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)

Dynamic Programming

- Consider each element in the list one at a time
  - For which M values will this new item give a solution?
  - Store a record of those solutions in an array indexed from 1 to M
  - Each time we get a solution for a value of M, put it in the array and use it for future solutions
- For example:
  - Let's say our desired M value for the problem is 50
  - Let's say the first two items in our list are 15 and 30
    - > Item 1 (15) could solve the problem by itself if M happened to be 15
    - > Item 2 (30) could solve the problem by itself if M happened to be 30
    - > Both together could solve the problem if M happened to be 45

- We continue in this way with each new element in our list
  - Which value of M could this solve directly (M = to its value)
  - Which other values of M could this solve if we add it to other items that have been considered
  - For each new answer we find we fill in the value in the array
  - Once the value for our goal M has been filled in, we have a solution
- Let's see how this can actually be done in the next slide

Subset Sum Using Dynamic Programming

store

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
					4		6			5			6	1		5		6		3		6		6

26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
5	4		6	2		5		6	6	3		5		6	5	4		6	2	6	5		6	6

j j j j j j

size

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

there is a solution size[j] positions before this, so adding item j will make a new solution

- Assume our store array is initially all 0s
- Loop j from 1 to N
  - Loop i from 1 to M
    - if (i == size[j]) and store[i] is 0, set store[i] = j
    - if store[i] is 0 and (store[i – size[j]] > 0) and (store[i – size[j]] != j)
      set store[i] = j

not yet assigned

previous solution is not j (since we can use j only one time in a solution)

Subset Sum Using Dynamic Programming

- Once store[M] is filled in we can extract the individual components in the answer
- In the example shown

  store[50] = 6 → size[6] = 8

  store[50-size[6]] = store[42] = 4 → size[4] = 6

  store[42-size[4]] = store[36] = 3 → size[3] = 21

  store[36-size[3]] = store[15] = 1 → size[1] = 15

  store[15-size[1]] = store[0] → sentinel → done
- Look at the code in subset.java
  - Run-time is clearly Theta(MN) → WHY?
  - But note that we are using a lot of space here
- This solution can be very good (pseudo polynomial) but it can also be very poor
- Let's consider some scenarios...

Subset Sum

N = 20, M = 1000

10 20 30 5 15 25 8 10 16 22 24 26 2 4 6 1 3 7 9 1000

- This instance shows the worst case behavior of the branch and bound solution
  - All combinations of the elements (~2^{19}) before last must be tried but to no avail until the last element is tried by itself
  - It is poor because we don't exceed the "bound" until we add the last item so the technique doesn't help us eliminate execution paths
  - The dynamic programming solution in this case is much better

    $MN == (1000)(20) == 20000 \ll 2^{19} == 524288$

Subset Sum

N = 4, M = 1111111111

1234567890 1357924680 1470369258 1111111111

- This instance shows the worst case behavior of the dynamic programming solution
  - Recall that our array must be size M, or 1111111111
    - First, we are using an incredible amount of memory
    - We must also process all answers from 1 up to M = 1111111111, so this is a LOT of work

    MN == (1111111111)(4) == 4444444444
  - Since N = 4, the branch and bound version will actually work in very few steps and will be much faster – worst case 2^4 == 16
  - This is why we say dynamic programming solutions are "pseudo-polynomial" and not actually polynomial

- Consider another problem:
  - Given N types of items, each with size S_i and value V_i
  - Given a KnapSack of capacity M
  - What is the maximum value of items that can fit into the Knapsack?
    - Note the number of each item available is indefinite
      - Alternate versions can restrict the number of each item
- Idea: We want to cram stuff into our knapsack so that it has the highest value
  - But different items are bigger / smaller so we have to choose our items carefully

- Simple Case:
  - If all items have the same size
  - Easy solution!
    - Take only the most valuable item

      Ex: US paper currency
- Ok, but can't we just figure out the optimal value/size ratio and use only that item?
  - Yes if we can take a "fraction" of an item, but no otherwise

KnapSack Problem

- Consider the following example with M = 9

Item	Size	Value	Value/Size
1	3	7	7/3
2	6	16	8/3
3	7	19	19/7
4	5	15	**3**

- Item 4 has the highest value/size ratio
- It we take 1 4/5 of item 4, our KnapSack will be full and have the optimal value of 27
  - But we cannot take a "fraction" of an item
- In this case the actual solution is 1 of item 1 and 1 of item 2 for a value of 23
  - Item 4 is not chosen at all!

- So how to solve KnapSack?
  - One way is Recursive Exhaustive Search using Branch and Bound
    - Add as many of item 1 as we can without overfilling (passing the bound)
    - Then backtrack, substituting item 2 (if it will fit) rather than the last item 1
    - Continue substituting until all combinations of items have been tried
    - This is similar to Subset Sum but more complex because we are considering two different variables:
      - Sum of sizes (must stay within bound M)
      - Sum of values (try to maximize)

- How about using dynamic programming?
  - Approach is similar to that of subset sum
    - We first consider solving the problem with ONLY the first item type (allowing mult. occurrences)
    - For this type we want to solve KnapSack for ALL M values from 1 up to our actual desired value
    - We then consider the second item type and repeat the process – substitute second item when it will give a bigger value
  - Fundamental differences from Subset Sum
    - We are maximizing a value
      - We need an extra array
    - We can consider items more than one time

KnapSack Problem

```
for (int j = 1; j <= N; j++)   // Try each item, one at a time
{
     // Given the current item, solve the knapsack problem for each
     // i from 1 up to the stated value for M.  Note that, unlike the
     // Subset Sum problem, the sizes of the items do not have to equal
     // M -- instead they must be <= M.
     for (int i = 1; i <= M; i++)    // Try each capacity i up to M
     {
          if (i >= size[j])    // Will item j fit into knapsack of size i?
          {
          // If current solution for knapsack of size i is less than
          // solution would be by adding item j, then add item j. Unlike
          // subset sum, j can be added multiple times
                    if (curstore[i] < curstore[i - size[j]] + val[j])
                    {
                              curstore[i] = curstore[i - size[j]] + val[j];
                              maxstore[i] = j;
                    }
          }
     }
}
```

- *Note the nested for loops*
- *Same basic structure as subset sum*
- *Same runtime of Theta(NM)*
- *See Knap.java for the rest of the code*

Row j indicates state of the curstore array after item j has been considered

Knapsack Problem

- Ex: consider N = 5, M = 15

Index:	1	2	3	4	5
Size:	3	4	7	8	9
Value:	4	5	10	11	13

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1			4	4	4	8	8	8	12	12	12	16	16	16	20
2				5	5		9	10		13	14		17	18	
3							10			14	15		18	20	
4								11							21
5									13			17			

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

maxstore array

- Consider one more problem:
  - Given a string S of length n
  - Given a string T of length m
  - We want to find the minimum Levenshtein Distance (LD) between the two, or the minimum number of character changes to convert one to the other
    - Consider changes to be one of the following:
      - **Change** a character in a string to a different char
      - **Delete** a character from one string
      - **Insert** a character into one string

- For example:

  LD("WEASEL", "SEASHELL") = 3

  - Why? Consider "WEASEL":
    - Change the W in position 1 to an S -> SEASEL
    - Add an H in position 5 -> SEASHEL
    - Add an L in position 8 -> SEASHELL
  - Result is SEASHELL
    - We could also do the changes from the point of view of SEASHELL if we prefer – try as an exercise
- How can we determine this distance?
  - We can define it in a recursive way initially
  - Later we will use dynamic programming to improve the run-time

- Generally speaking:
  - We want to calculate D[n, m] where n is the length of S and m is the length of T
    - From this point of view we want to determine the distance from S to T
      - If we reverse the arguments, we get the (same) distance from T to S (but the edits may be different)

      If n = 0 // base case S is empty

      return m (m appends will create T from S)

      else if m = 0 // base case T is empty

      return n (n deletes will create T from S)

      else

      Consider character n of S and character m of T
      - Now we have some possibilities

- If characters **match**
  - return D[n-1, m-1]
    - > Result is the same as the strings with the last character removed (since it matches)
  - Recursively solve the same problem with both strings one character smaller
- If characters **do not match** -- more poss. here
  - We could have a **mismatch** at that char**:**
    - return D[n-1, m-1] + 1
    - Example:
      - > `S = ------------X`
      - > `T = ------------Y`
    - Change X to Y, then recursively solve the same problem but with both strings one character smaller

- S could have an **extra** character
  - return D[n-1, m] + 1
  - Example:
    - S = ----------X**Y**
    - T = -----------**X**
  - Delete Y, then recursively solve the same problem, with S one char smaller but with T the same size
- S could be **missing** a character there
  - return D[n, m-1] + 1
  - Example:
    - S = -----------**Y**
    - T = ----------Y**X**
  - Add X onto S, then recursively solve the same problem with S the original size and T one char smaller

Edit Distance

- Unfortunately, we don't know which of these is correct until we try them all!
- So to solve this problem recursively we must try them all and choose the one that gives the minimum result
  - This yields 3 recursive calls for each original call (in which a mismatch occurs) and thus can give a worst case run-time of Theta(3^n)
- How can we do this more efficiently?
  - Let's build a table of all possible values for n and m using a two-dimensional array
  - Basically we are calculating the same values but from the bottom up rather than from the top down
- See pseudocode from handout
  - http://en.wikipedia.org/wiki/Levenshtein_distance

- Idea: As we proceed column by column (or row by row) we are finding the edit distance for prefixes of the strings
  - We use these to find out the possibilities for the successive prefixes
  - For each new cell D[i, j] we are taking the minimum of the cells
    - D[i-1, j] + 1
      - > Delete a char from S at this point to generate T
    - D[i, j-1] + 1
      - > Insert a char into S at this point to generate T
    - D[i-1, j-1] + cost[i,j]
      - > Where cost is 1 if characters don't match, 0 otherwise
      - > Change char at this point in S if necessary

D[i-1,j-1]

D[i-1,j]

D[i,j-1]

Edit Distance

		S	E	A	S	H	E	L	L
	0	1	2	3	4	5	6	7	8
W	1	1	2	3	4	5	6	7	8
E	2	2	1	2	3	4	5	6	7
A	3	3	2	1	2	3	4	5	6
S	4	3	3	2	1	2	3	4	5
E	5	4	3	3	2	2	2	3	4
L	6	5	4	4	3	3	3	2	3

Edit Distance

```
function LevenshteinDistance(char s[1..m], char t[1..n]):
// for all i and j, d[i,j] will hold the Levenshtein distance between
// the first i characters of s and the first j characters of t
declare int d[0..m+1, 0..n+1]   set each element in d to zero
// source prefixes can be transformed into empty string by
// dropping all characters
for i from 1 to m + 1:
     d[i, 0] := i
// target prefixes can be reached from empty source prefix
// by inserting every character
for j from 1 to n + 1:
     d[0, j] := j

for j from 1 to n + 1:
     for i from 1 to m + 1:
         if s[i] = t[j]:
            substitutionCost := 0
         else:
            substitutionCost := 1
         d[i, j] := minimum(d[i-1, j] + 1,     // deletion
                            d[i, j-1] + 1,     // insertion
                            d[i-1, j-1] + substitutionCost)  // subst
return d[m + 1, n + 1] // final answer is in bottom right location
```

Edit Distance

  - At the end the value in the **bottom right corner** is our final edit distance
  - See example in on web site
- Try one yourselves to see what is going on
  - Fill in the squares on the next slide
  - To see the solution see the slide after next
  - We are starting with ROTTEN
  - We want to generate PROTEIN
    - Note the initialization of the first row and column
      - Base cases
  - We will trace this during our synchronous lecture

Edit Distance

		P	R	O	T	E	I	N
	0	1	2	3	4	5	6	7
R	1							
O	2							
T	3							
T	4							
E	5							
N	6							

Edit Distance

		P	R	O	T	E	I	N
	0	1	2	3	4	5	6	7
R	1	1	1	2	3	4	5	6
O	2	2	2	1	2	3	4	5
T	3	3	3	2	1	2	3	4
T	4	4	4	3	2	2	3	4
E	5	5	5	4	3	2	3	4
N	6	6	6	5	4	3	3	***3***

Edit Distance

- Why is this cool?
  - Run-time is Theta(MN) worst case
    - As opposed to the 3^n of the recursive version
  - Unlike the pseudo-polynomial subset sum and knapsack solutions, this solution does not have any anomalous worst case scenarios
    - There is a price, which is the space required for the matrix
    - Optimized versions can reduce this from Theta(MN) space to Theta(M+N) space
- For more actual implementations, see http://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance

- Dynamic programming is also useful for a lot of other problems:
  - TSP (run-time Theta($n^2 2^n$) – exponential but better than Theta(n!) of the brute force algo)
  - Various other sequencing problems
    - Ex: DNA / RNA sequence alignment
    - Ex: String longest common subsequences
  - All Pairs Shortest Path (in a directed graph)
  - Optimizing word wrap in a document
  - See: http://en.wikipedia.org/wiki/Dynamic_programming