Greedy Algorithms

Optimization Problems

- Some problems can have many possible/ feasible solutions with each solution having a specific cost. We wish to find the best solution with the optimal cost.
  - *Maximization problem* finds a solution with maximum cost
  - *Minimization problem* finds a solution with minimum cost
- A set of choices must be made in order to arrive at an optimal (min/max) solution, subject to some constraints.
- Is “Sorting a sequence of numbers” optimization problem?

Optimization Problems

- Two common techniques:
  - Greedy Algorithms (local)
    - Make the greedy choice and THEN
    - Solve sub-problem arising after the choice is made
    - The choice we make may depend on previous choices, but not on solutions to sub-problems
    - Top down solution, problems decrease in size
  - Dynamic Programming (global)
    - We make a choice at each step
    - The choice depends on solutions to sub-problems
    - Bottom up solution, smaller to larger sub-problems

Greedy Algorithm

- A greedy algorithm works in phases. At each phase:
  - makes the best choice available right now, without regard for future consequences
  - hopes to end up at a *global* optimum by choosing a *local* optimum at each step. For some problem, it works
- Greedy algorithms sometimes works well for optimization problems
- Greedy algorithms tend to be easier to code
- Greedy algorithms frequently used in everyday problem solving
  - Choosing a job
  - Route finding
  - Playing cards
  - Invest on stocks

Greedy Algorithm

- How to know if a greedy algorithm will solve a particular optimization problem?
- Two key ingredients
  - Greedy choice property
  - Optimal sub-structure

- If a problem has these properties, then we can develop a greedy algorithm for it.

Greedy Algorithm

- Greedy choice property: A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
  - Make whatever choice seems best at the moment and then solve the sub-problem arising after the choice is made
  - The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems

- Optimal sub-structure: A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub-problems

Counting Money Problem

Counting Money Problem

Suppose you want to count out a certain amount of money, using the fewest possible bills and coins

A greedy algorithm would do this would be:

At each step, take largest possible bill/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 optimum solution

Counting Money Problem

Greedy algorithm (C, N)

1. sort coins so C1 ≥ C2 ≥ . . . ≥ Ck
2. S = Φ;
3. Change = 0
4. i = 1 \\ Check for next coin
5. **while** Change ≠ N do \\ all most valuable coins
6. **if** Change + Ci ≤ N **then**
7. Change = Change + Ci
8. S = S ∪ {Ci}
9. **else** *i = i+1*

Counting Money Problem

In Pakistan, our currency notes are

$C_1 = 5000$, $C_2 = 1000$, $C_3 = 500$, $C_4 = 100$,

$C_5 = 50$, $C_6 = 20$, $C_7 = 10$

Applying above greedy algorithm to N = 13,660, we get

$S = \{C_1, C_1, C_2, C_2, C_2, C_3 , C_4 , C_5 , C_7\}$

Does this algorithm always find an optimal solution?

For Pakistani currency the greedy algorithm always gives optimum solution.

Counting Money Problem

A failure of the greedy algorithm:

In some (fictional) monetary system, “kron” comes in 1 kron, 7 kron, and 10 kron coins.

Applying greedy algorithm to count out 15 krons, we 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, requiring only tree coins

The greedy algorithm results in a solution, but not in an optimum solution.

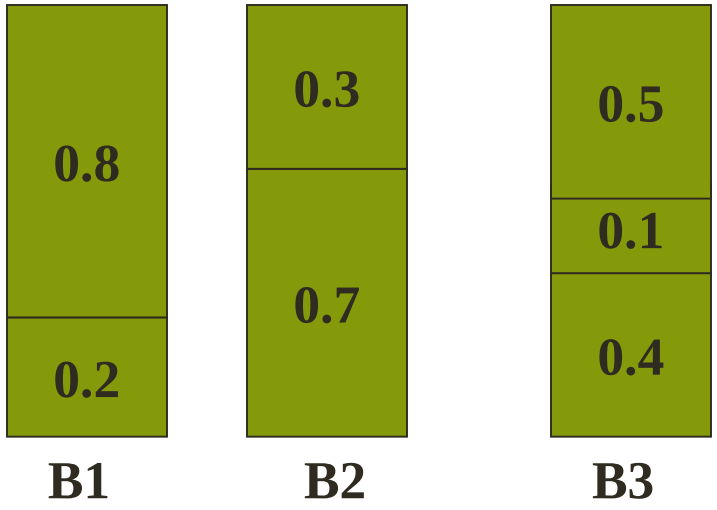
Bin Packing Problem

Approximate Bin Packing Problem

We are given n items of sizes S_1, S_2,…..,S_n. All sizes satisfy $0 < S_i <= 1$.

The problem is to pack these items in the fewest number of bins, given that each bin has unit capacity.

Suppose we have item list with sizes 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8. Optimal packing of these sizes is



Approximate Bin Packing Problem

Algorithm 1: Next Fit

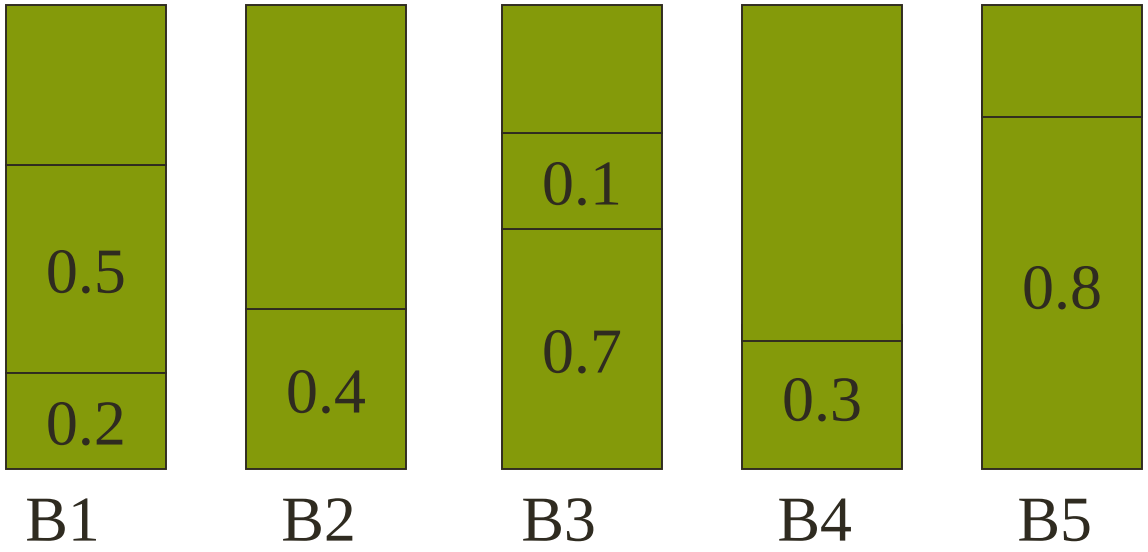
When processing any item, we check to see whether it fits in the same bin as the last item; if it does it is placed there; otherwise a new bin is created.

```
void NextFit ( )
{   read item1;
    while ( read item2 ) {
        if ( item2 can be packed in the same bin as item1 )
                place item2 in the bin;
        else
                create a new bin for item2;
        item1 = item2;
    } /* end-while */
}
```

【Theorem】 Let *M* be the optimal number of bins required to pack a list *I* of items. Then *next fit* never uses more than $2M$ bins. There exist sequences such that *next fit* uses $2M - 2$ bins.

Approximate Bin Packing Problem

Next Fit for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8 is



Approximate Bin Packing Problem

Algorithm 2: First Fit

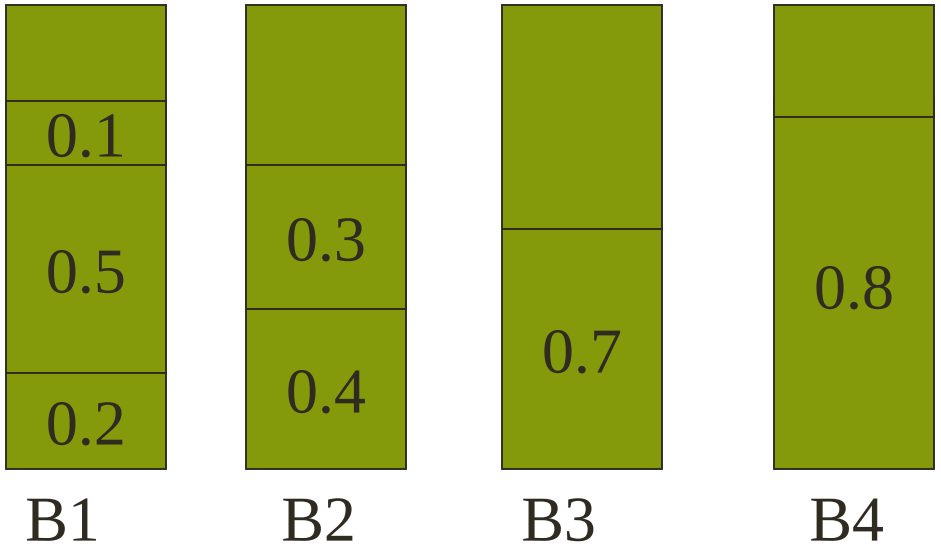
- We scan the bins in order and place the new item in the first bin that is large enough to hold it. A new bin is created only when the result of previous placements have left no other alternative.

```
void FirstFit ( )
{   while ( read item ) {
        scan for the first bin that is large enough for item;
        if ( found )
                place item in that bin;
        else
                create a new bin for item;
    } /* end-while */
}
```

【Theorem】 Let M be the optimal number of bins required to pack a list I of items. Then *first fit* never uses more than $\lceil 17M / 10 \rceil$ bins. There exist sequences such that *first fit* uses $17(M - 1) / 10$ bins.

Approximate Bin Packing Problem

First Fit for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8 is

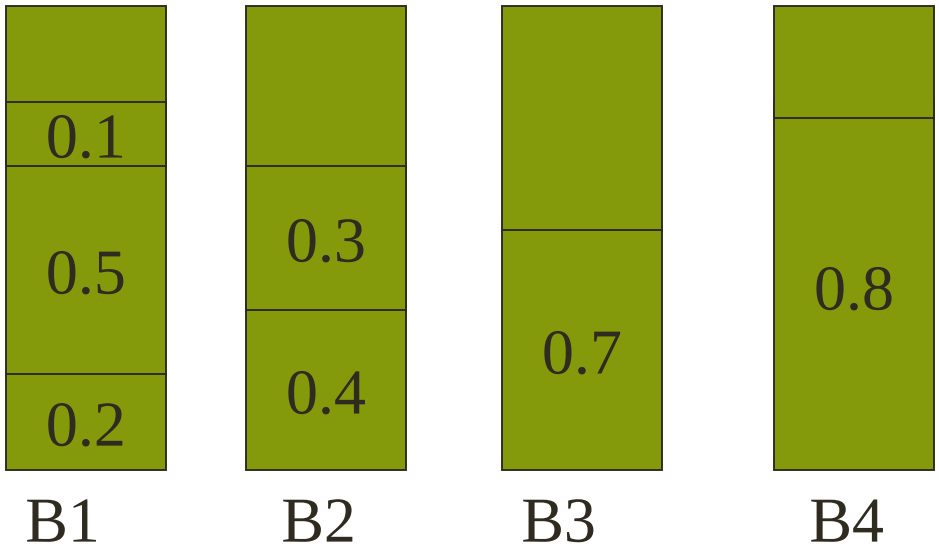


Approximate Bin Packing Problem

Algorithm 3: Best Fit

- Instead of placing a new item in the first spot that is found. It is placed in the tightest spot among all bins

Best Fit for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8 is



T = O(N log N) and bin no. < 1.7M

Approximate Bin Packing Problem

Offline Algorithms

- The major problem with all the on-line algorithms is that it is hard to pack the large items, especially when they occur late in the input.
- The natural way around this is to sort the items, placing the largest items first.

- Then apply first fit or best fit, i.e.
  - Best Fit Decreasing
  - First Fit Decreasing

Approximate Bin Packing Problem

Offline Algorithm

- Best Fit Decreasing for 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1 is