St. Francis Xavier University
Department of Computer Science
CSCI 550: Approximation Algorithms
Lecture 4: Rounding Data and Dynamic Programming
Fall 2022

1 A Dynamic Situation

If you've previously taken an advanced or upper-level course in algorithm design, then you may already be familiar with the notion of *dynamic programming.* In the dynamic programming paradigm, an algorithm iteratively constructs an optimal solution to a problem by computing and storing optimal subsolutions for subproblems. The algorithm can then refer back to these stored subsolutions instead of having to recompute them over and over.

In order for a problem to be amenable to dynamic programming, it must satisfy two properties: it must have *optimal substructure* (as we mentioned earlier, where the optimal solution can be constructed from optimal subsolutions) and *overlapping subproblems* (i.e., subproblems that reoccur throughout the course of the computation).

It should come as no big surprise that we can use the dynamic programming paradigm to construct approximation algorithms for certain problems. Often, such an algorithm relies on rounding data in order for the algorithm to work efficiently. In this lecture, we will consider two approaches to using both rounding and dynamic programming techniques in an approximation algorithm. In the first approach, we will observe that encoding input values in unary sometimes yields a performance advantage, and rounding these input values allows us to exploit this advantage to solve the original problem instance efficiently. In the second approach, we will see how to split up a single input into "large" and "small" components, and use dynamic programming to find an optimal solution just for these "large" components. We can then factor the "small" components into our solution to get an approximately-optimal solution.

1.1 Knapsack Problem

Like the scheduling problems we saw previously, the *knapsack problem* is one that may be quite familiar to students. Suppose a student has a knapsack of fixed size, and they must fit as many course materials and books into their knapsack as possible with the goal of maximizing the "value" of the contents; for instance, selecting course materials that correspond to the lectures they'll be attending that day.

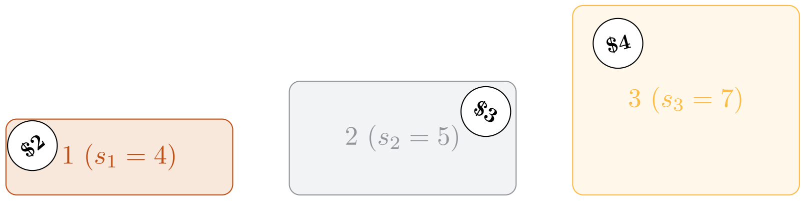
A more natural instance of the knapsack problem in real life (though not a very legal instance) is that of a bank robbery. A robber enters a bank with a duffle bag of fixed size and must fit as much money into the bag as they can. Their goal, again, is to maximize the value of the contents in the duffle bag.

With these examples in mind, we can formalize the problem as follows:

<u>Knapsack</u>
Given: a knapsack with integer capacity B where $B \geq 1$, a set of items $\{1, 2, \ldots, n\}$ and, for each item i, an associated integer value v_i and integer size s_i where $v_i, s_i \geq 1$
Determine: a subset S of items that maximizes the value $\sum_{i \in S} v_i$ with the constraint $\sum_{i \in S} s_i \leq B$

Note that we can make some natural assumptions about instances of the knapsack problem. For example, we can assume that $s_i \leq B$ for all items i, since otherwise we could never include item i in any feasible solution.

Example 1. Suppose we have a knapsack with a capacity of $B = 10$ units, and we have three items: item 1 has a value of \$2 and a size of 4 units; item 2 has a value of \$3 and a size of 5 units, and item 3 has a value of \$4 and a size of 7 units.



If we took the item of greatest value first, then we would add item 3 to our knapsack and use 7 out of our 10 units of space. However, now we can't fit either item 1 or item 2 into our knapsack, so the value in this case is \$4.

If, on the other hand, we took the item of least size first, then we would add item 1 to our knapsack and use 4 out of our 10 units of space. This leaves us with enough room to also add item 2 to our knapsack, using another 5 units of space for a total of 9 units. The value in this case is \$2 + \$3 = \$5.

The knapsack problem, like so many other problems we study in this course, is known to be NP-hard. However, using a little cleverness, we can develop a dynamic programming algorithm for the problem.

Before we continue, let's define one piece of notation. We will say that $A(i, v)$ denotes the size of the "smallest" subset of items $\{1, 2, \ldots, i\}$ with a value equal to v. If no such subset exists, then $A(i, v) = \infty$. We use this value $A(i, v)$ in our algorithm.

Our dynamic programming algorithm takes on the usual form: we begin by initializing all values $A(i, 0)$ to be zero, and we then initialize all values $A(1, v)$. For the remainder of the algorithm, we "fill in" each of the other values $A(i, v)$ using previous values we computed and stored.

Algorithm 1: Knapsack—dynamic programming

```
$V \leftarrow \max_i v_i$
for all $1 \le i \le n$ do
    $A(i, 0) \leftarrow 0$
for all $1 \le v \le nV$ do            ▷ if $V = \max_i v_i$, then $nV$ is an upper bound for any solution
    if $v_1 = v$ then
        $A(1, v) \leftarrow s_1$
    else
        $A(1, v) \leftarrow \infty$
for all $2 \le i \le n$ do
    for all $1 \le v \le nV$ do
        if $v_i \le v$ then
            $A(i, v) \leftarrow \min(A(i-1, v), s_i + A(i-1, v - v_i))$
        else
            $A(i, v) \leftarrow A(i-1, v)$
```

Though we won't do it here, we can prove that this algorithm computes all values A correctly. Moreover, we can use this algorithm to obtain the solution

$$\arg\max_v \{A(n, v) \mid A(n, v) \le B\},$$

which gives the largest number of items both maximizing the value v and fitting in the knapsack.

However, observe the time complexity of our algorithm. In particular, the nested for loops at the bottom of the algorithm run in time $O(n^2 V)$. How can this algorithm possibly run in polynomial time, given the fact that the knapsack problem is NP-hard? As it turns out, there's a little bit of hidden trickery going on here. The input to an algorithm is usually encoded in binary, since that's the "language" of the computer. Thus, we require $\log_2(v_i)$ bits to encode a value v_i. Even though the running time of the algorithm is *polynomial* in $V = \max_i v_i$, it is actually *exponential* in the input size of that value V. In other terms, the size of the input, s, is $\log_2(V)$, and so $2^s = 2^{\log_2(V)} = V$.

We can sneak around this issue by using a different encoding for our input. If we encode the input in unary instead of binary, then we would require exactly v_i bits to encode a value v_i, and the running time of our algorithm would remain polynomial in the size of the input. We give a special name to such algorithms.

Definition 2 (Pseudopolynomial-time algorithm)**.** An algorithm for a problem P is said to run in pseudopolynomial time if its running time is polynomial in the size of the input when the input is encoded in unary.

If our value V were some polynomial in n, then the running time of our algorithm would be polynomial in the input size. This is the key idea we will use to obtain our approximation algorithm for the knapsack problem: we will round our input so that V becomes polynomial in n, and then use our dynamic programming algorithm on the rounded input. Depending on how much we round our input, we can control the precision of our solution, and so this approach will in fact give us a *polynomial-time approximation scheme* for the knapsack problem. Recall the definition from our first lecture:

Definition 3 (Polynomial-time approximation scheme)**.** A polynomial-time approximation scheme, or PTAS, for an optimization problem is a family of approximation algorithms $\{A_\epsilon\}$ where, for each $\epsilon > 0$, there exists a $(1 + \epsilon)$-approximation algorithm (for minimization problems) or a $(1 - \epsilon)$-approximation algorithm (for maximization problems) in the family.

By the definition, each algorithm A_ϵ in a polynomial-time approximation scheme is polynomial in the input size, but could depend arbitrarily on the value $1/\epsilon$. This dependence could possibly be exponential in $1/\epsilon$, which would mean that finding solutions nearer to the optimal solution becomes progressively more difficult. If we can establish the additional condition that the dependence is polynomial in $1/\epsilon$, however, then we obtain an even better scheme.

Definition 4 (Fully polynomial-time approximation scheme)**.** A fully polynomial-time approximation scheme, or FPTAS, for an optimization problem is a polynomial-time approximation scheme where the running time of each algorithm A_ϵ is bounded by some polynomial in $1/\epsilon$.

With these notions established, we can now present our approximation scheme for the knapsack problem. Here, we will measure values in terms of a number K (which is itself defined in terms of ϵ). We then round each original value v_i down to the nearest integer multiple of K. We then run our dynamic programming algorithm on the instance of the knapsack problem we were given, but with the original values v_i replaced by our rounded values.

Algorithm 2: Knapsack—approximation scheme

```
$K \leftarrow \epsilon V/n$
for all $i$ do
    $v'_i \leftarrow \lfloor v_i/K \rfloor$
run Algorithm 1 on the given problem instance with values $v'_i$
```

How did we settle on this particular value for K? Consider that, with our approximation algorithm, we want the error in our algorithm's solution to be bounded by at most ϵ times a lower bound on the optimal solution. Taking V to be the maximum value of any item in our set, as we did earlier, we get that V itself is a lower bound on the optimal solution (since we can just put the most valuable item in our knapsack and nothing else). Thus, we want to take our value K so that $nK = \epsilon V$, and rearranging, we get that $K = \epsilon V/n$.

Let's wrap up by proving that our algorithm is a rather good approximation scheme for the knapsack problem.

Theorem 5. *Algorithm 2 is a fully polynomial-time approximation scheme for the knapsack problem.*

Proof. Since the knapsack problem is a maximization problem, we must prove that Algorithm 2 returns a solution whose value is within a factor of $(1-\epsilon)$ times the value of the optimal solution.

Let S be the set of items selected by Algorithm 2, and denote the optimal set of items by O. As we observed before, we have a lower bound $V \leq \text{OPT}$, since we can put the most valuable item in our knapsack and then immediately finish. We also know, by the definition of v_i', that

$$Kv_i' \leq v_i \leq K(v_i'+1)$$

and, as a result, we have that

$$Kv_i' \geq v_i - K.$$

Combining this observation with the fact that S is an optimal solution for the rounded instance of the problem, we get the following:

$$\begin{aligned}\sum_{i\in S} v_i &\geq K\sum_{i\in S} v_i' \\ &\geq K\sum_{i\in O} v_i' \\ &\geq \sum_{i\in O} v_i - |O|K \\ &\geq \sum_{i\in O} v_i - nK \\ &= \sum_{i\in O} v_i - \epsilon V \\ &\geq \text{OPT} - \epsilon\cdot\text{OPT} = (1-\epsilon)\cdot\text{OPT}.\end{aligned}$$

Note also that, using the modified values, we have $V' = \sum_{i=1}^{n} v_i' = \sum_{i=1}^{n} \lfloor v_i/(\epsilon K/n)\rfloor$, and this expression is $O(n^2\cdot 1/\epsilon)$. Overall, the running time of Algorithm 2 is $O(n\cdot\min\{B, V'\}) = O(n^3\cdot 1/\epsilon)$, and therefore it is a fully polynomial-time approximation scheme. □

1.2 Parallel Scheduling on Identical Machines

In this section, we will revisit the problem of scheduling on multiple machines that can complete jobs in parallel. We redefine the problem here for convenience.

Scheduling-Multiple-Identical-Machines
Given: a set of m identical machines, a set of n jobs S, and, for each job j, a processing time p_j
Determine: a schedule that minimizes the total processing time $C_{\max} = \max_{j=\{1,\ldots,n\}} C_j$

Recall that, the last time we considered this problem, we developed a 2-approximation algorithm using a local search technique. Here, we will improve on our previous result by developing a polynomial-time approximation scheme for the problem. Each algorithm in our polynomial-time approximation scheme will divide the given set of jobs into "long jobs" and "short jobs", depending on the processing time required to complete the job. The algorithm will then use two different techniques to schedule both the long jobs and the short jobs while constraining the length of the schedule to meet some target length T.

Before we proceed, we need to define what constitutes a short job versus a long job. Let k be a fixed positive integer, and consider algorithm A_k from our polynomial-time approximation scheme. We will say that a job i is short if $p_i \leq T/k$; that is, if the job's processing time p_i is less than or equal to the target schedule length divided by the constant k.

The technique we will use to schedule short jobs is straightforward; in fact, it is one of the earliest-known examples of an approximation algorithm for *any* problem, having been studied by Ronald Graham in the 1960s.

Algorithm 3: Multiple identical machine scheduling—short jobs

```
for $1 \le i \le n$ do
    schedule job $i$ on the machine with the least work assigned to it so far
```

This *list scheduling* algorithm simply schedules all of the short jobs in order, with each job being assigned to whatever machine currently has the least amount of work to do.

To schedule long jobs, we will again use rounding and dynamic programming. After we schedule each of the long jobs, we can then use Algorithm 3 to extend and complete the schedule.

By our earlier definition, we know that a job i is long if $p_i > T/k$. We will begin by rounding down the processing time of each job to the nearest multiple of T/k^2, and then attempt to find a schedule for these rounded jobs that completes within time T. If such a schedule exists, then we can use it as the basis to schedule our original jobs. If no such schedule exists, on the other hand, then we can conclude that we also can't schedule the original jobs within time T.

Algorithm 4: Multiple identical machine scheduling—long jobs

```
for each job $j$ with processing time $p_j > T/k$ do
    $p'_j = T/k^2$
if there exists a job $\ell$ with $p'_\ell > T$ then
    no schedule of length at most $T$ exists
else
    construct $(k+1)^{k^2}$ vectors $(s_1, \ldots, s_{k^2})$    ▷ since $k$ is constant, the set of vectors is constant-size
    for each vector $v$ do
        if $\sum_{i=1}^{k^2} s_i \cdot iT/k^2 \le T$ then
            $C \leftarrow C \cup \{v\}$
    while there are jobs left to schedule do
        $\mathrm{OPT}(n_1, \ldots, n_{k^2}) \leftarrow 1 + \min_{(s_1,\ldots,s_{k^2}) \in C} \mathrm{OPT}(n_1 - s_1, \ldots, n_{k^2} - s_{k^2})$
```

This algorithm uses dynamic programming to compute a schedule for the set of rounded long jobs. If a schedule of length T can exist, then the algorithm considers all possible *machine configurations* $(s_1, \ldots, s_{k^2})$, or job assignments to a single machine. Since each machine must process one of $k+1$ rounded long jobs, there are a total of $(k+1)^{k^2}$ configurations, and since the original jobs have processing time at least T/k, at most k jobs can be assigned to one machine. The algorithm finally computes the value $\mathrm{OPT}(n_1, \ldots, n_{k^2})$, which corresponds to the minimum number of machines sufficient to schedule the given input and is calculated by assigning some jobs to one machine (according to that machine's configuration) and then using as few machines as possible to assign the remaining jobs.

Altogether, algorithm A_k uses both Algorithms 3 and 4 to schedule the given set of jobs. As before, we analyze algorithm A_k to show that it gives an approximate solution.

Theorem 6. *The family of algorithms $\{A_k\}$ is a polynomial-time approximation scheme for the multiple identical machine scheduling problem.*

Proof. We first prove that, for some constant k, the algorithm A_k produces a schedule of length at most $(1 + \frac{1}{k})T$ whenever there exists a schedule of length at most T.

When there exists a schedule of length at most T for the original jobs, then there must exist a schedule for the rounded long jobs, and so the algorithm works as expected. Otherwise, suppose that a schedule for the rounded long jobs is computed by the algorithm, and that this schedule has length at most T. Let S be the set of jobs assigned to one machine by this schedule. Since each job in S is a long job and therefore has processing time at least T/k, it follows that $|S| \leq k$. Moreover, the difference between the original and rounded processing time of any job $j \in S$ is at most T/k^2. Thus, after the long jobs are scheduled, we have that

$$\sum_{j \in S} p_j \leq T + k(T/k^2) = \left(1 + \frac{1}{k}\right) T.$$

Next, we must schedule the short jobs. Each short job ℓ is assigned to whatever machine currently has the least work assigned to it. Since $\sum_{j=1}^{n} p_j/m \leq T$, the average load assigned to any one machine is $\sum_{j \neq \ell} p_j/m < T$. Therefore, there must exist a machine whose currently-assigned jobs have a total processing time less than T. If the algorithm assigns job ℓ to this machine, then the machine's new workload is

$$p_\ell + \sum_{j \neq \ell} p_j/m < T/k + T = \left(1 + \frac{1}{k}\right) T.$$

Therefore, after all rounded long jobs and short jobs are scheduled, the schedule has length at most $(1 + \frac{1}{k})T$.

We next show how to convert the family of algorithms $\{A_k\}$ to a polynomial-time approximation scheme. Suppose $\epsilon > 0$ is the error term. We will use a bisection approach (i.e., binary search) to determine a suitable target value T for the schedule. Let

$$L_0 = \max \left\{ \max_{j=\{1,2,\ldots,n\}} p_j, \left\lceil \sum_{j=1}^{n} p_j/m \right\rceil \right\};$$

recall that these two values are the lower bounds on the value of the optimal schedule. Similarly, let

$$U_0 = \max_{j=\{1,2,\ldots,n\}} p_j + \left\lceil \sum_{j=1}^{n} p_j/m \right\rceil.$$

The bisection approach will maintain an interval $[L, U]$ that satisfies two invariants: (1) $L \leq C^*_{\max}$, and (2) the algorithm can compute a schedule of length at most $(1 + \epsilon)U$. These invariants hold for the initial values L_0 and U_0, and it is straightforward to show that the invariants also hold after each iteration where we set $T = \lfloor (L + U)/2 \rfloor$ and run algorithm A_k with $k = \lceil 1/\epsilon \rceil$. If algorithm A_k returns a schedule, then we take $U = T$; otherwise, we take $L = T + 1$. The bisection approach terminates when $L = U$.

Since both of the invariants must hold for the interval $[L, L]$, then we know both that $L \leq C^*_{\max}$ and that the final schedule produced by the algorithm has length at most $(1 + \epsilon)L \leq (1 + \epsilon)C^*_{\max}$. Therefore, it is a polynomial-time approximation scheme. □

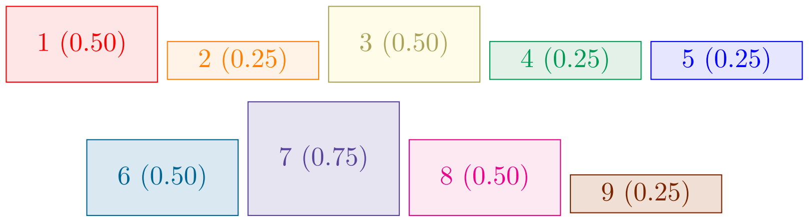
1.3 Bin Packing

The next problem we consider, the *bin packing problem*, has taken on a renewed importance with the advent of e-commerce and shipping items to our doors directly. In the bin packing problem, we must determine the minimum number of identical bins needed to pack a set of items, each of a given size. The bins we use each have a capacity limit, so we can't overload any one bin. Naturally, this problem is very relevant to post offices and shipping companies, where workers load packages into bins, bags, and trucks. However, the bin packing problem also appears regularly in the digital world; for instance, back in the days when physical media was popular and music was released on CDs, a classical music label might have needed to figure out how best to organize the movements of a symphony on disc: can all movements fit on one disc, or will a second disc be required to fit the final one or two movements?

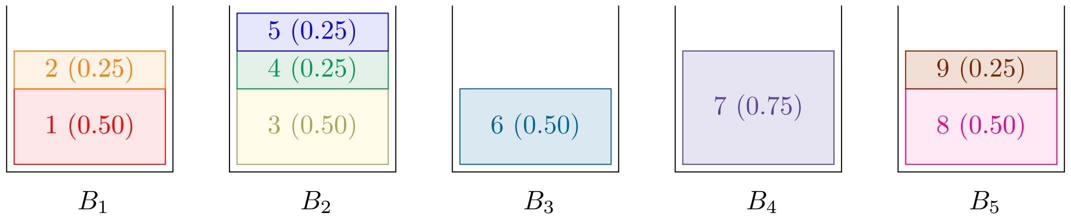
The bin packing problem, formally speaking, asks us to find the smallest value k such that all of our items can be packed into k bins without going over any single bin's capacity limit.

Bin-Packing
Given: a set of items $\{1, \ldots, n\}$ and an associated size s_i for each item i, where $0 \leq s_i \leq 1$
Determine: a partitioning of items into k bins $B_1, \ldots, B_k$ such that $\sum_{i \in B_j} s_i \leq 1$ and k is minimum

Example 7. Suppose we have a set of nine items, $\{1, \ldots, 9\}$, where the items have the following sizes:



We can pack these nine items into five bins, each with size limit 1, in the following way:



As we would expect, the bin packing problem is NP-hard (in fact, NP-complete). Going beyond establishing the difficulty of finding an *exact* solution to the bin packing problem, though, we can establish the difficulty of finding an *approximate* solution. We do so by way of a reduction from the partition problem:

Partition
Given: a set of items $\{1, \ldots, n\}$ and an associated size s_i for each item i, where $s_1 \geq \cdots \geq s_n$
Determine: a partitioning of items into two sets A and B such that $\sum_{i \in A} s_i = \sum_{i \in B} s_i$

With a little thought, we can formulate an instance of the partition problem in terms of an instance of the bin packing problem where, if the set of items can be partitioned into two sets of equal size, then the optimal packing requires two bins. Otherwise, it must be the case that three or more bins are needed for the packing. In the case where OPT is either two or three bins, then any approximation algorithm for the bin packing problem with performance guarantee $\alpha < 3/2$ will give us the exact answer to the partition problem. Since the partition problem itself is NP-complete, we get the following result:

Theorem 8. *There exists no α-approximation algorithm for the bin packing problem with $\alpha < 3/2$, unless* $\mathsf{P} = \mathsf{NP}$.

This should not dissuade us from trying to construct approximation algorithms for the bin packing problem, though. We can find approximate solutions to the problem using the same techniques we've used before, but with one small change that is new to us: our performance guarantees will incur an additive factor.

To gain a better understanding of this additive factor, let's begin by analyzing the most straightforward approximation algorithm for the bin packing problem, where we simply place each item into the first bin that will accommodate it. If the item will not fit into any of the existing bins, then we implicitly create a new empty bin and place the item there.

Algorithm 5: Bin packing—first fit

```
for $1 \leq i \leq n$ do
    $j \leftarrow$ the first bin such that item $i$ fits in bin $j$
    put item $i$ in bin $j$
```

Theorem 9. *Let $FF(I)$ be the solution given by Algorithm 5 on an input instance I. Then*

$$FF(I) \leq 2 \cdot \mathrm{OPT}(I) + 1.$$

Proof. Let $S = \sum_i s_i$. Clearly, we have the lower bound $S \leq \mathrm{OPT}(I)$ on the optimal solution for the given input instance I.

We claim that, in the solution given by Algorithm 5, at most one bin can be half-full. This is because if the solution contained two bins that were each half-full, then the last item added to the second half-full bin should have instead been added to the first half-full bin. Thus,

$$\frac{1}{2}\left(FF(I) - 1\right) \leq S,$$

and therefore $FF(I) \leq 2 \cdot S + 1$. By our lower bound, $FF(I) \leq 2 \cdot \mathrm{OPT}(I) + 1$. □

Next, in the spirit of the other results of this lecture, we will create a "polynomial-time approximation scheme" for the bin packing problem. We put it in quote marks, though, because the additive factor prevents it from being a true polynomial-time approximation scheme. More precisely, we will be creating the following approximation scheme.

Definition 10 (Asymptotic polynomial-time approximation scheme)**.** An asymptotic polynomial-time approximation scheme, or APTAS, for an optimization problem is a polynomial-time approximation scheme where, for some constant c, the running time of each algorithm A_ϵ returns a solution of value at most $(1 + \epsilon)\mathrm{OPT} + c$ (for minimization problems) or $(1 - \epsilon)\mathrm{OPT} + c$ (for maximization problems).

Given an instance I of the bin packing problem with error term $\epsilon > 0$, suppose the size of each item i is bounded above by $s_i \leq \epsilon/2$. Using an argument similar to that used in the proof of Theorem 9, we can show that at most one bin can be at most $\left(1 - \frac{\epsilon}{2}\right)$-full. Therefore,

$$\left(1 - \frac{\epsilon}{2}\right)\left(FF(I) - 1\right) \leq S,$$

and, if $\epsilon < 1$, then we have

$$\begin{aligned} FF(I) &\leq \frac{1}{1 - \epsilon/2} \cdot S + 1 \\ &\leq (1 + \epsilon) \cdot S + 1 \\ &\leq (1 + \epsilon) \cdot \mathrm{OPT}(I) + 1. \end{aligned}$$

From this observation, we can use an approach quite similar to the one we used with the multiple identical machine scheduling problem to develop an approximation algorithm for the bin packing problem. The core idea will be to divide the set of items into "large items" (where $s_i > \epsilon/2$) and "small items" (where $s_i \leq \epsilon/2$), and then use two different techniques to pack the large items into b bins and to pack the small items into the existing b bins, or into additional bins if needed. If we do require additional bins, then we can show that all but at most one bin will contain items of total size at most $\left(1 - \frac{\epsilon}{2}\right)$ and, as a result, we will need at most $(1 + \epsilon) \cdot \mathrm{OPT}(I) + 1$ bins. We can then use this algorithm to find an approximate solution to the original instance.

To pack the large items, we can use an algorithm essentially identical to Algorithm 4, where we make the appropriate modifications: instead of finding the number of *machines* needed to *schedule jobs* within a *time* T, we find the number of *bins* needed to *pack items* within a *capacity* T.

In order to use Algorithm 4, though, we require a constant number of item sizes, and so we must round the size of each item. However, rounding down is not ideal, since doing so wouldn't allow us to "re-expand" the item sizes after packing items into a bin of fixed size 1. Thus, we must round up in this case.

We will follow a "grouping" heuristic to round item sizes up. First, we arrange the original set of items in decreasing order of size. Then, starting from the item with largest size, place ℓ consecutive items in one group. After handling all items, we will be left with $\lceil n/\ell \rceil$ groups $G_1, \ldots, G_{\lceil n/\ell \rceil}$, where the earlier groups contain the larger items.

We then obtain our rounded instance in the following way: discard the group G_1, and for all other groups G_i, round the size of each item in that group to the size of the largest item in G_i. This will produce at most $\lceil n/\ell \rceil - 1$ item sizes, and we now have the constant number we require.

To pack the small items, we can simply use the first-fit procedure given in Algorithm 5. Altogether, this procedure is summarized by the following algorithm:

Algorithm 6: Bin packing—approximation scheme

```
$LG \leftarrow$ items with size $> \frac{\epsilon}{2}$
$SM \leftarrow$ items with size $\leq \frac{\epsilon}{2}$
$\ell \leftarrow \lceil \epsilon \cdot S \rceil$                                  ▷ recall that $S = \sum_i s_i$
group all items in $LG$ to get a rounded instance $I'$
pack items in $G_1$ into $\ell$ bins
run modified Algorithm 4 on $I'$ to pack items optimally into $b$ bins
run Algorithm 5 on items in $SM$
```

As we did in the proof of Theorem 5, we can obtain upper and lower bounds on the optimal solution for the original instance in terms of the rounded instance.

Lemma 11. *For any instance I of the bin packing problem and its rounded instance I',*

$$\mathrm{OPT}(I') \leq \mathrm{OPT}(I) \leq \mathrm{OPT}(I') + \ell.$$

We then use this lemma to establish the performance of our approximation scheme.

Theorem 12. *Let $AS(I)$ be the solution given by Algorithm 6 on an input instance I. Then*

$$AS(I) \leq (1+\epsilon) \cdot \mathrm{OPT}(I) + 1.$$

Proof. We consider two cases: either the first-fit step of the algorithm adds new bins to the solution, or it does not. The case where the first-fit step adds new bins was proved in Theorem 9.

Suppose then that the first-fit step does not add new bins. In this case, the items were packed into a total of $b + \ell$ bins, and we have that

$$\begin{aligned} AS(I) &\leq b + \ell \\ &\leq \mathrm{OPT}(I) + \lceil \epsilon \cdot S \rceil \\ &\leq (1+\epsilon) \cdot \mathrm{OPT}(I) + 1, \end{aligned}$$

where the second line follows from the fact that $b = \mathrm{OPT}(I') \leq \mathrm{OPT}(I)$ (given by Lemma 11) and the third line follows from our earlier lower bound, $S \leq \mathrm{OPT}(I)$. □