(a) Let $\{w_1, w_2, w_3\} = \{1, 2, 1\}$, and $K = 2$. Then the greedy algorithm here will use three trucks, whereas there is a way to use just two.

(b) Let $W = \sum_i w_i$. Note that in *any* solution, each truck holds at most K units of weight, so W/K is a lower bound on the number of trucks needed.

Suppose the number of trucks used by our greedy algorithm is an odd number $m = 2q+1$. (The case when m is even is essentially the same, but a little easier.) Divide the trucks used into consecutive groups of two, for a total of $q + 1$ groups. In each group but the last, the total weight of containers must be *strictly* greater than K (else, the second truck in the group would not have been started then) — thus, $W > qK$, and so $W/K > q$. It follows by our argument above that the optimum solution uses at least $q + 1$ trucks, which is within a factor of 2 of $m = 2q + 1$.

[1] ex667.592.236

(a) For each protein p, we define a set S_p consisting of all proteins similar to it; we do this by simply enumerating all proteins q for which $d(p, q) \leq \Delta$. With respect to these sets, a representative set $R \subseteq P$ is simply a set for which $\{S_p : p \in R\}$ if a set cover for P.

Thus, to approximate the size of the smallest representative set, we can use the approximation algorithm for Set Cover from this chapter, obtaining an approximation guarantee of $O(\log n)$.

(b) The problem with using the approximation algorithm for Center Selection is that we'd obtain a set R of proteins for which every protein is within distance 2Δ of some element of R. But this doesn't satisfy the requirements for a representative, which stipulated that every protein had to be within distance Δ of some element of R.

[1] ex815.903.104

(a) Let $a_1 = 1$ and $a_2 = 100$, and consider the bound $B = 100$. Only a_1 will be chosen, while an optimal solution would choose a_2.

(b) In fact, this can be done in $O(n)$ time. We first go through all the numbers a_i and delete any whose value exceeds B — such numbers cannot be used in any solution, including the optimal one, so we have not changed the value of the optimum by doing this.

We then go through the numbers $a_1, a_2, \ldots, a_n$ in order until the sum of numbers we've seen so far first exceeds B. Let a_j be the number on which this happens. Thus we have $\sum_{i-1}^{j} a_i \geq B$, but $\sum_{i-1}^{j-1} a_i \leq B$ and also $a_j \leq B$. Thus, one of the sets $\{a_1, a_2, \ldots, a_{j\ 1}\}$ or $\{a_j\}$ is at least $B/2$ and at most B; we select this set as our solution. Since the optimum has a sum of at most B, our solution is at least half the optimal value.

[1] ex650.691.264

Note that in case when all sets B_i have exactly 2 elements (i.e. $b = 2$), the Hitting Set problem is equivalent to the Vertex Cover problem (two-element sets B_i correspond to edges). In the chapter we saw two approximation algorithm for Vertex Cover; here we generalize the one based on linear programming to arbitrary b.

Consider the following problem for Linear Programming:

$$\begin{aligned} \text{Min} \quad & \sum_{i=1}^{n} w_i x_i \\ \text{s.t.} \quad & 0 \le x_i \le 1 \qquad \text{for all } i = 1, \ldots, n \\ & \sum_{i:a_i \in B_j} x_i \ge 1 \quad \text{for all } j = 1, \ldots, m \text{ (all sets are hit)} \end{aligned}$$

Let x be the solution of this problem, and w_{LP} is a value of this solution (i.e. $w_{LP} = \sum_{i=1}^{n} w_i x_i$).

Now define the set S to be all those elements where $x_i \ge 1/b$:

$$S = \{a_i \mid x_i \ge 1/b\}$$

(1) *S is a hitting set.*

Proof. We want to prove that any set B_j intersects with S. We know that the sum of all x_i where $a_i \in B_j$ is at least 1. The set B_j contains at most b elements. Therefore some $x_i \ge 1/b$, for some $a_i \in B_j$. By definition of S, this element $a_i \in S$. So, B_j intersects with S by a_i. ■

(2 *The total weight of all elements in S is at most $b \cdot w_{LP}$.*

Proof. For each $a_i \in S$ we know that $x_i > 1/b$, i.e., $1 < bx_i$. Therefore

$$w(S) = \sum_{a_i \in S} w_i \le \sum_{a_i \in S} w_i \cdot bx_i \le b \sum_{i=1}^{n} w_i x_i = b w_{LP}$$

■

(3 *Let S^* be the optimal hitting set. Then $w_{LP} \le w(S^*)$.*

Proof. Set $x_i = 1$ if a_i is in S^*, and $x_i = 0$ otherwise. Then the vector x satisfy constrains of our problem for Linear Programming:

$$\begin{aligned} & 0 \le x_i \le 1 \qquad \text{for all } i = 1, \ldots, n \\ & \sum_{i:a_i \in B_j} x_i \ge 1 \quad \text{for all } j = 1, \ldots, m \text{ (because all sets are hit)} \end{aligned}$$

Therefore the optimal solution is not worse that this particular one. That is,

$$w_{LP} \le \sum_{i=1}^{n} w_i x_i = \sum_{a_i \in S} w_i = w(S^*)$$

■

Therefore we have a hitting set S, such that $w(S) \le b \cdot w(S^*)$.

[1] ex53.496.888

In the textbook we prove that

$$T - t_j \leq T^*.$$

where T is our makespan, t_j is a size of a job and T^* is the optimal makespan. We also proved that the optimal makespan is at least the average load, which is at least 300 in our case:

$$T^* \geq \frac{1}{m} \sum_j t_j \geq \frac{1}{10} 3000 = 300.$$

We also know that $t_j \leq 50$. Therefore the ratio of difference between our makespan and the optimal makespan to the optimal makespan is at most

$$\frac{T - T^*}{T^*} \leq \frac{t_j}{T^*} \leq \frac{50}{300} = \frac{1}{6} \leq 20\%$$

[1] ex995.878.454

One way to do this works as follows: When each job arrives, we put it on the machine that currently ends the soonest. (Note that this determination involves taking into account the speeds of the machines.)

To give a bound on this algorithm, we first give some lower bounds on the optimum makespan T^*. The total time of all jobs is $\sum_j t_j$. Let

$$t = \frac{\sum_j t_j}{m + 2k}.$$

If jobs could be assigned to machines so that each slow machine had a set of jobs summing to less than t, and each fast machine had a set of jobs summing to less than $2t$, then we would have

$$\sum_j t_j < mt + 2kt = \sum_j t_j,$$

a contradiction. Thus, some machine runs for at least t time units, and hence

$$T^* \geq \frac{\sum_j t_j}{m + 2k}.$$

Also, we have

$$T^* \geq \frac{1}{2} t_j,$$

for every job j, since at best it runs on one of the fast machines.

Let $M(r)$ denote the set of jobs assigned to machine r. Consider a machine i that achieves the makespan, and let j be the last job to go on it. Let x denote the time it uses for all jobs before j. (This means that $\sum_{j \in M(i)} t_j$ is equal to x if it's a slow machine, and it is equal to $2x$ if it's a fast machine.) Then at the moment j is added, every slow machine s has $\sum_{j \in M(s)} t_j \geq x$, and and every fast machine f has $\sum_{j \in M(f)} t_j \geq 2x$. Thus we have $\sum_j t_j \geq mx + 2kx$, and hence $T^* \geq x$. Also, $2T^* \geq t_j$.

Since the makespan is achieved by i, it is at most $x + t_j \leq T^* + 2T^* = 3T^*$.

An alternate solution is to simply sort the jobs in decreasing order of size, and then run the Greedy-Balance algorithm as though all machines were slow. We know from the chapter that this would give a $\frac{3}{2}$-approximation if all machines really were slow. However, we are comparing to the optimum as though all its machines are slow; in reality, the optimum's makespan might be half as large as we think, since some of its machines are fast. Thus, this gives a 3-approximation.

[1] ex829.220.704

(a) We process the customers in an arbitrary order. At any given point in time, let V_j denote the total value of all customers who have been shown ad j. As we see each new customer, we show him or her the ad for which V_j is as small as possible.

Let s' denote the spread of this algorithm. We first claim that $s' \geq \overline{v}/2m$. To prove this, suppose that ad j is the one achieving the spread (i.e., $V_j = s'$), and let i be any other ad. Let c be the last customer to be shown ad i. Before c was shown ad i, the value of V_i was at most V_j (by the definition of our greedy algorithm), and so $V_i \leq V_j + v_c \leq V_j + (\overline{v}/2m)$ by our assumption about the maximum customer value. Thus, if $s' = V_j < \overline{v}/2m$, then

$$\overline{v} = \sum_j V_j < V_j + (m-1)(V_j + (\overline{v}/2m)) < mV_j + (\overline{v}/2m) < (\overline{v}/2m) + (\overline{v}/2m) = \overline{v},$$

a contradiction.

Next we claim that the optimum spread s satisfies $s \leq \overline{v}/m$. Indeed, the total customer value is $\overline{v}$, and there are m ads, so one must be allocated at most a customer value of $\overline{v}/m$.

Combining these two claims, we get $s \leq \overline{v}/m \leq 2s'$.

(b) Suppose the input begins with $N + m$ customers of value 1, for some very large N, and then $m/2$ customers of value 2. (Suppose m is even and N is divisible by m.) Then our greedy algorithm will produce a spread of $1 + N/m$, while the optimal spread is $2 + N/m$, obtained by grouping the final m customers of value 1 onto $m/2$ ads, and showing the remaining $m/2$ ads to the customers of value 2.

[1] ex43.640.595

The conjecture is true. Consider the assignment of jobs to machines in an arbitrary optimal solution, and order the jobs arbitrarily on each machine. We say that the *base height* of a job j is the total time requirements of all jobs that precede it on its assigned machine.

We order all jobs by their base heights (breaking ties arbitrarily), and we feed them to the Greedy-Balance algorithm in this order. (We will label the jobs $1, 2, \ldots, n$ according to this order.)

We claim the following by induction on r: after the first r jobs have been processed by Greedy-Balance, the set of machine loads is the same as the set of machine loads if we consider the assignment of these r jobs made by the optimal solution.

This is clearly true for $r = 1$, since one machine will have load t_1, and all others will have load 0. Now suppose it is true up to some r, with loads $T_1, \ldots, T_m$, and consider job $r + 1$. Because we have sorted jobs by base height, job $r + 1$ comes from the machine that, in the optimal solution, has load $\min_i T_i$. By the definition of Greedy-Balance, this is the machine on which job $r + 1$ will be placed, giving it a load of $t_{r+1} + \min_i T_i$. This completes the induction step.

[1] ex286.347.713

We will use the following simple algorithm. Consider triples of T in any order, and add them if they do not conflict with previously added triples. Let M denote the set returning by this algorithm and M^* be the optimal three-dimensional matching.

(1) *The size of M is at least $1/3$ of the size of M^*.*

Proof. Each triple (a, b, c) in M^* must intersect at least one triple in our matching M (or else we could extend M greedily with (a, b, c)). One triple in M can only be in conflict with at most 3 triples in M^* as edges in M^* are disjoint. So M^* can have at most 3 times as many edges as M has. ■

[1] ex271.721.76

(a) If $v \notin S$, it must have never been chosen by the greedy algorithm. This means that it was deleted in some iteration by the selection of a node v': by the definition of the selection rule, this node v' must both be a neighbor of v, and have at least as much weight as v.

(b) Consider any other independent set T. For each node $v \in T$, we *charge* it to a node in S as follows. If $v \in S$, then we charge v to itself. Otherwise, by (a), v is a neighbor of some node $v' \in S$ whose weight is at least as large. We charge v to v'.

Now, if v is charged to itself, then no other node is charged to v, since S and T are independent sets. Otherwise, at most four neighboring nodes of no greater weight are charged to v. Either way, the total weight of all nodes charged to v is at most $4w(v)$. Since these charges account for the total weight of T, it follows that the total weight of nodes in T is at most four times the total weight of nodes in S.

[1] ex727.874.96

This means that our knapsack has capacity $(1+2\epsilon)W$. We throw out all items of weight exceeding W, since these can't be used in the solution we're comparing against.

We now round all remaining weights down to the nearest multiple of $\epsilon W/n$, and then multiply them all by $n/(\epsilon W)$. This means that all weights are now integers between 0 and n/ϵ. (Note that the items of weight less than $\epsilon W/n$ do get rounded down to 0, and yes, this means we will probably take them all, but as we'll see this is not a problem.)

So in time polynomial in n and $1/\epsilon$, by the dynamic programming algorithm for the knapsack problem with small weights, we can find the subset of (rounded) weight at most W which achieves the greatest value. The solution of (true) weight W and value V that was promised to exist must be available as an option, since its weight only went down, and since we find the best subset, we find one of value at least V. When we put all these items in our knapsack, each has a weight that may be up to $\epsilon W/n$ more than we thought (due to rounding down), so we use a weight of at most $W + n(\epsilon W/n) = W(1+\epsilon)$.

[1] ex662.412.328

We'll select the sites and the users they cover using the idea of the Set-Cover greedy algorithm. If a side s is used to cover the a subset U_s of users, then the average user cost is $(f_s + \sum_{u \in U_s} d_{us})/|U_s|$. The idea behind the greedy algorithm is to select the site s with a subset U_s that minimizes this quantity. First we need to argue that this minimum can be found.

(1) *Given a set R of uncovered users, and a site possible s, one can find the subset $U_s \subset U$ that minimizes the average cost $(f_s + \sum_{u \in U_s} d_{us})/|U_s|$ in polynomial time.*

Proof. Sort the users by increasing distance d_{us} from site s. The set U_s will be an initial set of this sorted sequence: $U_s = \{u \in R : d_{su} \leq \alpha\}$ for some value α. ■

Now the algorithm will be analogous to the Set Cover greedy algorithm. We select sites s with subsets U_s by the above greedy rule: selecting the site and the set that minimizes the average cost of covering a new user. There is one more option to consider. Suppose T is the subset of sites already selected. For a site $s \in T$ we can add a new node $u \in R$ to U_s, covering the new user u at the cost of d_{us}. In the algorithm given below, we will also save the cost c_u at which user u got covered by the algorithm. These values will be used by the analysis.

```
Start with R = U and T = ∅.
While R is not empty
   Let c = min_{u∈R,s∈T} d_us
   Select s ∈ S−T, and set U_s ⊂ R that minimizes
          c' − (f_s + Σ_{u∈U_s} d_us)/|U_s|.
   If c' ≤ c then
      Select the site s and set U_s used to obtain c' above.
      Add s to T, and delete U_s from R.
      Set c_u = c' for all u ∈ U_s.
   Else
      Select s and u obtaining the first minimum.
      Add u to U_s,
      Set c_u = c.
Endwhile
```

First, note that if we select the set of sites T, and have each site $s \in T$ cover the users in U_s then we get a solution to the problem with total cost $\sum_{u \in U} c_u$. Also, the algorithm runs in polynomial time. It remains to show that this is an $H(n)$ approximation algorithm.

The proof of the approximation ratio follows very closely the proof for the set cover algorithm. Consider an optimum solution. Assume it contains a subset T^* of sites, and $s \in T^*$ is used to cover a set U_s^* of users. The cost of using s to cover U_s^* is $f_s + \sum_{u \in U_s^*} d_{us}$. We will want to compare the optimum's cost, and $\sum_{u \in U_s^*} c_u$, which is the cost our greedy algorithm paid for the users in U_s^*.

[1] ex37.588.671

(2) *Using the notation introduced above, and the costs defined by the algorithm, we have that* $\sum_{u \in U_s^*} c_u \leq H(d)(f_s + \sum_{u \in U_s^*} d_{us})$, *where* $d = |U_s^*|$.

Proof. For notational simplicity, let $C = f_s + \sum_{u \in U_s^*} d_{us}$. Consider the elements in U_s^* in the order the algorithm covered them. Assume they are $u_1, u_2, \ldots, u_d$. Consider the moment the algorithm covers the ith node u_i from U_s^*. There are two cases to consider.

Case 1 At this point of the algorithm $s \notin T$.

Case 2 At this point of the algorithm $s \in T$.

When the algorithm covered u_i it selected the smallest average cost. In Case 1 this implies that the cost c_{u_i} is at most the cost of selecting cite s with the set $U_s^* \cap R$, which is at most $c_{u_i} \leq C/(d - i + 1)$ (as $i - 1$ previously covered nodes are no longer in the set). In Case 2, this implies that $c_{u_i} \leq d_{us}$. Assume that Case 1 applies when the first k nodes are covered, and after that Case 2 applies (k may be equal to d). Now summing all costs in U_s^* we get that

$$\sum_{u \in U_s^*} c_u \leq C/d + C/(d-1) + \ldots C/(d-k+1) + \sum_{i>d} d_{u_i,s}.$$

Now if $d = k$ then the upper bound on the cost is $H(d)C$ as claimed. If $k < d$ then note that the costs $\sum_{i>d} d_{u_i,s}$ is bounded by C, and so we also can bound the total cost by $H(d)C$. ∎

Now we are ready to prove that the algorithm is an $H(n)$ approximation algorithm. Let T^* and U_s^* be the optimal solution. The total cost of the solution is $\sum_{s \in T^*}(f_s + \sum_{u \in U_s^*} d_{us})$. We use the above Lemma to bound each term of the cost, and upper bound $II(d)$ by $II(n)$ for each set U_s^* in the optimum, to get the following.

$$OPT - \sum_{s \in T^*}(f_s + \sum_{u \in U_s^*} d_{us}) \leq \sum_{s \in T^*} H(n) \sum_{u \in U_s^*} c_u - H(n) \sum_{u \in U} c_u,$$

where the last sum is the algorithm's cost as claimed by the first Lemma.