Topic 25: Approximation Algorithms -- SOLUTIONS

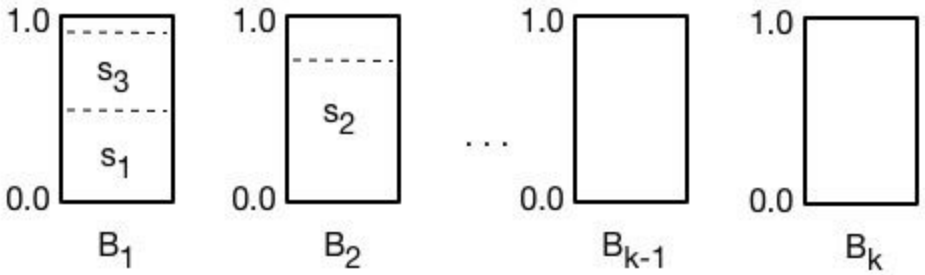
It's Moving Day!

Suppose we have *n* objects of different sizes s_1, s_2, … s_n, and we want to pack them into boxes -- which we will call "bins"-- using as few bins as possible. We can get as many bins B_1, B_2, B_3 … as we need. To simplify the problem, we will treat all objects as one-dimensional and scale the sizes such that

- the size of each object s_i is $0 < s_i < 1$
- the size of each bin is 1.0 (so for each bin B_i, $\sum s_j \in B_i \leq 1$)

Define $S = \sum s_i$, the sum of the sizes of all *n* objects that we want to pack.

We want to find the minimum *k* where bins B_1, B_2, B_3 … B_k hold all the objects.



The corresponding decision problem (whether the n objects will fit in k bins) is called the **Bin Packing Problem** and is NP-Complete (by reduction of Subset Sum, Partition, or 3-Partition). Thus, finding the minimum k is NP-Hard: we probably need an approximation algorithm to get an efficient (polynomial time) solution.

1. Specify a polynomial time approximation algorithm for bin packing. It should be a 2-approximation algorithm (see next question). *Do it in these steps:*

a. Give pseudocode of an approximation algorithm for bin packing:
You will place objects s_1, s_2, … s_n in bins B_1, B_2, … B_k and return k, the number of bins used. You may use informal pseudocode, for example, "if object s_i fits in bin B_j then put s_i in B_j", "allocate a new bin B_k", etc.

This is based on the greedy heuristic:

Since you want to minimize the number of bins, don't start a new bin if the next object being placed fits into one that has already been started.

```
Bin-Packing-Approx ((s1, s2, … sn), n) // returns final k
1  k = 1
2  allocate a new bin B1
3  for i = 1 to n     // place each of the n si
4       j = 1
5       placed = FALSE
6       while (j <= k) and not placed
7            if object si fits in bin Bj then
8                 put si in Bj
9             placed = TRUE
10           j = j + 1
11      if not placed   // place si in new bin
12           k = k + 1
13           Allocate a new bin Bk
14           put si in Bk
15 return k
```

Your code may differ!

Some student groups found an alternative solution that may not be as optimal as the above, but is faster and is still within the 2-approximation bound: sort the items to pack, and then fill each bin with the smallest items remaining first before starting a new bin. You never need to go back, because larger items will not fit in prior bins if smaller ones did not. For example:

```
Bin-Packing-Approx ((s1, s2, … sn), n) // returns final k
1  sort (s1, s2, … sn) from smallest to largest
2  k = 1
3  allocate a new bin B1
4  for i = 1 to n:
5         if object si fits in bin Bk then
6              put si in Bk
7         else
8              k = k + 1
9              allocate new bin Bk
10             put si in Bk
11  return k
```

A counter-example to optimality is 10 objects of size 0.1 and 10 objects of size 0.9. This can be packed into 10 bins by pairing smaller objects with larger ones, but the above algorithm puts all 10 0.1 in the first bin and then needs 10 more. However, the first greedy algorithm above may have other anomalies, and we have not yet formalized a way to compare their performance.

b. Analyze the asymptotic time complexity of your algorithm in terms of n. (It is sufficient to show it is polynomial: the analysis need not be a tight bound.)

> **Solution:** The first solution above is $O(n^2)$: The outer loop executes $\Theta(n)$ times. Since k cannot be greater than n, the inner loop is O(n). (This is a loose bound that can be tightened with some work, but it is sufficient to show it is polynomial, which is all that is required.) All the other work is constant time.
>
> The second solution above is O(n). The O(n) loop in line 4 has constant operations. The sort takes O(n lg n) with a comparison-based sort, but since the keys are in range [0,1] we can use Bucket Sort (see Topic 10 notes) in O(n) time.

2. Prove that your approximation algorithm is a 2-approximation algorithm. In other words, if your algorithm uses C bins and the optimal number of bins is C*, show that $C \leq 2C^*$ bins. Since we don't know C* we prove it relative to a lower bound on C*. *Use the following steps. Hints are given as questions that you should answer here.*

a. Write a lower bound on C* as an inequality in terms of S. But first, answer this question: Can you pack S amount of stuff into fewer than S bins? (S is the sum of object sizes.)

> **Solution (either algorithm):** No you can't. Each bin has capacity 1.0, so S bins have capacity S, and cannot hold more than S stuff. This means that the optimal number of bins C* cannot be less than S. So the inequality is:
>
> $$S \leq C^*$$

b. Write an upper bound on C as an inequality in terms of S. But first, answer this question: How many nonempty bins does your algorithm leave no more than half-full, and why?

Solution (first algorithm): There are no more than 1 nonempty bins no more than half full. Proof by contradiction: if more than one bin was half full or less, the contents of the later bin in the sequence, being ≤ 0.5, can fit into the earlier bin, which has ≥ 0.5 remaining. The "first fit" algorithm (implemented as lines 6-7 in the code above) would have placed these contents in the earlier bin, as it is encountered first, so the later bin would not have been allocated or needed.

Since only one bin can be no more than half full, to construct the worst case we can have all the bins filled to ½ + ε capacity for arbitrarily small but nonzero ε, and we need 2S bins to fit it all. So the inequality is:

$$C < 2S$$

(Strict inequality because if C = 2S then we have S stuff in 2S bins, so ½S stuff in each bin on average, meaning we would have more than 1 bin half full.)

Solution (second algorithm): For a bin to be no more than half full, by definition it has to have <= 0.5 in it, *and* there must not be any remaining unplaced items s_k in the set of objects with value <= 0.5, because if there were at least one of such items would be placed in this half-full bin due to the sorting, making the bin have > 0.5 in it. So, all remaining items have size > 0.5 and therefore are more than half full. Thus, only one bin can be no more than half full. Again we get C < 2S.

c. Use the answers to (a) and (b) to prove (with simple algebra) that C ≤ 2C*, and hence your algorithm does not require more than twice the lower bound on the number of bins.

Solution: We have C < 2S from (b) and S ≤ C* from (a). Putting the two formulas together, we get C < 2S ≤ 2C*. Thus, **C < 2C*:** our approximation algorithm is a 2-approximation algorithm, being bounded by twice the optimal solution.