Dynamic Programming

CSE 2320 – Algorithms and Data Structures
University of Texas at Arlington

Alexandra Stefan
(Includes images, formulas and examples from CLRS and Dr. Bob Weems)

Last updated: 10/25/2018

Dynamic Programming (DP) - CLRS

- Dynamic programming (DP) applies when a problem has both of these properties:
  1. **Optimal substructure**: "optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently".
  2. **Overlapping subproblems**: "a recursive algorithm revisits the same problem repeatedly".
- Dynamic programming is typically used to:
  - Solve optimization problems that have the above properties.
  - Solve counting problems –e.g. Stair Climbing or Matrix Traversal.
  - Speed up existing recursive implementations of problems that have overlapping subproblems (property 2) – e.g. Fibonacci.
- Compare **dynamic programming** with **divide and conquer**.

Bottom-Up vs. Top Down

- There are two versions of dynamic programming.
  - Bottom-up.
  - Top-down (or memoization).

- Bottom-up:
  - Iterative, solves problems in sequence, from smaller to bigger.
- Top-down:
  - Recursive, start from the larger problem, solve smaller problems as needed.
  - For any problem that we solve, **store the solution**, so we never have to compute the same solution twice.
  - This approach is also called **memoization**.

Top-Down Dynamic Programming (Memoization)

- Maintain an array/table where solutions to problems can be saved.
- To solve a problem P:
  - See if the solution has already been stored in the array.
  - If yes, return the solution.
  - Else:
    - Issue recursive calls to solve whatever smaller problems we need to solve.
    - Using those solutions obtain the solution to problem P.
    - Store the solution in the solutions array.
    - Return the solution.

Bottom-Up Dynamic Programming

- Requirements for using dynamic programming:
  - The answer to our problem, P, can be easily obtained from answers to smaller problems.
  - We can order problems in a sequence (P_0, P_1, P_2, ..., P_K) of reasonable size, so that:
    - P_k is our original problem P.
    - The initial problems, P_0 and possibly P_1, P_2, ..., P_R up to some R, are easy to solve (they are **base cases**).
    - For i > R, each P_i can be easily solved using solutions to P_0, ..., P_{i-1}.
- If these requirements are met, we solve problem P as follows:
  - Create the sequence of problems P_0, P_1, P_2, ..., P_K, such that P_k = P.
  - For i = 0 to K, solve P_K.
  - Return solution for P_K.

Fibonacci Numbers

Fibonacci Numbers

- Generate Fibonacci numbers
  - 3 solutions: inefficient recursive, memoization (top-down dynamic programming (DP)), bottom-up DP.
  - Not an optimization problem but it has overlapping subproblems => DP eliminates recomputing the same problem over and over again.
- Weighted interval scheduling
- Matrix multiplication

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2:

  Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- How can we write a function that computes Fibonacci numbers?

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Consider this function: what is its running time?

Notice the mapping/correspondence of the mathematical expression and code.

```
int Fib(int i)
{
  if (i < 1) return 0;
  if (i == 1) return 1;
  return Fib(i-1) + Fib(i-2);
}
```

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Consider this function: what is its running time?
  - g(N) = g(N-1) + g(N-2) + constant

  $\Rightarrow$ g(N) $\geq$ Fibonacci(N) => g(N) = Ω(Fibonacci(N)) => g(N) = $\Omega(1.618^N)$

  Also g(N) $\leq$ 2g(N-1)+constant => g(N) $\leq c2^N$ => g(N) = $O(2^N)$

  => g(N) is exponential
  - We cannot compute Fibonacci(40) in a reasonable amount of time (with this implementation).
  - See how many times this function is executed.
  - Draw the tree

```
int Fib(int i)
{
  if (i < 1) return 0;
  if (i == 1) return 1;
  return Fib(i-1) + Fib(i-2);
}
```

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Alternative to inefficient recursion: compute from small to large and store data in an array.

Notice the mapping/correspondence of the mathematical expression and code.

linear version (Iterative, bottom-up):

```
int Fib_iter (int i)  {
   int F[i+1];
   F[0] = 0;    F[1] = 1;
   int j;
   for (j = 2; j <= i; j++) F[j] = F[j-1] + F[j-2];
   return F[i];
}
```

exponential version:

```
int Fib(int i)  {
   if (i < 1) return 0;
   if (i == 1) return 1;
   return Fib(i-1) + Fib(i-2);
}
```

Applied scenario

- F(N) = F(N-1)+F(N-2), F(0) = 0, F(1) = 1,
- Consider a webserver where clients can ask what the value of a certain Fibonacci number<, F(N) is, and the server answers it.

  How would you do that? (the back end, not the front end)
- Constraints:
  - Each loop iteration or function recursive call costs you 1cent.
  - Each loop iteration or function recursive call costs the client 0.001seconds wait
  - Memory is cheap
- How would you charge for the service?

- Think of some scenarios of requests that you could get. Think of it with focus on:
  - “good sequence of requests”
  - “bad sequence of requests”
  - Is it clear what good and bad refer to here?

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Alternative: remember values we have already computed.
- Draw the new recursion tree and discuss time complexity.

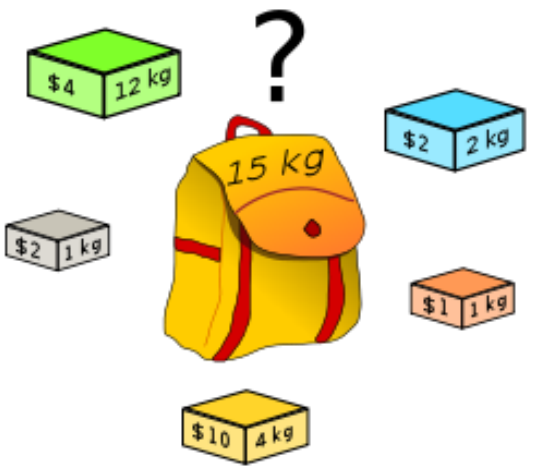
memoized version (helper function only):

```
int Fib_aux (int i, int[] sol)  {
  if (sol[i]!=-1) return sol[i];
  int res;
  if (i < 1)  res = 0;
  else
     if (i==1) res =1 ;
     else
        res = Fib_aux(i-1, sol) + Fib_aux(i-2, sol);
  sol[i] = res;
  return res;
}
```

exponential version:

```
int Fib(int i)  {
  if (i < 1) return 0;
  if (i == 1) return 1;
  return Fib(i-1) + Fib(i-2);
}
```

The Knapsack Problem



Problem:

- A thief breaks into a store.
- The maximum total weight that he can carry is *W*.
- There are *N* types of items at the store.
- Each type t_i has a value v_i and a weight w_i.
- What is the maximum total **value** that he can carry out?
- What items should he pick to obtain this maximum value?

Variations based on item availability:

- Unlimited amounts – *Unbounded* Knapsack
- Limited amounts – *Bounded* Knapsack
- Only one item – *0/1* Knapsack

- Items can be 'cut' – *Continuous* Knapsack (or *Fractional* Knapsack)

Image from Wikipedia:
https://en.wikipedia.org/wiki/Knapsack_problem

Variations of the Knapsack Problem

Unbounded:
Have unlimited number of each object.
Can pick any object, any number of times.
(Same as the stair climbing with gain.)

Fractional:
For each item can take the whole quantity, or a fraction of the quantity.

flour soda

Bounded:
Have a limited number of each object.
Can pick object i, at most x_i times.

0-1 (special case of Bounded):
Have only one of each object.
Can pick either pick object i, or not pick it.
This is on the web.

All versions have:	
N	number of different types of objects
W	the maximum capacity (kg)
$v_1, v_2, \ldots, v_N$	Value for each object. ($$)
$w_1, w_1, \ldots, w_N,$	Weight of each object. (kg)

The bounded version will have the amounts: $c_1, c_2, \ldots, c_N$ of each item.

Unbounded Knapsask

- This problem is the same as the stair climbing with gain.

Stair Climbing with Gain	Unbounded Knapsack
Jump	Object/item type
N – number of stairs	W – knapsack capacity
Jump size	Item weight
Gain (health points)	Value ($$ value)

Example (solve for max capacity, W=22):

Item type:	A	B	C	D	E
Weight (kg)	3	4	7	8	9
Value ($$)	4	6	11	13	15

Worksheet: Unbounded Knapsack

Item	Item value ($$)	Item weight (Kg)	Remaining capacity (Kg)	Sol[rem_capacity] ($$)	New total value ($$)
A	4	3			
B	6	4			
C	11	7			
D	13	8			
E	15	9			

Value_using_item_i = Sol[pb_size-weight_of_i] + value_of_i

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Sol																		
Picked																		
A, 3, 4																		
B, 4, 6																		
C, 7,11																		
D, 8,13																		
E, 9,15																		

See the Stair Climbing with Gain problem for method explanation.

Answers: Unbounded Knapsack

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Sol	0	0	0	4	6	6	8	11	13	15	15	17	19	21	22	24	26	28
Picked	-	-	-	A	B	B	A	C	D	E	A	A	A	B	C	C	C	D
A, 3, 4	-	-	-	0,4	1,4	2,4	3,8	4, 10	5, 10	6, 12	7, 15	8, 17	9, 19	10, 19	11, 21	12, 23	13, 25	14, 26
B, 4, 6	-	-	-	-	0, 6	1,6	2, 6	3, 10	4, 12	5, 12	6, 14	7, 17	8, 19	9, 21	10, 21	11, 23	12, 25	13, 27
C, 7,11	-	-	-	-	-	-	-	0, 11	1, 11	2, 11	3, 15	4, 17	5, 17	6, 19	7, 22	8, 24	9, 26	10, 26
D,8,13	-	-	-	-	-	-	-	-	0, 13	1, 13	2, 13	3, 17	4, 19	5, 19	6, 21	7, 24	8, 26	9, 28
E,9,15	-	-	-	-	-	-	-	-	-	0, 15	1, 15	2, 15	3, 19	4, 21	5, 21	6, 23	7, 26	8, 28

See the Stair Climbing with Gain problem for method explanation.

Unbounded Knapsack – recover the items

Find the items that give the optimal value. For example in the data below, what items will give me value 31 for a max weight of 22?

Note that the item values are different from those on the previous page. (They are from a different problem instance.)

Item type:	A	B	C	D	E
Weight (kg)	3	4	7	8	9
Value ($$)	4	5	10	11	13

ID of picked item

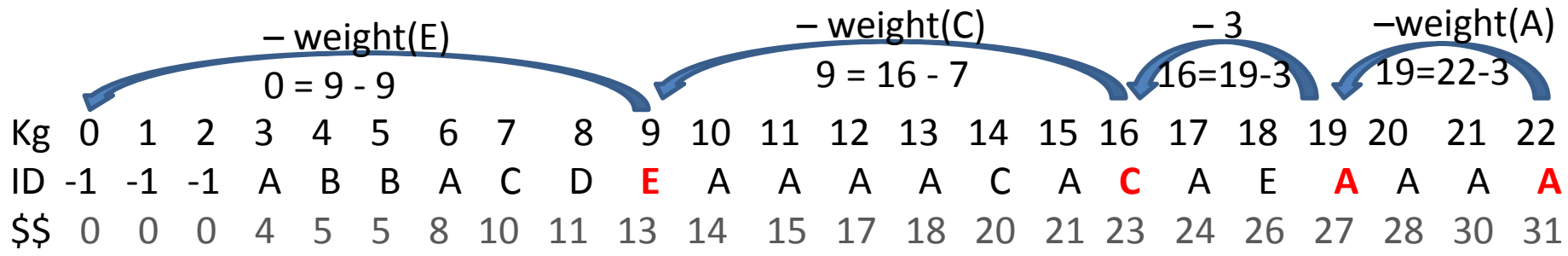
Kg	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
ID	-1	-1	-1	A	B	B	A	C	D	E	A	A	A	A	C	A	C	A	E	A	A	A	A
$$	0	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24	26	27	28	30	31

Unbounded Knapsack – recover the items

Find the items that give the optimal value. For example in the data below, what items will give me value 31 for a max weight of 22?

Note that the item values are different from those on the previous page. (They are from a different problem instance.)

Item type:	A	B	C	D	E
Weight (kg)	3	4	7	8	9
Value ($$)	4	5	10	11	13



Answer: E, C, A, A

Warning!

- Wrong information regarding solving the unbounded Knapsack: https://www.codeproject.com/Articles/706838/Bounded-Knapsack-Algorithm

  What is described there is the Greedy version, which will NOT give the optimal answer for all problems.

- Try to verify what you read on the web:
  - Use trusted sites such as Wikipedia,
  - See if you get the same answer from several sources.
  - Check blog posts

Recursive Solution

```
struct Items {
   int number;
   char ** types;
   int * weights;
   int * values;
};
```

```
int knapsack(int max_weight, struct Items items){
   if (max_weight == 0) return 0;
   int max_value = 0;
   int i;
   for (i = 0; i < items.number; i++)   {
      int rem = max_weight - items.weights[i];
      if (rem < 0) continue;
      int value = items.values[i] +
                              knapsack(rem, items);
      if (value > max_value) max_value = value;
   }
   return max_value;
}
```

solution to smaller problem

How Does This Work?

- We want to compute: knap(17).
- knap(17) can be computed from which values?

item type:	A	B	C	D	E
weight:	3	4	7	8	9
value	4	6	11	13	15

- val_A = ???
- val_B = ???
- val_C = ???
- val_D = ???
- val_E = ???

```
int knapsack(int max_weight, struct Items items){
   if (max_weight == 0) return 0;
   int max_value = 0;
   int i;
   for (i = 0; i < items.number; i++)   {
      int rem = max_weight - items.weights[i];
      if (rem < 0) continue;
      int value = items.values[i] +
                              knapsack(rem, items);
      if (value > max_value) max_value = value;
   }
   return max_value;
}
```

How Does This Work?

- We want to compute: knap(17).
- knap(17) will be the maximum of these five values:

item type:	A	B	C	D	E
weight:	3	4	7	8	9
value	4	6	11	13	15

- val_A = 4 + knap(14)
- val_B = 6 + knap(13)
- val_C = 11 + knap(10)
- val_D = 13 + knap(9)
- val_E = 15 + knap(8)

```
int knapsack(int max_weight, struct Items items){
   if (max_weight == 0) return 0;
   int max_value = 0;
   int i;
   for (i = 0; i < items.number; i++)   {
      int rem = max_weight - items.weights[i];
      if (rem < 0) continue;
      int value = items.values[i] +
                              knapsack(rem, items);
      if (value > max_value) max_value = value;
   }
   return max_value;
}
```

Recursive Solution for Knapsack

running time?

```
struct Items {
   int number;
   char ** types;
   int * weights;
   int * values;
};
```

```
int knapsack(int max_weight, struct Items items){
   if (max_weight == 0) return 0;
   int max_value = 0;
   int i;
   for (i = 0; i < items.number; i++)   {
      int rem = max_weight - items.weights[i];
      if (rem < 0) continue;
      int value = items.values[i] +
                              knapsack(rem, items);
      if (value > max_value) max_value = value;
   }
   return max_value;
}
```

Recursive Solution for Knapsack

running time?

very slow (exponential)

How can we make it faster?

```
struct Items {
   int number;
   char ** types;
   int * weights;
   int * values;
};
```

```
int knapsack(int max_weight, struct Items items){
   if (max_weight == 0) return 0;
   int max_value = 0;
   int i;
   for (i = 0; i < items.number; i++)    {
      int rem = max_weight - items.weights[i];
      if (rem < 0) continue;
      int value = items.values[i] +
                              knapsack(rem, items);
      if (value > max_value) max_value = value;
   }
   return max_value;
}
```

Draw the recursion tree

- Inefficient recursion
- Memoized version

Bottom-Up Solution pseudocode

```
int knapsack(int max_weight, Items items)
  Create array of solutions.
  Base case: solutions[0] = 0.
  For each weight = 1 to max_weight
     max_value = 0.
     For each item in items:
       • remainder = weight - weight of item.
       • if (remainder < 0) continue;
       • value = value of item + solutions[remainder].
       • if (value > max_value) max_value = value.
     solutions[weight] = max_value.
  Return solutions[max_weight].
```

Top-Down Solution

Top-level function (almost identical to the top-level function for Fibonacci top-down solution):

```
int knapsack(int max_weight, Items items)
   Create array of solutions.
   Initialize all values in solutions to "unknown".
   result = knap_helper(max_weight, items, solutions)
   If needed, free up the array of solutions.
   Return result.
```

Top-Down Solution: Helper Function

```
int knap_helper(int weight, Items items, int * solutions)
   // Check if this problem has already been solved.
   if (solutions[weight] != "unknown") return solutions[weight].
   If (weight == 0) result = 0.      // Base case
   Else:
      result = 0. // same as max_value in iterative solution.
      For each item in items:
          remainder = weight - weight of item.
          if (remainder < 0) continue;
          value = value of item + knap_helper(remainder,items,solutions)
          If (value > result) result = value.
   solutions[weight] = result.          // Memoization
   return result.
```

Time complexity

- Note that the recursive inefficient version is a brute force solution (it generates all sets with total weight less than the knapsack capacity)
  - B.c. it recalculates every smaller problem
  - Draw the recursion tree.
- Discuss the time complexity of all 3 methods.
  - Let *W* = maxWeight, *n* = number of different items
  - Bottom-up (iterative): $O(W * n)$
    - Imagine case with an item of weight 1.
    - Pseudo-polynomial – polynomial in the *value* of one of the inputs: W.
  - Recursive with memoization: $O(W*n)$
    - Worst case tree height is W, every problem size is at most solved once (as an internal node) every internal node will have n branches.
  - Inefficient recursion: $O(n^W)$ (assume all items have weight 1)

Performance Comparison

- Recursive version: (knapsack_recursive.c)
  - Runs reasonably fast for max_weight <= 60.
  - Starts getting noticeably slower after that.
  - For max_weight = 75 I gave up waiting after 3 minutes.
- Bottom-up version: (knapsack_bottom_up.c)
  - Tried up to max_weight = 100 million.
  - No problems, very fast.
  - Took 4 seconds for max_weight = 100 million.
- Top-down version: (knapsack_top_down.c)
  - Very fast, but crashes around max_weight = 57,000.
  - The system cannot handle that many recursive function calls.

Worksheet: 0/1 Knapsack

optimal solution (for **this** problem size), excluding item i

optimal solution (for a **smaller** problem size), excluding item i

Value_using_**first_i_items:** Sol**[i]** [pb_size] = **max**{Sol**[i-1]** **[** pb_size - weight_of_i**]** + value_of_i, Sol**[i-1]** [pb_size

index	0	1	2	3	4	5	6	7	8	9	10	11
No item												
A, 3, 4												
B, 4, 6												
C, 7,11												
D, 8,13												
E, 9,15												

Worksheet: 0/1 Knapsack

optimal solution (for **this** problem size), excluding item i

optimal solution (for a **smaller** problem size), excluding item i

Value_using_**first_i_items:** Sol**[i]** [pb_size] = **max**{Sol**[i-1]** **[** pb_size - weight_of_i**]** + value_of_i, Sol**[i-1]** [pb_size

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
No item																		
A, 3, 4																		
B, 4, 6																		
C, 7,11																		
D, 8,13																		
E, 9,15																		

Answer: 0/1 Knapsack

optimal solution (for a **smaller** problem size), excluding item i

optimal solution (for **this** problem size), excluding item i

Value_using_**first_i_items:**
Sol**[i]** [pb_size] = **max**{Sol**[i-1]** **[** pb_size - weight_of_i**]** + value_of_i, Sol**[i-1]** [pb_size]

E.g.:
Value_using_**first_3_items(A,B,C):** Sol**[3]** [14] = **max**{Sol**[2]** **[14 - 7]** +11, Sol**[2]** [7] = max{10+11, 10} = 21

	index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	No item	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	A, 3, 4	0	0	0	4*	4*	4*	4*	4*	4*	4*	4*	4*	4*	4*	4*	4*	4*
2	B, 4, 6	0	0	0	4	6*	6*	6*	10*	10*	10*	10*	10*	10*	10*	10*	10*	10*
3	C, 7,11	0	0	0	4	6	6	6	11*	11*	11*	15*	17*	17*	17*	21*	21*	21*
4	D, 8,13	0	0	0	4	6	6	6	11	13*	13*	15	17*	19*	19*	21	24*	24*
5	E, 9,15	0	0	0	4	6	6	6	11	13	15*	15*	17	19*	21*	21*	24	26*

+11

Unbounded vs 0/1 Knapsack Solutions

- Unbounded (unlimited number of items)
  - Need only one (or two) 1D arrays: sol (and picked).
  - The other rows (one per item) are added to show the work that we do in order to figure out the answers that go in the table. There is NO NEED to store it.
  - Similar problem: Minimum Number of Coins for Change (solves a minimization, not a maximization problem): https://www.youtube.com/watch?v=Y0ZqKpToTic
- 0/1 (most web resources show this problem)
  - MUST HAVE one or two 2D tables, of size: (items+1) x (max_weight+1).
  - Each row (corresponding to an item) gives the solution to the problem using items from the beginning up to and including that row.
  - Whenever you look back to see the answer for a precomputed problem you look precisely on the row above because that gives a solution with the items in the rows above (excluding this item).
    - Unbounded knapsack can repeat the item => no need for sol excluding the current item => 1D

Hint for DP problems

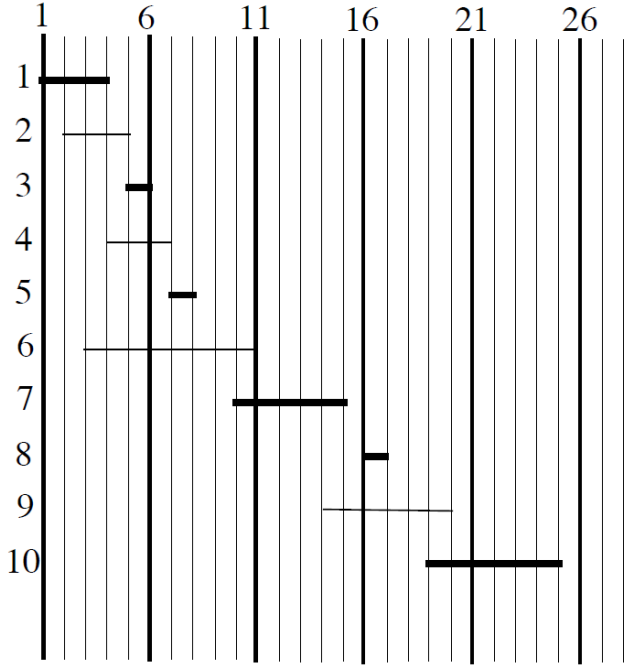
- For a DP problem you can typically write a MATH function that gives the solution for problem of size N in terms of smaller problems.
- It is straightforward to go from this math function to code:
  - Iterative: The math function ‘maps’ to the sol array
  - Recursive: The math function ‘maps’ to recursive calls
- Typically the math function will be a
  - Min/max (over itself applied to smaller N)
  - Sum (over itself applied to smaller N)

Weighted Interval Scheduling

(Job Scheduling)

Weighted Interval Scheduling (a.k.a. Job Scheduling)

- Problem:
  - Given n jobs of values $v_1,\ldots, v_n$, select a subset that do not overlap and give the largest total value.
- Preprocessing:
  - Sort jobs in increasing order of their finish time. –already done here
  - For each job ,*i*, compute the last job prior to *i*, p_i, that does not overlap with *i*.



	v_i	p_i	$m(i)$
1	5	0	
2	4	0	
3	3	2	
4	2	1	
5	1	4	
6	5	0	
7	4	5	
8	3	7	
9	2	6	
10	1	8	

(Recursive solution view: the last step is to pick or not pick job 10.)
Steps: one step for each job.
Option: pick it or not

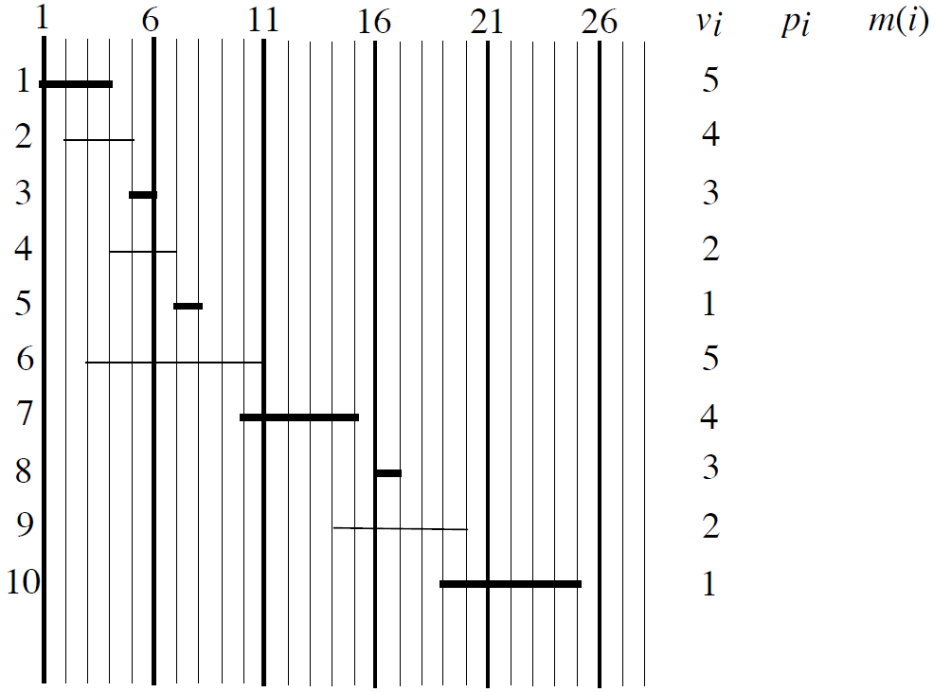
Cost:
m(i) – optimal solution value for jobs *1,2,…,i*
$m(i) = max\{v_i + m(p_i), m(i-1)\}$
$m(0) = 0$

Time complexity: O(n)
- Fill out *m(i)* with constant operations

Reference: Dr. Bob Weems, notes 07.

Job Scheduling - Work sheet

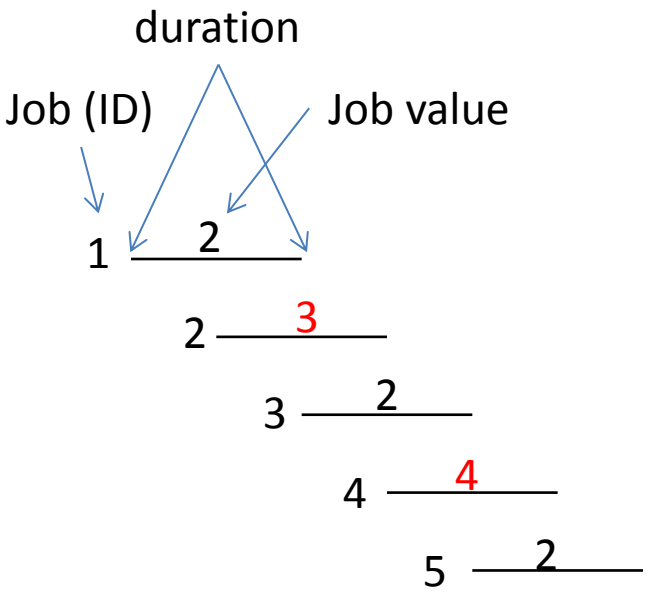
- Problem:
  - Given n jobs of values $v_1,\ldots, v_n$, select a subset that do not overlap and give the largest total value.
- Preprocessing:
  - Sort jobs in increasing order of their finish time. –already done here
  - For each job ,*i*, compute the last job prior to *i*, p_i, that does not overlap with *i*.



To do:
- Fill-out the p_i data.
- Figure out the DP solution (write the optimization function, m(i)).

Recovering the Solution

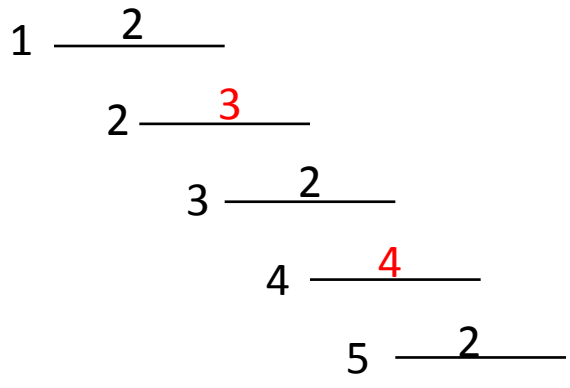
- Notations conventions:
  - Jobs are already sorted by end time
  - Horizontal alignment is based on time: jobs 1 and 2 overlap, jobs 1 and 3 do not overlap (3 starts immediately after 1 finishes).



Time complexity: O(n)

Recovering the Solution

- Example showing that when computing the optimal gain, we cannot decide which jobs will be part of the solution and which will not. We can only recover the jobs picked AFTER we computed the optimum gain and by going from end to start.

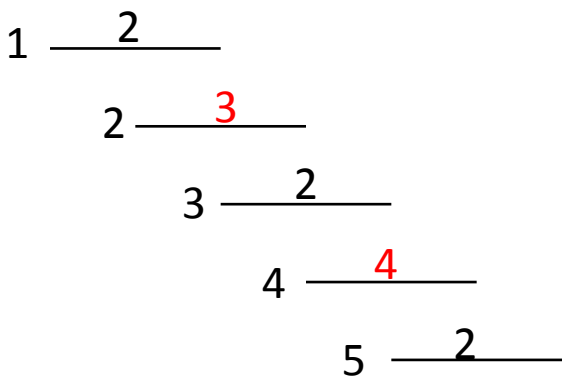


i	v_i	p_i	m(i)	m(i) used i	In optimal solution
0	0	0	0	-	-
1	2	0	2	Yes	-
2	3	0	3	Yes	Yes
3	2	1	4	Yes	-
4	4	2	7	Yes	Yes
5	2	3	7	No	-

Time complexity: O(n)

Job Scheduling – Brute Force Solution

- For each job we have the option to include it (0) or not(1). Gives:
  - The power set for a set of 5 elements, or
  - All possible permutations with repetitions over n positions with values 0 or 1=> $O(2^n)$
  - Note: exclude sets with overlapping jobs.
- Time complexity: $O(2^n)$



1	2	3	4	5	Valid	Total value
0	0	0	0	0	yes	0
0	0	0	0	1	yes	2
0	0	0	1	0	yes	4
0	0	0	1	1	no	
0	0	1	0	0	yes	2
0	0	1	0	1	yes	4 (=2+2)
0	0	1	1	1	no	
…	…	…	…	…	…	…
1	1	1	1	1	no	

Bottom-up (BEST)

```
// Bottom-up (the most efficient solution)
int js_iter(int* v, int*p, int n){
    int j, with_j, without_j;
    int m[n+1];  // where allowed. Else use malloc
    // optionally, may initialize it to -1 for safety
    m[0] = 0;
    for(j = 1; j <= n; j++){
        with_j = v[j] + m[p[j]];
        without_j = m[j-1];
        if ( with_j >= without_j)
            m[j] = with_j;
        else
            m[j] = without_j;
    return m[n];
}
```

Recursive (inefficient)

```
// Inefficient recursive solution:
int jsr(int* v, int*p, int n){
    if (n == 0) return 0;
    int res;
    int with_n = v[n] + jsr(v,p,p[n]);
    int without_n = jsr(v,p,n-1);
    if ( with_n >= without_n)
       res = with_n;
    else
       res = without_n;
    return res;
}
```

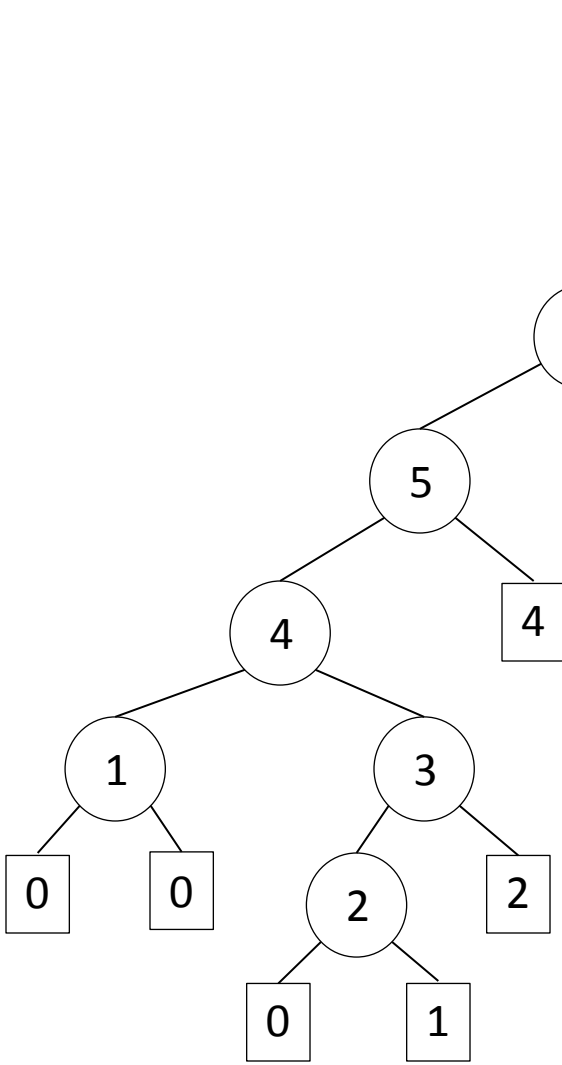
Memoization (Efficient Recursion) recursive function

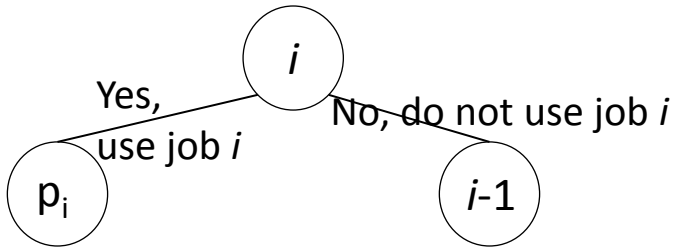
```
// Memoization efficient recursive solution:
int jsr(int* v, int*p, int n, int* m){
    if (m[n] != -1)  // already computed.
       return m[n];   // Used when rec call for a smaller problem.
    int res;
    int with_n = v[n] + jsr(v,p,p[n]);
    int without_n = jsr(v,p,n-1);
    if ( with_n >= without_n)
       res = with_n;
    else
       res = without_n;
    m[n] = res;
    return res;
}
```

Memoization – wrapper versions

```
// Memoization efficient recursive solution:
int jsr_out(int* v, int*p, int n){
    int* m = malloc( (n+1) * sizeof(int));
    int j, res;
    m[0] = 0;
    for (j = 1; j<=n; j++)    m[j] = -1;  //unknown, not computed
    jsr(v,p,n,m);
    res = m[n];
    free(m);
    return res;
}
// or:
int jsr_out(int* v, int*p, int n){
    int m[n+1];  // where the compiler allows it
    int j;
    m[0] = 0;
    for (j = 1; j<= n; j++)   m[j] = -1;  //unknown, not computed
    jsr(v,p,n,m);
    return m[n];
}
```

Function call tree for the memoized version





Round nodes – internal nodes. Require recursive calls.
Square nodes – leaves, show calls that return without any new recursive calls.

To estimate the number of method calls note that every problem size is an internal node only once and that every node has exactly 0 or 2 children. A property of such trees states that the number of leaves is one more than the number of internal nodes => there are at most (1+2N) calls. Here: N = 10 jobs to schedule.

Longest Common Subsequence (LCS)

Longest Common Subsequence (LCS)

- Given 2 sequences, find the longest common subsequence (LCS)
- Example:
  - A B C B D A B
  - B D C A B A
- Examples of subsequences of the above sequences:
  - BCBA (length 4)
  - BDAB
  - CBA (length 3)
  - CAB
  - BB (length 2)

Show the components of the solution. Can you show a solution similar that of an Edit distance problem?

LCS
Smaller Problems

- Original problem:

  A B C B D A B

  B D C A B A

- Smaller problems:

- Smaller problems that can be base cases:

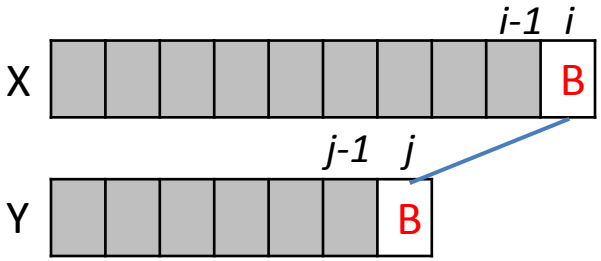
Original problem (LCS length)	A B C B D A B B D C A B A (4)				
Smaller problems (LCS length)	"ABCB" "BD" (1)	"AB" "DC" (0)			
Smaller problems that can be base cases (LCS length)	"" "" (0)	"" "B" (0)	"" "BDCABA" (0)	"A" "" (0)	"ACBDAB" "" (0)

Dependence on Subproblems (recursive case)

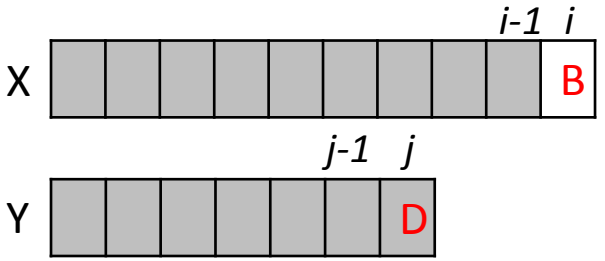
c(i,j) – depends on *c(i-1,j-1), c(i-1, j), c(i,j-1)*

(grayed areas show solved subproblems)

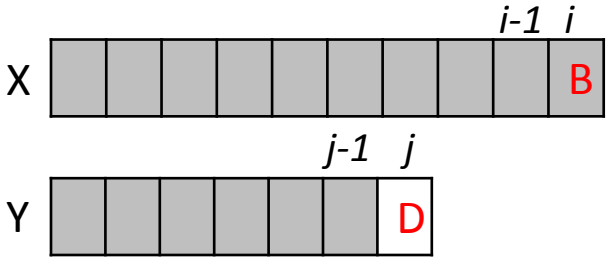
$$c(i,j) = \begin{cases} c(i-1, j-1)+1, & x_i = y_j,\ i, j > 0,\ or \\ \max\{c(i-1, j), c(i, j-1)\}, & x_i \neq y_j,\ i, j > 0 \end{cases}$$



⇒ *c(i-1,j-1) + 1, if* $x_i = y_j$
This case makes the solution grow
(finds an element of the subsequence)



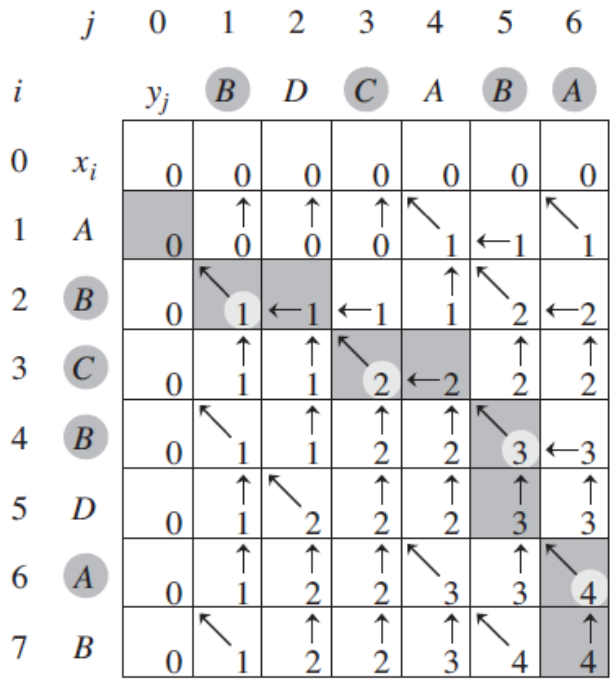
⇒ *c(i-1,j)*
x_i is ignored



⇒ *c(i,j-1)*
y_j is ignored

Here indexes start from 1

Longest Common Subsequence



i		j	0	1	2	3	4	5	6
				y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

CLRS – table and formula

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

Iterative solution

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

CLRS – pseudocode

Recover the subsequence

CLRS pseudcode

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Longest Increasing Subsequence (LIS)

Longest Increasing Subsequence

Given an array of values, find the longest increasing subsequence.
Example: A = { 3,5,3,9,3,4,3,1,2,1,4}

Variations:

Repetitions NOT allowed: strictly increasing subsequence. E.g.: 3,5,9

Repetitions allowed: increasing subsequence. E.g.: 3,3,3,4,4

Simple solution: reduce to LCS problem.
For a more efficient solution tailored for the LIS problem see Dr. Weems notes.

	No repetitions	With repetitions
Solution for given instance:	1,2,4 (also: or 3,5,9)	3,3,3,3,4
Solution concept: reduce to LCS	X = {min(A), min(A)+1, ….,max(A)} LIS(A) = LCS (A,X)	X = sorted(A) LIS(A) = LCS(A, sorted(A))
Time complexity	$\Theta(n+v+nv) = \Theta(nv)$ depends on min and max values from A: v = max(A)-min(A)+1	Sort A : $O(n^2)$ LCS(A, sorted(A)) : $\Theta(n^2)$ => $\Theta(n^2)$
Space complexity	$\Theta(nv)$	$\Theta(n^2)$

LIS to LCS reduction

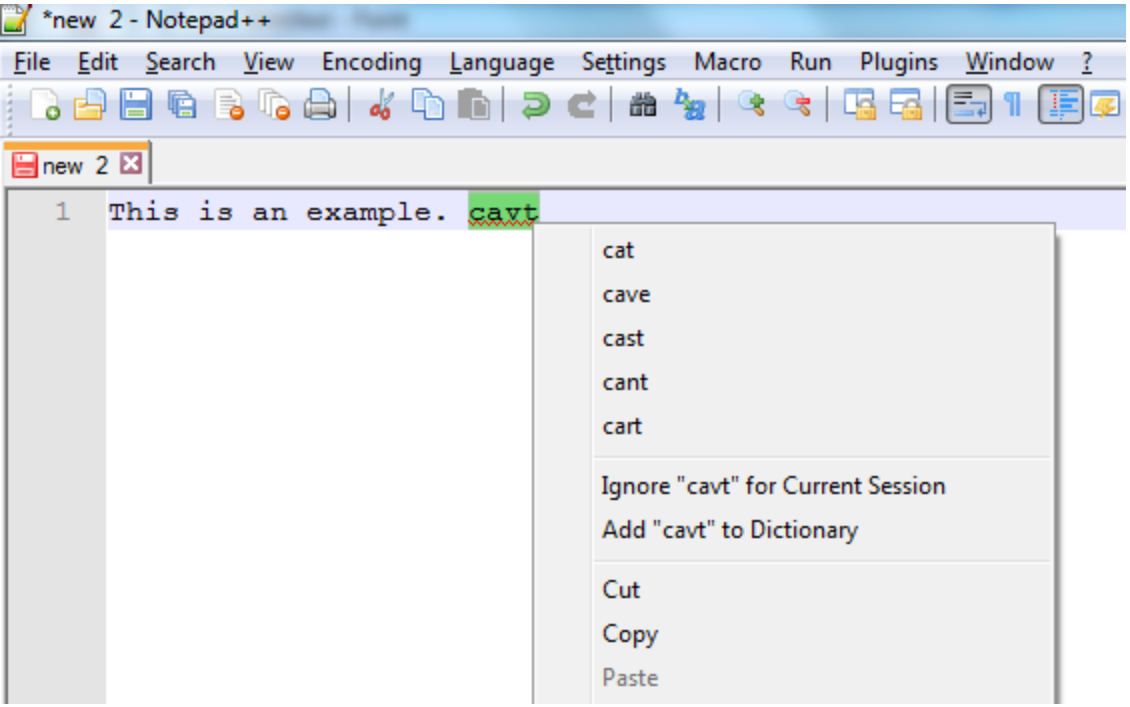
- A = { 3,5,3,9,3,4,3,1,2,1,4}
- LIS with NO repetitions:
  - Produce: X = {1,2,3,4,5,6,7,8,9}
  - LIS(A) = LCS(A,X)

  - If v>>n , where v = |X| = max(A)-min(A)+1), use X = {1,2,3,4,5,9} (unique elements of A sorted in increasing order)

- LIS WITH repetitions:
  - Produce X = {1,1,2,3,3,3,3,4,4,5,9}
  - LIS(A) = LCS(A,X)

The Edit Distance

Application: Spell-checker



Spell checker

- Computes the "edit distance" between the words: the smaller distance, the more similar the words.

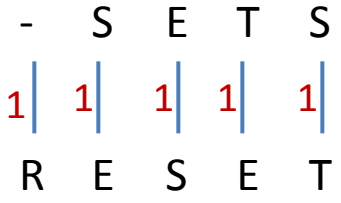
Edit distance

- Minimum cost of all possible alignments between two words.

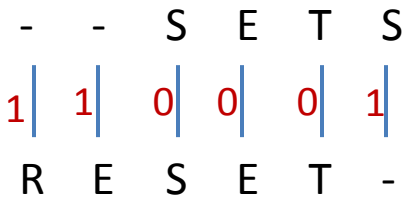
- Other: search by label/title (e.g. find documents/videos with cats)
- This is a specific case of a more general problem: time series alignment.
- Another related problem is: Subsequence Search.

Alignments

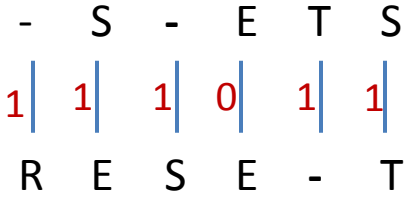
Examples of different alignments for the same words



Cost/distance: 5



Cost/distance: 3



Cost/distance: 5

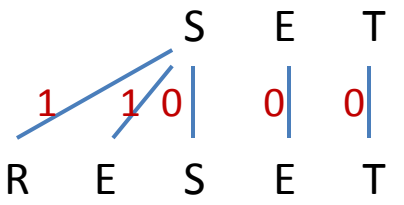
- No cross-overs: The letters must be in the order in which they appear in the string.

Incorrect alignment

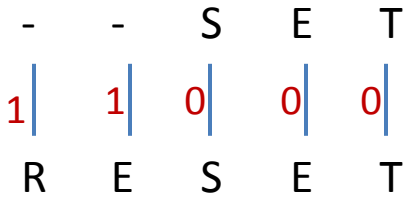
Pair cost:
Same letters: 0
Different letters: 1
Letter-to-dash: 1
Alignment cost: sum of costs of all pairs in the alignment.
Edit distance: minimum alignment cost over all possible alignments.

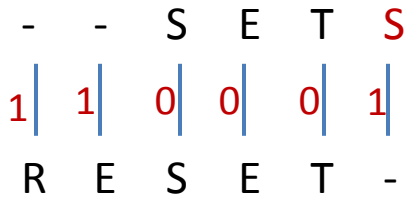
The Edit Distance

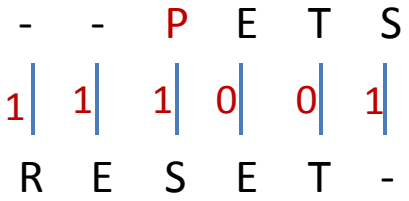
- Edit distance – the cost of the best alignment
  - Minimum cost of all possible alignments between two words.
  - (The smaller distance, the more similar the words)



Edit distance: minimum alignment cost over all possible alignments.
Alignment cost: sum of costs of all pairs in the alignment.
Pair cost:
Same letters: 0
Different letters: 1
Letter to dash: 1







Examples

Notations, Subproblems

- Notation:
  - $X = x_1, x_2, \ldots, x_n$
  - $Y = y_1, y_2, \ldots, y_m$
  - Dist(i,j) = the smallest cost of all possible alignments between substrings $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots, y_j$.
  - Dist(i,j) will be recorded in a matrix at cell [i,j].
- Subproblems of ("SETS", "RESET"):
  - Problem size can change by changing either X or Y (from two places):
  -
  -
  -
  -
- What is Dist for all of the above problems?

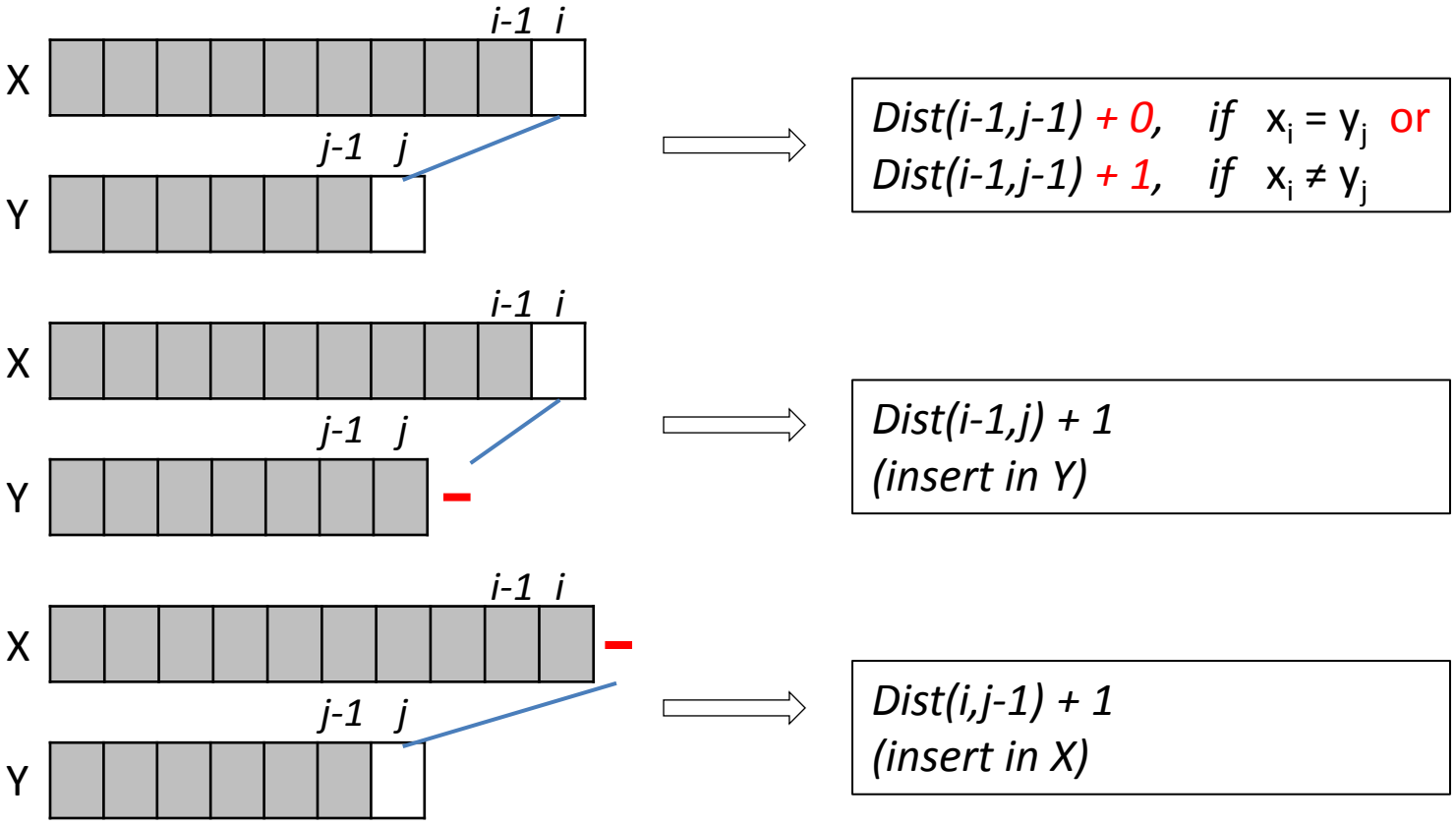
Notations, Subproblems

- Notation:
  - $X = x_1, x_2, \ldots, x_n$
  - $Y = y_1, y_2, \ldots, y_m$
  - Dist(i,j) = the smallest cost of all possible alignments between substrings $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots, y_j$.
  - Dist(i,j) will be recorded in a matrix at cell [i,j].
- Subproblems of ("SETS", "RESET"):
  - Problem size can change by changing either X or Y (from two places):
  - ("S", "RES")
  - ("", "R"), ("", "RE"), ("", "RES"), …, ("", "RESET")
  - ("S", ""), ("SE", ""), ("SET", ""), ("SETS", "")
  - ("" , "")
- What is Dist for all of the above problems?

Dependence on Subproblems

- *Dist(i,j)* – depends on *Dist(i-1,j-1), Dist (i-1, j), Dist(i,j-1)*

(below, grayed areas show the solved subproblems)

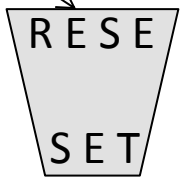


Edit Distance

Filling out the distance matrix

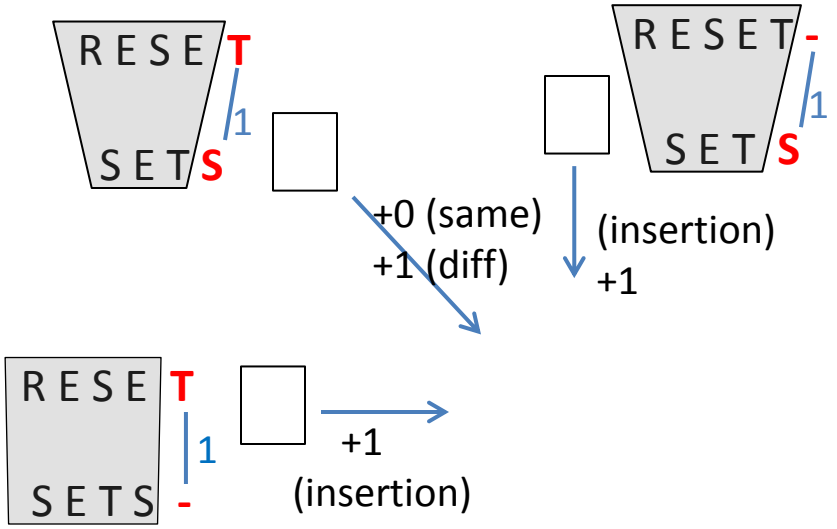
- Each cell will have the answer for a specific subproblem.
- Special cases:
  - Dist(0,0) =
  - Dist(0,j) =
  - Dist(i,0) =
  - Dist(i,j) =

Represents some alignment between "RESE" and "SET"



- Complexity (where: |X| = n, |Y| = m): Time: Space:

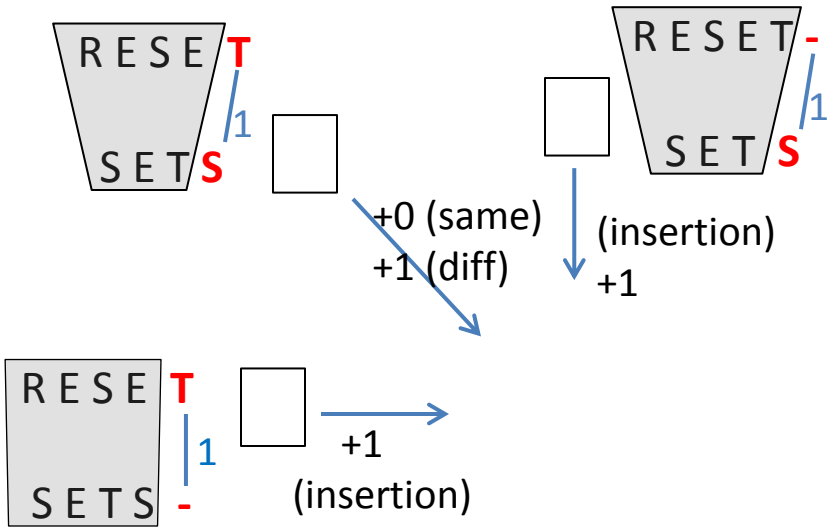
		0	1	2	3	4	5
		""	R	E	S	E	T
0	""						
1	S						
2	E						
3	T						
4	S						



Edit Distance

- Each cell will have the answer for a specific subproblem.
- Special cases:
  - Dist(0,0) = 0
  - Dist(0,j) = 1 + Dist(0,j-1)
  - Dist(i,0) = 1 + Dist(i-1,0)
  - Dist(i,j) = $\begin{cases} \min\{ \text{Dist}(i-1,j)+1,\ \text{Dist}(i,j-1)+1,\ \text{Dist}(i-1,j-1) \} & \text{if } x_i = y_j \text{ or} \\ \min\{ \text{Dist}(i-1,j)+1,\ \text{Dist}(i,j-1)+1,\ \text{Dist}(i-1,j-1)+1 \} & \text{if } x_i \neq y_j \end{cases}$
- Complexity (where: |X| = n, |Y| = m): Time: O(n*m) Space: O(n*m)

		0	1	2	3	4	5
	""	R	E	S	E	T
0	""	**0**	**1**	**2**	**3**	**4**	**5**
1	S	**1**	**1**	**2**	**2**	**3**	**4**
2	E	**2**	**2**	**1**	**2**	**2**	**3**
3	T	**3**	**3**	**2**	**2**	**3**	**2**
4	S	**4**	**4**	**3**	**2**	**3**	**3**



Edit Distance
Recover the alignment – Worksheet
(using the arrow information)

X = SETS
Y = RESET

Time complexity: O(……..)
(where: |X| = n, |Y| = m)

	Aligned Pair	Update
↖	x_i y_j	i = i-1 j = j-1
↑	x_i -	i = i-1
←	- y_j	j = j-1

i \ j		""	R	E	S	E	T
		↖ 0	← 1	← 2	← 3	← 4	← 5
0	""						
1	S	↑ 1	↖ 1	↖ 2	↖ 2	← 3	← 4
2	E	↑ 2	↖ 2	↖ 1	← 2	↖ 2	← 3
3	T	↑ 3	↖ 3	↑ 2	↖ 2	↖ 3	↖ 2
4	S	↑ 4	↖ 4	↑ 3	↖ 2	↖ 3	↑ 3

i							
j							
X							
Y							

Start at:
i = …….
j = …….

How big will the solution be (as num of pairs)?

Edit Distance

Recover the alignment

Here the pairs are filled in from the LEFT end to the RIGHT end and printed from RIGHT to LEFT.

X = SETS
Y = RESET

Time complexity: O(n+m)
(where: |X| = n, |Y| = m)

	j	0	1	2	3	4	5
i		""	R	E	S	E	T
0	""	↖ 0	← 1	← 2	← 3	← 4	← 5
1	S	↑ 1	↖ 1	↖ 2	↖ 2	← 3	← 4
2	E	↑ 2	↖ 2	↖ 1	← 2	↖ 2	← 3
3	T	↑ 3	↖ 3	↑ 2	↖ 2	↖ 3	↖ 2
4	S	↑ 4	↖ 4	↑ 3	↖ 2	↖ 3	↑ 3

	Aligned Pair	Update
↖	x_i y_j	i = i-1 j = j-1
↑	x_i -	i = i-1
←	- y_j	j = j-1

i	4	3	2	1	0	0	0
j	5	5	4	3	2	1	0
	↑	↖	↖	↖	←	←	
X	**S**	**T**	**E**	**S**	**-**	**-**	
Y	**-**	**T**	**E**	**S**	**E**	**R**	
	1	**0**	**0**	**0**	**1**	**1**	

Print from right to left.

Start at:
i = 4
j = 5

How big will the solution be (as num of pairs)?
n+m

Sum of costs of pairs in the alignment string is the same as table[4][5]: 1+0+0+0+1+1 = 3

- What is the best alignment between

abcdefghijk

cdXYZefgh

Edit Distance

Recover the alignment - Method 2:

(based only on distances)

Even if the choice was not recorded, we can backtrace based on the distances: see from what direction (cell) you could have gotten here.

		w	w	a	b	u	d	e	f
	0	1	2	3	4	5	6	7	8
a	1	1	2	2	3	4	5	6	7
b	2	2	2	3	2	3	4	5	6
c	3	3	3	3	3	3	4	5	6
d	4	4	4	4	4	4	3	4	5
e	5	5	5	5	5	5	4	3	4
f	6	6	6	6	6	6	5	4	3
y	7	7	7	7	7	7	6	5	4
y	8	8	8	8	8	8	7	6	5
y	9	9	9	9	9	9	8	7	6

first: abcdefyyy
second: wwabudef

edit distance:
Alignment:

Edit Distance

Recover the alignment - Method 2:

(based only on distances)

Even if the choice was not recorded, we can backtrace based on the distances: see from what direction (cell) you could have gotten here.

		w	w	a	b	u	d	e	f
	0	**1**	**2**	3	4	5	6	7	8
a	1	1	2	**2**	3	4	5	6	7
b	2	2	2	3	**2**	3	4	5	6
c	3	3	3	3	3	**3**	4	5	6
d	4	4	4	4	4	4	**3**	4	5
e	5	5	5	5	5	5	4	**3**	4
f	6	6	6	6	6	6	5	4	**3**
y	7	7	7	7	7	7	6	5	**4**
y	8	8	8	8	8	8	7	6	**5**
y	9	9	9	9	9	9	8	7	**6**

first: abcdefyyy
second: wwabudef

edit distance: 6
Alignment:
- - a b c d e f y y y
w w a b u d e f - - -
1 1 0 0 1 0 0 0 1 1 1

Sample Exam Problem

On the right is part of an edit distance table. CART is the complete second string. AL is the end of the first string (the first letters of this string are not shown).

a. (6 points) Fill-out the empty rows (finish the table).

b. (4 points) How many letters are missing from the first string (before AL)? Justify your answer.

c. (8 points) Using the table and the information from part b), for each of the letters **C and A** in the second string, CART, say if it could be one of the missing letters of the first string: **Yes** (it is one of the missing letters – 'proof'), **No** (it is not among the missing ones – 'proof'), **Maybe** (it may or may not be among the missing ones – give example of both cases).

   – C: Yes/No/Maybe. Justify:

   – A: Yes/No/Maybe. Justify:

		C	A	R	T
…	…	…	…	…	…
…	5	5	4	3	3
A					
L					

Edit Distance
Sliding Window

- Space complexity improvement:
  - Keep only two rows
  - Keep only one row

Motivation for DP

- Align time series:
  - Collected in a database temperatures at different hours over one day in various places (labelled with the name). Given a query consisting of temperatures collected in an unknown place, find the place with the most similar temperatures. Issues:
    - Not same number of measurements for every place and query.
    - Not at the exact same time. (E.g. maybe Mary recorded more temperatures throughout the day and none in the night, and John collected uniformly throughout the day and night.)
- Find shapes
- Find videos showing a similar sign

DP in an exam

- Must be able to solve these problems in the shown format and with the methods covered in class.
- Components of every DP problem:
  - Problem description and representation,
  - Cost function (math formulation and/or implementation), recover the solution, brute force method
    - time complexity for all three items above
  - Solve on paper:
    - Find optimum.
    - Recover choices that give optimum solution.

DP or not DP?

- I consider DP problems to be optimization problems.
  - CLRS gives only optimization problems as DP
- Other sources list non-optimization problems as DP as well.
- Main point: optimization problems are hard. Consider DP for them.
- DP requires two properties:
  - Optimal substructure – applicable to optimization problems
    - An optimal solution to the big problem gives optimal solutions to the small problems.
  - Overlapping subproblems – applicable to all problems (including Fibonacci, stair climbing, matrix traversal)
    - The pattern of the solution in both cases (e.g. matrix traversal with and without optimization) is the same except that in one case you add the answer from the subproblems and in the other you pick the max.
    - Results in an ordering of the subproblems, where each next depends on the previous ones.

Divide and Conquer vs DP vs Recursion

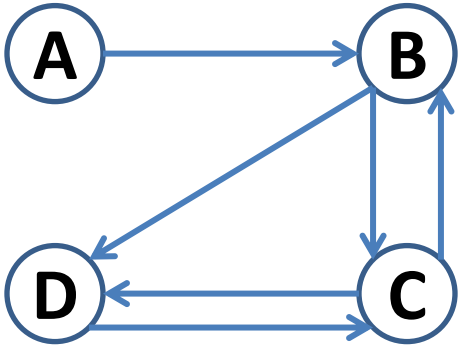
- Both DP and Divide and Conquer can be easily solved using recursion
  - For Divide and conquer you typically want to use recursion (ok if you do)
  - For DP you want to use memoization or the iterative solution (recursion is exponential)
- Recursive problem that does not allow neither DP nor Div & Conq:
  - Place N queens on an NxN check board (s.t. they do not attack each other)
- Divide and conquer problems:
  - Sum of elements in an array
    - Give the solution here
  - Fibonacci with splitting in half (or Sorting elements in an array (Mergesort))
  - Notes:
    - Problems of same size (e.g. 3 elements) are different (depending on the actual values and order of the elements). => cannot 'reuse' solutions to smaller problems.
    - Unlike DP, the solution does not depend on where we cut (ok to cut into subproblems at any place)
- Why Dynamic Programming (DP) and not Divide and Conquer
  - We cannot apply Div&Conq because we cannot commit to a 'point' in the solution (e.g. , for Job Scheduling, choose if a certain job will be selected or not).

When DP does not work

- Queens placement problem:
  - Fails overlapping subproblems: Cannot reuse solutions to smaller problems (even if just counting).
    - The placement on any column depends on the placement on previous columns => cannot reuse the solution from previous columns (it will in fact certainly not work).

Q		Q	
	Q		Q

- Longest path (with no cycles) in a graph
  - Fails optimal substructure: if C is part of an optimal solution for A->D, then the A->C and C->D parts of that solution, are optimal solution for those smaller problems. Fails. There is a longer path from C to D: C,B,D
  - Note: Shortest path has this property:
    - A->D: A,B,D
    - C is not part of an optimal solution



Dynamic Programming Space Requirements

- Quadratic or linear
  - Quadratic: edit distance, LCS, 0/1 Knapsack
  - Linear: Unbounded Knapsack
- If you only need the optimal cost, you may be able to optimize it,
  - Quadratic -> linear: keep only the previous row (e.g. LCS)
  - Linear -> constant: keep only the computations for the few immediately smaller problems that are needed (e.g. Fibonacci)
- If you need to also backtrace the solution (choices) that gave this optimal cost, you have to keep information (either cost or choice) for all the smaller subproblems.

List of DP Problems

- Stair Climbing
- LCS (Longest Common Subsequence) *
- LIS (Longest Increasing Subsequence)
- Edit distance *
- Job scheduling *
- Knapsack
  - Special case: Rod cutting
  - Equivalent: stair climbing with gain
- Matrix traversal
  - All ways to traverse a matrix – not an optimization problem
  - All ways to traverse it when there are obstacles
  - Most gain when there is fish that you can eat/catch.
    - Variant: add obstacles
  - Monotone decimal / Strictly monotone decimal (consecutive digits are ≤ or <))
- Shuttle to airport
- Subset sum
- Chain matrix multiplication
  - Order in which to multiply matrices to minimize the cost.

Motivation for DP

- Applications:
  - Edit distance and other similarity measures:
    - Word similarity
    - Time series similarity (ASL, compare trends, data that can be represented as sequences and require alignment) that requires
      - Can you guess how we used DP to find videos of similar signs in ASL?

- Note that for some problems it is even hard to find the Brute Force solution or the Greedy solution
  - e.g. for the Edit Distance

Analyzing recursive functions

- Look at the implementations for
  - Knapsack
  - Rod cutting
  - Memoization motivation example (the math definition)

Worksheet:
Inefficient Recursive Solution
(CLRS: Rod-Cutting)

Write the recurrence formula for this function: T(N) =……..

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Source: CLRS

Answers:

Inefficient Recursive Solution

(CLRS: Rod-Cutting)

$$T(n) = \sum_{i=1}^{n} T(n-i) + \Theta(n)$$

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Note the difference between
- what the function computes and returns (for this I need the information in the array p) and
- the time complexity: T(N) (for this I do not need to know p)

Source: CLRS

Example: recursion, time complexity, benefits of memoization

$$F(n) = \begin{cases} 5 & \text{if } 0 \le n \le 3 \\ F(n+3) - 2 & \text{if } n \text{ is odd} \\ F(n/2) + F(n-4) & \text{if } n \text{ is even} \end{cases}$$

Example used to show:
- Benefits of memoization
- Difference between evaluating a function and the time complexity of a program solving that function

- Evaluate the function in points: 2, 3, 4, 5, 6, 7.
- Give a C implementation (code) for this math function.
  - Recursive. Recursive with memoization. Iterative.
  - Draw the tree showing the recursive calls for both recursions: inefficient and memoized.
  - Use the tree drawn above to estimate the time complexity for the memoized version. (Hint: notice that each number (problem size) is an internal node at most once in a binary tree.)
- What is the time complexity of your function?
  - Write the recursive formula for your implementation. (For solving: bring to one recursive case.)
- Benefits of memoization:

  ** for F(10) not needed: F(7) and F(9)

  ** for F(20) not needed: F of: 7, 9, 11, 13, 14, 15, 17, 18, 19

Worksheet

$$F(n) = \begin{cases} 3 & \text{if } n \le 1 \\ F(n-1) & \text{if } n \text{ is odd} \\ F(n/2)+10 & \text{if } n \text{ is even} \end{cases}$$

```
int foo(int n){
    if (n <= 1) return 3;
    if (n%2 == 1)
       return foo(n-1);
    else
        return foo(n/2)+10;
}
```

- Math definition => code => recurrence formula => Θ
- Recursive inefficient => memoized

Answer

$$F(n) = \begin{cases} 3 & \text{if } n \le 1 \\ F(n-1) & \text{if } n \text{ is odd} \\ F(n/2)+10 & \text{if } n \text{ is even} \end{cases}$$

```
int foo(int n){
    if (n <= 1) return 3;
    if (n%2 == 1)
       return foo(n-1);
    else
        return foo(n/2)+10;
}
```

$$T(n) = c \quad \text{if } n \le 1$$
$$T(n) = T(n-1) + c \quad \text{if } n \text{ is odd}$$
$$\mathrm{T(n)} = \mathrm{T(n/2)} + \mathrm{c} \quad \text{if } n \text{ is even}$$

Note that in this case it is a coincidence that the recurrence formulas, T(n), for the time complexity are similar to what the function computes (and the math definition). Typically they are different. Even here you can see the difference in the case when n is odd: the +c in the T(n) formula.

Answer

$$F(n) = \begin{cases} 3 & \text{if } n \le 1 \\ 5F(n-1) & \text{if } n \text{ is odd} \\ F(n/2)+10 & \text{if } n \text{ is even} \end{cases}$$

```
int foo(int n){
    if (n <= 1) return 3;
    if (n%2 == 1)
       return 5*foo(n-1);
    else
        return foo(n/2)+10;
}
```

$$T(n) = c \quad \text{if } n \le 1$$
$$T(n) = T(n-1) + c \quad \text{if } n \text{ is odd}$$
$$\mathrm{T(n)} = \mathrm{T(n/2)} + \mathrm{c} \quad \text{if } n \text{ is even}$$

Note that when n is odd, we still use T(n-1) and not 5T(n-1), because still only one recursive call is made in the foo function when n is odd. (The result of that call is multiplied with 5, but that affects what the function computes, not its time complexity.)

From Recursive Formula to Code Worksheet

- Give a piece of code/pseudocode for which the time complexity recursive formula is:
  - $T(N) = N*T(N-1) + n$ or
  - $T(N) = N*T(N-1) + \Theta(N)$

  Assume $T(1) = c$

From Recursive Formula to Code Answers

- Give a piece of code/pseudocode for which the time complexity recursive formula is:
  - $T(N) = N*T(N-1) + n$ or
  - $T(N) = N*T(N-1) + \Theta(N)$

  Assume $T(1) = c$

```
int foo(int N){
    if (N <= 1) return 3;
    for(int i=1; i<=N; i++)
      foo(N-1);
}
```

Compare

```
int foo1(int N){
    if (N <= 1) return 3;
    for(int i=1; i<=N; i++){
       foo1(N-1);
    }
}
```

T(N) = **N***T(N-1) + cN

```
int foo2(int N){
    if (N <= 1) return 3;
    for(int i=1; i<=N; i++){
       printf("A");
    }
    foo2(N-1); // foo2(N-1) is not
               // in the loop
}
```

T(N) = T(N-1) + cN

DP - summary

- Most problems that can be solved with DP have a solution that can be expressed with a cost function.
- If the cost function is given the implementation is straight-forward. Implementation options:
  - Iterative
  - Recursive
  - Memoized
- Guidelines for finding the cost function:
  - Express the solution to the problem as a sequence of independent choices/steps
  - Try to write the cost of the current problem using answers to subproblems:
    - You must consider the choice and the answer to the subproblem.

Additional Problems

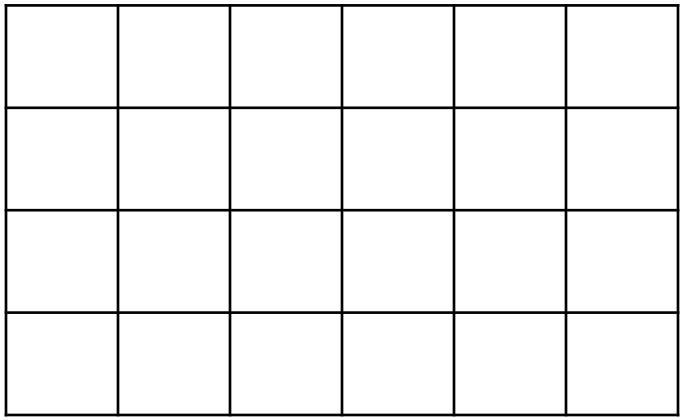
2D Matrix Traversal

P1. All possible ways to traverse a 2D matrix.

- Start from top left corner and reach bottom right corner.
- You can only move: 1 step to the right or one step down at a time. (No diagonal moves).
- Variation: Allow to move in the diagonal direction as well.
- Variation: Add obstacles (cannot travel through certain cells).

P2. Add fish of various gains. Take path that gives the most gain.

- Variation: Add obstacles.



Stair Climbing

Stair Climbing

- Problem:

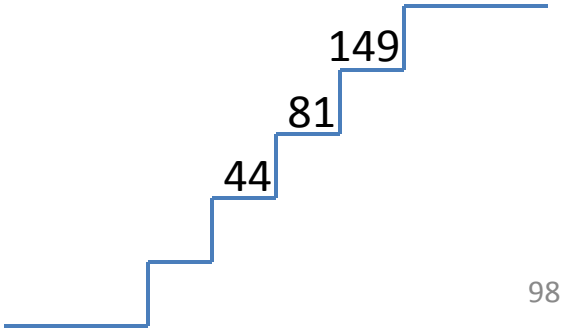
  A child has to climb N steps of stairs. He can climb 1,2 or 3 steps at a time.

  How many different ways are there to climb the stairs?

  - We count permutations as different ways, e.g.:
    - (3,1,3,1) is different than (3,3,1,1)

      (2 jumps of 3 and 2 jumps of 1 in both cases, but in different order)
  - Let *C(n)* – number of different ways to climb n stairs.

- Solution:
  - Hints:
    - What dictates the problem size (e.g. give 2 problems of different sizes).
    - What is the smallest problem size that you can solve?



Stair Climbing

- Problem:

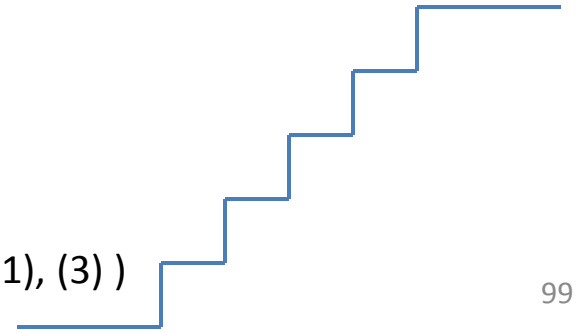
  A child has to climb N steps of stairs. He can climb 1,2 or 3 steps at a time.

  How many different ways are there to climb the stairs?

  - We count permutations as different ways, e.g.:
    - (3,1,3,1) is different than (3,3,1,1)

      (2 jumps of 3 and 2 jumps of 1 in both cases, but in different order)
  - Let *C(n)* – number of different ways to climb n stairs.

- Solution:
  - What is the last jump he can do before he reaches the top?
    - 1,2 or 3 => *C(n) = C(n-1) + C(n-2) + C(n-3)*
  - *Set-up base cases*
    - *C(0) = 1* (stays in place, this allows us to cover N = 1,2,3 by recursion)
    - *C(x) = 0, for all x<0*
  - Test that it works for the first few small n: 1,2,3,4:
    - C(1) = C(0) + C(-1) + C(-2) = 1 + 0 + 0 = 1
    - C(2) = C(1) + C(0) + C(-1) = 1 + 1 + 0 = 2 ((1, 1) or (2))
    - C(3) = C(2) + C(1) + C(0) = 2 + 1 + 1 = 4 ((1,1,1) , (1,2), (2,1), (3))
    - C(4) = C(3) + C(2) + C(1) = 4 + 2 + 1 = 7



Stair Climbing

- Write code

Stair Climbing Worksheet

Possible ways to climb this many stairs.

Count based on possible last steps.

index	0	1	2	3	4	5	6	7	8	9	10	11
Sol	1	1										
1	-	1										
2	-	-										
3	-	-										

Stair climbing with gain

Solve for 17 stairs with the Gain (e.g. health points, HP,) per jump size given below. Here the only jump sizes that give gain are the ones in the table.
You can assume that other jump sizes are allowed, but they give 0 gain. That allows us to consider ALL problems size (for example a problem of size 1 is allowed: 1 step we have 0 gain.)

Jump Size	2	3	9		
Gain	2	4	18		

Stair climbing with gain

Jump	Gain (health points (HP)	Jump size	Remaining stairs [rem_stairs] (smaller problem size)	Sol[rem_stairs] ($$) (answer to smaller problem)	New total Gain (HP) (for each possible choice)
	2	2	4 (=6-2)	4	6 (= 4+2)
	4	3	3 (=6-3)	4	8 (= 4+4)
	18	9	-	-	-

The top table shows the details of the work from the last 3 rows of the bottom table.

Pick the maximum as the answer for this problem size, 6.

Solution (gain) for problems of this size.

Last choice that gives the optimal solution.

Gain based on possible last steps.
These rows show the work needed to fill-in the answers in Sol and Picked.

index	0	1	2	3	4	5	6	7	8	9	10	11
Sol	0	0	2	4	4	6	8					
Picked	-1	-1	2	3	2	2	3					
2 (2)	-	-	0,2	1,2	2,4	3,6	4,6					
3 (4)	-	-	-	0,4	1,4	2,6	3,8					
9 (18)	-	-	-	-	-	-	-					

Remaining stairs

Gain with this jump

Stair climbing with gain - Answers

Jump	Gain (health points (HP)	Jump size	Remaining stairs [rem_stairs] (smaller problem size)	Sol[rem_stairs] ($$) (answer to smaller problem)	New total Gain (HP) (for each possible choice)
	2	2	4 (=6-2)	4	6 (= 4+2)
	4	3	3 (=6-3)	4	8 (= 4+4)
	18	9	-	-	-

The top table shows the details of the work from the last 3 rows of the bottom table.

Pick the maximum as the answer for this problem size, 6.

Solution (gain) for problems of this size.

Last choice that gives the optimal solution.

index	0	1	2	3	4	5	6	7	8	9	10	11
Sol	0	0	2	4	4	6	8	8	10	18	18	20
Picked	-1	-1	2	3	2	2	3	2	2	9	9	2
2 (2)	-	-	0,2	1,2	2,4	3,6	4,6	6,8	6, 10	7, 10	8, 12	9, 20
3 (4)	-	-	-	0,4	1,4	2,6	3,8	4,8	5, 10	6, 12	7, 12	8, 14
9 (18)	-	-	-	-	-	-	-	-	-	0, 18	1, 18	2, 20

Gain based on possible last steps.
These rows show the work needed to fill-in the answers in Sol and Picked.

Solution (gain) for problems of this size.

Last choice that gives the optimal solution.

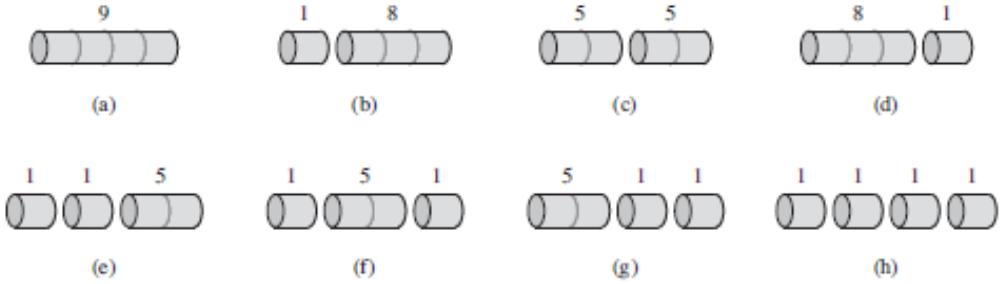
Gain based on possible last steps.
These rows show the work needed to fill-in the answers in Sol and Picked.
In an implementation (code) you will NOT need a 2D array.

index	0	1	2	3	4	5	6	7	8	9	10	11
Sol	0	0	2	4	4	6	8	8	10	18	18	20
Picked	-1	-1	2	3	2	2	3	2	2	9	9	2
2 (2)	-	-	0,2	1,2	2,4	3,6	4,6	6,8	6, 10	7, 10	8, 12	9, 20
3 (4)	-	-	-	0,4	1,4	2,6	3,8	4,8	5, 10	6, 12	7, 12	8, 14
9 (18)	-	-	-	-	-	-	-	-	-	0, 18	1, 18	2, 20

Rod Cutting Problem

- Given:
  - A rod of a certain length, n.
  - An array of prices for rods of sizes between 1 and n.
  - What is the best way to cut the cod so that you make the most money out of selling it?
- Optimization problem:
  - There are many ways to cut the rod, you want the best one.

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30



CLRS image & formula

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

Example of an Inefficient Recursive Solution (CLRS: Rod-Cutting)

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Source: CLRS

Recursive solution idea:

- Consider all possible sizes for the first piece and make a recursive call for cutting the remaining piece
- Add the values (of the first piece and the result of the recursive call)
- Return the max of all these combinations

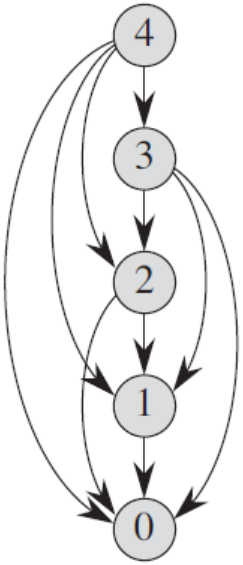
Rod Cutting Problem

- Recursive solution:
  - Try cutting a first piece of all possible size and make a recursive call for cutting the remaining piece
  - Add the values (of the first piece and the result of the recursive call)
  - Return the max of all these combinations
- Recursion tree
  - Example for n = 4
    - Notice how many times a recursive call is made for problems of the same size: 8 times for size 0, 4 times for size 1, 2 times for size 2
      - We will fix this inefficiency.
  - Properties of a tree for rod of size n:
    - Total nodes (including leaves): 2^n. (proof by induction)
    - Total leaves: 2^{n-1}. (proof by induction)
    - Number of leaves = number of different ways to cut the rod
      - Each path down to a leaf corresponds to a different way of cutting the rod.

Rod Cutting Problem

- 3 solutions:
  - Recursive – inefficient (NOT Dynamic Programming (DP)) –
    - $\Theta(2^n)$
  - Top-down DP (Memoization) - recursive with saving the computed results
    - $\Theta(n^2)$
  - Bottom-up DP - non-recursive
    - $\Theta(n^2)$
- Recursion tree
  - For plain recursion
  - For memoization
    - Also see the solution array
- DP solution complexity - $\Theta(n^2)$

Subproblem graph
(CLRS image)

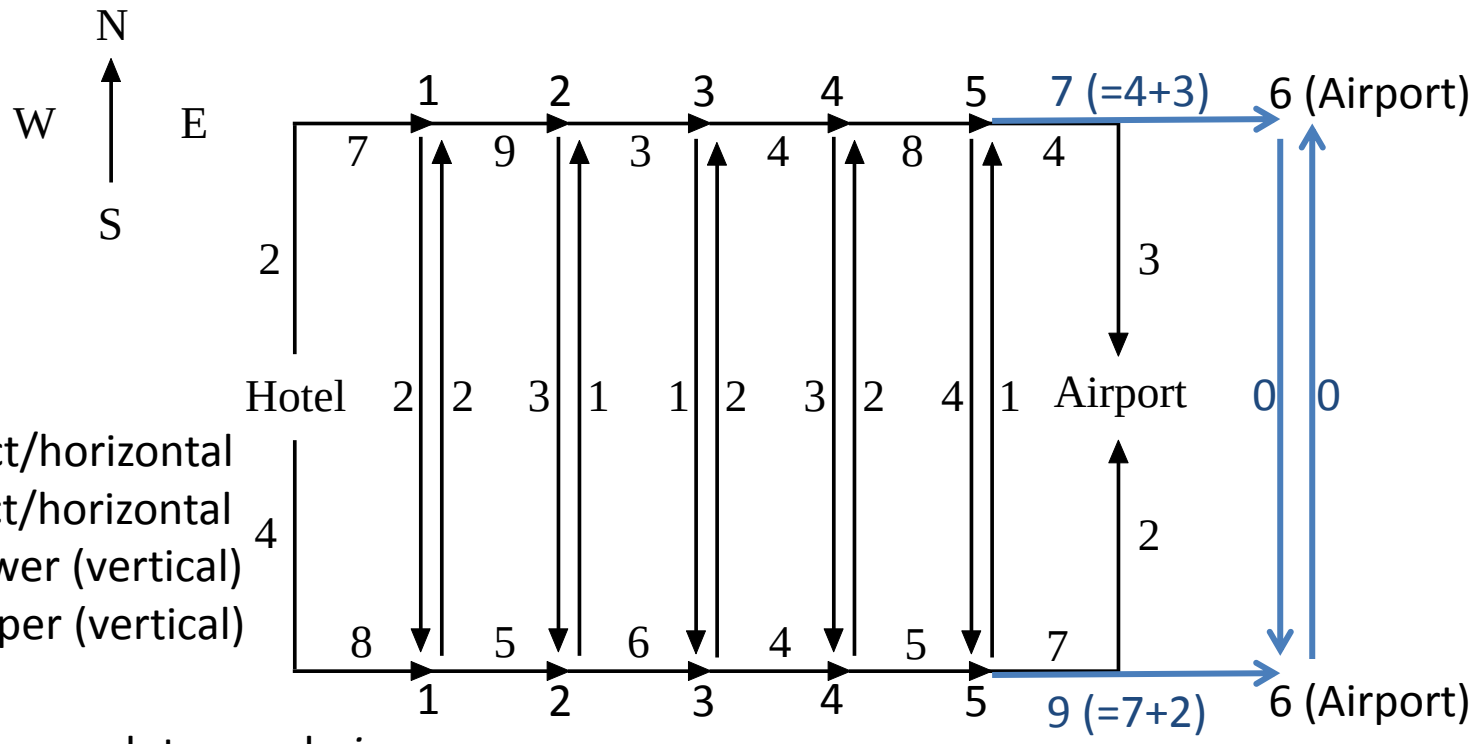


Shuttle-to-Airport Problem

Shuttle-to-Airport Problem

(Dr. Bob Weems, notes07)

(We may cover this at the end)



UD(i) – Upper direct/horizontal
LD(i) – Lower direct/horizontal
UL(i) – Upper to lower (vertical)
LU(i) – Lower to upper (vertical)

U(i) – optimal cot to reach top node *i*
L(i) - optimal cot to reach bottom node *i*
U(0) = L(0) = 0
U(i) = min{U(i-1) + UD(i), L(i-1) + LD(i)+LU(i)}
L(i) = min{L(i-1) + LD(i), U(i-1) + UD(i)+UL(i)}

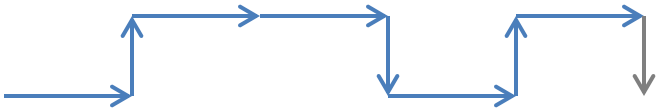
Time complexity : O(n)
Fill out U and L arrays with constant local cost.

Shuttle-to-Airport Problem

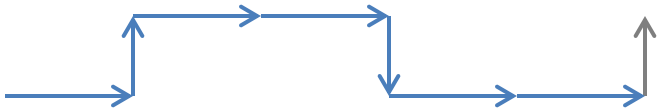
Brute Force

- Evaluate all routes that end at the top n-th point.
  - The top n-th and bottom n-th points will have the same value , so we will use the top n-th point as the airport.
    - These are really the same point (the airport). We artificially split it into two points to make the problem representation more uniform (as opposed to having it as a special case).
    - Any route that ends at the bottom n-th point is a duplicate of the same route ending at the top point (with the use of one of the vertical n-th arrows with cost 0).

A route ending at the top.
Using the gray arrow, we get the duplicate route ending at the bottom.



A route ending at the bottom.
Using the gray arrow, we get the duplicate route ending at the top.



Shuttle-to-Airport Problem

Brute Force: Version 1

- VERSION 1: count all the ways to get to any point.
- Let:
  - *top(i)* be the total number of different ways to get to the top point, i.
  - *bottom(i)* be the total number of different ways to get to the bottom point, i.
- Then:
  - *top(0) = bottom(0) = 1* (we are already at the starting point)
  - *top(i) = bottom(i) = top(i-1) +bottom(i-1)*
  - *=> top(i) = 2*top(i-1) => top(i) = 2^i.*
    - Straight forward proof by induction that *top(i) == bottom(i)* and that *top(i) = 2^i*.
  - *=> top(n) = 2^n* routes
- => $O(2^n)$ complexity of brute force

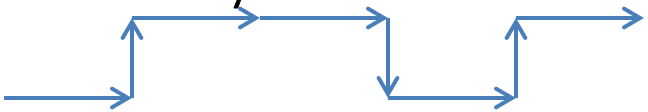
Shuttle-to-Airport Problem

Brute Force: version 2

- Version 1: count representations of all different routes.
- Notice that:
  - A horizontal arrow will always be picked (it moves us closer to the airport)
  - A vertical arrow may or may not be picked.
  - There cannot be two consecutive vertical arrows. Between any two horizontal arrows in a solution, there may be nothing, or a vertical arrow.

- The difference between two routes is in the placement of vertical arrows.

⇒ Encode only the vertical movement: 1-used (either up or down), 0-not used

⇒  is encoded as: 1 0 1 1

⇒ 2^n routes => $O(2^n)$

⇒ Note that the 'reflection' routes will lead to the n-th bottom point and that is a duplicate solution, thus ignored.

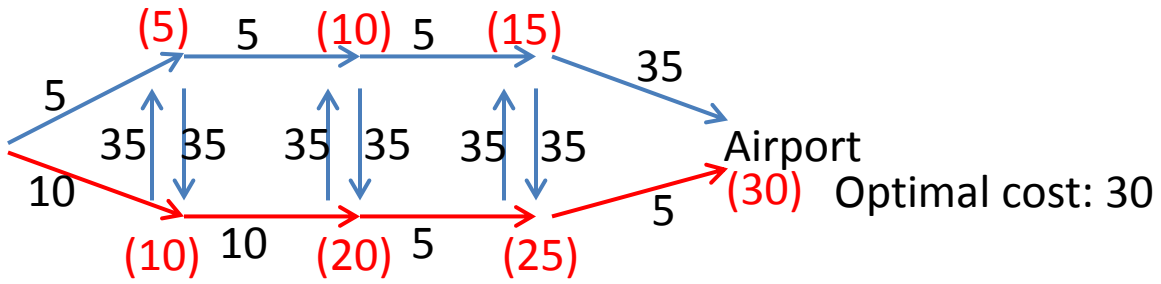
Shuttle-to-Airport Problem Backtracing

- Version 1: record choices (h - horizontal or v-vertical arrow to reach that node)
- Version 2: use the cost function to see which way you came from:
  - *If ((U(i-1) + UD(i)) <= (L(i-1) + LD(i)+LU(i)))*
    - *Horizontal (upper direct arrow used)*

    Else
    - *Vertical (lower to upper arrow used)*
  - *Similar for (L(i-1) + LD(i)), and (U(i-1) + UD(i)+UL(i))*
- Hint, if you store U and L in a 2D array as rows, you can easily switch from top to bottom by using (row+1)%2. The row value indicates: 0-top, 1-bottom
- Time complexity: O(n)

Shuttle-to-Airport Backtracing

Example showing that in the first path (when computing the optimal cost) we do not know what nodes will be part of the solution. Here the top 3 nodes have costs smaller than their bottom correspondents, but the final top cost is too large and the bottom route becomes optimal.



	0	1	2	3	4	5	6
U	0	9(h)	17(v)	20(h)	24(h)	31(V)	38(h)
L	0	11(v)	16(h)	21(v)	25(h)	30(h)	38(v)

Route (in reversed order, starting from 6 top):
6 top, 5 top, 5 bottom, 4 bottom, 3 bottom, 3 top, 2 top, 2 bottom, 1 bottom, start point

Worksheet Knapsack Example 2

Item	Item value ($$)	Item weight (Kg)	Remaining capacity (Kg)	Sol[rem_capacity] ($$)	New total value ($$)
A	4	3			
B	5	4			
C	10	7			
D	11	8			
E	13	9			

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Sol																		
Picked																		
3 (4)																		
4 (5)																		
7 (10)																		
8 (11)																		
9 (13)																		

Answers Knapsack Example 2

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Sol	0	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
Picked	-	-	-	A	B	B	A	C	D	E	A	A	A	A	C	A	C	A
A, 3, 4	-	-	-	0,4	1,4	2,4	3,8	4,9	5,9	6, 12	7, 14	8, 15	9, 17	10, 18	11, 19	12, 21	13, 22	14, 24
B, 4, 5	-	-	-	-	0,5	1,5	2,0	3,9	4, 10	5, 10	6, 13	7, 15	8, 16	9, 18	10, 19	11, 20	12, 22	13, 23
C, 7,10	-	-	-	-	-	-	-	0, 10	1, 10	2, 10	3, 14	4, 15	5, 15	6, 18	7, 20	8, 21	9, 23	10, 24
D,8,11	-	-	-	-	-	-	-	-	0, 11	1, 11	2, 11	3, 15	4, 16	5, 16	6, 19	7, 21	8, 22	9, 24
E,9,13	-	-	-	-	-	-	-	-	-	0, 13	1, 13	2, 13	3, 17	4, 18	5, 18	6, 21	7, 23	8, 24