.iisc.ernet.in/dsa/node187.html



If excluding (x, y) makes it impossible for x or y to have two adjacent edges in the tour, include (x, y).

If including (x, y) would cause x or y to have more than two edges adjacent in the tour, or complete a non-tour cycle with edges already included, exclude (x, y).

http://lcm.csa.iisc.ernet.in/dsa/node187.html

Brute force algorithm

- A brute force algorithm simply tries *all* possibilities until a satisfactory solution is found
- Such an algorithm can be:
  - Optimizing: Find the *best* solution. This may require finding all solutions, or if a value for the best solution is known, it may stop when any best solution is found
    - Example: Finding the best path for a traveling salesman
  - Satisficing: Stop as soon as a solution is found that is *good enough*
    - Example: Finding a traveling salesman path that is within 10% of optimal

Improving brute force algorithms

- Often, brute force algorithms require exponential time
- Various *heuristics* and *optimizations* can be used
  - Heuristic: A "rule of thumb" that helps you decide which possibilities to look at first
  - Optimization: In this case, a way to eliminate certain possibilities without fully exploring them

Randomized algorithms

- A randomized algorithm uses a random number at least once during the computation to make a decision
  - Example: In Quicksort, using a random number to choose a pivot
  - Example: Trying to factor a large number by choosing random numbers as possible divisors

Dynamic Programming

Lecture 11: Dynamic Programming, Avrim Blum

https://www.cs.cmu.edu/~avrim/451f09/lectures/lect1001.pdf

Dynamic Programming

- General approach to solving problems
  - general method like “divide-and-conquer”
- Unlike divide-and-conquer, the subproblems will typically overlap
- **Basic Idea (version 1)**: take our problem and break it into a reasonable number of subproblems ($O(n^2)$) that can be optimally solved to give the optimal solution to the larger one.
- Unlike divide-and-conquer (as in mergesort or quicksort) it is OK if our subproblems overlap, so long as there are not too many of them.

Longest Common Subsequence (LCS)

- We are given two strings: string S of length n, and string T of length m. Our goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings.
  - Genomics, “diff” in code repositories (edit distance)

$$S_1 = AAACCGTGAGTTATTCGTTCTAGAA$$
$$S_2 = CACCCCTAAGGTACCTTTGGTTC$$

$$ACCTAGTACTTTG$$

S = ABAZDC
T = BACBAD

LCS = ABAD

http://www.columbia.edu/~cs2035/courses/csor4231.F11/lcs.pdf

LCS

- Say **LCS[i,j]** is the length of the LCS of **S[1..i]** with **T[1..j]**. How can we solve for LCS[i,j] in terms of the LCS's of the smaller problems?
- **Case 1**: S[i] <> T[j]
  - The subsequence has to ignore one of S[i] or T[j]
  - LCS[i, j] = max(LCS[i − 1, j], LCS[i, j − 1])
- **Case 2**: S[i] = T[j]
  - The LCS of S[1..i] and T[1..j] might as well match them up.
  - A common subsequence that matched S[i] to an earlier location in T could always match it to T[j] instead
  - LCS[i, j] = 1 + LCS[i − 1, j − 1]

LCS

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1); //
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) )
  return result;
}
```

Traceback
D A B A
and reverse
A B A D

		B	A	C	B	A	D
	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
B	0	1	1	1	2	2	2
A	0	1	2	2	2	3	3
Z	0	1	2	2	2	3	3
D	0	1	2	2	2	3	4
C	0	1	2	3	3	3	4

R = (GAC), and *C* = (AGCAT)

Traceback example

	Ø	A	G	C	A	T
Ø	0	**0**	0	0	0	0
G	0	←0	↖1	←1	←1	←1
A	0	↖1	←1	←1	↖2	←2
C	0	↑1	←1	↖2	←2	←2

Exponential time!

e.g. no characters match

https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

Memoization

- **Basic Idea (version 2)**: Suppose you have a recurrence where many of the subproblems in the recursion tree are the same. Then you can get a savings only if you store your computations so that you compute each different subproblem just once.
- You can store these solutions in an array or hash table. This is called ***memoizing***.

LCS with Memoization

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (arr[n][m] != unknown) return arr[n][m];  // <- added this line (*)
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1);
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
  arr[n][m] = result;                          // <- and this line (**)
  return result;
}
```

Complexity is O(mn)
(Size of array)

Knapsack Problem

- We are given a set of n items, where each item i is specified by a size s_i and a value v_i. We are also given a size bound S (the size of our knapsack).
- The goal is to find the subset of items of maximum total value such that sum of their sizes is at most S (they all fit into the knapsack).
  - Exponential time to try all possible subsets
  - O(n.S) using DP

Knapsack Problem

- 0-1 Knapsack:
  - *n* items (can be the same or different)
  - Have **only one** of each
  - Must **leave or take** (i.e. 0-1) each item (e.g. bars of gold)
  - DP works, greedy does not

- Fractional Knapsack:
  - *n* items (can be the same or different)
  - Can take **fractional part** of each item (e.g. gold dust)
  - Greedy works and DP algorithms work

http://www.radford.edu/~nokie/classes/360/greedy.html

Greedy Solution 1

- From the remaining objects, select the object with maximum value that fits into the knapsack
- *Does not guarantee an optimal solution*
- E.g., n=3, s=[100,10,10], v=[20,15,15], S=105

Greedy Solution 2

- Select the one with minimum size that fits into the knapsack
- *Also, does not guarantee optimal solution*
- E.g., n=2, s=[10,20], v=[5,100], c=25

Greedy Solution 3

- Select the one with the maximum value density v_i/s_i that fits into the knapsack
- E.g., n=3, s=[20,15,15], v=[40,25,25], c=30
- Greedy works…if fractional items possible!

DP for 0-1 Knapsack

```
// Recursive algorithm: either we use the last element or we don't.
Value(n,S)    // S = space left, n = # items still to choose from
{
   if (n == 0) return 0;
   if (arr[n][S] != unknown) return arr[n][S];  // <- added this
   if (s_n > S) result = Value(n-1,S);
   else result = max{v_n + Value(n-1, S-s_n), Value(n-1, S)};
   arr[n][S] = result;                          // <- and this
   return result;
}
```

Reading

- Online resources on algorithm types

- https://www.cs.cmu.edu/~avrim/451f09/lectures/lect1001.pdf