SIT221: Data Structures and Algorithms

Lecture 10: Algorithm Design Paradigms & Dynamic Programming

Week 9 recording for prac

	Recording Name	Session Name	Date	Duration
Week 9	Week 9 - prac - recording 1	Week 9 - prac	9/09/2017 4:23 pm	00:33:14
Week 8	Week 8 - prac - recording 1	Week 8 - prac	2/09/2017 4:08 pm	01:09:28
Week 7	Week 7 - prac - recording 1	Week 7 - prac	26/08/2017 4:11 pm	00:39:42
Week 4	SIT221 - Data Structures And Algorithms - recording 1	SIT221 - Data Structures And Algorithms	3/08/2017 11:22 am	00:42:34
Week 3	SIT221 - Data Structures And Algorithms - recording 1	SIT221 - Data Structures And Algorithms	27/07/2017 10:55 am	00:40:01
Week 2	Week 2 - prac - recording 1	Week 2 - prac	20/07/2017 11:01 am	00:31:17

Algorithm design paradigms

- These are techniques that we can use to solve a given problem.
  - Brute Force
  - Reduce to a known problem
  - Divide & Conquer
  - Dynamic Programming
  - Greedy Algorithms

Example

- Given an array of n elements, can we check if it has repeated elements?

Brute force

- Try all possible solutions.
- Get a correct solution, and do not pay attention to the running time of your solution.

Brute forcing repeated array elements

- Loop on all elements, for each element I will compare it with all other elements in the array – if a match is found, the algorithm is stopped and reports element X is repeated at index i.
- What’s your BigO?

```
for(int i = 0 ; I < data.Count; i++) {
    for(int j = 0 ; j < data.Count; j++) {
        If( data[i] == data[j] && i!=j) {
        Console.WriteLine(string.Format(“element {0} is repeated..
    }
}
```

Reduce to known problem

- Same problem can be solved by sorting your data using O(nlog(n)) algorithm – give one?
- Now the data is sorted, we can loop on all of them comparing every consecutive elements – O(n)
- So the total running time is O(???) – Is this more efficient?

Divide and conquer

- **Divide** the problem into a number of smaller sub-problems of the same type.
- **Conquer** (solve) the smaller problems (recursively)
- **Merge** solutions of sub-problems into a solution to the original problem.
- Top-down, we start with the big problem and keep dividing until we reach atomic problem. see next slide

Find Min/Max	Factorial	Fibonacci

Max(data) or Min(data)
 	
int Max(int[] data, int start, int end)
{
if(start == end) return data[start];
int mid = (start+end)/2);
int max1 = Max(data, 0, mid);
int max2 = Max(data, mid+1, end);
return (max1 > max2)? max1 : max2;
}
$T(n) = T(n/2) + T(n/2)$
$T(n) = 2 * T(n/2) = 2 * 2 * T(n/4)$
$T(n) = 2^i * T(n/2^i)$
@ $n/2^i = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$
$T(n) = 2^{\log_2(n)} = O(n)$
	Fact(n) = n * Fact(n-1)
	
	int Fact(int n)
	{
	if(n == 1) return 1;
	return n * Fact(n-1);
	}
	
	$T(n) = T(n-1) + 1;$
	$T(n) = T(n-2) + 1 + 1;$
	$T(n) = T(n-3) + 1 + 1 + 1;$
	$T(n) = T(n – i) + i$
	@ $n – i = 1 \Rightarrow i = n – 1$
	$T(n) = T(1) + n -1$
	$O(n)$
		Fib(n) = Fib(n-1) + Fib(n-2)
		
		int Fib(int n)
		{
		if(n == 1 || n == 0) return 1;
		return Fib(n-1) + Fib(n-2)
		}
		
		$T(n) = T(n-1) + T(n-2)$
		$T(n) \sim= 2 * T(n – 1)$
		$T(n) \sim= 2 * 2 * T(n-2)$
		$T(n) \sim= 2*2*2 * 2(n-3)$
		$T(n) \sim= 2^n * 1$
		
		EXPONENTIAL === ooppsss!

What else?

- Tree traversal (pre-order, post-order, in-order)
- List traversal, search, etc.
- Binary search
- Merge & Quick sort
- Other recursive problems

Problems with Divide & Conquer

- Stack overflow – so deep, nested sub-problems?

- Repeated problems are repeatedly solved ==> we should use **memorization as in dynamic programming**

- Base case size ==> shall we keep breaking down the problem until we have empty list or input of size zero?

Dynamic programming (DP)- Careful brute force!

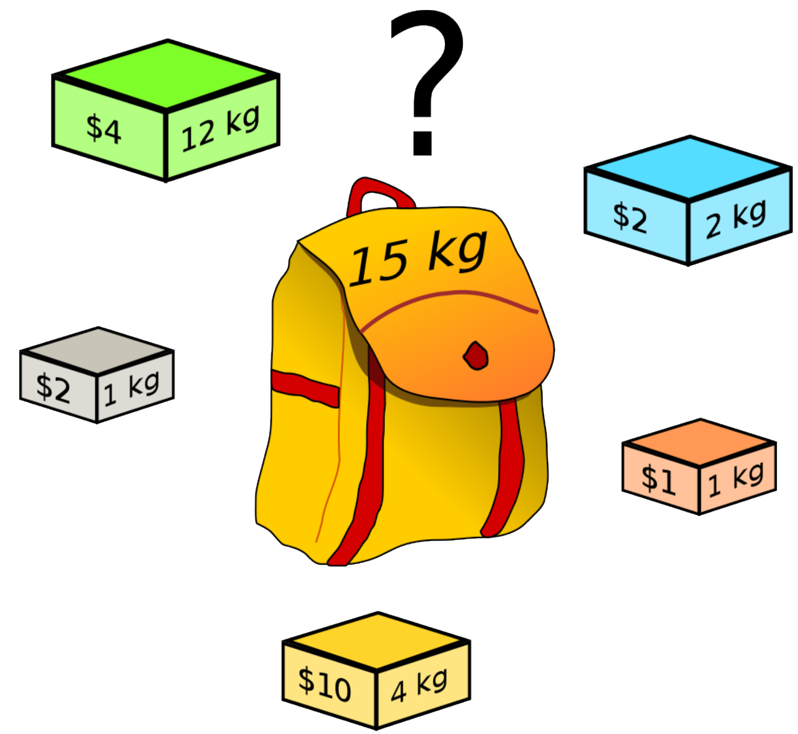
- Divide & Conquer + Memorize

- It moves bottom-up (instead of top-down) while storing solutions to sub-problems to use in solving the original problems.

0/1 Knapsack problem

- Given a set of n items, each with a weight and a value.
- You have a knapsack with max weight (W)
- Which items to take such that the total weight (sum w_i) is less than or equal to the knapsack limit and the benefit (b) is the maximum.

$$\max \sum_{i \in T} b_i \text{ subject to} \sum_{i \in T} w_i \leq W$$



0/1 Knapsack problem – solution development

If we have a knapsack of max weight 10 and items with weight and value as show below

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

- Brute force? try all possible combinations → $O(2^4)$
- Dynamic programming? Let’s see

0/1 Knapsack problem – solution development

- Let $w_1, w_2, \dots, w_n$ and $v_1, v_2, \dots, v_n$ be the weights and values of item $1, 2, \dots, n$.
- Let W be the capacity of the bag
- $w_1, w_2, \dots, w_n$ and W are strictly positive integers.
- Let $m[i, w]$ be the maximum value that can be attained with weight less than or equal to w using items up to i (first i items)
- We can define
  - $m[0, w] = 0$
  - $m[i, w] = m[i-1, w]$ if $w_i > w$
  - $m[i, w] = \max\{m[i-1, w], m[i-1, w-w_i] + v_i\}$ if $w_i \leq w$
- The solution can be found by $m[n, W]$

0/1 Knapsack problem – solution development

```
for (int j=0; j<=W; j++)
        m[0][j] = 0;
for (int i=1; i<=n; i++)
{
        for (int j=0; j<=W; j++)
        {
                if (w[i] > j)
                        m[i][j] = m[i-1][j];
                else
                        m[i][j] = max(m[i-1][j], m[i-1][j-w[i]] + v[i]);
        }
}
```

0/1 Knapsack - Example

Up to the first item:

Value	Weight	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0
10	5	0	0	0	0	0	10	10	10	10	10	10
40	4	0										
30	6	0										
50	3	0										

Weight 0 to 4, value is zero
Weight 5 to 10, value is 10

$$m[i,w] = \max\{m[i-1,w], m[i-1,w-w_i]+v_i\}$$

0/1 Knapsack - Example

Up to the second item:

Value	Weight	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0
10	5	0	0	0	0	0	10	10	10	10	10	10
40	4	0	0	0	0	40	40	40	40	40	50	50
30	6	0										
50	3	0										

Weight 0 to 3, value is zero
Weight 4 to 8, value is 40
Weight 9 to 10, value is 50

$$m[i,w] = \max\{m[i-1,w], m[i-1,w-w_i]+v_i\}$$

0/1 Knapsack - Example

Up to the third item:

Value	Weight	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0
10	5	0	0	0	0	0	10	10	10	10	10	10
40	4	0	0	0	0	40	40	40	40	40	50	50
30	6	0	0	0	0	40	40	40	40	40	50	**60**
50	3	0										

Weight 0 to 3, value is zero
Weight 4 to 8, value is 40
Weight 9 to 9, value is 50
Weight at 10, value is 60

$$m[i, w] = \max\{m[i-1, w], m[i-1, w-w_i] + v_i\}$$

0/1 Knapsack - Example

Up to the fourth item:

Value	Weight	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0
10	5	0	0	0	0	0	10	10	10	10	10	10
40	4	0	0	0	0	40	40	40	40	40	50	50
30	6	0	0	0	0	40	40	40	40	40	50	60
50	3	0	0	0	50	50	50	50	**90**	90	90	90

Weight 0 to 2, value is zero
Weight 3 to 6, value is 50
Weight 7 to 10, value is 90

Items to choose include: weight 3 & 4

$$m[i,w] = \max\{m[i-1,w], m[i-1,w-w_i]+v_i\}$$

Greedy Algorithms

- "take what you can get now" strategy.
- Work in phases, in each phase the currently best decision is made
- Not guaranteed optimal solution.

Coins – change problem

- You were asked to write a coin change program for a supermarket. Your program should choose the **minimum number of coins** to give as a change.
- You decided to use a greedy algorithm { choose max every round}
- Coins available {1cent, 5 cents, 10 cents, 20 cents, 50 cents}
- For a change of 63 cents, what coins should your code generate?
  - 50 + 10 + 3 * 1 { 4 coins}
  - What if you are running out of 5 and 10 cents?
    - 50 + 13 * 1 { 14 coins}, you could have chosen 3 * 20 + 3 * 1 { 6 coins }

Implementation

```
void change(int cents) {
    while (cents > 0)
    {
        for (int i = coins.Length - 1; i >= 0; i--)
        {
            if (cents / coins [i] > 0)
            {
                Console.WriteLine(string.Format("{0} of {1} " , cents/coins [i] , coins[i]));
                cents = cents % tems[i];
             }
         }
     }
}
```