1 Dynamic Programming: Multi-Parameter Problems

Subset Sum: A Weighted Scheduling Problem

Suppose now that our jobs don’t have start/end times, but merely durations, and that the resource is available for a limited amount of time. Is the problem easier or more difficult? Can we still use dynamic programming?

The Problem: Given \( n \) jobs \( J_1, \ldots, J_n \), each with duration \( w_i \) and a single resource available for a total amount of time \( W \), find a subset of jobs \( s_1, \ldots, s_k \) (each \( s_i \in \{1, \ldots, n\} \)) such that

1. \( w_{s_1} + \ldots + w_{s_k} \) is as large as possible, subject to
2. \( w_{s_1} + \ldots + w_{s_k} \leq W \)

This problem is typically called the subset sum problem, since it is essentially a problem about positive integers. Note that the "value" (or weight) of a job, for now, is just its duration. We’ll bring back separate job values later.

In this situation, as in the previous example, there appears to be no known natural greedy algorithm that successfully provides an efficient solution. But dynamic programming can be used.

The challenge is to find the right set of (not too many) subproblems to solve! In this situation, there is no obvious natural order to put on the jobs (small to large or large to small doesn’t seem to help). Let \( S(i) \) represent the weight of an optimal (maximum weight) solution to the problem using only jobs \( J_1, \ldots, J_i \). We want to determine \( S(n) \).

We begin with the obvious observation: An optimal solution either uses item \( J_n \) or it doesn’t.

If the solution doesn’t use \( J_n \) then \( S(n) = S(n - 1) \); that is, using a subset of the first \( n - 1 \) jobs, we seek a maximum-weight subset whose weight doesn’t exceed \( W \).

However, if the solution uses \( J_n \) then there will only be \( W - w_n \) units of time remaining and so the solution will have the value \( S(n) = w_n + "the best solution for J_1, \ldots, J_{n-1} that uses at most W - w_n time". \)

Hmmm. Our notation is failing us. Let’s try again:

Let \( S(i, w) \) denote the optimal solution to the problem that only uses jobs chosen from \( J_1, \ldots, J_i \) and does not exceed total weight \( w \).

Then we are interested in the value \( S(n, W) \).

- If the solution uses \( J_n \) then there will only be \( W - w_n \) units of time remaining and so \( S(n, W) = w_n + S(n - 1, W - w_n) \).
- If the solution doesn’t use \( J_n \) then \( S(n, W) = S(n - 1, W) \).

Thus

\[
S(j, w) = \max\{S(j - 1, w), w_j + S(j - 1, w - w_j)\}
\]

Note: this assumes \( w - w_j \geq 0 \). We’ll assume that \( S(i, w) = 0 \) if \( w \leq 0 \) or if \( i = 0 \).

So, to solve the problem, we merely need to build an \( n \times W \) array \( S[1..n, 1..W] \). Observe that

- It will be easier to make \( S \) an \( (n + 1) \times (W + 1) \) array \( S[0..n, 0..W] \), where \( S[0, -] = 0 = S[-, 0] \).
- The array will be built “row-by-row” and to some extent “left-to-right”. DRAW A PICTURE!
Whenever \( w - w_j < 0 \) we replace \( w - w_j \) with 0.

**Complexity** The table can clearly be built in time proportional to its size, so the time and space complexity of the algorithm are both \( O(nW) \).

So, is that good?

Not necessarily: \( W \) could be, say, \( 2^n \). If some constant number of weights, along with \( W \), were \( n \) bits long, the algorithm could be exponential, since the input would still be of size \( O(n) \) and the table of size \( O(n2^n) \).

An algorithm like this is called *pseudo-polynomial*.

**The Knapsack Problem**

We can continue to generalize our problem by adding, for each item \( i \), in addition to a weight \( w_i \), a value \( v_i \). The problem then becomes:

Find a subset of the items \( I \subset \{1, \ldots, n\} \) that maximizes \( \sum_{i \in I} v_i \) subject to satisfying the constraint \( \sum_{i \in I} w_i \leq W \).

The only thing that changes in our optimization formula is that \( S \) now holds values, not weights:

\[
S(j,w) = \max\{S(j-1,w), v_j + S(j-1,w-w_j)\}
\]

Note: this assumes \( w - w_j \geq 0 \).

And so the Knapsack problem can be solved with the same technique used for the Subset Sum problem.

Frequently, finding a good decomposition into sub-problems requires generalizing the form of the subproblem beyond, say, describing \( Opt(n) \) in terms of \( Opt(k) \) for some set of \( k \)'s smaller than \( n \).

We’ll look at more examples, the next of which will be an important problem on strings.

## 2 The Secondary Structure of RNA

RNA molecules consist of long strands of nucleotides of four types: adenine, guanine, cytosine, an uracil.

Unlike DNA, RNA appears in single strands, communicating information in the cell and directing production of proteins.

Understanding the structure of RNA is the focus of much research.

A number of the fundamental questions that arise are, in large part, complex string processing questions.

The *primary structure* of RNA just refers to the sequential ordering of the nucleotides on a single strand of RNA.

The *secondary structure* of RNA refers to the particular hydrogen-bond pairings between nucleotides on a single strand of RNA: cytosine bonds with guanine (C-G) and adenine bonds with uracil (A-U).

We could represent a strand of RNA by a string \( \sigma = \sigma_1 \ldots \sigma_n \) over the alphabet \( \{A,C,G,U\} \).

The secondary structure might, schematically, look like this: DRAW A PICTURE!

RNA seems to follow certain rules with respect to secondary structure:

- Bonds are A-U and C-G
- Bonds do not cross: If we have pairs \( \sigma_i - \sigma_j \) and \( \sigma_k - \sigma_l \), the sequence \( i < k < j < l \) is forbidden.
• Geometry prevents any bond \( \sigma_i - \sigma_j \) in which \( |i - j| < 4 \).
• Therefore, the bonded pairs (base pairs) form a (non-perfect) matching.

Researchers are concerned with how much free energy (energy that can be converted into work) can be stored in a strand. One way of modeling this quantity is by counting the number of bond pairs—the more bond pairs, the more free energy.

**The Problem:** Given a strand of RNA, find the secondary structure that maximizes the free energy.

We begin, as seems to be typical with a stab at notation and an obvious observation.

**First Attempt**

Let \( \text{Opt}(n) \) be the size of the maximum matching of base pairs for a given strand \( \sigma = \sigma_1 \ldots \sigma_n \) subject to the constraints.

Can we identify an optimality relation that would let us apply dynamic programming?

**Observation 1.** Either \( \sigma_n \) bonds with some \( \sigma_t, t < n \) or it doesn’t.

- If it does not, then \( \text{Opt}(n) = \text{Opt}(n-1) \).
- If it does, then we have two sub-problems: one on \( \sigma_1 \ldots \sigma_{t-1} \) and one on \( \sigma_{t+1} \ldots \sigma_{n-1} \).

The second case breaks our notation (again)!

**Second Attempt**

Let \( \text{Opt}(i,j) \) be the size of the maximum matching of base for a portion \( \sigma_i \ldots \sigma_j \) of strand \( \sigma \), satisfying the constraints.

**Observation 2.** Either \( \sigma_j \) bonds with some \( \sigma_t, i \leq t < j \) or it doesn’t.

- If it does not, then \( \text{Opt}(i,j) = \text{Opt}(i,j-1) \).
- If it does, then we have two sub-problems: one on \( \sigma_i \ldots \sigma_{t-1} \) and one on \( \sigma_{t+1} \ldots \sigma_{j-1} \).
- So \( \text{Opt}(i,j) = \max_t \{ \text{Opt}(i,j-1), 1 + \text{Opt}(i,t-1) + \text{Opt}(t+1,j-1) \} \) where \( i \leq t \leq j-4 \) and \( \sigma_t \sigma_j \) is a legal pair.

Why is this the correct optimality condition?

All of the constraints appear either explicitly or implicitly in our optimality condition. Moreover, the constraint that there is no bond \( \sigma_i - \sigma_j \) when \( |i - j| < 4 \) also defines the base cases of our \( \text{Opt()} \) recurrence: \( \text{Opt}(i,j) = 0 \) if \( |i - j| < 4 \).

Note that we have become more creative in choice of subproblems; this is common.

It might help to visualize the data: **DRAW A TRIANGULAR TABLE WITH MANY ZEROS!**

Each element of the table can be built in \( O(n) \) time, and there are \( O(n^2) \) entries, giving an \( O(n^3) \) algorithm.

**Did We Over-Count?**

An element on diagonal \( k = j - i - 4 \) requires, approximately \( k \) comparisons.

There are \( n - k + 1 \) entries on diagonal \( k \). This gives \( \sum_{k=4}^{k=n-5} (n-k+1)(k-4) = \frac{3-n^2-64n+576}{6} \).

**Theorem 1.** The free energy of the optimal secondary structure for a strand of RNA of length \( n \) can be computed in \( O(n^3) \) time via dynamic programming.

---

1 According to Wolfram Alpha!
Note that as usual, we can compute not just the size of the matching but the matching itself either by including additional information in the table or by unwinding the recurrence using the table. The former method adds $O(n)$ time; the latter $O(n^2)$ time.

A Note on Building Partial Solutions

Can we better manage the additional storage produced by computing actual solution sets to subproblems?

Sure: The recurrence relation imposes the structure of a DAG on the entries of the array, where each entry "points to" the entries that it depended on to compute its own optimal value.

So, just have each table entry also store the "marginal" portion of the solution that the entry contributed, and then traverse this graph to find the solution.