Problem Set 4—Due Monday, March 7. 3PM

This problem set will consider approximate solutions to NP-hard problems.

The first problem we will be addressing is **bin packing**. This problem is intended to explore some properties of bin packing algorithms, as well as getting more experience with timing and testing algorithms.

You are to implement various bin-packing algorithms. In each setting you will be packing integers in the range $1..k$ and packing them into bins of capacity k. For each setting do your experiments using random integers (you can modify part of the randgraph.c code from ps2 to get random integers in the desired range), and try your experiments both for $k = 100$ and $k = 1,000$. Also, do all your experiments on list of at least 100,000 items.

Your goal is to pack the items as quickly as possible for the given strategy (and report your times), and also to find good strategies for packing the items effectively (using as few bins as possible). Thus you should report how well you have packed the items (a good measure is the amount of waste space per bin: e.g. if you pack items into 13 bins each of capacity 100 and the sum of your item sizes is 1,170, then the average waste fraction is $(1300 - 1170)/1300 = .1$.

In designing your algorithms you may exploit the fact that you are packing uniform random numbers.

1[32]) On-line bin-packing. For this setting you must pack each item before seeing the rest of the list. However, you can keep as many bins active as you like and you may assume that you know n the number of items to be packed.

i) Use First-Fit (FF) to pack the items. Note: FF tells you WHERE to put the next item (the first bin into which it fits), but not HOW to find this bin quickly. In order to make this work quickly you should use some extra data structure(s) to find the proper FF bin.

ii) Find a better on-line scheme for this setting (better meaning it has less waste than FF and is still on-line and fast).

2[33]) Off-line bin-packing. You can look at the entire list before packing the items.

i) Use First-Fit decreasing to pack the items.

ii) Try to find a better (and still fast) packing scheme (this may be hard to do since FFD should work quite well).

3[15]) Optional extra projects:

i) Consider distributions that are not uniform over the entire range (e.g. values uniform on the range 20..100 for bins of capacity 100).

ii) Your solutions to A/B could exploit the fact that the item sizes were integers in a known range. Generalize your solutions to work for arbitrary integer values.

You should turn in:

I) Your code for 1) and 2) (emailed to Sam so he can actually run your code.)

II) A high level description of how you implemented the algorithms to find the first bin that fits, and your improved algorithms.

III) A summary of your timing experiments on programs A/B (a table listing waste as well as the run time would be nice for each implementation).

IV) Your conclusions as to the best algorithm for bin-packing. Briefly justify why you made your choice.

pencil and paper questions:

4[70]) We explore some knapsack issues/extension

A[20]) We showed we could solve the fractional knapsack problem optimally in $O(n \log n)$ time by sorting the v_i/w_i ratios, then including all the $b-1$ best items, and just enough of the bth item. We now consider solving this in $O(n)$ time. Note that we can find the k smallest items in an unsorted list in $O(n)$ time (this can be done using a simple variant of quicksort in expected linear time, for simplicity we will assume $O(n)$ worst case time which is also possible; you don't actually need to know any details about the algorithm. Just assume a function **Select**(A, k) that takes an unsorted array $A[1..n]$, and an integer $1 \leq k \leq n$ and rearranges A so the k smallest values in A are in position $1..k$ (in an arbitrary order) and does this in $O(n)$ time.

Hint: use the **Select** algorithm to find the value of b and the associated ratio.

B[40]) Zero-One knapsack:

The DP algorithm given in 11.8 runs in $O(nV)$ time where V is the number of columns in the table. We must have $V \geq V^*$ where V^* is the optimal value. Since we don't actually know V^* we want an estimate V that is i) as small as possible, and ii) $V \geq V^*$.

The book uses $V = v_1 + \ldots v_n$ which meets ii) above but is not too great for meeting i).

i) show that there is an infinite family of inputs (where the number of items n is growing) where V is significantly larger than V^*, and the ratio V/V^* grows as n grows.

ii) Suggest a rather better way to find V.

iii) The algorithm given in the book finds the optimal **value** V^*, but does not find the actual solution set S. Describe an efficient algorithm to find S once we have constructed the table M and know $V^*$and give its run time.(Hint first figure out the largest index i of an item in the optimal solution, include that item, then continue on a smaller size problem.)

iv) A suggestion is made to discover the number of columns, V, as the algorithm runs (that is we start with a small value of V, we know is achievable, and let the table grow as we discover larger and larger achievable values). We will assume that we can expand the size of our array as needed (ignoring some practical issues with implementing that): that is, if our current array has c columns, we can expand it to have $c' > c$ columns and still keep all our perviously stored data in place..

To use this idea we will assume that an array entry $M[i, j]$ is the smallest weight with value exactly j using a subset of the first i items.

To start with, for the first row we could use $V = v_1$ since we can't get a value greater than that with the first item. $M[1, 0] \leftarrow 0$, $M[1, v_1] \leftarrow w_1$ and the other entries are all set to $W + 1$, which shows it is an invalid solution.

Let V be the number of columns in an already computed row $i-1$. Describe how to compute a new row i, and the (possibly) new number of columns V' in the new row. Briefly justify your solution method and give its run time.

C[10]) The standard 0/1 knapsack problem assumes we have only one copy of each item. We now extend this to the case where we have multiple copies of each item. Now assume that for the *ith* item we can use up to d_i copies of the item (thus if $d_1 = 3$ we can use zero, one, two, or three copies of the item, getting respectively value $0, v_1, 2v_1$, or $3v_1$ and incurring weight $0, w_1, 2w_1$, or $3w_1$.

Formulate this version of the 0/1 knapsack problem as an integer programming problem.

5[20]) Problem 11-6 in the Kleinberg-Tardos Text