• This means that we can assume each list $A(j)$ is of the form $(t_1, w_1), \ldots, (t_k, w_k)$ with $t_1 < t_2 < \ldots < t_k$ and $w_1 < w_2 < \ldots < w_k$.

• Since the sizes of the items are integers, there are at most $B + 1$ pairs in each list.

• Furthermore, if we let $V = \sum_{i=1}^{n} v_i$ be the maximum possible value for the knapsack, then there can be at most $V + 1$ pairs in the list.

• Finally, we ensure that for each feasible set $S \subseteq \{1, \ldots, j\}$ (with $\sum_{i \in S} s_i \leq B$), the list $A(j)$ contains some pair $(t, w)$ that dominates $(\sum_{i \in S} s_i, \sum_{i \in S} v_i)$.

• Alg. 3.1 is the dynamic program that constructs the lists $A(j)$ and solves the knapsack problem.

• We start out with $A(1) = \{(0, 0), (s_1, w_1)\}$.

• For each $j = 2, \ldots, n$, we do the following:

• We first set $A(j) \leftarrow A(j - 1)$, and for each $(t, w) \in A(j - 1)$, we also add the pair $(t + s_j, w + v_j)$ to the list $A(j)$ if $t + s_j \leq B$.

• We finally remove from $A(j)$ all dominated pairs:
  – by sorting the list with respect to their space component,
  – retaining the best value for each space total possible, and
removing any larger space total that does not have a corresponding larger value

One way to view this process is to generate two lists, \( A(j - 1) \) and the one augmented by \( (s_j, w_j) \), and then perform a type of merging of these two lists

We return the pair \( (t, w) \) from \( A(n) \) of maximum value as our solution

---

**Algorithm 3.1:**

A dynamic programming algorithm for the knapsack problem

1. \( A(1) \leftarrow \{(0,0), (s_1, w_1)\} \)
2. for \( j \leftarrow 2 \) to \( n \) do
3. \( A(j) \leftarrow A(j - 1) \)
4. for each \( (t, w) \in A(j - 1) \) do
5. if \( t + s_j \leq B \) then
6. Add \( (t + s_j, w + v_j) \) to \( A(j) \)
7. Remove dominated pairs from \( A(j) \)
8. return \( \max_{(t, w) \in A(n)} w \)
**Theorem 3.1**: Algorithm 3.1 correctly computes the optimal value of the knapsack problem.

**Proof**: By induction on \( j \) we prove that \( A(j) \) contains all non-dominated pairs corresponding to feasible sets \( S \subseteq \{1, \ldots, j\} \). Certainly this is true in the base case by setting \( A(1) \) to \( \{(0,0), (s_1, w_1)\} \).

Now suppose it is true for \( A(j-1) \). Let \( S \subseteq \{1, \ldots, j\} \), and let \( t = \sum_{i \in S} s_i \leq B \) and \( w = \sum_{i \in S} v_i \). We claim that there is some pair \((t', w') \in A(j)\) such that \( t' \leq t \) and \( w' \geq w \).

First, suppose that \( j \in S \). Then the claim follows by the induction hypothesis and by the fact that we initially set \( A(j) \) to \( A(j-1) \) and removed dominated pairs.

Now suppose \( j \notin S \). Then for \( S' = S \setminus \{j\} \), by the induction hypothesis, there is some \((t', \bar{w}) \in A(j-1)\) that dominates \( (\sum_{i \in S'} s_i, \sum_{i \in S'} v_i) \), so that \( t' \leq \sum_{i \in S'} s_i \) and \( \bar{w} \geq \sum_{i \in S'} v_i \). Then the algorithm will add the pair \((t + s_j, \bar{w} + v_j) \to A(j)\), where \( t + s_j \leq t \) and \( \bar{w} + v_j \geq w \). Thus, there will be some pair \((t', w') \in A(j)\) that dominates \((t, w)\).

- Algorithm 3.1 takes \( O(n \min(B, V)) \) time
- This is not a polynomial-time algorithm, since we assume that all input numbers are encoded in binary; thus, the size of the input number \( B \) is essentially \( \log_2 B \), and so the running time \( O(nB) \) is exponential in the size of the input number \( B \), not polynomial
- If we were to assume that the input is given in unary, then \( O(nB) \) would be a polynomial in the size of the input
- It is sometimes useful to make this distinction between problems
Definition 3.2: An algorithm for a problem Π is said to be pseudopolynomial if its running time is polynomial in the size of the input when the numeric part of the input is encoded in unary.

- If the maximum possible value \( V \) were some polynomial in \( n \), then the running time would indeed be a polynomial in the input size.
- We now show how to get a PTAS for the knapsack problem by rounding the values of the items so that \( V \) is indeed a polynomial in \( n \).
- The rounding induces some loss of precision in the value of a solution, but this does not affect the final value by too much.

Definition 3.3: A polynomial-time approximation scheme (PTAS) is a family of algorithms \( \{A_\varepsilon\} \), where there is an algorithm for each \( \varepsilon > 0 \), such that \( A_\varepsilon \) is a \((1 + \varepsilon)\)-approximation algorithm (for minimization problems) or a \((1 - \varepsilon)\)-approximation algorithm (for maximization problems).

- The running time of the algorithm \( A_\varepsilon \) is allowed to depend arbitrarily on \( 1/\varepsilon \): this dependence could be exponential in \( 1/\varepsilon \), or worse.
- We often focus attention on algorithms for which we can give a good bound of the dependence of the running time of \( A_\varepsilon \) on \( 1/\varepsilon \).
Definition 3.4: A fully polynomial-time approximation scheme (FPAS, FPTAS) is an approximation scheme such that the running time of $A_{\varepsilon}$ is bounded by a polynomial in $1/\varepsilon$.

- We can now give a FPTAS for knapsack
  - Let us measure value in (integer) multiples of $\mu$ (we set $\mu$ below), and convert each value $v_i$ by rounding down to the nearest multiple of $\mu$
  - Precisely, we set $v'_i = \lfloor v_i/\mu \rfloor$ for each item $i$
  - We then run the Algorithm 3.1 on the items with sizes $s_i$ and values $v'_i$, and output the optimal solution for the rounded data as a near-optimal solution for the true data

- The main idea here is that we wish to show that the accuracy we lose in rounding is not so great, and yet the rounding enables us to have the algorithm run in polynomial time
  - Let us first do a rough estimate; if we used values $\tilde{v}_i = v'_i \mu$ instead of $v'_i$, then each value is inaccurate by at most $\mu$, and so each feasible solution has its value changed by at most $n \mu$
  - We want the error introduced to be $\leq \varepsilon$ times a lower bound on the optimal value (and so be sure that the true relative error is at most $\varepsilon$)
  - Let $M$ be the maximum value of an item; that is, $M = \max_{i \in I} v_i$
• $M$ is a lower bound on $\text{OPT}$, since one can pack the most valuable item in the knapsack by itself
• Thus, it makes sense to set $\mu$ so that $n\mu = \epsilon M$
or, in other words, to set $\mu = \epsilon M / n$
• Note that with the modified values,

$$V' = \sum_{i=1}^{n} v'_i = \sum_{i=1}^{n} \left| \frac{v_i}{\epsilon M / n} \right| = O \left( \frac{n^2}{\epsilon} \right)$$
• Thus, the running time of the algorithm is $O(n \min(B, V')) = O \left( \frac{n^3}{\epsilon} \right)$ and is bounded by a polynomial in $1/\epsilon$
• The algorithm returns a solution whose value is at least $(1 - \epsilon)$ times that of an optimal solution

---

ALGORITHM 3.2:
An approximation scheme for the knapsack problem

1. $M \leftarrow \max_{i \in I} v_i$
2. $\mu \leftarrow \frac{\epsilon M}{n}$
3. $v'_i \leftarrow \lfloor v_i / \mu \rfloor$ for all $i \in I$
4. Run Algorithm 3.1 for knapsack instance with values $v'_i$
**Theorem 3.5:** Algorithm 3.2 is a FPTAS for the knapsack problem.

**Proof:** We need to show that the algorithm returns a solution whose value is at least \((1 - \epsilon)\) times the value of an optimal solution. Let \( S \) be the set of items returned by the algorithm. Let \( O \) be an optimal set of items. Certainly \( M \leq \text{OPT} \), since one possible solution is to put the most valuable item in a knapsack by itself. Furthermore, by the definition of \( v'_i \), \( \mu v'_i \leq v_i \leq \mu (v'_i + 1) \), so that \( \mu v'_i \geq v_i - \mu \).

Applying the definitions of the rounded data, along with the fact that \( S \) is an optimal solution for the values \( v'_i \), we can derive the following inequalities:

\[
\sum_{i \in S} v_i \geq \mu \sum_{i \in S} v'_i \\
\geq \mu \sum_{i \in O} v'_i \\
\geq \sum_{i \in O} v_i - |O|\mu \\
\geq \sum_{i \in O} v_i - n\mu \\
= \sum_{i \in O} v_i - \epsilon M \\
\geq \text{OPT} - \epsilon \text{OPT} = (1 - \epsilon)\text{OPT}.
\]
3.2 Scheduling jobs on identical parallel machines

- Earlier we saw that by first sorting the jobs in order of non-increasing processing requirement, and then using a list scheduling rule, we find a schedule of length guaranteed to be at most $4/3$ times the optimum.
- This result contains the seeds of a PTAS.
- For any given value of $\rho > 1$, we give an algorithm that runs in polynomial time and finds a solution of objective function value at most $\rho$ times the optimal value.

As earlier, let the processing requirement of job $j$ be $p_j$, $j = 1, \ldots, n$, and let $C^*_{\text{max}}$ denote the length of a given schedule with job completion times $C_j$, $j = 1, \ldots, n$; the optimal value is denoted $C^*_{\text{max}}$.
- Each processing requirement is positive integer.
- In the analysis of the list scheduling rule its error can be upper bounded by the processing requirement of the last job to complete.
- The $4/3$-approximation was based on this fact, combined with the observation that when each job’s processing requirement is more than $C^*_{\text{max}}/3$, this natural greedy-type algorithm actually finds the optimal solution.
We present an approximation scheme for this problem based on a similar principle. We focus on a specified subset of the longest jobs, and compute the optimal schedule for that subset. Then we extend that partial schedule by using list scheduling on the remaining jobs. We will show that there is a trade-off between the number of long jobs and the quality of the solution found.

Let $k$ be a fixed positive integer; we will derive a family of algorithms, and focus on $A_k$ among them. We partition the job set into two parts: the long jobs and the short jobs, where a job $\ell$ is considered short if $p_\ell \leq \frac{1}{km} \sum_{j=1}^{m} p_j$. This implies that there are at most $km$ long jobs. Enumerate all schedules for the long jobs, and choose one with the minimum makespan. Extend this schedule by using list scheduling for the short jobs. I.e., given an arbitrary order of the short jobs, schedule these jobs in order, always assigning the next job to the machine currently least loaded.
Consider the running time of algorithm $A_k$

To specify a schedule for the long jobs, we simply indicate to which of the $m$ machines each long job is assigned; thus, there are at most $m^{km}$ distinct assignments.

If we focus on the special case of this problem in which the number of machines is a constant (say, 100, 1,000, or even 1,000,000), then this number is also a constant, not depending on the size of the input.

Thus, we can check each schedule, and determine the optimal length schedule in polynomial time in this special case.

As in the analysis of the local search algorithm earlier, we focus on the last job $\ell$ to finish.

Recall that we derived the equality that

$$C_{\text{max}} \leq p_\ell + \sum_{j \neq \ell} p_j/m$$

The validity of this inequality relied only on the fact that each machine is busy up until the time that job $\ell$ starts.

To analyze the algorithm that starts by finding the optimal schedule for the long jobs, we distinguish now between two cases.
If the last job to finish (in the entire schedule), job $\ell$, is a short job, then this job was scheduled by the list scheduling rule, and it follows that the previous inequality holds.

Since job $\ell$ is short, and hence $p_\ell \leq \sum_{j=1}^{n} p_j/(mk)$, it also follows that $C_{\text{max}} \leq \sum_{j=1}^{n} p_j/m + \sum_{j=\ell}^{n} p_j/m \leq \left(1 + \frac{1}{k}\right) \sum_{j=1}^{n} p_j/m \leq \left(1 + \frac{1}{k}\right) C^*_{\text{max}}$.

If $\ell$ is a long job, then the schedule delivered is optimal, since its makespan equals the length of the optimal schedule for just the long jobs, which is clearly no more than $C^*_{\text{max}}$ for the entire input.

The algorithm $A_K$ can easily be implemented to run in polynomial time (treating $m$ as a constant).

**Theorem 3.6:** The family of algorithms $\{A_K\}$ is a polynomial-time approximation scheme for the problem of minimizing the makespan on any constant number of identical parallel machines.

A significant limitation is that the number of machines needs to be a constant.

It is not too hard to extend these techniques to obtain a PTAS even if the number of machines is allowed to be an input parameter.
The key idea is that we didn’t really need the schedule for the long jobs to be optimal.

We used the optimality of the schedule for the long jobs only when the last job to finish was a long job.

If we had found a schedule for the long jobs that had makespan at most $1 + \frac{1}{k}$ times the optimal value, then that would have been sufficient.

We will see how to obtain this near-optimal schedule for long jobs by rounding input sizes and dynamic programming, as we saw previously on the knapsack problem.

Let us first set a target length $T$ for the schedule.

As before, we also fix a positive integer $k$; we will design a family of algorithms $\{B_k\}$ where $B_k$ either proves that no schedule of length $T$ exists, or else finds a schedule of length $\left(1 + \frac{1}{k}\right)T$.

Later we will show how such a family of algorithms also implies the existence of a polynomial-time approximation scheme.

We can assume that $T \geq \frac{1}{m} \sum_{j=1}^{n} p_j$, since otherwise no feasible schedule exists.
The algorithm $B_k$ is quite simple
- We again partition the jobs into long and short, but require that $p_j > T/k$ for $j$ to be long
- We round down the processing requirement of each long job to its nearest multiple of $T/k^2$
- We will determine in polynomial time whether or not there is a feasible schedule for these rounded long jobs that completes within time $T$
- If yes, we interpret it as a schedule for the long jobs with their original processing requirements
- If not, we conclude that no feasible schedule of length $T$ exists for the original input

Finally, we extend this schedule to include the short jobs by using the list scheduling algorithm for the short jobs
- We need to prove that the algorithm $B_k$ always produces a schedule of length at most $\left(1 + \frac{1}{k}\right)T$ whenever there exists a schedule of length at most $T$
- When the original input has a schedule of length $T$, then so does the reduced input consisting only of the rounded long jobs (which is why we rounded down the processing requirements); in this case, the algorithm does compute a schedule for the original input
• Suppose that a schedule is found
• It starts with a schedule of length at most $T$ for the rounded long jobs
• Let $S$ be the set of jobs assigned by this schedule to one machine
• Since each job in $S$ is long, and hence has rounded size at least $T/k$, it follows that $|S| \leq k$
• Furthermore, for each job $j \in S$, the difference between its true processing requirement and its rounded one is at most $T/k^2$ (because rounded value is a multiple of $T/k^2$)

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

• Now consider the effect of assigning the short jobs: each job $\ell$, in turn, is assigned to a machine for which the current load is smallest
• Since $\sum_{j=1}^{\ell} p_j / m \leq T$, also $\sum_{j \neq \ell} p_j / m \leq T$
• Since the average load assigned to a machine is less than $T$, there must exist a machine that is currently assigned jobs of total processing requirement less than $T$
• So, when we choose the machine that currently has the lightest load, and then add job $\ell$, this machine’s new load is at most

$$p_\ell + \sum_{j \neq \ell} p_j / m < T/k + T = \left(1 + \frac{1}{k}\right)T$$
Hence, the schedule produced by list scheduling will also be of length at most \((1 + 1/k)T\).

To complete \(B_k\), we must still show that we can use dynamic programming to decide if there is a schedule of length \(T\) for the rounded long jobs.

Clearly if there is a rounded long job of size greater than \(T\), then there is no such schedule.

Otherwise, describe an input by a \(k^2\)-dimensional vector, where the \(i\)th component specifies the number of long jobs of rounded size equal to \(iT/k^2\), \(i = 1, \ldots, k^2\).

We know that for \(i < k\), there are no such jobs, since that would imply that their original processing requirement was less than \(T/k\), and hence not long.

So there are at most \(n^{k^2}\) distinct inputs— a polynomial number!

**How many distinct ways are there to feasibly assign long jobs to one machine?**

Each rounded long job still has processing time at least \(T/k\).

\[ \therefore \text{at most } k \text{ jobs are assigned to one machine} \]

Again, an assignment to one machine can be described by a \(k^2\)-dimensional vector, where again the \(i\)th component specifies the number of long jobs of rounded size equal to \(iT/k^2\) that are assigned to that machine.
Consider the vector \((s_1, s_2, \ldots, s_{k^2})\); we call it a **machine configuration** if
\[
\sum_{i=1}^{k^2} s_i \cdot iT / k^2 \leq T
\]

Let \(C\) denote the set of all machine configurations.

Note that there are at most \((k + 1)^k\) distinct configurations, since each machine must process a number of rounded long jobs that is in the set \(\{0, 1, \ldots, k\}\).

Since \(k\) is fixed, this means that there are a constant number of configurations.

Let \(\text{OPT}(s_1, \ldots, s_{k^2})\) be the min # of machines sufficient to schedule this arbitrary input.

This value is given by the following recurrence (assign some jobs to one machine, and then using as few machines as possible for the rest):
\[
\text{OPT}(s_1, \ldots, s_{k^2}) = 1 + \min_{(s_1, s_2, \ldots, s_{k^2}) \in C} \text{OPT}(n_1 - s_1, \ldots, n_{k^2} - s_{k^2})
\]

View as a table with a polynomial number of entries (one for each possible input type).

To compute each entry, find the minimum over a constant number of previously computed values.

The desired schedule exists exactly when the corresponding optimal value is at most \(m\).
Finally, one can convert the family of algorithms \( \{B_k\} \) into a PTAS.

**Theorem 3.7:** There is a PTAS for the problem of minimizing the makespan on an input number of identical parallel machines.

- Note that since we consider \((k + 1)^{k^2}\) configurations and \(k = \lceil 1/\epsilon \rceil\), the running time in the worst case is exponential in \(O(1/\epsilon^2)\).
- Thus, in this case, we did not obtain a fully polynomial-time approximation scheme (in contrast to the knapsack problem).

This is for a fundamental reason:

- This scheduling problem is strongly NP-complete.
- Even if we require that the processing times be restricted to values at most \(q(n)\), a polynomial function of the number of jobs, this special case is still NP-complete.
- We claim that if a fully polynomial-time approximation scheme exists for this problem, it could be used to solve this special case in polynomial time, which would imply that \(P = NP\).