> Sections (like this one) which marked with a vertical line on the left side are commentary — discussions about the algorithm development process that wouldn't be part of a writeup.

0-1 Knapsack

Establish the problem.

> Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *Specifications.*

  > State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each? For optimization problems, identify and distinguish between legal solutions and optimal ones.

  Given n items, each with a value v_i and weight w_i, find the maximum value set of items that fit in a knapsack with capacity W. Only whole items can be taken.

  Input: n items, each with a value $v_i > 0$ and weight $w_i > 0$; knapsack capacity $W > 0$

  Output: a subset of the items

  Legal solution: a subset of items with total weight $\leq W$

  Optimization goal: maximize total value of items taken

- *Examples.*

  > If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

Identify avenues of attack.

> Determine the algorithmic approach(es) to try.

- *Targets.*

  > Backtracking algorithms are fundamentally exponential; with dynamic programming we hope to reduce the problem to polynomial or pseudopolynomial, but that is not always possible.

- *Paradigms and patterns.*

  Paradigm: dynamic programming.

  > (This is dictated.)

  > Consider the backtracking patterns.

  This is a subset task — which of the items to include in the pack.

  This is a find-the-best-solution task.

  Produce output: repeatedly find the next item to take

  Process input: for the next item, take it or not?

- *The series of choices.*

  > Identify the series of choices. Take into account efficiency considerations when choosing between alternatives.

> For subset problems, process input results in a branching factor of 2. While there will be n decisions required (one about each item), $O(2^n)$ is likely to be more efficient than repeatedly choosing the next item (a branching factor of up to n and a solution path length also up to n).

For the next item, take it or not?

Define the algorithm.

> Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Size.*

  > The size of the problem is the number of choices left to make. The smallest problem size is 0 (a complete solution).

  The number of choices left to make = the number of items left to make a decision about.

- *Generalize / define subproblems.*

  - *Partial solution.*

    > What constitutes a solution-so-far? This will be the same kind of thing as a legal solution for the whole problem, but less complete (fewer choices have been made).

    The set of items taken so far.

  - *Alternatives.*

    > What are the legal alternatives for the next choice?

    If there's room in the pack for the item, take it or not. Otherwise, the only choice is not to take it.

  - *Subproblem.*

    > Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified. Take into account efficiency considerations when defining the input and output.
    >
    > For dynamic programming, the subproblem's output is just the solution for the subproblem (not a complete solution).

    > The subproblem task is reframed slightly — we're no longer trying to find the best complete set of items, but rather the best way to fill the rest of the pack. The output will also be just the value of the items picked, since the items themselves can be found after the array has been filled in.

    The subproblem's task is to solve the rest of the problem in light of the choices made so far — picking of items from amongst those not yet considered as to maximize the total value of those items, given the remaining capacity of the pack.

    Task: `knapsack(`S'`,`W'`)` — find the highest-value subset of items in S' whose total weight does not exceed the remaining capacity W' of the knapsack

    Input: set S' of items left to consider, remaining (unfilled) capacity of the knapsack W'

    Output: total value (of those picked from S')

    Legal solution: a subset of items with total weight $\leq W'$

    Optimization goal: highest value

  - *Memoization.*

    > Identify how to parameterize subproblem state for efficient lookup, typically in an array.

    > The subproblem is defined by S' and W' — the set of items to choose from and the remaining capacity of the pack. Two parameters means a 2D array. We need to figure out how to represent a set of items and a remaining capacity as array indexes (integers ≥ 0).

Subproblems are defined by the set S' of items left to consider and the remaining capacity W' of the pack.

The order items are considered in doesn't matter, so S' can be represented with an array of all of the items in S and an index k — S' would be the elements $S[k..n-1]$, inclusive.

If the pack's capacity W' and the weights w_i are all integers, W' will be integer and can be used directly as an array index since it will be ≥ 0.

We will use `V[k][w]` to store the highest value possible from the items `S[k..n-1]` and pack capacity `w`. The initial subproblem is `V[0][W]`.

- *Base case(s).*

  A complete solution has been found — address what to do with it.

  In the original backtracking formulation, we had: A complete solution is when all of the items have been considered — $|S'| = 0$. Return I and V.

  Translate this into the scheme of returning just the subproblem solution and the memoization notation.

  A complete solution is when all of the items have been considered — $|S'| = 0$ and thus $k = n$. The total value is 0 because there are no more elements left from which to pick. `V[n][w] = 0` for $0 \leq w \leq W$.

- *Main case.*

  Address how to solve a typical large problem instance (one that is not a base case).

  In the original backtracking formulation, we had:

  ```
  let s be an element of S'
  if w_s ≤ W'
    (I1,V1) ← knapsack(I ∪ {s},V + v_s,S' − {s},W' − w_s)    // take s
  (I2,V2) ← knapsack(I,V,S' − {s},W')                        // don't take s
  if V1 > V2 return (I1,V1) otherwise return (I2,V2)
  ```

  Translate this into the scheme of returning just the subproblem solution and the memoization notation.

  The current subproblem is `V[k][w]` where $S' = S[k..n-1]$ and $W' = w$.

  ```
  if w_k ≤ w   // item k fits in the pack - consider both take and don't take
    V[k][w] = max(V[k+1][w-w_k]+v_k,V[k+1][w])
  else         // item k doesn't fit in the pack - consider don't take only
    V[k][w] = V[k+1][w]
  ```

- *Top level.*

  The top level puts the context around the recursion.

  - *Initial subproblem.*

    Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem.

    `knapsack({},0,S,W)` — S is the original set of items, W is the original capacity of the pack

  - *Setup.*

    Whatever must happen before the initial subproblem is solved.

    Nothing to do.

  - *Wrapup.*

> Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

Nothing to do — just return the items returned.

- *Special cases.*

> Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).

If all of the items fit in the pack, take them all. This will be detected in the normal course of the algorithm, but could be checked specifically at the first step to greatly improve performance in that case without much impact to the running time when the items don't all fit.

The case of none of the items fitting will be detected in the first main case, so no need to handle it specially.

The cases of no space left in the pack or not enough space left for any of the remaining items can be handled through pruning, but don't require special handling and don't need to be included as base cases.

Our memoization scheme requires integer weights for the items and an integer capacity for the pack. If the weights aren't integer, the problem can be solved with an arbitrary degree of precision by multiplying W and w_i by a power of 10 — with a corresponding increase in time and space.

- *Algorithm.*

> Assemble the algorithm from the previous steps and state it.
>
> There shouldn't be new elements here, instead bring together the base case(s), main case, and top level with any handling needed for special cases and state the whole algorithm.

> The base case is $k = n$, so the loop for k goes from large $(n-1)$ to small (0). The order of computation for w doesn't matter because `V[k][...]` is defined in terms of `V[k+1][...]`.

`algorithm knapsack(`S`,`W`)` — find the highest-value subset of items in S whose total weight does not exceed the capacity W of the knapsack

Input: set S of n items, each with a value $v_i > 0$ and weight $w_i > 0$; knapsack capacity $W > 0$

Output: a subset of the items

```
// initialize the base cases - k=n
for w = 0 to W do
  V[n][w] = 0

// fill in the rest of the array
for k = n-1 downto 0
  for w = 1..W
    if w_k ≤ w  // item k fits in the pack - consider both take and don't take
      V[k][w] = max(V[k+1][w-w_k]+v_k,V[k+1][w])
    else        // item k doesn't fit in the pack - consider don't take only
      V[k][w] = V[k+1][w]

return the set of items for V[0][W]
```

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.*

> Show that the recursion — and thus the algorithm — always terminates.

The fill-in-the-array loops are counting loops and the array is finite, so all of the slots will eventually be filled and the loops will end. Reconstructing the solution involves one repetition per step in the solution path, which is also finite.

- *Correctness.*

  > Show that the algorithm is correct.

  - *Establish the base case(s).*

    > Explain why the solution is correct for each base case: the right thing is done with a complete solution.

    There aren't any more items to consider, so there's nothing else to pick. The max value obtainable from no items is 0.

  - *Show the main case.*

    > Explain why the desired solution will be found (all possible legal alternatives for the next choice are either considered or are safe to skip) and showing that the correct partial solutions and subproblems are handed to the friends.

    All legal alternatives are considered: Since the solution is a subset of the input items, every item is either in that subset or not. Adding the item to the pack is only considered if there's room for it.

    The right subproblems: s is removed from the set of items left to be considered, and its weight is deducted from the remaining capacity if it is taken.

    The friends return the complete solutions resulting from choosing s or not choosing s, so we simply need to return the better of the two.

  - *Final answer.*

    > Explain why the top level — the setup plus a correct solution to the initial subproblem followed by the wrapup — means that the final result is a correct answer to the problem. For the initial subproblem, establish that the correct input is provided – the (empty) solution-in-progress is legal and the rest of the problem is actually the whole problem.

    The initial subproblem `V[0][W]` is the original problem, and the subset of elements to include is the desired answer. There is no setup or wrapup to consider.

Determine efficiency.

> Evaluate the running time and space requirements of the algorithm.

- *Implementation.*

  > Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.

- *Time and space.*

  > Assess the running time and space requirements of the algorithm given the implementation identified.

The array has nW slots and computing each value is $O(1)$, so the total running time is $O(nW)$. The space requirements are also $O(nW)$.

> This is *pseudopolynomial* — W doesn't depend on n, but it also isn't a constant factor that can just be eliminated. However, $O(nW)$ it is still a vast improvement over $O(2^n)$ in most cases, though space requirements become an issue if W is large.

- *Room for improvement.*

  > Algorithm design decisions — the series of choices pattern, the specific input and output for the subproblems — should have already been considered, but revisit them if not.