**Greedy Algorithms**

* A ***greedy algorithm*** always makes the choice that looks best at the moment.
* It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
* In some problems, can achieve optimal solutions, for Other problems, achieves near-optimal solutions fast.
* Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.
* If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. But Greedy algorithms cannot always be applied.
* An algorithm to solve the activity-selection problem does not need to work

bottom-up, like a table-based dynamic-programming algorithm. Instead, it can

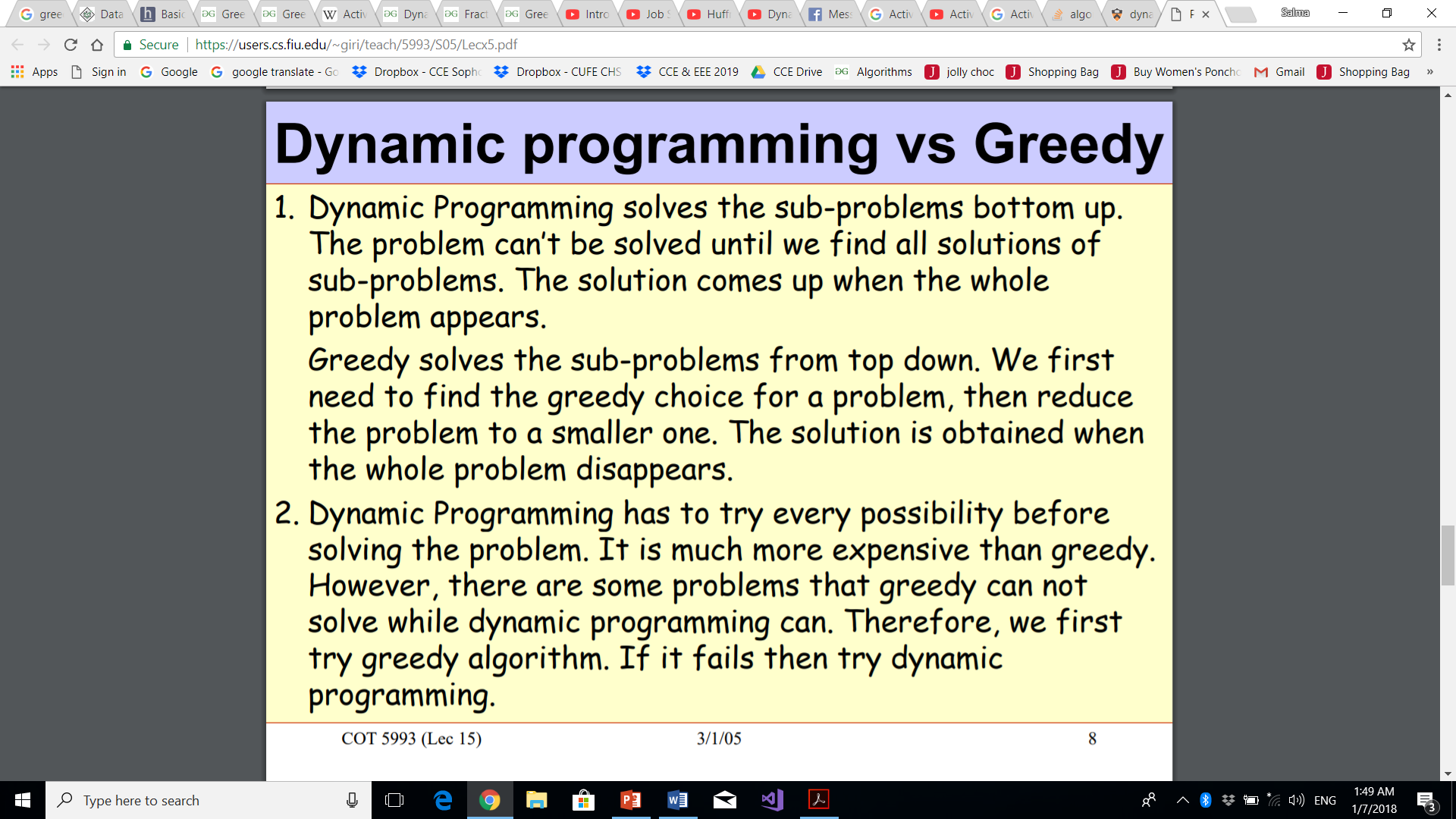
work top-down, choosing an activity to put into the optimal solution and then solving

the subproblem of choosing activities from those that are compatible with those

already chosen. Greedy algorithms typically have this top-down design: make a

choice and then solve a subproblem, rather than the bottom-up technique of solving

subprobems before making a choice.



**Greedy Algorithms (Slides):**

1.Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.

2.Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.

3.Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

**Activity Selection Algorithm:**

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

**Example 1 :** Consider the following 3 activities sorted by

by finish time.

start[] = {10, 12, 20};

finish[] = {20, 25, 30};

A person can perform at most **two** activities. The

maximum set of activities that can be executed

is {0, 2} [ These are indexes in start[] and

finish[] ]

The process that we can followed in to develop a greedy algorithm was

a bit more involved than is typical. We go through the following steps:

1. Determine the optimal substructure of the problem.

2. Develop a recursive solution. (For the activity-selection problem, we formulated

recurrence (16.2), but we bypassed developing a recursive algorithm based

on this recurrence.)

Hint: This recurrence is using **dynamic programming**

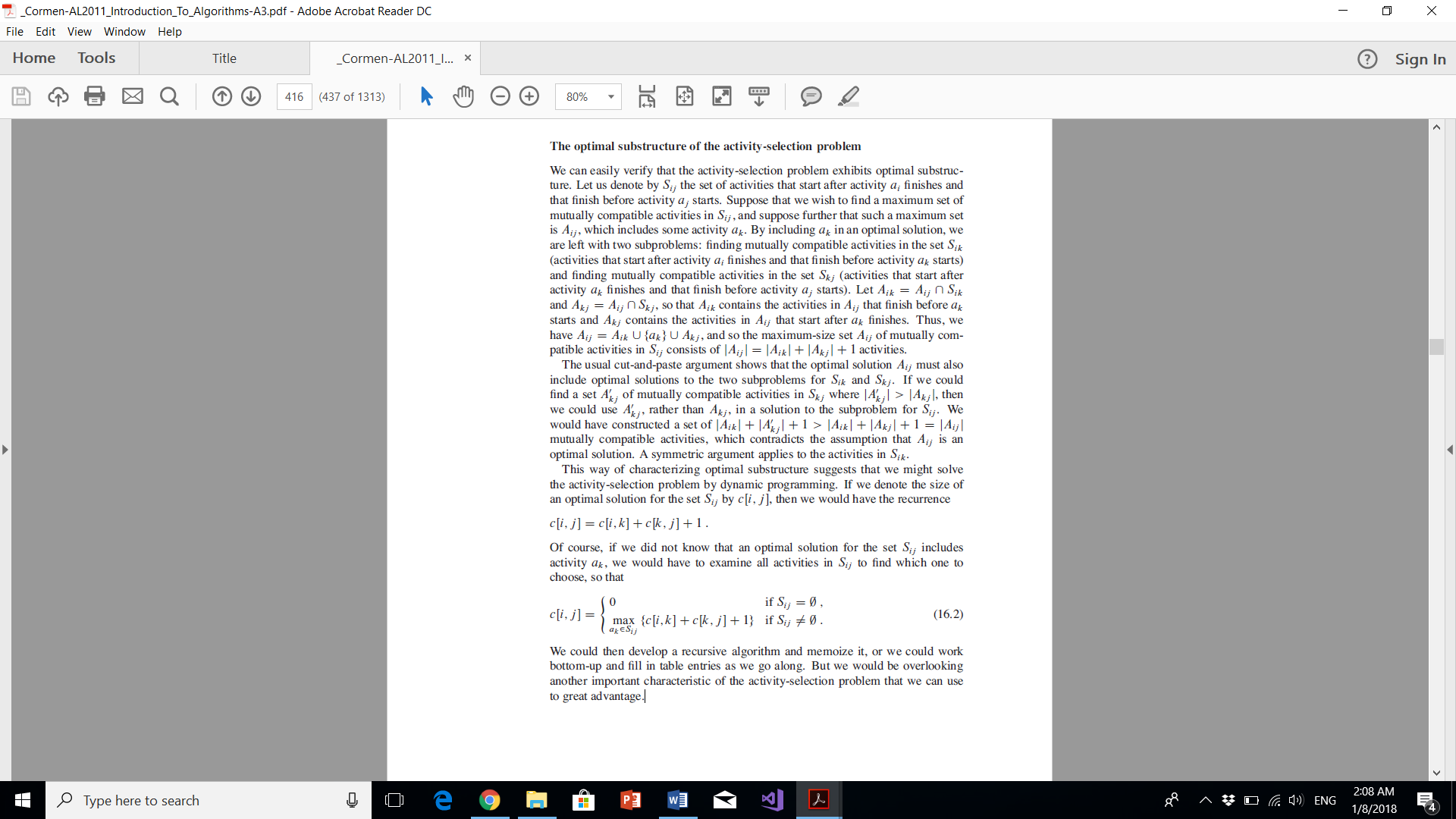
3. Show that if we make the greedy choice, then only one subproblem remains.

4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur

in either order.)

5. Develop a recursive algorithm that implements the greedy strategy.

6. Convert the recursive algorithm to an iterative algorithm.

****Step 1 and 2:

Step 3:

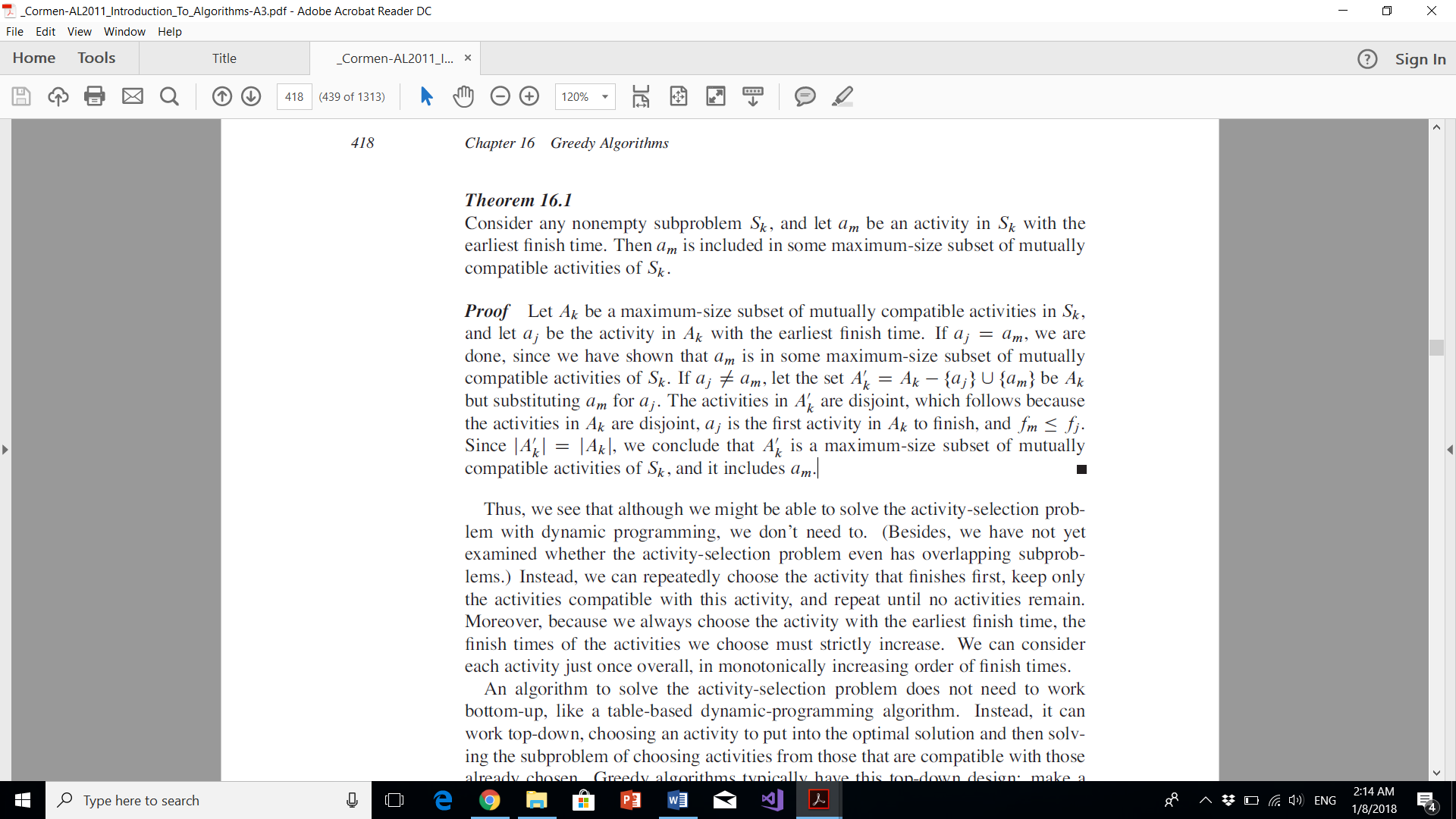
What if we could choose an activity to add to our optimal solution without having

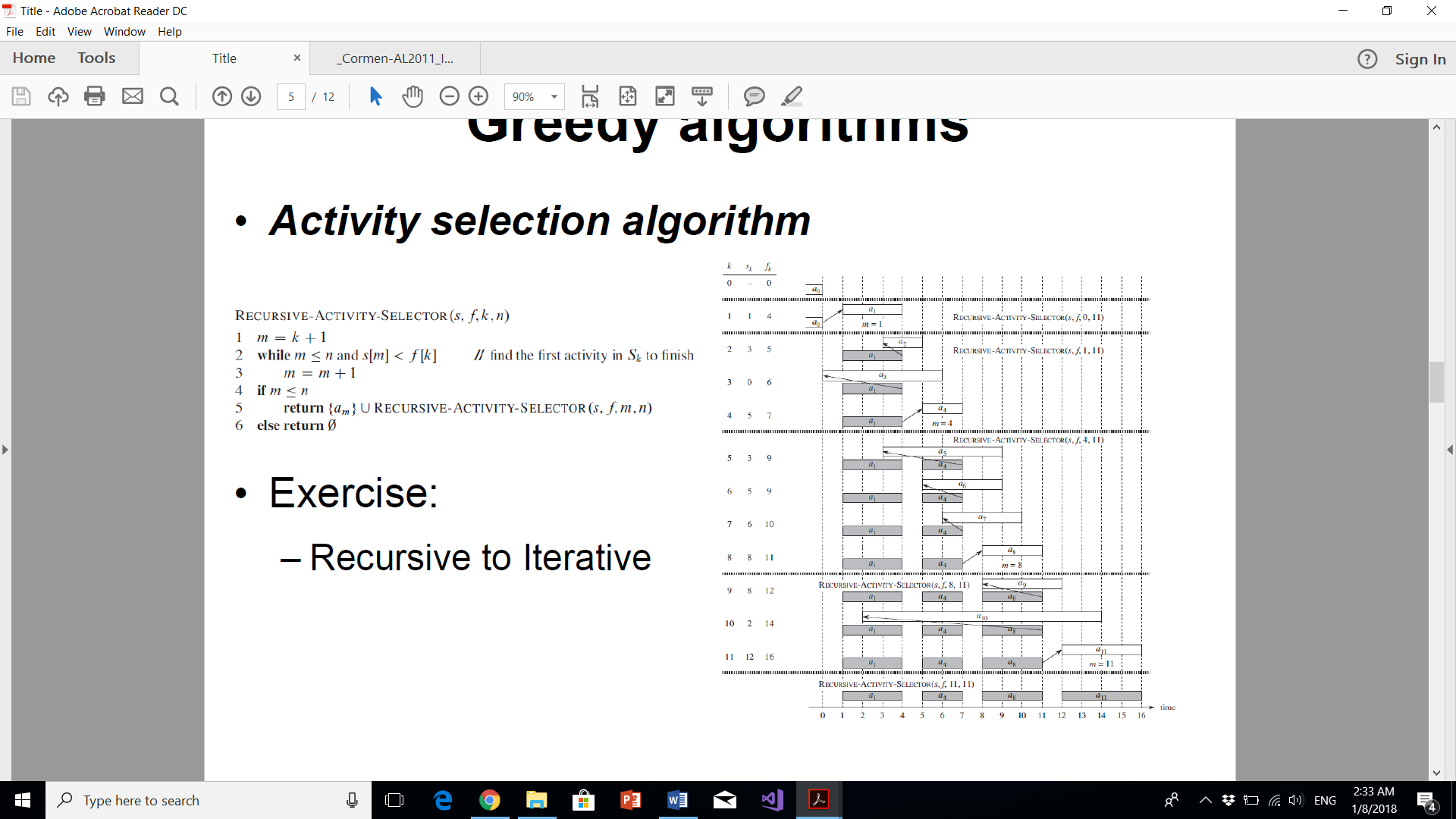
to first solve all the subproblems? That could save us from having to consider all

the choices inherent in recurrence (16.2). In fact, for the activity-selection problem,

we need consider only one choice: the greedy choice.

Step 4 (Proof):

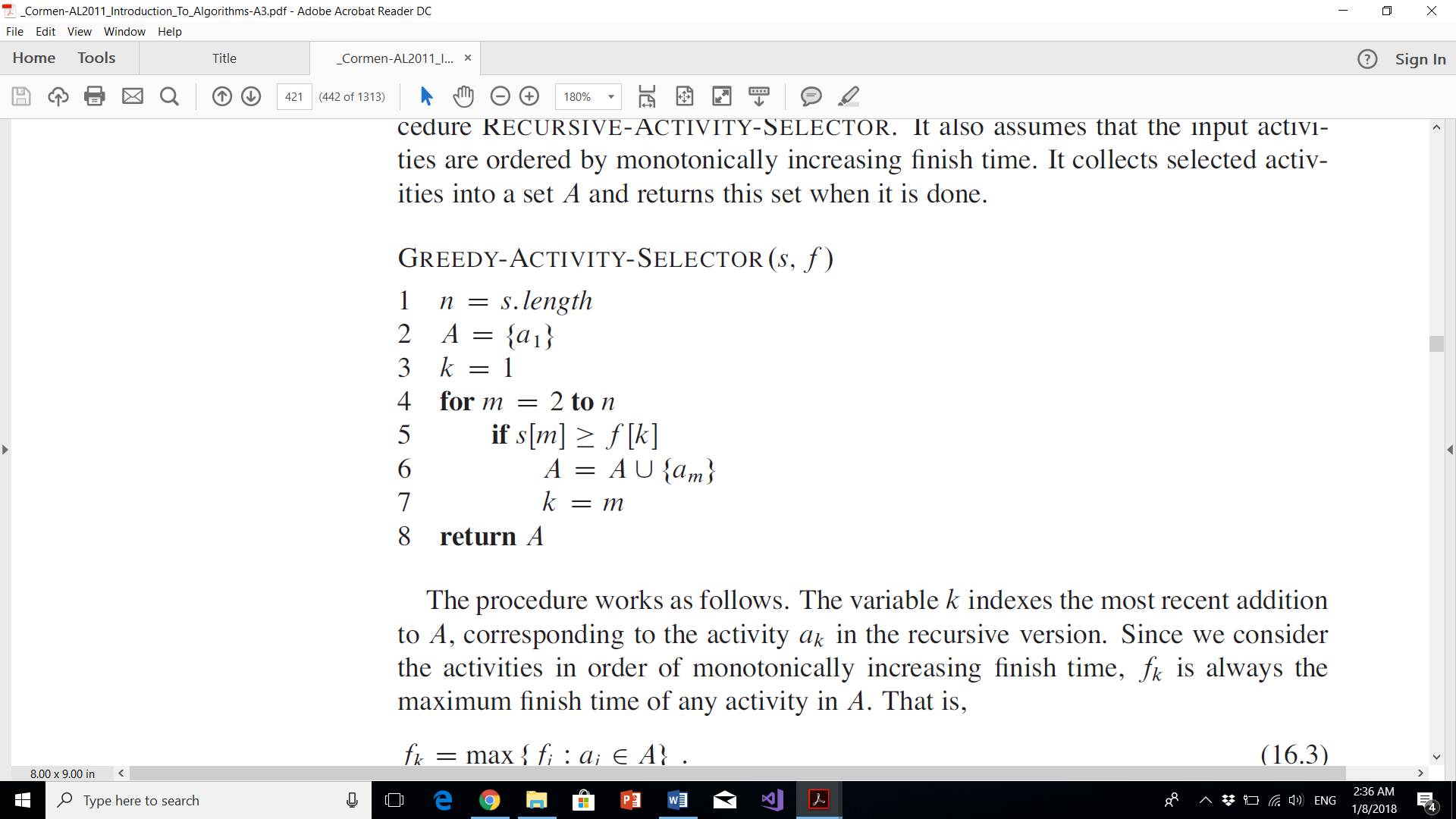


Step 5 (Greedy Recursive Algorithm):

Step 6 (Iterative greedy Algorithm):

We easily can convert our recursive procedure to an iterative one. The procedure

RECURSIVE-ACTIVITY-SELECTOR is almost “tail recursive”, it ends with a recursive call to itself followed by a union operation. It is usually a straightforward task to transform a tail-recursive procedure to an iterative form; in fact, some compilers for certain programming languages perform this task automatically.



**Time Complexity :** It takes O(n log n) time if input activities may not be sorted. It takes O(n) time when it is given that input activities are always sorted.

**0-1 Knapsack:**

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

**Optimal Substructure:**

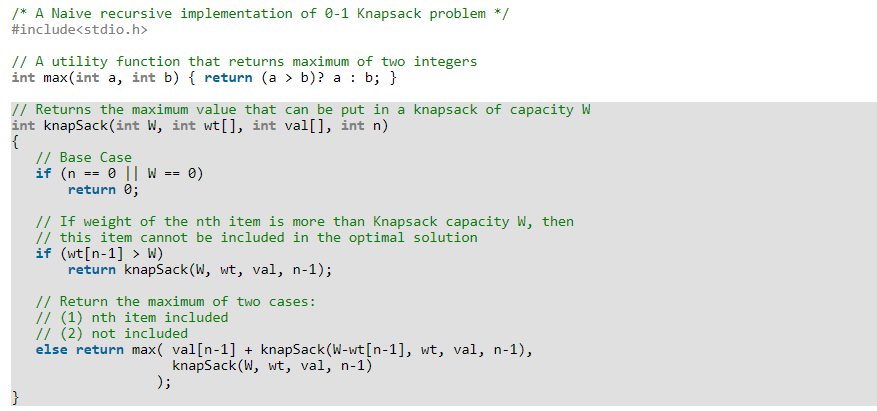
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.  
Therefore, the maximum value that can be obtained from n items is max of following two values.  
1) Maximum value obtained by n-1 items and W weight (excluding nth item).  
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

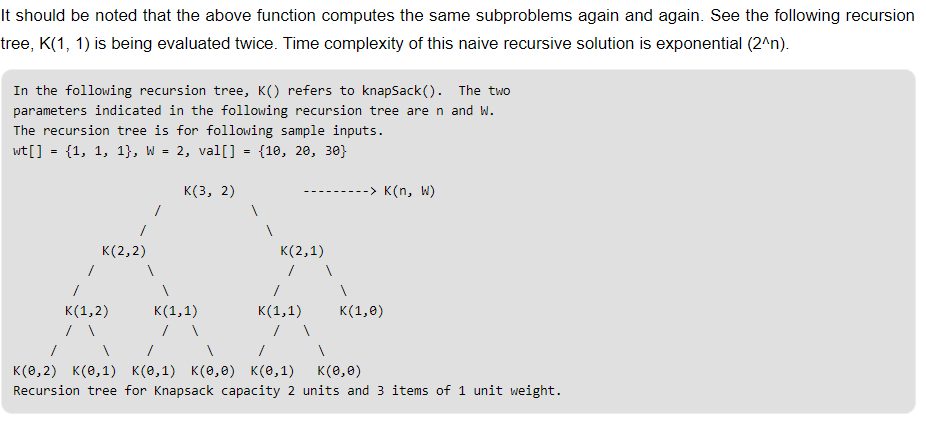
If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

**Overlapping subproblems:**

Following is recursive implementation that simply follows the recursive structure mentioned above

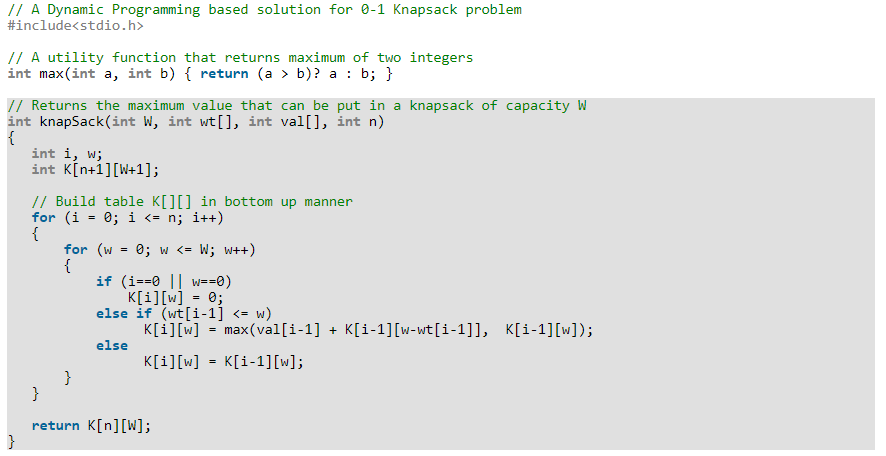
Recursive solution without DP:





Since subproblems are evaluated again, this problem has Overlapping Subproblems property. So the 0-1 Knapsack problem has both properties (Optimal substructure and overlapping subproblems) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

DP solution:



Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.

**Fractional Knapsack:**

Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In the [0-1 Knapsack problem](https://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/), we are not allowed to break items. We either take the whole item or don’t take it.

Input:

Items as (value, weight) pairs

arr[] = {{60, 10}, {100, 20}, {120, 30}}

Knapsack Capacity, W = 50;

Output:

Maximum possible value = 220

by taking items of weight 20 and 30 kg

In **Fractional Knapsack**, we can break items for maximizing the total value of knapsack. This problem in which we can break item also called fractional knapsack problem.

Input :

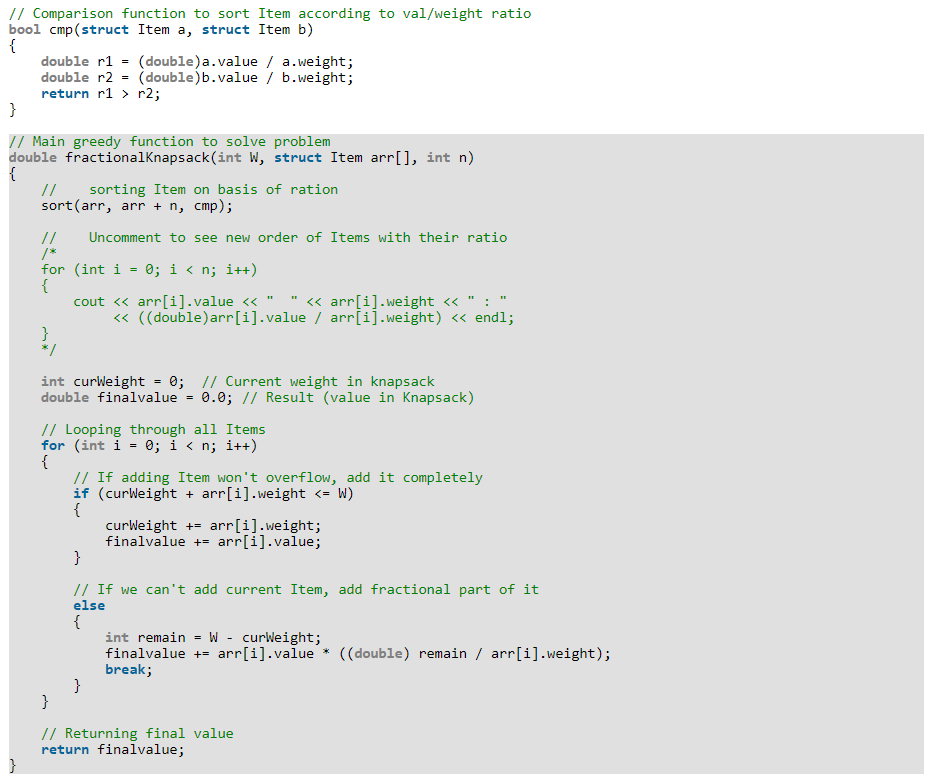
Same as above

Output :

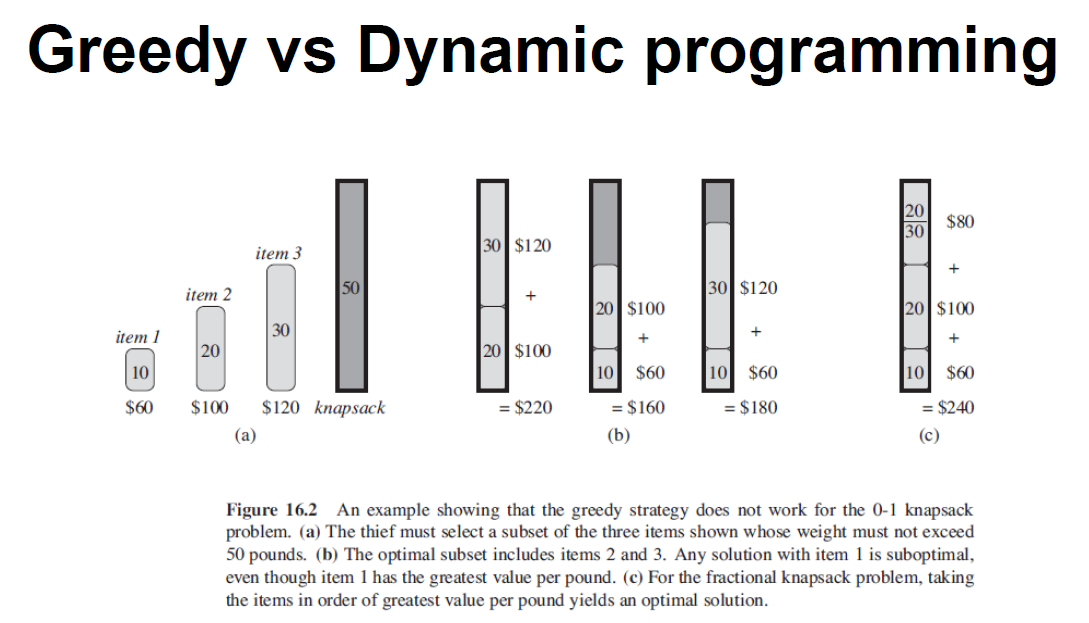
Maximum possible value = 240

By taking full items of 10 kg, 20 kg and

2/3rd of last item of 30 kg

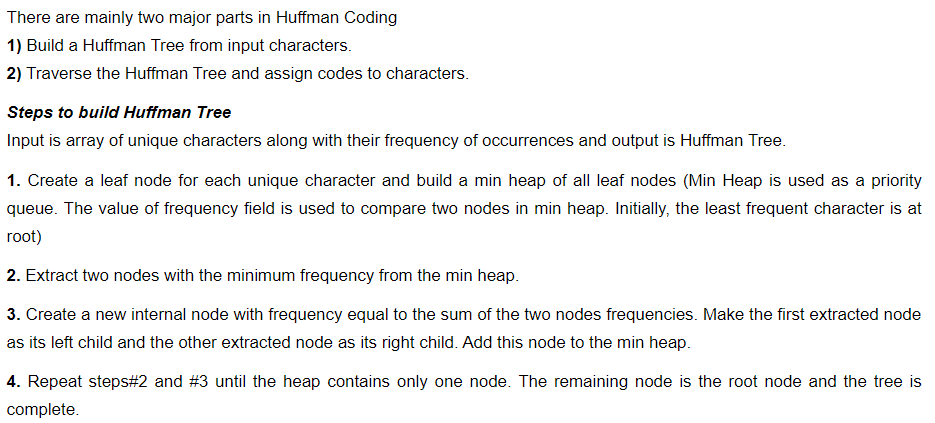


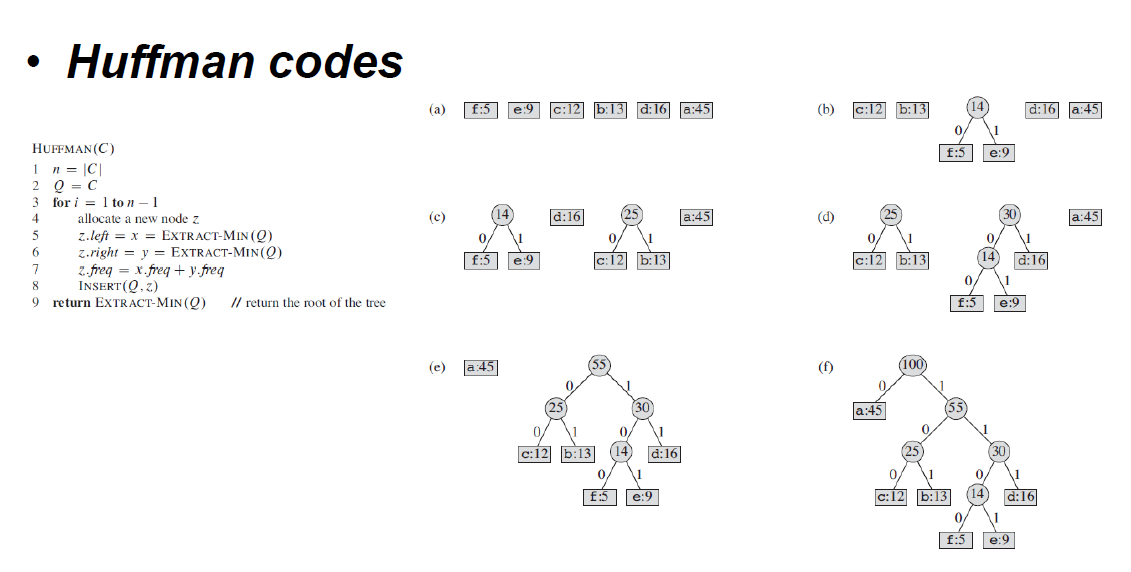
Time Complexity:  O(n log n)

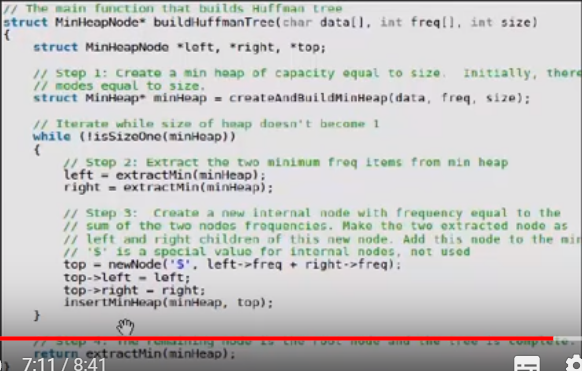


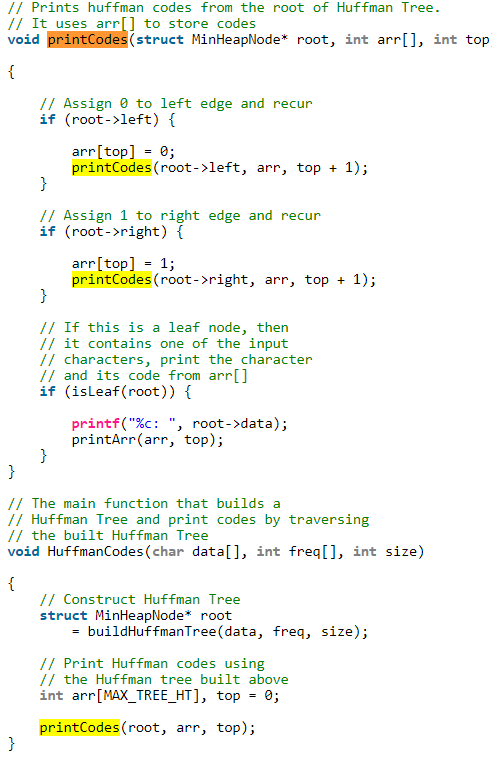
**Huffman Coding:**

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.  
The variable-length codes assigned to input characters are [Prefix Codes](http://en.wikipedia.org/wiki/Prefix_code), means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.  
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.



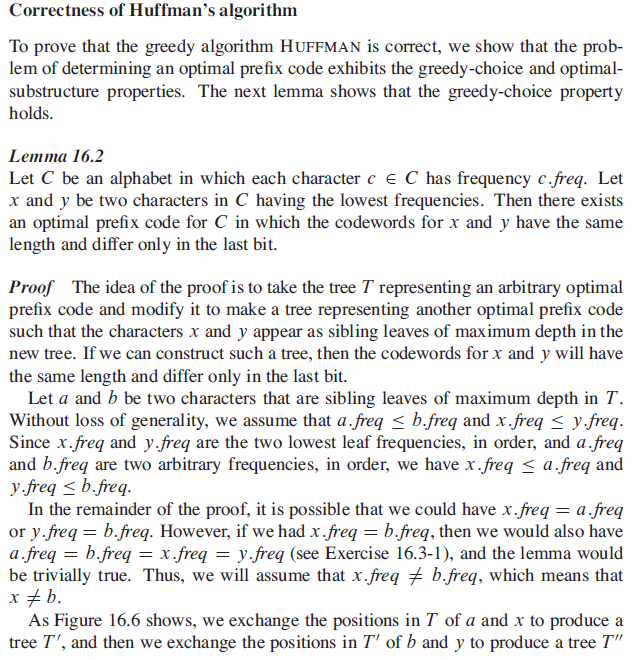


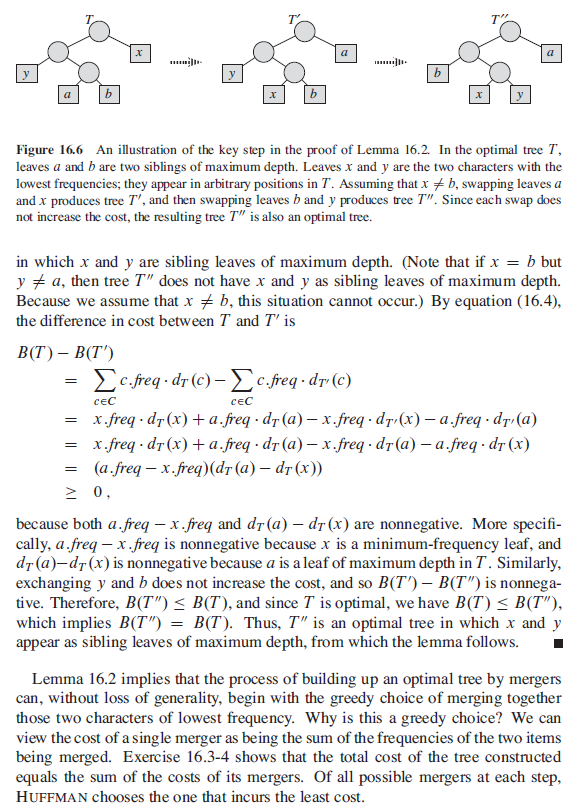


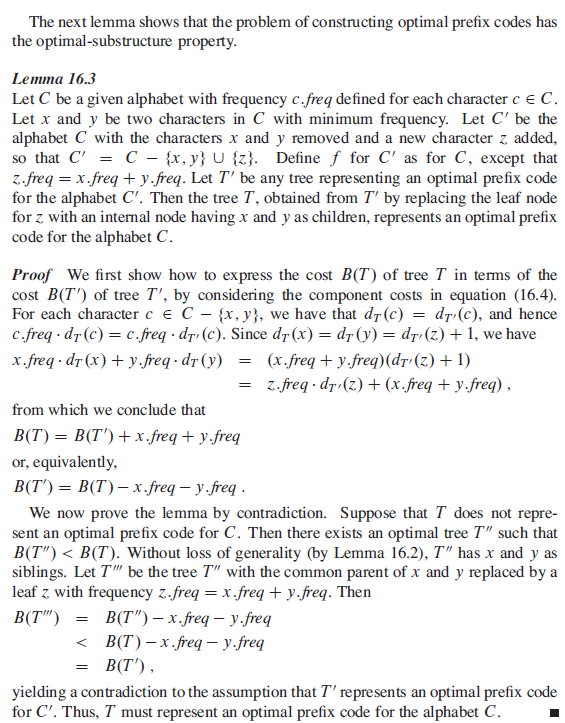


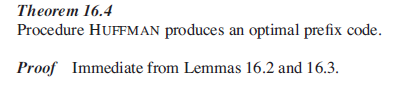
**Time complexity:** O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2\*(n – 1) times. extractMin() takes O(logn) time as it calles minHeapify(). So, overall complexity is O(nlogn).

**Huffman Correctness:**

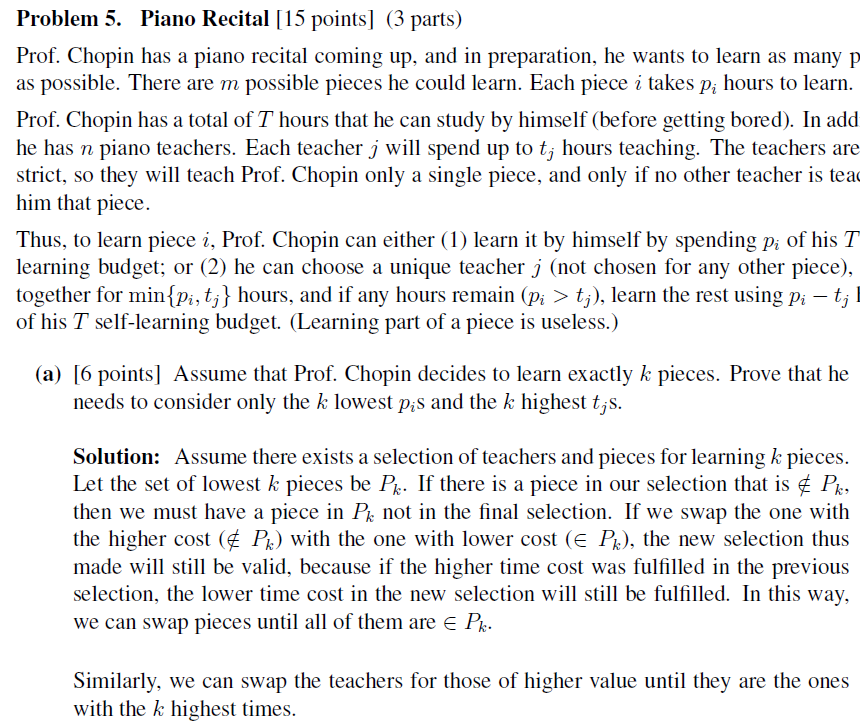


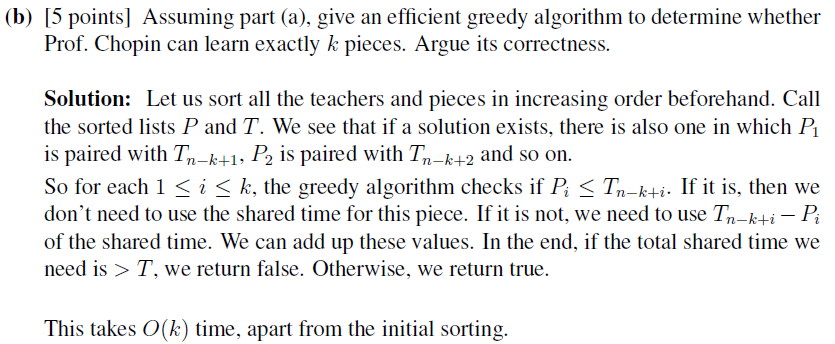


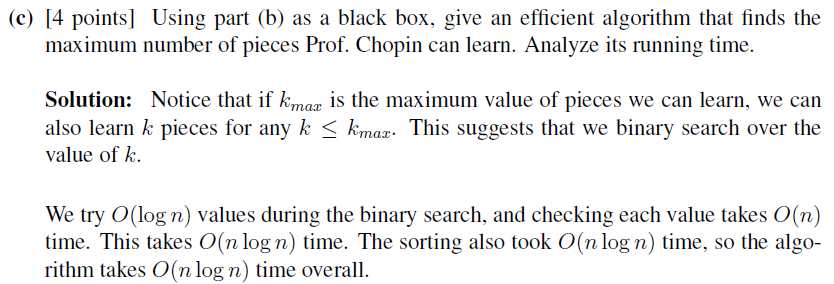


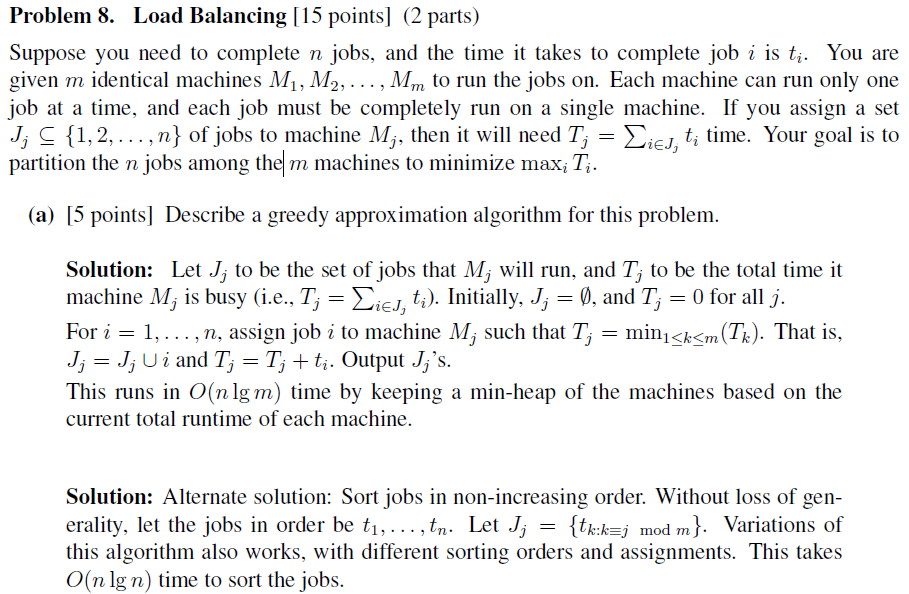


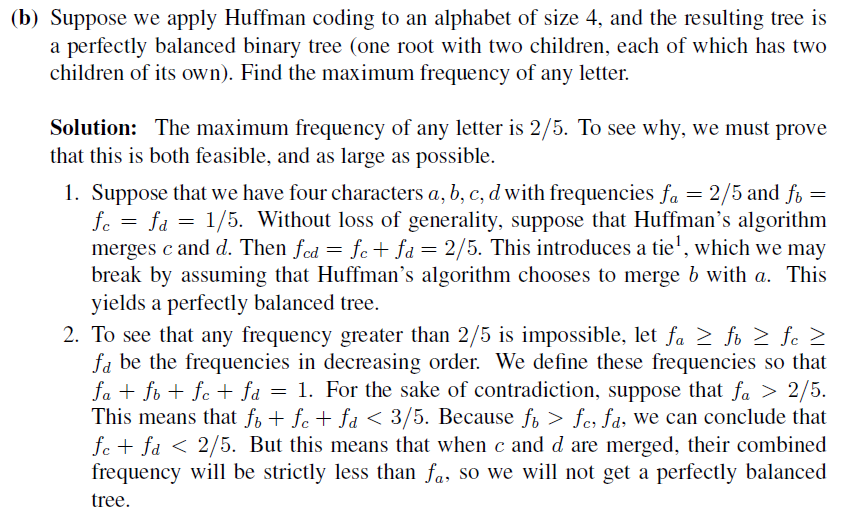
**Exams Problems:**











General rule: X <= 2(1-X)/3

