7.1 Knapsack Problem

- **Input**: Items $[n]$, profits $f_p : [n] \to \mathbb{N}$, sizes $f_s : [n] \to \mathbb{N}$, and the size of the knapsack $k \in \mathbb{N}$.
- **Feasible Solutions**: $I \subseteq [n]$ such that $\sum_{i \in I} f_s(i) \leq k$
- **Objective**: Maximize $\sum_{i \in I} f_p(i)$

7.1.1 Greedy Algorithms for the Knapsack Problem

Algorithm 1 Greedy KNAPSACK Algorithm 1

Input: $[n]$, f_p, f_s, k. We assume $[n]$ is sorted by nonincreasing $\frac{f_p(i)}{f_s(i)}$
Output: $I \subseteq [n]$

```
I ← ∅
for i ∈ [n] do
  if Σ_{j∈I} f_s(j) + f_s(i) < k then
    I ← I ∪ {i}
  end if
end for
return I
```

Theorem 7.1.1 *This algorithm is an $\Omega(k)$-approximation.*

Proof: Consider an instance of this problem such that there are exactly two items, such that $f_s(1) = 1$, $f_p(1) = 2$, $f_s(2) = k$, and $f_p(2) = k$. In such an instance, the algorithm chooses item 1 instead of 2 (the optimal solution). Thus the value of the algorithm is 2 while the optimal value is k, so it is no bettr than a $k/2$-approximation. ■

This algorithm performed poorly because the item right *after* it got stuck was the actual correct item to add. The next algorithms adds the items in greedy order (as before), but once it gets stuck it simply compares what it chose to the next item and chooses the better of the two.

Algorithm 2 Greedy KNAPSACK Algorithm 2

Input: $[n]$, f_p, f_s, k. We assume $[n]$ is sorted by nonincreasing $\frac{f_p(i)}{f_s(i)}$
Output: $I \subseteq [n]$

```
i = 1
while ∑_{j<i} f_s(j) + f_s(i) < k do
  i ← i + 1
end while
if ∑_{j<i} f_p(j) > f_p(i) then
  return {1, 2, ..., i − 1}
else
  return {i}
end if
```

Theorem 7.1.2 *This algorithm is an 2-approximation.*

Proof: Let i^* be the first item that cannot fit into the knapsack. Because the items are arranged in decreasing order with respect to their "bang-for-buck", $[i^*]$ is the minimum set of maximized "bang-for-buck" items, such that the sum of their sizes exceeds k. Clearly, this value is strictly greater than OPT. More formally, $f_p(i^*) + \sum_{i=1}^{i^*-1} f_p(i) > OPT$. Hence

$$\max\left(f_p(i^*), \sum_{i=1}^{i^*-1} f_p(i)\right) \geq \frac{OPT}{2}$$

This algorithm chooses $\max\left(f_p(i^*), \sum_{i=1}^{i^*-1} f_p(i)\right)$, so it is a 2-approximation. ■

7.1.2 Pseudo-Polynomial Algorithm for the Knapsack Problem

$M[i][v]$ denotes the minimum size of a subset of $[i]$ that yields profit exactly v. Then

$$M[i][v] = \begin{cases} 0 & : i = 0, v = 0 \\ \infty & : i = 0, v > 0 \\ M[i-1][v] & : i > 0, f_p(i) > v \\ \min\left(M[i-1][v], f_s(i) + M[i-1][v - f_p(i)]\right) & : i > 0, f_p(i) \leq v \end{cases}$$

Note that if we can compute this function, we can solve the knapsack problem by doing a binary search over the possibilities for v to find the largest v such that $M[n][v] \leq k$. Let $S = \max_{i \in [n]} f_p(i)$. Using dynamic programming, we can clearly compute the above function in time $O(n^2 S)$, since there are at most nS options for v, at most n options for i, and evaluating a single table entry takes $O(1)$ time. Because of the factor of S, the runtime of this algorithm is pseudo-polynomial and (and hence knapsack is only weakly NP-complete).

7.1.3 Fully Polynomial-time Approximation Scheme for the Knapsack Problem

A fully polynomial-time approximation scheme (FPTAS) is an algorithm which takes an instance of an optimization problem and a parameter $\epsilon > 0$, and, in time polynomial in the instance size and $1/\epsilon$, produces a solution that is within a factor of $1 + \epsilon$ of being optimal. Note that this the *fully* in this definition is what requires the running time to be polynomial in $1/\epsilon$ – later we will see PTASes which run in times like $n^{1/\epsilon}$ and hence are not FPTASes. We will now construct an FPTAS for Knapsack.

Let $S = \max_{i \in [n]} f_p(i)$ and let $\delta = \frac{\epsilon S}{n}$. We construct a "rounded" instance, in which $f_{p'}(i) = \lfloor \frac{f_p(i)}{\delta} \rfloor$ for all $i \in [n]$ and $S' = \max_{i \in [n]} f_{p'}(i)$. Think of $f_{p'}(i)$ as first scaling $f_p(i)$ so the largest profit is approximately n/ϵ and then rounding down to an integer value.

Note that $f_{p'}(i)$ compresses $f_p(i)$ and also applies rounding (resulting in a loss of precision). Our algorithm simply runs the previous dynamic programming algorithm, substituting $f_{p'}$ for f_p.

Theorem 7.1.3 *This algorithm runs in $O(n^3/\epsilon)$ time.*

Proof: We know from the last subsection that the running time is $O(n^2)$ times the value of the largest profit. Hence it is at most $O(n^2 S') = O\big(n^2 \lfloor \frac{S}{\delta} \rfloor\big) = O\Big(n^2 \Big\lfloor \frac{S}{\frac{\epsilon S}{n}} \Big\rfloor\Big) = O\big(\frac{n^3}{\epsilon}\big)$. ■

Theorem 7.1.4 *This algorithm is a $(1-\epsilon)$-approximation.*

Proof: Let I be the solution returned from this algorithm, and let I^* denote the optimal solution (under the original profits f_p). First note that I is a feasible solution – since we did not change the sizes, the total size of items in I must be at most k. To bound the total profit, note that from the definition of $f_{p'}$, we know that $f_p(i) \geq \delta f_{p'}(i)$. Hence

$$\sum_{i \in I} f_p(i) \geq \delta \sum_{i \in I} f_{p'}(i) \geq \delta \sum_{i \in I^*} f_{p'}(i),$$

since I is the optimal solution under the scaled and rounded $f_{p'}$ profits. But now from the definition of $f_{p'}$ we have that $f_{p'}(i) \geq (f_p(i)/\delta) - 1$. Thus

$$\begin{aligned} \delta \sum_{i \in I^*} f_{p'}(i) &\geq \delta \sum_{i \in I^*} \left(\frac{f_p(i)}{\delta} - 1 \right) = \sum_{i \in I^*} f_p(i) - \delta |I^*| \geq OPT - \delta n = OPT - \epsilon S \\ &\geq OPT - \epsilon OPT = (1-\epsilon) OPT \end{aligned}$$

Putting these together, we get that the profit of the algorithm, $\sum_{i \in I} f_p(i)$, is at least $(1-\epsilon)OPT$. ■

Theorem 7.1.5 *This algorithm is an FPTAS.*

Proof: By 7.1.3 and 7.1.4 this algorithm runs in time polynomial in n and $1/\epsilon$, and produces a solution that is within a factor of $1 - \epsilon$ of being optimal. ■

7.2 Min-Makespan on Identical Parallel Machines

- **Input**: Jobs $J = \{j_1, \ldots, j_n\}$, machines $M = \{m_1, \ldots, m_k\}$, and processing times $f_p : J \to \mathbb{R}$.
- **Feasible Solutions**: $\Phi : J \to M$
- **Objective**: Minimize makespan $\implies$ Minimize $\max_{m \in M} \left(\sum_{j \in \Phi^{-1}(m)} f_p(j) \right)$

Algorithm 3 Greedy Min-Makespan Algorithm

Input: J, M, f_p
Output: $\Phi : J \to M$

```
Φ ← ∅
for j ∈ J do
    m ← least loaded machine in Φ
    Set Φ to assign job j to machine m.
end for
return Φ
```

Theorem 7.2.1 *This algorithm is a 2-approximation.*

Proof: Consider the following two simple observations:

1. $\forall j \in J, f_p(j) \leq \text{OPT}$ (since every job needs to be scheduled on some machine)
2. Let $I \subseteq J$. Then $\frac{\sum_{j \in I} f_p(j)}{|I|} \leq \text{OPT}$ (the makespan, i.e. the maximum load, must be at least the average load of any subset of jobs)

Let m be the machine whose load equals the makespan of the greedy algorithm (i.e. the most heavily loaded machine after the algorithm completes). Let j be the last job assigned to m by the algorithm. Then just before j was assigned to m, by the second observation we know that the load of m was at most OPT. By the first observation the processing time for j is at most OPT, and since j was the last job assigned to m we get that the total load on m is at most 2 OPT. ■