> 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.

- *Paradigms and patterns.*

  Paradigm: backtracking.

  > (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.

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

    Task: `knapsack(`I`,`V`,`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 I of items already picked and their total value V, set S' of items left to consider, remaining (unfilled) capacity of the knapsack W'

    Output: items chosen ($I \cup$ those picked from S'), total value (V + value of those picked from S')

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

    Optimization goal: highest value

- *Base case(s).*

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

  A complete solution is when all of the items have been considered — $|S'| = 0$. Return I and V.

- *Main case.*

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

```
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)
```

- *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.

- *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.

  `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

  ```
  (items,value) ← knapsack({},0,S,W)
  return items
  ```

`algorithm knapsack(`I`,`V`,`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 I of items already picked and their total value V, set S' of items left to consider, remaining (unfilled) capacity of the knapsack W'

Output: items chosen ($I \cup$ those picked from S'), total value ($V+$ value of those picked from S')

```
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)
```

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

- *Termination.*
  Show that the recursion — and thus the algorithm — always terminates.
  - *Making progress.*
    Explain why what each of your friends get is a smaller instance of the problem: in each step, another choice is made.

    The friends get $S' - \{s\}$ so the set of elements is one smaller.
  - *The end is reached.*
    Explain why a base case is always reached: eventually all choices have been made.

    The size of S' is decreased by one in each subsequent subproblem, so eventually $|S'| = 0$.
- *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 the partial solution is complete. I and V are that complete solution.
  - *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. s is also added to the set of items in the partial solution and its value is added to the total vale of the partial solution 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 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.

  We pass $S' - \{s\}$ to the friends. Since it doesn't matter what order we consider the elements in, s can be picked however is convenient. We can represent S' with an array `S` containing all of the items and an index `k` — $S' = S[k..n-1]$. Pick `s = S[k]`, thus $S' - \{s\}$ is `S[k+1..n-1]`.

  We do need to update the set of items in the partial solution for the subproblems, but if "don't take" is considered first the work adding/removing s can be minimized — pass I to the "don't take" subproblem, add s to I and pass $I \cup \{s\}$ to the "take" subproblem, then remove s from I if "don't take" yields the better result.

  There are collections that support $O(1)$ add and remove of the just-added item (appending to an array-based list, a stack, a hashtable implementation of a set).

- *Time and space.*

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

  With the implementation described, the work other than the recursive calls is O(1).

  The branching factor is 2 (take or not take) and the longest path length is n (the number of items), leading to $\Theta(2^n)$ overall.

- *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.

  The produce output approach results in a branching factor of $\leq n - k$, where k is the number of items already chosen, and the longest path length is $\leq n$. Which is better depends on how many items will fit in the pack, but since $n - k \geq 2$ until the very end, the process input approach is likely to be better unless there's only room for a few items.