If $T + a_i \leq B$ then
  \[ S \leftarrow S \cup \{a_i\} \]
  \[ T \leftarrow T + a_i \]
Endif
Endfor

Give an instance in which the total sum of the set $S$ returned by this algorithm is less than half the total sum of some other feasible subset of $A$.

(b) Give a polynomial-time approximation algorithm for this problem with the following guarantee: It returns a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$. Your algorithm should have a running time of at most $O(n \log n)$.

4. Consider an optimization version of the Hitting Set Problem defined as follows. We are given a set $A = \{a_1, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of $A$. Also, each element $a_i \in A$ has a weight $w_i \geq 0$. The problem is to find a hitting set $H \subseteq A$ such that the total weight of the elements in $H$, that is, $\sum_{a \in H} w_a$, is as small as possible. (As in Exercise 5 in Chapter 8, we say that $H$ is a hitting set if $H \cap B_i$ is not empty for each $i$.) Let $b = \max_i |B_i|$ denote the maximum size of any of the sets $B_1, B_2, \ldots, B_m$. Give a polynomial-time approximation algorithm for this problem that finds a hitting set whose total weight is at most $b$ times the minimum possible.

5. You are asked to consult for a business where clients bring in jobs each day for processing. Each job has a processing time $t_i$ that is known when the job arrives. The company has a set of ten machines, and each job can be processed on any of these ten machines.

At the moment the business is running the simple Greedy-Balance Algorithm we discussed in Section 11.1. They have been told that this may not be the best approximation algorithm possible, and they are wondering if they should be afraid of bad performance. However, they are reluctant to change the scheduling as they really like the simplicity of the current algorithm: jobs can be assigned to machines as soon as they arrive, without having to defer the decision until later jobs arrive.

In particular, they have heard that this algorithm can produce solutions with makespan as much as twice the minimum possible; but their experience with the algorithm has been quite good: They have been running it each day for the last month, and they have not observed it to produce a makespan more than 20 percent above the average load, $\frac{1}{10} \sum_i t_i$. 
To try understanding why they don't seem to be encountering this factor-of-two behavior, you ask a bit about the kind of jobs and loads they see. You find out that the sizes of jobs range between 1 and 50, that is, $1 \leq t_i \leq 50$ for all jobs $i$, and the total load $\sum_i t_i$ is quite high each day: it is always at least 3,000.

Prove that on the type of inputs the company sees, the Greedy-Balance Algorithm will always find a solution whose makespan is at most 20 percent above the average load.

6. Recall that in the basic Load Balancing Problem from Section 11.1, we're interested in placing jobs on machines so as to minimize the makespan—the maximum load on any one machine. In a number of applications, it is natural to consider cases in which you have access to machines with different amounts of processing power, so that a given job may complete more quickly on one of your machines than on another. The question then becomes: How should you allocate jobs to machines in these more heterogeneous systems?

Here's a basic model that exposes these issues. Suppose you have a system that consists of $m$ slow machines and $k$ fast machines. The fast machines can perform twice as much work per unit time as the slow machines. Now you're given a set of $n$ jobs; job $i$ takes time $t_i$ to process on a slow machine and time $\frac{1}{2}t_i$ to process on a fast machine. You want to assign each job to a machine; as before, the goal is to minimize the makespan—that is the maximum, over all machines, of the total processing time of jobs assigned to that machine.

Give a polynomial-time algorithm that produces an assignment of jobs to machines with a makespan that is at most three times the optimum.

7. You're consulting for an e-commerce site that receives a large number of visitors each day. For each visitor $i$, where $i \in [1, 2, \ldots, n]$, the site has assigned a value $v_i$, representing the expected revenue that can be obtained from this customer.

Each visitor $i$ is shown one of $m$ possible ads $A_1, A_2, \ldots, A_m$ as they enter the site. The site wants a selection of one ad for each customer so that each ad is seen, overall, by a set of customers of reasonably large total weight. Thus, given a selection of one ad for each customer, we will define the spread of this selection to be the minimum, over $j = 1, 2, \ldots, m$, of the total weight of all customers who were shown ad $A_j$.

**Example** Suppose there are six customers with values 3, 4, 12, 2, 4, 6, and there are $m = 3$ ads. Then, in this instance, one could achieve a spread of
9 by showing ad $A_1$ to customers 1, 2, 4, ad $A_2$ to customer 3, and ad $A_3$ to customers 5 and 6.

The ultimate goal is to find a selection of an ad for each customer that maximizes the spread. Unfortunately, this optimization problem is NP-hard (you don’t have to prove this). So instead, we will try to approximate it.

(a) Give a polynomial-time algorithm that approximates the maximum spread to within a factor of 2. That is, if the maximum spread is $s$, then your algorithm should produce a selection of one ad for each customer that has spread at least $s/2$. In designing your algorithm, you may assume that no single customer has a value that is significantly above the average; specifically, if $\bar{v} = \frac{\sum_{i=1}^{n} v_i}{n}$ denotes the total value of all customers, then you may assume that no single customer has a value exceeding $\bar{v}/(2m)$.

(b) Give an example of an instance on which the algorithm you designed in part (a) does not find an optimal solution (that is, one of maximum spread). Say what the optimal solution is in your sample instance, and what your algorithm finds.

8. Some friends of yours are working with a system that performs real-time scheduling of jobs on multiple servers, and they’ve come to you for help in getting around an unfortunate piece of legacy code that can’t be changed.

Here’s the situation. When a batch of jobs arrives, the system allocates them to servers using the simple Greedy-Balance Algorithm from Section 11.1, which provides an approximation to within a factor of 2. In the decade and a half since this part of the system was written, the hardware has gotten faster to the point where, on the instances that the system needs to deal with, your friends find that it’s generally possible to compute an optimal solution.

The difficulty is that the people in charge of the system’s internals won’t let them change the portion of the software that implements the Greedy-Balance Algorithm so as to replace it with one that finds the optimal solution. (Basically, this portion of the code has to interact with so many other parts of the system that it’s not worth the risk of something going wrong if it’s replaced.)

After grumbling about this for a while, your friends come up with an alternate idea. Suppose they could write a little piece of code that takes the description of the jobs, computes an optimal solution (since they’re able to do this on the instances that arise in practice), and then feeds the jobs to the Greedy-Balance Algorithm in an order that will cause it to allocate them optimally. In other words, they’re hoping to be able to
reorder the input in such a way that when Greedy-Balance encounters the input in this order, it produces an optimal solution.

So their question to you is simply the following: Is this always possible? Their conjecture is,

*For every instance of the load balancing problem from Section 11.1, there exists an order of the jobs so that when Greedy-Balance processes the jobs in this order, it produces an assignment of jobs to machines with the minimum possible makespan.*

Decide whether you think this conjecture is true or false, and give either a proof or a counterexample.

9. Consider the following maximization version of the 3-Dimensional Matching Problem. Given disjoint sets $X$, $Y$, and $Z$, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, a subset $M \subseteq T$ is a 3-dimensional matching if each element of $X \cup Y \cup Z$ is contained in at most one of these triples. The *Maximum 3-Dimensional Matching Problem* is to find a 3-dimensional matching $M$ of maximum size. (The size of the matching, as usual, is the number of triples it contains. You may assume $|X| = |Y| = |Z|$ if you want.)

Give a polynomial-time algorithm that finds a 3-dimensional matching of size at least $\frac{1}{3}$ times the maximum possible size.

10. Suppose you are given an $n \times n$ grid graph $G$, as in Figure 11.13.

Associated with each node $v$ is a weight $w(v)$, which is a nonnegative integer. You may assume that the weights of all nodes are distinct. Your

![Figure 11.13 A grid graph.](image)
The "heaviest-first" greedy algorithm:
Start with \( S \) equal to the empty set
While some node remains in \( G \)
    Pick a node \( v_j \) of maximum weight
    Add \( v_j \) to \( S \)
    Delete \( v_j \) and its neighbors from \( G \)
Endwhile
Return \( S \)

(a) Let \( S \) be the independent set returned by the "heaviest-first" greedy algorithm, and let \( T \) be any other independent set in \( G \). Show that, for each node \( v \in T \), either \( v \in S \), or there is a node \( v' \in S \) so that \( w(v) \leq w(v') \) and \((v, v')\) is an edge of \( G \).

(b) Show that the "heaviest-first" greedy algorithm returns an independent set of total weight at least \( \frac{1}{2} \) times the maximum total weight of any independent set in the grid graph \( G \).

11. Recall that in the Knapsack Problem, we have \( n \) items, each with a weight \( w_i \) and a value \( v_i \). We also have a weight bound \( W \), and the problem is to select a set of items \( S \) of highest possible value subject to the condition that the total weight does not exceed \( W \)—that is, \( \sum_{i \in S} w_i \leq W \). Here's one way to look at the approximation algorithm that we designed in this chapter. If we are told there exists a subset \( S' \) whose total weight is \( \sum_{i \in S'} w_i \leq W \) and whose total value is \( \sum_{i \in S'} v_i = V \) for some \( V \), then our approximation algorithm can find a set \( A \) with total weight \( \sum_{i \in A} w_i \leq W \) and total value at least \( \frac{1}{1+\epsilon} \). Thus the algorithm approximates the best value, while keeping the weights strictly under \( W \). (Of course, returning the set \( \emptyset \) is always a valid solution, but since the problem is NP-hard, we don't expect to always be able to find \( \emptyset \) itself; the approximation bound of \( 1+\epsilon \) means that other sets \( A \), with slightly less value, can be valid answers as well.)

Now, as is well known, you can always pack a little bit more for a trip just by "sitting on your suitcase"—in other words, by slightly overflowing the allowed weight limit. This too suggests a way of formalizing the approximation question for the Knapsack Problem, but it's the following, different, formalization.