CS 91 USACO

Bronze Division

Unit 3: Problem Solving Using Algorithms

LECTURE 13: GREEDY ALGORITHMS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Problem Solving: Finding the optimal solution at the solution space.
- Interval Scheduling

Optimization Problems

SECTION 1

Optimization Problems

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

Example: Counting Money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm to do this would be:
  At each step, take the largest possible bill or coin that does not overshoot
  - Example: To make $6.39, you can choose:
    - a $5 bill
    - a $1 bill, to make $6
    - a 25¢ coin, to make $6.25
    - A 10¢ coin, to make $6.35
    - four 1¢ coins, to make $6.39
- For US money, the greedy algorithm always gives the optimum solution

Greedy Algorithm Failure

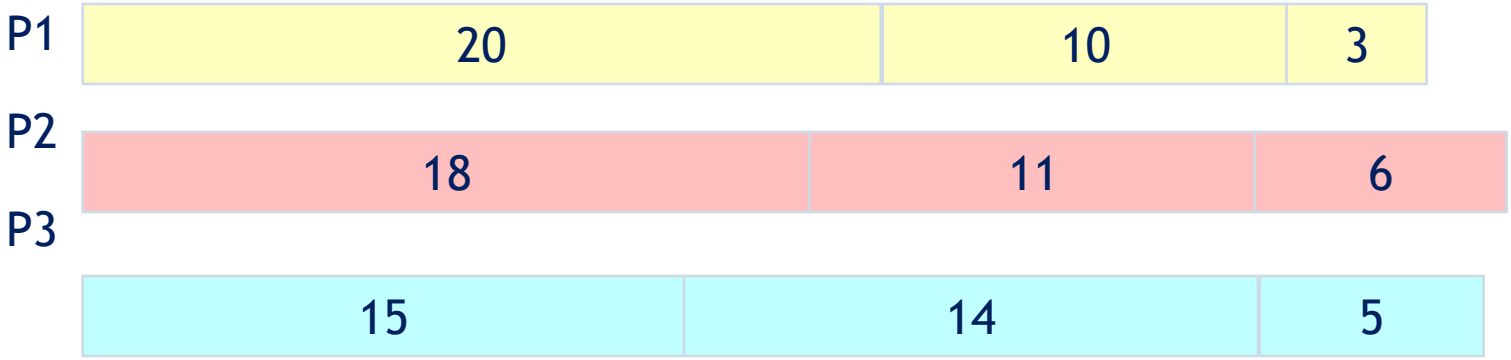
- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces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

Interval Scheduling

SECTION 2

A Scheduling Problem

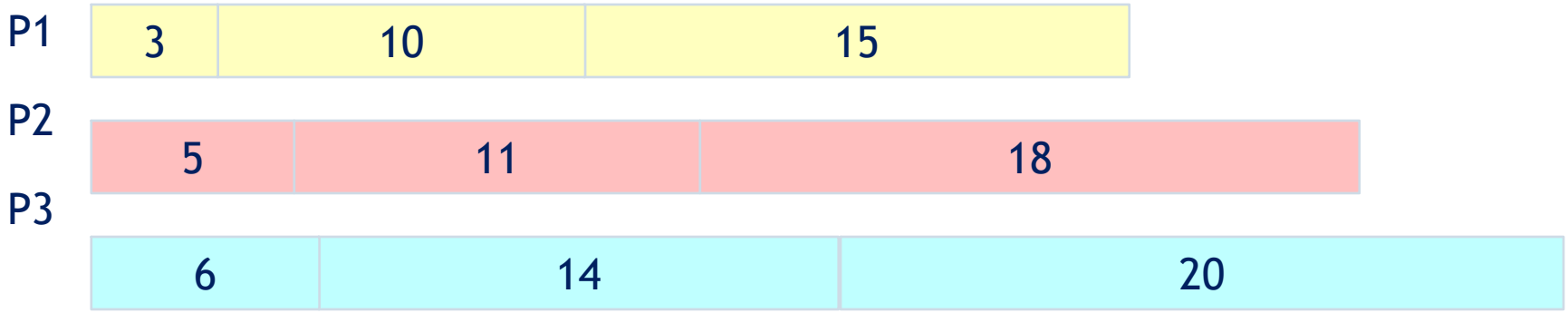
- You have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes.
- You have three processors on which you can run these jobs.
- You decide to do the longest-running jobs first, on whatever processor is available.



- Time to completion: 18 + 11 + 6 = 35 minutes
- This solution isn't bad, but we might be able to do better

Another Approach

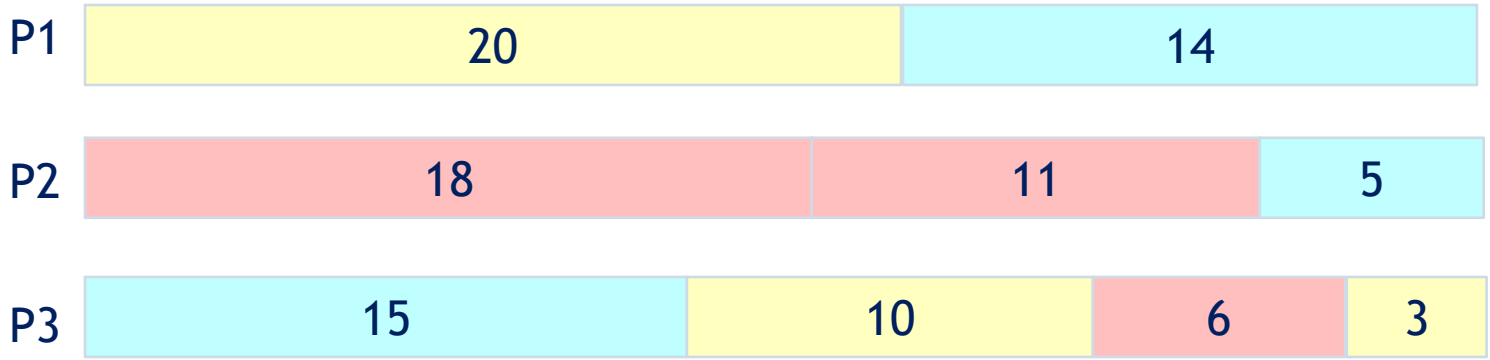
- What would be the result if you ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes



- That wasn't such a good idea; time to completion is now 6 + 14 + 20 = 40 minutes
- Note, however, that the greedy algorithm itself is fast
  - All we had to do at each stage was pick the minimum or maximum

An Optimum Solution

- Better solutions do exist:



  - This solution is clearly optimal (why?)
  - Clearly, there are other optimal solutions (why?)
  - How do we find such a solution?
    - One way: Try all possible assignments of jobs to processors
    - Unfortunately, this approach can take exponential time

```java
import java.util.*;
public class Interval_Scheduling
{
   static Integer[] intervals = {3, 5, 6, 10, 11, 14, 15, 18, 20};

   public static int findMinProcessor(ArrayList<ArrayList<Integer>> processor){
       int min = Integer.MAX_VALUE;
       int minIndex = 0;

       for (int i=0; i<processor.size(); i++){
            int sum = 0;
            for (Integer interval: processor.get(i)){
               sum += interval;
            }
            if (sum<min) { minIndex = i; min = sum; }
        }
       return minIndex;
    }
```

Interval_Scheduling.java

```
  public static int max(ArrayList<Integer> alist){
      int m = Integer.MIN_VALUE;
      for (int i=0; i<alist.size(); i++){
          if (m<alist.get(i)) {
              m = alist.get(i);
           }
       }
      return m;
    }
   public static void main(String[] args){
      System.out.print("\f");
      int N=3;
      ArrayList<ArrayList<Integer>> processor = new ArrayList<ArrayList<Integer>>();
      for (int i=0; i<N; i++) processor.add(new ArrayList<Integer>());
      ArrayList<Integer> ilist = new ArrayList<Integer>(Arrays.asList(intervals));

      int M = intervals.length;
      for (int i=0; i<M; i++){
          int maximum = max(ilist);
          processor.get(findMinProcessor(processor)).add(maximum);
          ilist.remove(new Integer(maximum));
       }
      for (ArrayList<Integer> list: processor){
          System.out.println(list);
       }
    }
}
```

```
[20, 10, 3]
[18, 11, 6]
[15, 14, 5]
```

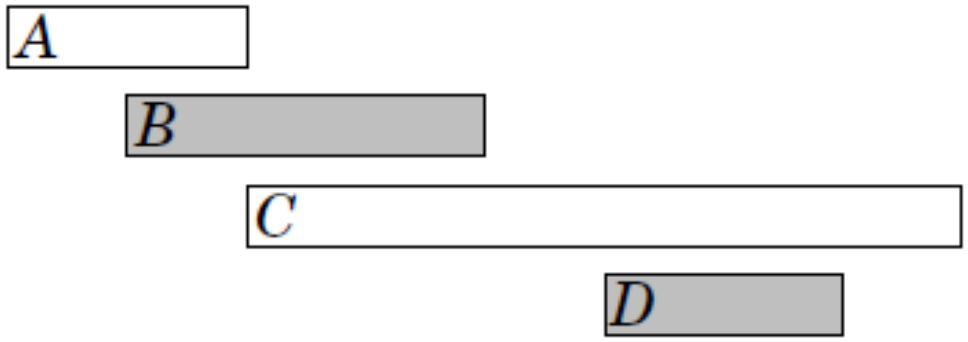
Scheduling 6.2

SECTION 3

Maximum Number of Compatible Intervals

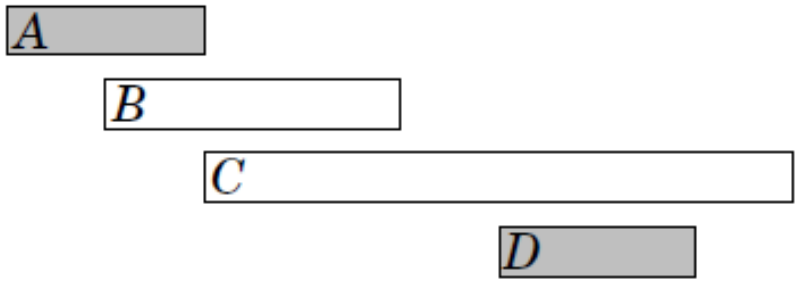
- Many scheduling problems can be solved using greedy algorithms.
- A classic problem is as follows: Given n events with their starting and ending times, our goal is to plan a schedule that includes as many events as possible. It is not possible to select an event partially. For example, consider the following events:

event	starting time	ending time
A	1	3
B	2	5
C	3	9
D	6	8

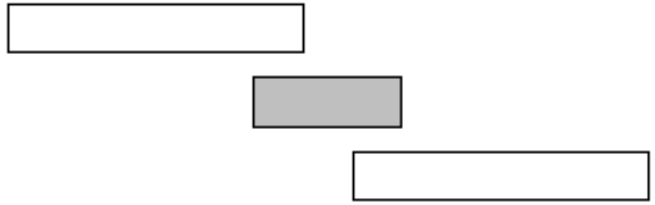


Algorithm 1

The first idea is to select as *short* events as possible. In the example case this algorithm selects the following events:



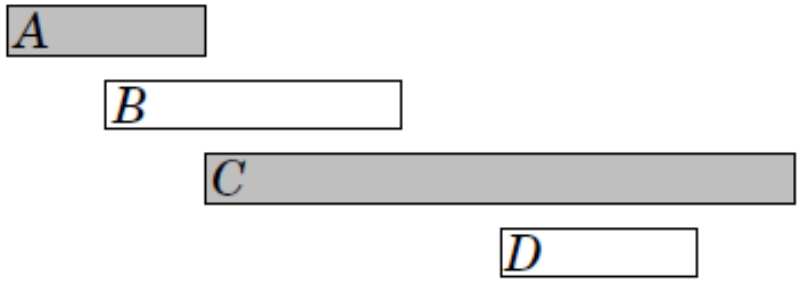
However, selecting short events is not always a correct strategy, but the algorithm fails, for example, in the following case:



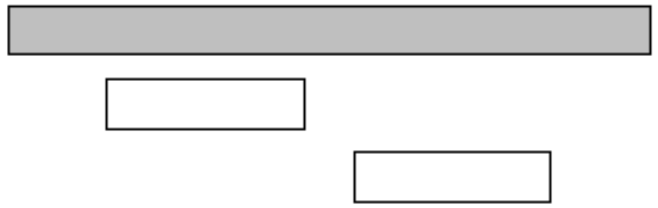
If we select the short event, we can only select one event. However, it would be possible to select both the long events.

Algorithm 2

Another idea is to always select the next possible event that *begins* as *early* as possible. This algorithm selects the following events:



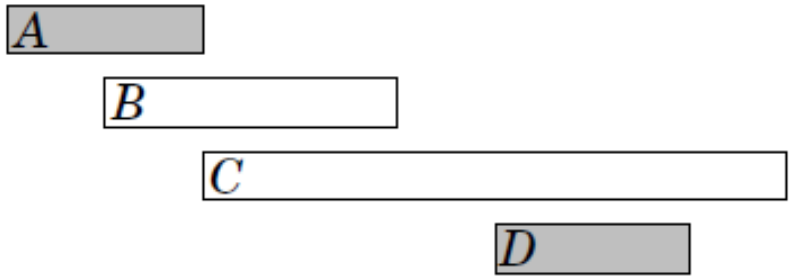
However, we can find a counterexample also for this algorithm. For example, in the following case, the algorithm only selects one event:



If we select the first event, it is not possible to select any other events. However, it would be possible to select the other two events.

Algorithm 3

The third idea is to always select the next possible event that *ends* as *early* as possible. This algorithm selects the following events:



It turns out that this algorithm *always* produces an optimal solution. The reason for this is that it is always an optimal choice to first select an event that ends as early as possible. After this, it is an optimal choice to select the next event using the same strategy, etc., until we cannot select any more events.

One way to argue that the algorithm works is to consider what happens if we first select an event that ends later than the event that ends as early as possible. Now, we will have at most an equal number of choices how we can select the next event. Hence, selecting an event that ends later can never yield a better solution, and the greedy algorithm is correct.

```java
import java.util.*;
public class MaxSchedule{
    public static boolean isCompatible(Interval x, Map<String, Interval> t){
        if (t.size()==0) return true;
        for (String z: t.keySet()){
            if (x.start>=t.get(z).start && x.start<=t.get(z).end
                || x.end >=t.get(z).start && x.end <=t.get(z).end)
                return false;
        }
        return true;
    }

    public static String findEarliest(Map<String, Interval> s, Map<String, Interval> t){
       int early_end = Integer.MAX_VALUE;
       String early = "";
       for (String x: s.keySet()){
            if (s.get(x).end < early_end && isCompatible(s.get(x), t)){
                early = x;
                early_end = s.get(x).end;
            }
       }
       return early;
    }
```

MaxSchedule.java

```
    public static void main(String[] args) {
        System.out.print("\f");

        Map<String, Interval> schedule = new HashMap<String, Interval>();
        schedule.put("A", new Interval(1, 3));
        schedule.put("B", new Interval(2, 5));
        schedule.put("C", new Interval(3, 9));
        schedule.put("D", new Interval(6, 8));

        Map<String, Interval> max_schedule = new HashMap<String, Interval>();
        String e = findEarliest(schedule, max_schedule);
        while (!e.equals("")) {
            max_schedule.put(e, schedule.get(e));
            schedule.remove(e);
            e = findEarliest(schedule, max_schedule);
        }

        System.out.println(max_schedule);
        System.out.println(max_schedule.size());
    }
}
```

MaxSchedule.java

```java
public class Interval implements Comparable<Interval>{
    int start=0, end=0;
    Interval(int s, int e){ start = s; end = e; }
    public int compareTo(Interval other){
        if (this.start > other.start) return 1;
        if (this.start < other.start) return -1;
        return 0;
    }
    public String toString(){ return "<"+start+","+end+">"; }
}
```

Interval.java

Tasks and Deadlines 6.3

SECTION 4

Tasks and deadlines

Let us now consider a problem where we are given n tasks with durations and deadlines and our task is to choose an order to perform the tasks. For each task, we earn $d - x$ points where d is the task's deadline and x is the moment when we finished the task. What is the largest possible total score we can obtain?

For example, suppose that the tasks are as follows:

task	duration	deadline
A	4	2
B	3	5
C	2	7
D	4	5

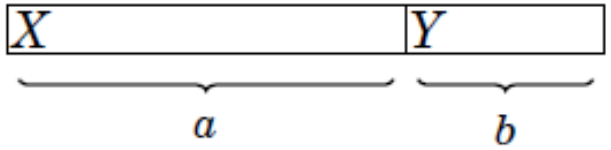
In this case, an optimal schedule for the tasks is as follows:

In this solution, C yields 5 points, B yields 0 points, A yields -7 points and D yields -8 points, so the total score is -10.

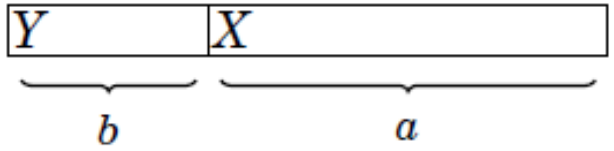
Tasks and deadlines

- Surprisingly, the optimal solution to the problem does not depend on the deadlines at all, but a correct greedy strategy is to simply perform the tasks sorted by their durations in increasing order.
- The reason for this is that if we ever perform two tasks one after another such that the first task takes longer than the second task, we can obtain a better solution if we swap the tasks. For example, consider the following schedule:

Tasks and deadlines



Here $a > b$, so we should swap the tasks:



Now X gives b points fewer and Y gives a points more, so the total score increases by $a - b > 0$. In an optimal solution, for any two consecutive tasks, it must hold that the shorter task comes before the longer task. Thus, the tasks must be performed sorted by their durations.

```java
import java.util.*;
public class Tasks
{
   public static void main(String[] args){
        System.out.print("\f");

        SortedMap<String, Interval> tasks = new TreeMap<String, Interval>();
        tasks.put("A", new Interval(4, 2));
        tasks.put("B", new Interval(3, 5));
        tasks.put("C", new Interval(2, 7));
        tasks.put("D", new Interval(4, 5));

        System.out.println(tasks);
    }
}
```

Tasks.java

Minimizing Sums
6.4

SECTION 5

Minimizing Sums

Solved Problem [Minimum Sum of Distance]

We will next consider a problem where we are given n numbers $a_1, a_2, \ldots, a_n$ and our task is to find a value x that minimizes the sum

$$|a_1 - x|^c + |a_2 - x|^c + \cdots + |a_n - x|^c.$$

We will focus on the cases $c = 1$ and $c = 2$.

Minimizing Sums

Solved Problem

Case $c = 1$

In this case, we should minimize the sum

$$|a_1 - x| + |a_2 - x| + \cdots + |a_n - x|.$$

For example, if the numbers are $[1,2,9,2,6]$, the best solution is to select $x = 2$ which produces the sum

$$|1-2| + |2-2| + |9-2| + |2-2| + |6-2| = 12.$$

In the general case, the best choice for x is the *median* of the numbers, i.e., the middle number after sorting. For example, the list $[1,2,9,2,6]$ becomes $[1,2,2,6,9]$ after sorting, so the median is 2.

The median is an optimal choice, because if x is smaller than the median, the sum becomes smaller by increasing x, and if x is larger then the median, the sum becomes smaller by decreasing x. Hence, the optimal solution is that x is the median. If n is even and there are two medians, both medians and all values between them are optimal solutions.

Minimizing Sums

Solved Problem

Case $c = 2$

In this case, we should minimize the sum

$$(a_1 - x)^2 + (a_2 - x)^2 + \cdots + (a_n - x)^2.$$

For example, if the numbers are $[1,2,9,2,6]$, the best solution is to select $x = 4$ which produces the sum

$$(1-4)^2 + (2-4)^2 + (9-4)^2 + (2-4)^2 + (6-4)^2 = 46.$$

In the general case, the best choice for x is the *average* of the numbers. In the example the average is $(1+2+9+2+6)/5 = 4$. This result can be derived by presenting the sum as follows:

$$nx^2 - 2x(a_1 + a_2 + \cdots + a_n) + (a_1^2 + a_2^2 + \cdots + a_n^2)$$

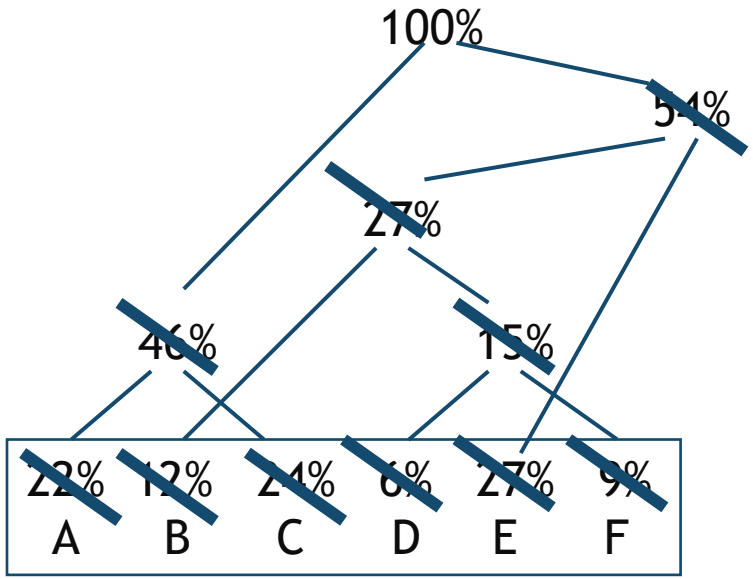
The last part does not depend on x, so we can ignore it. The remaining parts form a function $nx^2 - 2xs$ where $s = a_1 + a_2 + \cdots + a_n$. This is a parabola opening upwards with roots $x = 0$ and $x = 2s/n$, and the minimum value is the average of the roots $x = s/n$, i.e., the average of the numbers $a_1, a_2, \ldots, a_n$.

Huffman Encoding

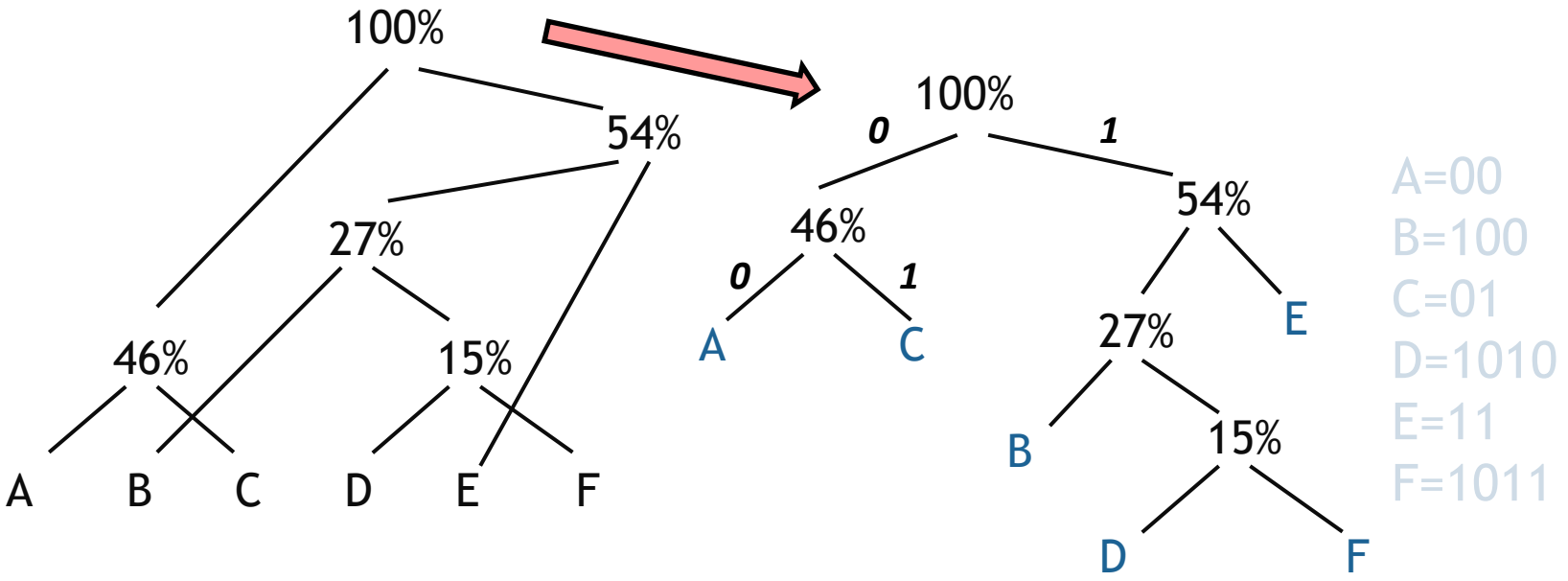
SECTION 6

Huffman encoding

- The Huffman encoding algorithm is a greedy algorithm
- Given the percentage the each character appears in a corpus, determine a variable-bit pattern for each char.
- You always pick the two smallest percentages to combine.



Huffman Encoding



- Average bits/char:
  0.22*2 + 0.12*3 + 0.24*2 + 0.06*4 + 0.27*2 + 0.09*4 = 2.42
- The solution found doing this is an optimal solution.
- The resulting binary tree is a **full tree**.

Analysis

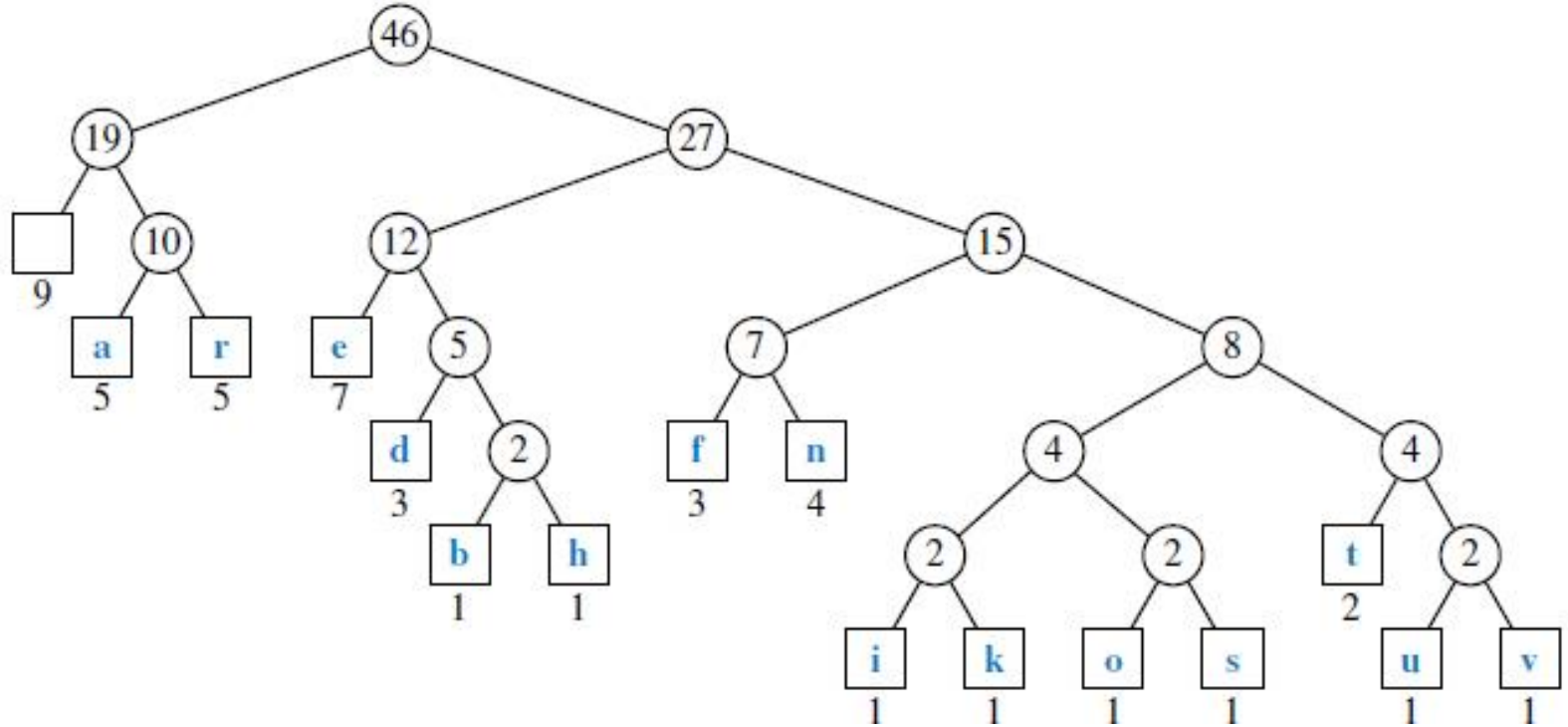
- A greedy algorithm typically makes (approximately) n choices for a problem of size n
  - (The first or last choice may be forced)
- Hence the expected running time is:
  O(n * O(choice(n))), where choice(n) is making a choice among n objects
  - Counting: Must find largest useable coin from among k sizes of coin (k is a constant), an O(k)=O(1) operation;
    - Therefore, coin counting is (n)
  - Huffman: Must sort n values before making n choices
    - Therefore, Huffman is O(n log n) + O(n) = O(n log n)

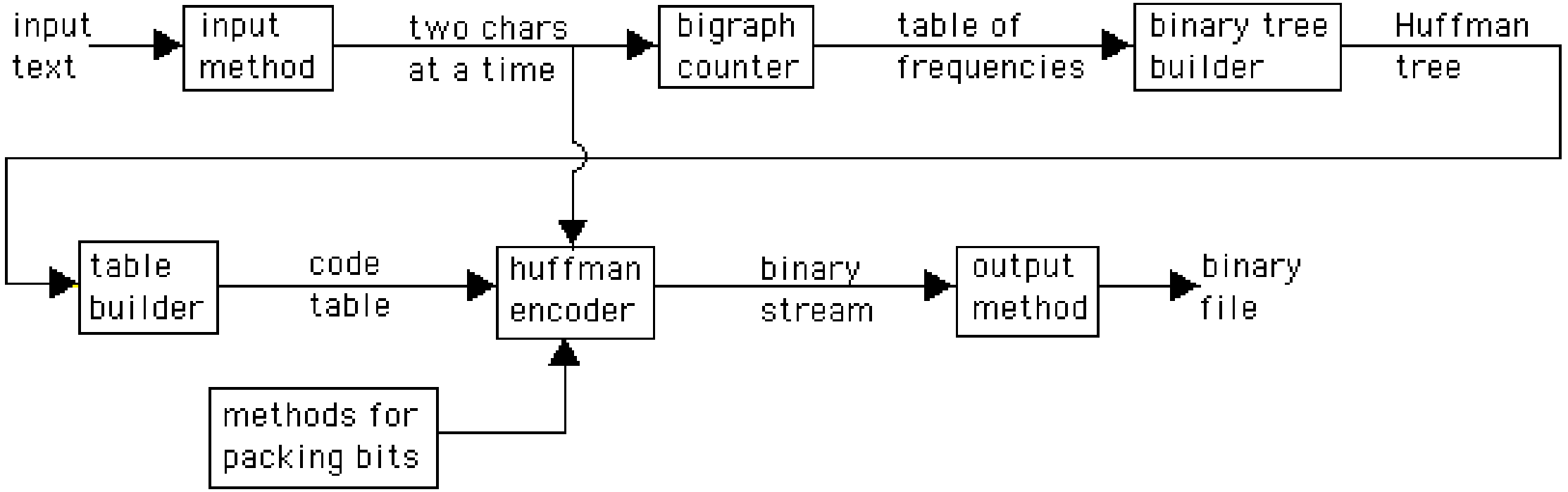
Huffman Encoding/Decoding

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)




input text
input method
two chars at a time
bigraph counter
table of frequencies
binary tree builder
Huffman tree
table builder
code table
huffman encoder
binary stream
output method
binary file
methods for packing bits

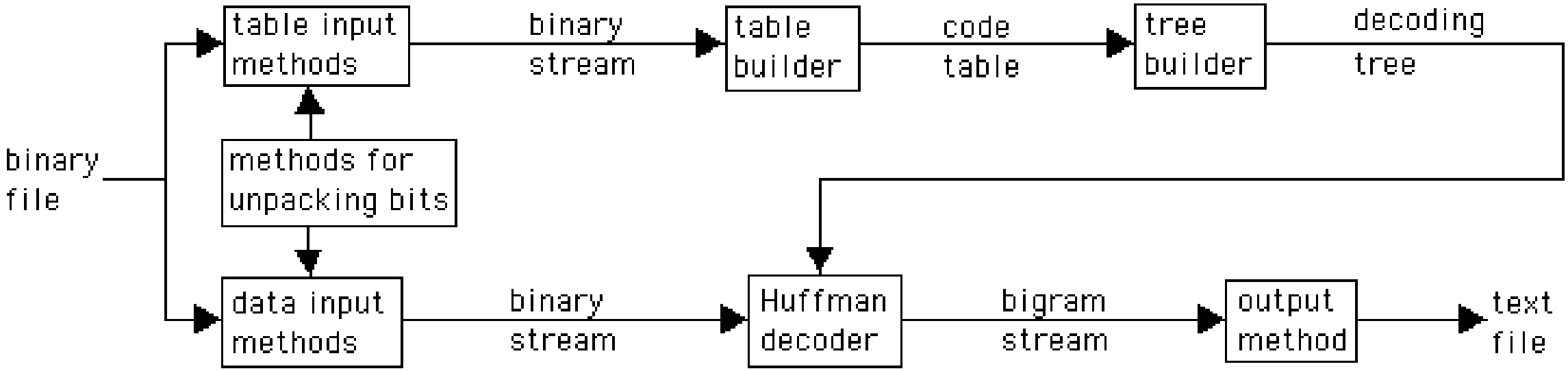


table input methods
binary stream
table builder
code table
tree builder
decoding tree
binary file
methods for unpacking bits
data input methods
binary stream
Huffman decoder
bigram stream
output method
text file

Other Greedy Algorithms

SECTION 7

Other Greedy Algorithms

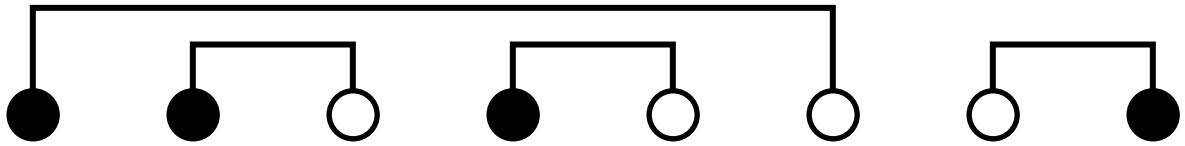
- **Dijkstra's** algorithm for finding the shortest path in a graph
  - Always takes the *shortest* edge connecting a known node to an unknown node
- **Kruskal's** algorithm for finding a minimum-cost spanning tree
  - Always tries the *lowest-cost* remaining edge
- **Prim's** algorithm for finding a minimum-cost spanning tree
  - Always takes the *lowest-cost* edge between nodes in the spanning tree and nodes not yet in the spanning tree

Connecting Wires

There are n white dots and n black dots, equally spaced, in a line

You want to connect each white dot with some one black dot, with a minimum total length of "wire"

Example:



Total wire length above is 1 + 1 + 1 + 5 = 8

Do you see a greedy algorithm for doing this?

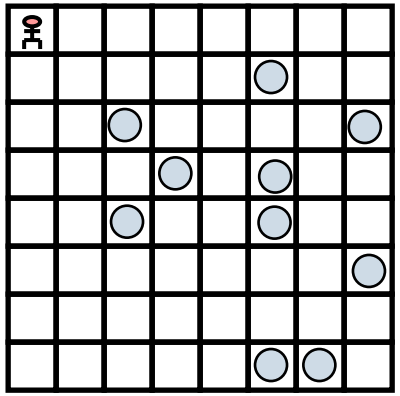
Does the algorithm guarantee an optimal solution?

- Can you prove it?
- Can you find a counterexample?

Collecting Coins

- A checkerboard has a certain number of coins on it
- A robot starts in the upper-left corner, and walks to the bottom left-hand corner
  - The robot can only move in two directions: right and down
  - The robot collects coins as it goes
- You want to collect *all* the coins using the *minimum* number of robots

Example:



Do you see a greedy algorithm for doing this?

Does the algorithm guarantee an optimal solution?

- Can you prove it?
- Can you find a counterexample?

Knapsack Problem

SECTION 8

0-1 Knapsack Problem

- 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).

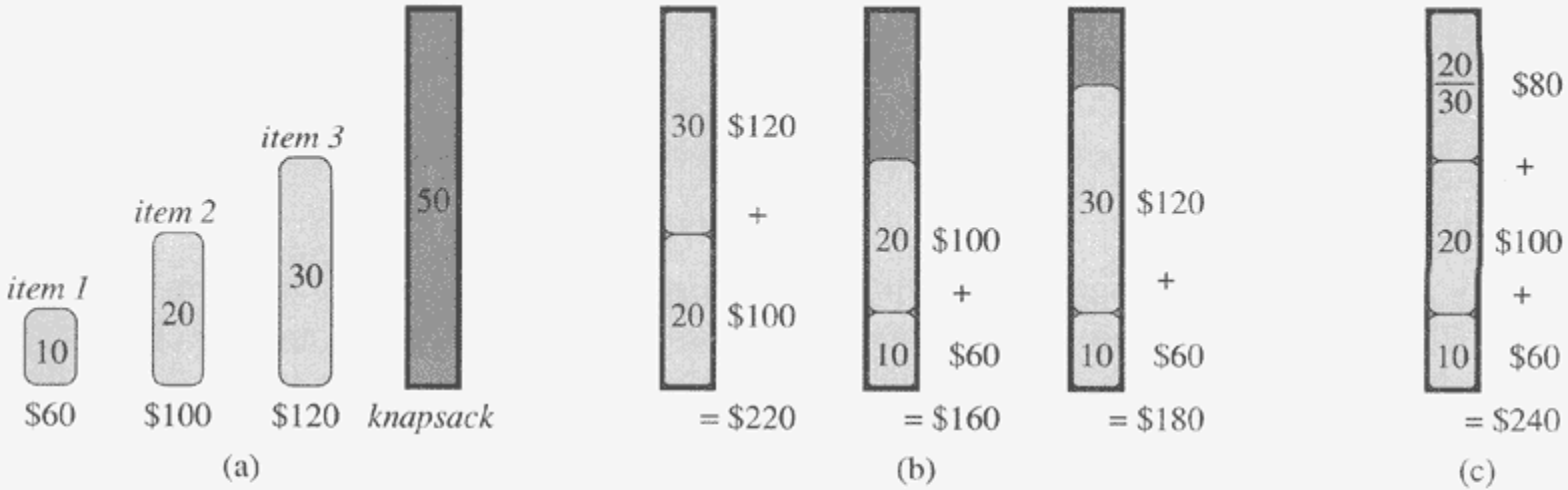


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Knapsack

- 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.

0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

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

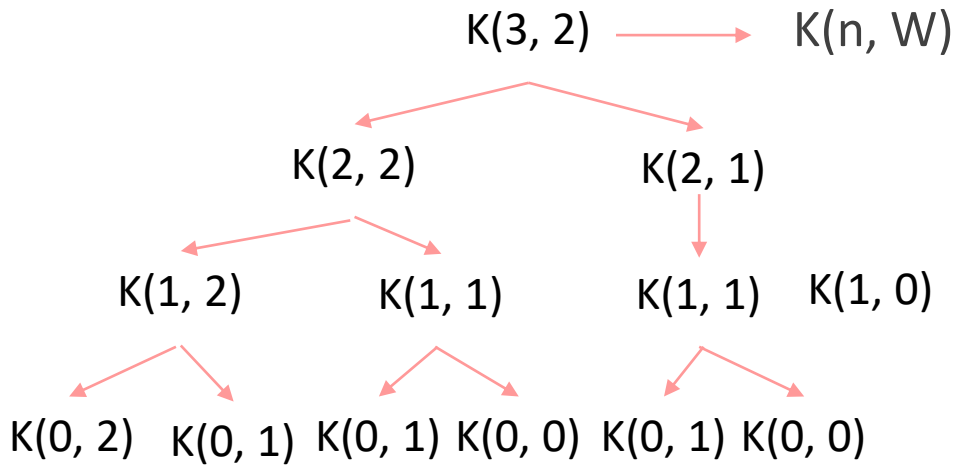
- Following is recursive implementation that simply follows the recursive structure mentioned above.

Overlapping Subproblems

- It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice.
- Time complexity of this naive recursive solution is exponential (2^n).

knapSack()

- In the following recursion tree, K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W.
- The recursion tree is for following sample inputs.
- wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}



- Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

Dynamic Programming

- Since suproblems are evaluated again, this problem has Overlapping Subprolems property.
- So the 0-1 Knapsack problem has both properties (see this and this) of a dynamic programming problem.
- Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

```java
import java.util.*;
public class Knapsack{
  public static int knapSack(int W, int[] wt, int[] val, int n){
      int[][] K = new int[n+1][W+1];
      for (int i=0; i<n+1; i++){
         for (int j=0; j<W+1; j++){
            if (i == 0 || j == 0)
                K[i][j] = 0;
            else if (wt[i - 1] <= j)
                K[i][j] = Math.max(val[i-1]+K[i-1][j - wt[i-1]], K[i-1][j]);
            else
                K[i][j] = K[i-1][j];
         }
      }
      return K[n][W];
    }
  public static void main(String[] args){
      System.out.print("\f");
      int[] val = {60, 100, 120}, wt  = {10, 20, 30};
      int W = 50, n = val.length;
      System.out.println(knapSack(W, wt, val, n));
    }
}
```

Knapsack.java

Summary

SECTION 9

Other Algorithm Categories

- Brute Force
  - Selection Sort
- Divide-and-Conquer
  - Quicksort
- Decrease-and-Conquer
  - Insertion Sort
- Transform-and-Conquer
  - Heapsort
- Dynamic Programming
- Greedy Algorithms
- Iterative Improvement
  - Simplex Method, Maximum Flow

From The Design & Analysis of Algorithms, Levitin

Practice: Mixing Milk (milk)

SECTION 1

Problem Statement

- The Merry Milk Makers company buys milk from farmers, packages it into attractive 1- and 2-Unit bottles, and then sells that milk to grocery stores so we can each start our day with delicious cereal and milk.
- Since milk packaging is such a difficult business in which to make money, it is important to keep the costs as low as possible. Help Merry Milk Makers purchase the farmers' milk in the cheapest possible manner. The MMM company has an extraordinarily talented marketing department and knows precisely how much milk they need each day to package for their customers.

Problem Statement

- The company has contracts with several farmers from whom they may purchase milk, and each farmer has a (potentially) different price at which they sell milk to the packing plant. Of course, a herd of cows can only produce so much milk each day, so the farmers already know how much milk they will have available.
- Each day, Merry Milk Makers can purchase an integer number of units of milk from each farmer, a number that is always less than or equal to the farmer's limit (and might be the entire production from that farmer, none of the production, or any integer in between).

Problem Statement

- Given:
  - The Merry Milk Makers' daily requirement of milk
  - The cost per unit for milk from each farmer
  - The amount of milk available from each farmer
- calculate the minimum amount of money that Merry Milk Makers must spend to meet their daily need for milk.
- Note: The total milk produced per day by the farmers will always be sufficient to meet the demands of the Merry Milk Makers even if the prices are high.

INPUT FORMAT (milk.in):

- Line 1:
  - Two integers, N and M. The first value, N, (0 <= N <= 2,000,000) is the amount of milk that Merry Milk Makers wants per day.
  - The second, M, (0 <= M <= 5,000) is the number of farmers that they may buy from.
- Lines 2 through M+1:
  - The next M lines each contain two integers: P_i and A_i. P_i (0 <= P_i <= 1,000) is price in cents that farmer i charges.
  - A_i (0 <= A_i <= 2,000,000) is the amount of milk that farmer i can sell to Merry Milk Makers per day.

INPUT FORMAT (milk.in):

SAMPLE INPUT:

```
100 5
5 20
9 40
3 10
8 80
6 30
```

```
100 5 -- MMM wants 100 units of milk from 5 farmers
5 20 -- Farmer 1 says, "I can sell you 20 units at 5 cents per unit"
9 40 etc.
3 10 -- Farmer 3 says, "I can sell you 10 units at 3 cents per unit"
8 80 etc.
6 30 -- Farmer 5 says, "I can sell you 30 units at 6 cents per unit"
```

OUTPUT FORMAT (milk.out):

- A single line with a single integer that is the minimum cost that Merry Milk Makers must pay for one day's milk.

SAMPLE OUTPUT:

```
630
```

Nature of the Problem

- Get all milk from the cheapest source until the goal is met.
- A farmer class with price and available milk is needed.
- Sort the farmer by the milk price in ascending order.
- Add up the milk volume and price until the milk acquisition goal is met.
- Return the minimum cost.