Dynamic Programming – The first few steps

Appie van de Liefvoort©, CSEE, UMKC

October 15, 2018

Please study the foundation of Dynamic Programming (DP) in your text book. The examples and exercises that follow are intended to demonstrate the algorithms for very specific problems, and to prepare you to apply these. They are highly condensed and give only incidental insight in Dynamic Programming as a concept. For a solid working knowledge you need additional explanations and examples so that you apply this design principle in a different problem settings.

1 What to expect

There are two observations worth making at this time. First, a problem of size n for which DP is appropriate can often be divided into a number of smaller subproblems, just like Divide&Conquer. Unless the D&C type problems, the smaller subproblems do overlap are not independent of one another. At the algorithmic level, a pure recursive implementation gives arguably the best insight, and is arguably easiest to prove correct with general induction. At the implementation level however, a program that adopts recursion will repeat certain calculations. To avoid solving an identical (sub-)problem multiple times, solve the smallest sub-problems first while storing their solutions, then solve slightly larger sub-problems while storing their solutions, and so on. There is a time-space trade-off: Solve small problems and store their solutions in appropriate data structures, which can then be used and incorporated in the solution of larger problems.

Some languages incorporate a memorization feature (MAPLE for instance has an option *remember* for their procedures) which will free the programmer from writing iterative implementations of recursive algorithms and free the programmer from designing efficient data structures for intermediate results. In general, a characteristic of DP is therefore: The algorithm is best presented (and proven correct) recursively, but when implementing it, be sure to use the memorization feature of your language or write the programs carefully with an iteration while building efficient data structures for finding the smaller solutions back. In this hand-out we will not use a memorization approach and guide you to iterative implementations which are then analyzed.

Secondly, just about all problems for which dynamic algorithms have been designed are optimization problems, where the minimum or maximum value is sought over the various combinatorial structures. When trying to find such an optimal value, generating all possible combinatorial possibilities would be too costly, it is typically exponential in cost. Thus rather than generating *all* possibilities, these possibilities are decomposed and grouped together in sub-problems using common features, and the solution to these sub-problems are stored in a table and referred to when needed.

Furthermore, knowing the optimal cost is only the first part of the problem. The follow-up second question is then "Which particular structure gives rise to this optimal value?" For instance, the optimization question is "Determine the shortest path between S and T", and the follow-up question is "Determine an actual path, whose path-length is equal to the minimal path length." There are two passes needed for these two related questions: The first pass determines the optimum value, and the second pass constructs an optimal solution. The optimum value is unique (so it makes sense to use the word "optimum", which implies uniqueness), whereas there may be multiple constructs which give rise to the same optimum value, and these constructs are called optimal constructs (the word does not imply uniqueness). Often the second pass is performed most efficient when certain pieces of decision information is stored during the first pass, which can then be used in the second pass. This additional information might at first be confusing, but they are already included in the short notes below. The second pass typically starts at the end of the structure: For instance, an actual optimal path in the shortest path problem is constructed by starting from the terminal node T, and following the stored information back to the starting node S. Thus most dynamic programming algorithms incorporate tracking information to allow the easy (re-)construction of optimal configuration. Such tracking information is stored in a relevant data structure, such as labels, arrays, tables, and so on, depending on the circumstance.

These remarks will make more sense after you understand several dynamic programming algorithms.

I apologize in advance for typo's that are ever present.
Please let me know the ones you find and you find misleading or irritating.

Contents

2 DP: Motivating Simple Cases

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

2.1 DP: Binomial Numbers

Take for instance the computation of $\binom{30}{10}$. You might be tempted to evaluate Pascal's Triangle in a row-by-row fashion

```
algorithmBinCoef.v1($n, k$)
for $row$ from $1$ to $n$ do
    $BC[row, 0] \leftarrow 1$
    for $entry$ from $1$ to $\min\{k, row-1\}$ do
        $BC[row, entry] \leftarrow BC[row-1, entry-1] + BC[row-1, entry]$
    end-for
    $BC[row, row] \leftarrow 1$
end-for
end algorithmBinCoef.v1
```

This appears a straightforward approach to finding the binomial coefficients. A careful inspection of the Pascal's Triangle, you can observe that the above algorithm computes more coefficients as needed, and costs $T(n) = \frac{1}{2}\, n^2$ (or $T(n) = \frac{1}{4}\, n^2$ if you use the symmetry of the coefficients). Note, it does not really depend on k. Can this be done cheaper? Using the formal definition, we need

$$\begin{aligned}
\binom{30}{10} &= \binom{29}{9} + \binom{29}{10} && (1)\\
&= \binom{28}{8} + 2\binom{28}{9} + \binom{28}{10} && (2)\\
&= \binom{27}{7} + 3\binom{27}{8} + 3\binom{27}{9} + \binom{27}{10} && (3)\\
&= \binom{26}{6} + 4\binom{26}{7} + 6\binom{26}{8} + 4\binom{26}{9} + \binom{26}{10} && (4)\\
&= \vdots\\
&= \binom{20}{0} + \binom{10}{9}\binom{20}{1} + \binom{10}{9}\binom{20}{2} + \binom{10}{9}\binom{20}{3} + \ldots + \binom{10}{2}\binom{20}{8} + \binom{10}{1}\binom{20}{9} + \binom{20}{10} && (5)
\end{aligned}$$

So to compute $\binom{30}{10}$ we only need all the elements of Pascal's Triangle that are on row 10. After that, we only need the first 11 entries on rows 11 through 20 and we can use the expression on the last row. The time complexity is now $T(n) \sim \frac{1}{2}\, k^2 + (n-2k)\, k$. In the worst case, $k = n/2$, so that the complexity is $T(n) = \frac{1}{8}\, n^2$.

There are other ways to find the binomial numbers, some are more elegant, others use products (multiplications and divisions). And in case you were wondering, $\binom{30}{10} = 30045015$.

2.2 DP: Fibonacci Numbers

There are (at least) four distinct methods to find the value for the n^{th} Fibonacci Number.

1. Recursive. At a cost of $T(n) = \mathcal{O}(\textsc{Fib}(n)) = \mathcal{O}\left(1.61803^n\right)$
2. Iterative. At a cost of $T(n) = \mathcal{O}(n)$
3. Powering. At a cost of $T(n) = \mathcal{O}(\lg n)$, Use $\left(\begin{smallmatrix}1 & 1\\ 1 & 0\end{smallmatrix}\right)^n = \left(\begin{smallmatrix}F_{n+1} & F_n\\ F_n & F_n\end{smallmatrix}\right)$
4. Numeric. At a cost of $T(n) = \mathcal{O}(\ 1\ )$ Use $F_n = \textsc{Round}(\alpha\ \exp(n\ \ln(1.61803))$

These will be shown in class, and their pro's and cons discussed.

Suggested Exercise Write an algorithm for each of these four methods.

3 DP: Matrix Chain Multiplication Problem

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

Suppose you need to calculate the product of three matrices, $\mathbf{A}_1$, $\mathbf{A}_2$ and $\mathbf{A}_3$, of dimensions 8×3, 3×9, and 9×2 respectively. There are two different ways to calculate this, remembering that multiplying $\mathbf{A}_1$ and $\mathbf{A}_2$ costs $8\times 3\times 9 = 216$ multiplications and the result is a 8×9 matrix:

1. $\mathbf{A}_1 \odot \mathbf{A}_2 \odot \mathbf{A}_3 = \Big(\big(\mathbf{A}_1 \odot \mathbf{A}_2\big) \odot \mathbf{A}_3\Big)$ at a cost of $8\times 3\times 9 + 8\times 9\times 2 = 216 + 144 = 360$.
2. $\mathbf{A}_1 \odot \mathbf{A}_2 \odot \mathbf{A}_3 = \Big(\mathbf{A}_1 \odot \big(\mathbf{A}_2 \odot \mathbf{A}_3\big)\Big)$ at a cost of $3\times 9\times 2 + 8\times 3\times 2 = 54 + 48 = 102$.

So even though the answer is correct in both cases, one method is considerably cheaper than the other. Let us investigate this deeper.

Suppose you have n matrices $\mathbf{A}_1$ through $\mathbf{A}_n$, each with potentially differing dimensions. The matrix $\mathbf{A}_1$ has dimensions $d_0\times d_1$, matrix $\mathbf{A}_2$ has dimensions $d_1\times d_2$, and so on: matrix $\mathbf{A}_k$ has dimensions $d_{k-1}\times d_k$ and $\mathbf{A}_n$ has dimensions $d_{n-1}\times d_n$. Suppose you need to multiply them and calculate the product $\mathbf{A}_1 \odot \mathbf{A}_2 \odot \mathbf{A}_3 \odot \ldots \mathbf{A}_{n-1} \odot \mathbf{A}_n$.

There are a Catalan number of ways, $\frac{1}{n+1}\binom{2n}{n}$, to multiply this product and parenthesis are used to indicate the order in which these multiplications occur. (or, similarly, by using execution trees to indicate the operational order.) Each has perhaps a different execution cost in terms of t number of scalar operations, and finding the minimal cost by exhaustive enumeration is too costly. The **Matrix Chain Multiplication Problem** is the problem to find the minimal number of operations needed, for a given chain of matrices, with given dimensions. Additionally, tracking information is stored such that, in a second pass, the optimal order for the multiplication can be constructed. That is, the fully parenthesized expression (or equivalently, the execution tree) can be constructed so that the minimal number of scalar operations is attained, see also `https://en.wikipedia.org/wiki/Matrix_chain_multiplication`. Another good explanation is as `https://home.cse.ust.hk/~dekai/271/notes/L12/L12.pdf`

There are n matrices whose product needs to be computed: $\mathbf{A}_1 \odot \ldots \odot \mathbf{A}_n$. The matrix $\mathbf{A}_1$ has dimensions $d_0\times d_1$, matrix $\mathbf{A}_2$ has dimensions $d_1\times d_2$, and so on: matrix $\mathbf{A}_k$ has dimensions $d_{k-1}\times d_k$. Let $M[1\ ,\ n]$ be the minimal cost to evaluate the value for the chain $\mathbf{A}_1 \odot \ldots \odot \mathbf{A}_n$. If you know in advance that

$$\Big(\mathbf{A}_1 \odot \ldots \odot \mathbf{A}_p\Big) \ \odot\ \Big(\mathbf{A}_{p+1} \odot \ldots \odot \mathbf{A}_n\Big) \tag{6}$$

results in the smallest value for $M[1,\ n]$, then

$$M[1,\ n] = M[1\ ,\ p] + M[p+1\ ,\ n]\ + d_0 d_p d_n \tag{7}$$

is the smallest of all the possible ways to parenthesize the chain. You then also know that both $M[1\ ,p]$ and $M[p+1\ ,n]$ are the smallest possible. The index p indicates the location of the split.

However, if you do not know in advance that the matrix with index p is the last matrix included in the left sub-chain, then the index p is found as the index for which the expression is optimal when the index varies as an open parameter (let us use k). Thus every matrix $\mathbf{A}_k$ should be considered as a potential last matrix on the left, $\Big(\mathbf{A}_1 \odot \ldots \odot \mathbf{A}_k\Big)\ \odot\ \Big(\mathbf{A}_{k+1} \odot \ldots \odot \mathbf{A}_n\Big)$. The optimal value for $M[1,\ n]$ is then found by varying k . If A_1 is the only matrix on the left, then $M[1,1] = 0$ to indicate that there is no computational cost. With this, we have

$$M[1,\ n] \quad = \quad M[1\ ,\ p] + M[p+1\ ,\ n]\ + d_0 d_p d_n = \min_{k,\ \text{with}\ 1\le k\le n} \{\ M[1,\ k] + M[k+1,\ n]\ + d_0 d_k d_n\}$$

Once the value for p has been determined, we would like to track this information (i.e. store) so that an optimal Matrix Chain can be actually constructed from the stored tracking information. We often refer to p as the position or location where the expression will be split: the root of the execution tree. Notice also, that there are sometimes more than one execution tree that results in the same optimal value for for $M[1, n]$.

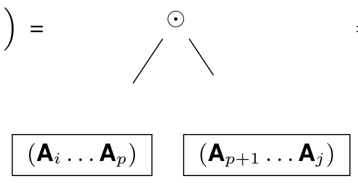
The expression for $M[1,\ n]$ relies on the values $M[1,\ k]$ and $M[k+1,\ n]$ being available for all k, and recursively for smaller chains as well. So we now present the formulation for any subset of consecutive matrices in the chain. Let $M[i,\ j]$ represent the minimal (unnormalized) cost that can be attained for a chain with the matrix chain $\mathbf{A}_i \odot \ldots \odot \mathbf{A}_j$. The driving equation for the minimal cost is then similarly,

$$\boxed{M[i\ ,\ j] = \begin{cases} 0 & j = i \\ d_{i-1} d_i d_{i+1} & j = i+1 \\ \min\limits_{k,\ \text{with}\ i\le k<j} \{\ M[i\ ,\ k] + M[k+1\ ,\ j]\ + d_{i-1} d_k d_j\} & j = i+2\ldots n \end{cases}} \tag{8}$$

Also, let us track the information gained from finding the minimal value and keep the value for k that indeed minimizes the expression: let this be p. (Technically, it depends on i and j as well, so it should be called $p[i,\ j]$). We will show the dependency on i and j since this is clear by context. It is the index of the last matrix in the left-subchain (or it is the p^{th} $\odot$-operation when counting matrix multiplications. It splits the matrix chain in a left- subchain and the right-subchain. Having all p's available allows the construction of the actual matrix chain execution tree, and thus this algorithm does not only find the optimum (i.e. minimum) value for the lowest cost possible for the computation of the matrix chain, it also provides a blueprint for the actual construction of the execution tree in a second phase. The fully parenthesized expression can be obtained, in this second pass, by executing a recursive algorithm, except that we need to track the information about "where best to split the expression", that is, we need to store the best value for p. This is done in a table MyP, where at index $MyP[i, j]$ you store the optimal value for p. It should be mentioned that there are sometimes multiple indices (read: execution trees) with the same minimal cost. Thus entries in the table MyP could be single indices, or a list of indices. The list could even reflect a priority ordering among options.

Anyway, the split at p is the best possible: $\Big(\mathbf{A}_i\ \mathbf{A}_{i+1}\ \ldots \mathbf{A}_{j-1}\ \mathbf{A}_j\Big) = \Big(\mathbf{A}_i \ldots \mathbf{A}_p\Big) \odot \Big(\mathbf{A}_{p+1} \ldots \mathbf{A}_j\Big)$
In terms of execution tree, the visualization is as:

$\Big(\mathbf{A}_i\ \mathbf{A}_{i+1}\ \ldots \mathbf{A}_{j-1}\ \mathbf{A}_j\Big) =$


$\odot$
$(\mathbf{A}_i \ldots \mathbf{A}_p)$
$(\mathbf{A}_{p+1} \ldots \mathbf{A}_j)$

$= \Big(\mathbf{A}_i \ldots \mathbf{A}_p\Big) \odot \Big(\mathbf{A}_{p+1} \ldots \mathbf{A}_j\Big)$

```
algorithm PrintParenths ($MyP$, $i$, $j$)
  if $i == j$
      then return ($MyP[i, j]$)
      else return concatenate("$\Big($" || PrintParenths($MyP$, $i$, $p$) || "$\Big)$" || $\odot$ || "$\Big($" || PrintParenths ($MyP$, $p+1$, $j$) || "$\Big)$" )
end-alg PrintParenths
```

Note also, that there may be two or more ways to split the expression into an optimal configuration. In this case, we show a list of possible positions where the expression can be broken up.

On the next pages, we show two examples.

3.1 Example 1

Consider an example with $n = 6$ matrices and dimensions $[d_0, d_1, \cdots d_6] = [4, 3, 2, 4, 2, 3, 1]$.

	1	2	3	4	5	6	
1	$\mathbf{A}_1$ 4×3 $M = 0$ $p = \oslash$	$\mathbf{A}_1 \cdot \mathbf{A}_2$ 4×2 $M = 24$ $p = 1$	$\mathbf{A}_1 \cdots \mathbf{A}_3$ 4×4 $M = 56$ $p = 2$	$\mathbf{A}_1 \cdots \mathbf{A}_4$ 4×2 $M = 52$ $p = 1$	$\mathbf{A}_1 \cdots \mathbf{A}_5$ 4×3 $M = 76$ $p = 2, 4$	$\mathbf{A}_1 \cdots \mathbf{A}_6$ $\times$ $M =$ $p =$	show final answer here $M = 40$ $p = 1$
2		$\mathbf{A}_2$ 3×2 $M = 0$ $p = \oslash$	$\mathbf{A}_2 \cdot \mathbf{A}_3$ 3×4 $M = 24$ $p = 2$	$\mathbf{A}_2 \cdots \mathbf{A}_4$ 3×2 $M = 28$ $p = 2$	$\mathbf{A}_2 \cdots \mathbf{A}_5$ 3×3 $M = 46$ $p = 2, 4$	$\mathbf{A}_2 \cdots \mathbf{A}_6$ 3×1 $M = 28$ $p = 2$	for option $k = 1$ $(\mathbf{A}_1) \cdot (\mathbf{A}_2 \cdots \mathbf{A}_6)$ $0 + 28 + 4 \cdot 3 \ \cdot 1 = 40$
3			$\mathbf{A}_3$ 2×4 $M = 0$ $p = \oslash$	$\mathbf{A}_3 \cdot \mathbf{A}_4$ 2×2 $M = 16$ $p = 3$	$\mathbf{A}_3 \cdots \mathbf{A}_5$ 2×3 $M = 28$ $p = 4$	$\mathbf{A}_3 \cdots \mathbf{A}_6$ 2×1 $M = 22$ $p = 3$	for option $k = 2$ $(\mathbf{A}_1 \cdot \mathbf{A}_2) \cdot (\mathbf{A}_3 \cdots \mathbf{A}_6)$ $24 + 22 + 4 \cdot 2 \ \cdot 1 = 54$
4				$\mathbf{A}_4$ 4×2 $M = 0$ $p = \oslash$	$\mathbf{A}_4 \cdot \mathbf{A}_5$ 4×3 $M = 24$ $p = 4$	$\mathbf{A}_4 \cdots \mathbf{A}_6$ 4×1 $M = 14$ $p = 4$	for option $k = 3$ $(\mathbf{A}_1 \cdots \mathbf{A}_3) \cdot (\mathbf{A}_4 \cdots \mathbf{A}_6)$ $56 + 14 + 4 \cdot 4 \ \cdot 1 = 86$
5					$\mathbf{A}_5$ 2×3 $M = 0$ $p = \oslash$	$\mathbf{A}_5 \cdot \mathbf{A}_6$ 2×1 $M = 6$ $p = 5$	for option $k = 4$ $(\mathbf{A}_1 \cdots \mathbf{A}_4) \cdot (\mathbf{A}_5 \cdot \mathbf{A}_6)$ $52 + 6 + 4 \cdot 2 \ \cdot 1 = 66$
6						$\mathbf{A}_6$ 3×1 $M = 0$ $p = \oslash$	for option $k = 5$ $(\mathbf{A}_1 \cdots \mathbf{A}_5) \cdot (\mathbf{A}_6)$ $76 + 0 + 4 \cdot 3 \ \cdot 1 = 88$

The fully parenthesized expression can be developed is $\mathbf{A}_1 \ (\mathbf{A}_2 \ (\mathbf{A}_3 \ (\mathbf{A}_4 \ (\mathbf{A}_5 \ \mathbf{A}_6))))$

3.2 Example 2

Consider another example, where the table below is a partially completed computation table for optimized Matrix Chain computation, for the dimensions $[d_0, d_1, \cdots d_6, d_7] = [2, 3, 3, 5, 5, 3, 4, 2]$.

	1	2	3	4	5	6	7	
1	$\mathbf{A}_1$ 2×3 $M = 0$ $p = \emptyset$	$\mathbf{A}_1 \cdot \mathbf{A}_2$ 2×3 $M = 18$ $p = 1$	$\mathbf{A}_1 \cdots \mathbf{A}_3$ 2×5 $M = 48$ $p = 2$	$\mathbf{A}_1 \cdots \mathbf{A}_4$ 2×5 $M = 98$ $p = 3$	$\mathbf{A}_1 \cdots \mathbf{A}_5$ 2×3 $M = 128$ $p = 4$	$\mathbf{A}_1 \cdots \mathbf{A}_6$ 2×4 $M = 152$ $p = 5$	$\mathbf{A}_1 \cdots \mathbf{A}_7$ 2×2 $M =$ $p =$	Final Answer $M =$ $p =$
2		$\mathbf{A}_2$ 3×3 $M = 0$ $p = \emptyset$	$\mathbf{A}_2 \cdot \mathbf{A}_3$ 3×5 $M = 45$ $p = 2$	$\mathbf{A}_2 \cdots \mathbf{A}_4$ 3×5 $M = 120$ $p = 2$	$\mathbf{A}_2 \cdots \mathbf{A}_5$ 3×3 $M = 147$ $p = 2$	$\mathbf{A}_2 \cdots \mathbf{A}_6$ 3×4 $M = 183$ $p = 5$	$\mathbf{A}_2 \cdots \mathbf{A}_7$ 3×2 $M = 152$ $p = 2$	for option $k = 1$ $(\mathbf{A}_1) \cdot (\mathbf{A}_2 \cdots \mathbf{A}_7)$ $0 + 152 + 2 \cdot 3 \cdot 2 = 164$
3			$\mathbf{A}_3$ 3×5 $M = 0$ $p = \emptyset$	$\mathbf{A}_3 \cdot \mathbf{A}_4$ 3×5 $M = 75$ $p = 3$	$\mathbf{A}_3 \cdots \mathbf{A}_5$ 3×3 $M = 120$ $p = 3$	$\mathbf{A}_3 \cdots \mathbf{A}_6$ 3×4 $M = 156$ $p = 5$	$\mathbf{A}_3 \cdots \mathbf{A}_7$ 3×2 $M = 134$ $p = 3$	for option $k = 2$ $(\mathbf{A}_1 \cdot \mathbf{A}_2) \cdot (\mathbf{A}_3 \cdots \mathbf{A}_7)$ $18 + 134 + 2 \cdot 3 \cdot 2 = 164$
4				$\mathbf{A}_4$ 5×5 $M = 0$ $p = \emptyset$	$\mathbf{A}_4 \cdot \mathbf{A}_5$ 5×3 $M = 75$ $p = 4$	$\mathbf{A}_4 \cdots \mathbf{A}_6$ 5×4 $M = 135$ $p = 5$	$\mathbf{A}_4 \cdots \mathbf{A}_7$ 5×2 $M = 104$ $p = 4$	for option $k = 3$ $(\mathbf{A}_1 \cdots \mathbf{A}_3) \cdot (\mathbf{A}_4 \cdots \mathbf{A}_7)$ $48 + 104 + 2 \cdot 5 \cdot 2 = 172$
5					$\mathbf{A}_5$ 5×3 $M = 0$ $p = \emptyset$	$\mathbf{A}_5 \cdot \mathbf{A}_6$ 5×4 $M = 60$ $p = 5$	$\mathbf{A}_5 \cdots \mathbf{A}_7$ 5×2 $M = 54$ $p = 5$	for option $k = 4$ $(\mathbf{A}_1 \cdots \mathbf{A}_4) \cdot (\mathbf{A}_5 \cdots \mathbf{A}_7)$ $98 + 54 + 2 \cdot 5 \cdot 2 = 172$
6						$\mathbf{A}_6$ 3×4 $M = 0$ $p = \emptyset$	$\mathbf{A}_6 \cdots \mathbf{A}_7$ 3×2 $M = 24$ $p = 6$	for option $k = 5$ $(\mathbf{A}_1 \cdots \mathbf{A}_5) \cdot (\mathbf{A}_6 \cdot \mathbf{A}_7)$ $128 + 24 + 2 \cdot 3 \cdot 2 = 164$
7							$\mathbf{A}_7$ 4×2 $M = 0$ $p = \emptyset$	for option $k = 6$ $(\mathbf{A}_1 \cdots \mathbf{A}_6) \cdot (\mathbf{A}_7)$ $152 + 0 + 2 \cdot 4 \cdot 2 = 168$

So the the minimal cost is $M[1, 7] = 164$ and there are three options to have such minimal cost, for $p = 1$ or $p = 2$ or $p = 5$. For each option, the parenthesized subexpression is:

$$\begin{aligned}
p = 1: &\quad \mathbf{A}_1 \odot (\mathbf{A}_2 \cdots \mathbf{A}_7) = \quad \mathbf{A}_1 \cdot (\mathbf{A}_2 \cdot (\mathbf{A}_3 \cdot (\mathbf{A}_4 \cdot (\mathbf{A}_5 \cdot (\mathbf{A}_6 \cdot \mathbf{A}_7)))))\\
p = 2: &\quad (\mathbf{A}_1 \cdot \mathbf{A}_2) \odot (\mathbf{A}_3 \cdots \mathbf{A}_7) = (\mathbf{A}_1 \cdot \mathbf{A}_2) \cdot (\mathbf{A}_3 \cdot (\mathbf{A}_4 \cdot (\mathbf{A}_5 \cdot (\mathbf{A}_6 \cdot \mathbf{A}_7))))\\
p = 5: &\quad (\mathbf{A}_1 \cdots \mathbf{A}_5) \odot (\mathbf{A}_6 \cdot \mathbf{A}_7) = (((((\mathbf{A}_1 \cdot \mathbf{A}_2)) \cdot \mathbf{A}_3) \cdot \mathbf{A}_4) \cdot \mathbf{A}_5) \odot (\mathbf{A}_6 \cdot \mathbf{A}_7)
\end{aligned}$$

3.3 Algorithms and Time Complexity

When writing an algorithm for this (which is left as an exercise), you just need to recognize that when $T[i,\ j]$ is evaluated, all table entries with a smaller value of both i and j are needed. For a bottom up approach, and referring to the square table as presented here, you could calculate start with the diagonal, first super diagonal, second super diagonal, and so forth.

The Key-and-Basic operation is the comparison between the various options when determining the value for $M[i, j]$. So we will count the number of "options" that are evaluated. We study them for each super-diagonal. Let $\ell = j - i$ indicate which super diagonal is studied.

$\ell = 0$ These are the n elements on the main diagonal, and each of it's M values is 0. No key-and-basic operation is involved.

$\ell = 1$ There are the $n - 1$ elements on the first super-diagonal. Each M value is determined by a single option: product of d's. One single option for each of the $n - 1$ elements: The cost for this diagonal is $(n - 1) \times 1$.

$\ell = 2$ There are the $n - 2$ elements on the second super-diagonal. Each M value is determined by selecting from two options. The cost for this diagonal is $(n - 2) \times 2$.

$\ell = 3$ There are the $n - 3$ elements on the third super-diagonal. Each M value is determined by selecting from three options. The cost for this diagonal is $(n - 3) \times 3$.

ℓ There are the $n - \ell$ elements on the ℓ^{st} super-diagonal. Each M value is determined by selecting from $n - \ell$ options. The cost for this diagonal is $(n - \ell) \times \ell$.

$\ell = n - 1$ There is 1 element on this last super-diagonal, (the $[1, n]^{\text{th}}$ element), and it's M value is determined by selecting from $n - 1$ options. The cost for this element is $(1) \times (n - 1)$.

The total cost in time to determine the best execution tree is thus:

$$T_{\text{MC}}(n) = \sum_{\ell=1}^{n-1} \{(n - \ell) \times \ell\} = \tfrac{1}{6}n^3 - \tfrac{1}{6}n \sim \tfrac{1}{6}n^3 \tag{9}$$

The total cost in space is $S_{\text{MC}}(n) = \frac{1}{2}n^2$

Although the cost to find the resulting Matrix Chain is unique, the actual execution tree might not be: Change the p-information to a list to track multiple possibilities, and change it to a priority-list for generating second best options.

Correctness Use a proof by counter example.

3.4 Suggested Exercises

1. For the example above, with $n = 6$ matrices and dimensions $[d_0, d_1, \cdots d_6] = [4, 3, 2, 4, 2, 3, 1]$,

   1. Draw the execution tree for $\mathbf{A}_1 \cdots \mathbf{A}_6$
   2. The table indicates that the expression tree for $\mathbf{A}_2 \cdots \mathbf{A}_5$ is not unique. Draw all execution trees with minimal costs, and validate the answers.
   3. Change the dimension of $\mathbf{A}_6$ from $3\times$ 1 to $3\times$ 3. Find again $M[1, 6]$, draw the new optimal execution tree(s) and find the fully parenthesized expression.

2. For the example above, with $n = 7$ matrices and dimensions $[d_0, d_1, \cdots d_6, d_7] = [2, 3, 3, 5, 5, 3, 4, 2]$,

   1. Draw the execution tree for $\mathbf{A}_1 \cdots \mathbf{A}_7$
   2. The shared dimension of matrices $\mathbf{A}_4$ and $\mathbf{A}_5$ is $d_4 = 5$. How robust are the execution trees to changes in d_4? Change d_4 to $6, 7, 8, \ldots$ and to $4, 3, 2, 1$ until the structure of the execution tree changes (The value of $M[1, 7]$ changes of course; you are asked to change d_4 so that the p-values do not change).

3. Write an algorithm in pseudo code for the Matrix Chain, implement it, and find both the minimum number of scalar operations needed and present a minimal expression (with parentheses) for the 13 matrices with dimensions $[d_0, d_1, \cdots d_{13}] = [4, 3, 2, 4, 2, 3, 1, 2, 3, 3, 5, 5, 3, 4, 2]$. (Answer: 138).

4. (Graduate Students) There are several heuristic ideas to use a "greedy" approach to find an execution tree for a matrix chain $\mathbf{A}_1 \times \mathbf{A}_2 \times \ldots \mathbf{A}_n$ with dimensions $[d_0, d_1, \cdots d_{n-1}, d_n]$. For all cases, analyze the time complexity to deliver the blue print, *and* give a counter example with 2, 3 or 4 matrices to show that the heuristic does not always result in the optimal multiplication cost. Hint: assume specific values for "inner" dimensions, and derive inequalities which can be solved. Or: implement the algorithm and experiment with varying values.

4:a One heuristic, *remove largest shared dimension first* is to find the largest value of the"inner" dimensions, $\{d_1, \cdots d_{n-1}\}$, and multiply the two matrices with this shared dimension. Repeat this process, until done.

4:b (Graduate Students) Another heuristic, *do most expensive matrix multiplication first* is to find the largest value of the possible matrix multiplications, and multiply the two matrices with this largest computation. In other words, find i such that the product $d_{i-1} \cdot d_i \cdot d_{i+1}$ is largest from $i = 1 \ldots n - 1$, and multiply matrices $\mathbf{A}_i$ and $\mathbf{A}_{i+1}$. Repeat this process, until done.

4:c Another heuristic, *remove smallest shared dimension first* is to find the smallest value of the"inner" dimensions, $\{d_1, \cdots d_{n-1}\}$, and multiply the two matrices with this shared dimension. Repeat this process, until done.

4:d (Graduate Students) Another heuristic, *do least expensive matrix multiplication first* is to find the smallest value of the possible matrix multiplications, and multiply the two matrices with this smallest computation. In other words, find i such that the product $d_{i-1} \cdot d_i \cdot d_{i+1}$ is smallest from $i = 1 \ldots n-1$, and multiply matrices $\mathbf{A}_i$ and $\mathbf{A}_{i+1}$. Repeat this process, until done.

5. (Graduate Students) The above algorithm is based on finding the minimum of several options, like

$$M[i\ ,\ j] = \min_{k,\text{ with } i \leq k < j} \{\ M[i\ ,\ k] + M[k+1\ ,\ j]\ + d_{i-1}d_k d_j\} \quad \text{for} \quad \text{j=i+2} \ldots \text{n} \tag{10}$$

If you change the "min " to "max " in above equation algorithm, does the resulting number $M[1, n]$ represent the most costly way to multiply a matrix chain? Make an example with only 3 matrices, and try to find a counterexample.

6. (Graduate Students) Notice that in the examples, and for every value of i, the sequences $\langle M[i, i]\, M[i, i{+}1], M[i, i{+}2] \ldots M[i, n]\ \rangle$ and $\langle M[i, i]\, M[i-1, i], M[i-2, i] \ldots M[1, i]\ \rangle$ are strictly increasing. Is this always the case? Can you prove or disprove this claim.

4 DP: Optimal Binary Search Trees

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

4.1 OBST with full coverage

An **Optimal Binary Search Tree** (OBST) is a binary search tree that is constructed and designed such that the average time complexity of a FIND-operation is minimal (lowest) from among all binary search trees that can possibly be constructed for these elements. Assume that there are n elements $\langle E_1 \dots\ E_n\rangle$ and that it is known that a FIND is always for an element that is actually present in the search tree. (In the next section, we incorporate the unsuccessful FINDs as well). The objective is to minimize the average number of comparisons for a FIND, and this involves probabilities. For reasons that will become clear later (but also for reasons to stay compatible with established notation by other authors), let F_i be the frequency of element $\langle E_i\rangle$, then the probability that a FIND is for element $\langle E_i\rangle$ is

$$\Pr[\text{FIND}(E_i)] = \frac{F_i}{F_1 + F_2 \dots F_n}.$$

Suppose we have an OBST with the elements $\langle E_1 \dots\ E_{R-1}\rangle$ in the left BST, with element $\langle E_R\rangle$ as root, and elements $\langle E_{R+1} \dots\ E_n\rangle$ in the right BST. A FIND(x) operation compares x with the root element $\langle E_R\rangle$, always, and subsequently stops (if the search is for $\langle E_R\rangle$, or if the element is not found at the root, then the FIND(x) continuous in either the left- or the right-subtree. The probabilities of these is given by the combined probabilities in the subtrees. Consider now the expected number of comparisons. Let $\mathsf{E}[1,\ n]$ be the expected number of comparison for a FIND(x) operation on the BST with elements $\langle E_1 \dots\ E_n\rangle$, then[1] (Note that this is the same was $T^{\text{Average}}[1\ ,\ n]$)

$$\mathsf{E}[1,\ n] = 1 + \mathsf{E}[1,\ R-1]\cdot\frac{F_1+\cdots+F_{R-1}}{F_1+\dots\quad\cdots+F_n} + 0\cdot\frac{F_R}{F_1+\dots\quad\cdots+F_n} + \mathsf{E}[R+1,\ n]\cdot\frac{F_{R+1}+\cdots+F_n}{F_1+\dots\quad\cdots+F_n} \tag{11}$$

Once you know that the element is in the particular subtree, the probability for a FIND(x) becomes a conditional probabilities (since you have narrowed down the search), and needs to be re-normalized. To avoid these extra complications, we will work with the frequencies F_i, instead of working with probabilities. This spares us from re-normalizing at each level in the tree, which will need to occur only once when completed. So now define

$$W[i\ ,\ j] = \begin{cases} F_i + \dots + F_j & i \le j \text{ and} \\ 0 & j < i \text{ (for convenience)} \end{cases} \tag{12}$$

and simplify the above equation to

$$\mathsf{E}[1,\ n]W[1,\ n] = W[1,\ n] + \mathsf{E}[1,\ R-1]\cdot W[1,\ R-1] + \mathsf{E}[R+1,\ n]\cdot W[R-1,\ n] \tag{13}$$

Finally, define

$$T[1\ ,\ n] = \mathsf{E}[1,\ n]W[1,\ n] \tag{14}$$

With these definitions, the average time for a FIND(x) - operation can be recovered as soon as $T[1,n]$ has been determined. If you know in advance that the BST with element $\langle E_R\rangle$ at the root is indeed optimal, then

$$T[1,\ n] = W[1,\ n] + \{\ T[1,\ R-1] + T[R+1,\ n]\ \}$$

So every other BST would have a higher cost. This means that if you do *not* know in advance that the element $\langle E_R\rangle$ is at the root, then R is found as the optimal value for an open parameter (let us use k). Thus every element $\langle E_k\rangle$ should be considered as a potential root, splitting the elements into three: the elements $\langle E_1 \dots\ E_{k-1}\rangle$ in the left BST, with the element $\langle E_k\rangle$ as root, and elements $\langle E_{k+1} \dots\ E_n\rangle$ in the right BST. The optimal value for $T[1,\ n]$ is then found by varying the root-element and finding the minimal value. If $\langle E_1\rangle$ is the root, then the left subtree is empty and we use $T[1,0]=0$ to indicate the cost for this empty subtree. In fact, we use $T[i,i-1]=0$ to indicate the zero cost for all leaves. With this, we have

$$T[1,\ n] \quad = \quad W[1,\ n] + \min_{k,\text{ with } 1\le k\le n} \{\ T[1,\ k-1] + T[k+1,\ n]\ \}$$

Similarly, the root element $\langle E_k\rangle$ is then also determined by tracking the actual value of the index k that minimized the sum in the expression above, and thus track R as the value of k that minimizes the expression.

The expression relies on the values $T[1,\ k-1]$ and $T[k+1,\ n]$ being available for all k, so we now present the formulation for any set of consecutive elements $\langle E_i \dots E_j\rangle$. First generalize $W[1,\ n]$, $\mathsf{E}[1,\ n]$ and $T[1\ ,\ n]$ to general indices: $W[i,\ j]$, $\mathsf{E}[i,\ j]$ and $T[i,\ j]$ which represent the relevant quantities for only the elements $\langle E_i \dots\ E_j\rangle$. The driving equation for the minimal cost is then

[1] Using both the linearity of expectation as well as the law of total expectation, see your probability text book for details.

similarly,

$$\boxed{T[i,\ j] = W[i,\ j] + \min_{k,\ \text{with}\ i \le k \le j} \{\ T[i,\ k-1] + T[k+1,\ j]\ \}} \tag{15}$$

Also, let us track the information gained from finding the minimal value and keep the value for k that indeed minimizes the expression: let this be $R[i,\ j]$. It is the root of the sought after BST for the consecutive elements $\langle E_i \dots\ E_j \rangle$. It splits the elements in a left- subtree, the root $E_{R[i,\ j]}$, and the right-subtree. Having all R's available allows the construction of the actual binary search tree, and thus this algorithm does not only find the optimum (i.e. minimum) value for average cost that is possible for a FIND on any BST that can be constructed from these elements, it also provides a blueprint for the actual construction of an OBST in a second phase.

The frequencies need to be summed several times, and in order to avoid the re-computation of these sums, please note that these can be easily generated form earlier values:

$$W[i\ ,\ j] = \begin{cases} 0 & j = i-1 \text{ (for convenience)} \\ W[i\ ,\ j-1] + F_j & i \le j \end{cases} \tag{16}$$

Note that $W[i\ ,\ i] = F_i$ (these are the single elements), and that $W[i\ ,\ i-1] = 0$ (representing empty sub-trees, or gaps.) Finally, the average time complexity of a FIND-operation on such an OBST for the elements $\langle E_1 \dots\ E_n \rangle$ is

$$\boxed{\mathsf{E}[1,\ n] = T^{\text{Average}}[1\ ,\ n] = \frac{T[1\ ,\ n]}{W[1\ ,\ n]}} \tag{17}$$

4.2 Algorithms and Time Complexity

The OBST and the Matrix Chain algorithms are very similar. The algorithm for the OBST (which is left as an exercise), you again need to recognize that when $T[i,\ j]$ is evaluated, all table entries with a smaller value of both i and j are needed. For a bottom up approach, and referring to the square table as presented here, you could start with the diagonal, then the first super diagonal, second super diagonal, and so forth.

The total cost in time to determine the OBST is also:

$$T_{\text{OBST}}(n) \sim \tfrac{1}{6} n^3 \tag{18}$$

The total cost in space is $S_{\text{OBST}}(n) = \frac{1}{2} n^2$

Although the average cost for a find on a OBST is unique, the actual OBST might not be: Change the root-information to a list of roots to track multiple possibilities, and change it to a priority-list for generating second best options.

4.2.1 Example w/o gaps

Consider an example with seven (7) elements

$$\begin{array}{c|ccccccc|}
index & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
Element & E_1 & E_2 & E_3 & E_4 & E_5 & E_6 & E_7 \\
Frequency & 6 & 4 & 2 & 6 & 8 & 3 & 3
\end{array}$$

	0	1	2	3	4	5	6	7	
1	$1\dots0$ $W=0$ $T=0$ gap	$1\dots1$ $W=6$ $T=6$ $F_1=6$	$1\dots2$ $W=10$ $T=14$ $R=1$	$1\dots3$ $W=12$ $T=20$ $R=1,2$	$1\dots4$ $W=18$ $T=34$ $R=2$	$1\dots5$ $W=26$ $T=54$ $R=4$	$1\dots6$ $W=29$ $T=63$ $R=4$	$1\dots7$ $W=$ $T=$ $R=$	final answer $T=$ $R=$
2		$2\dots1$ $W=0$ $T=0$ gap	$2\dots2$ $W=4$ $T=4$ $F_2=4$	$2\dots3$ $W=6$ $T=8$ $R=2$	$2\dots4$ $W=12$ $T=20$ $R=4$	$2\dots5$ $W=20$ $T=36$ $R=4$	$2\dots6$ $W=23$ $T=45$ $R=4$	$2\dots7$ $W=26$ $T=55$ $R=5$	For option $k=1$ $(\emptyset\ )\ E_1\ (E_2\dots\ E_7)$ $0+55+32=87$
3			$3\dots2$ $W=0$ $T=0$ gap	$3\dots3$ $W=2$ $T=2$ $F_3=2$	$3\dots4$ $W=8$ $T=10$ $R=4$	$3\dots5$ $W=16$ $T=26$ $R=4,5$	$3\dots6$ $W=19$ $T=32$ $R=5$	$3\dots7$ $W=22$ $T=41$ $R=5$	For option $k=2$ $(E_1\ )\ E_2\ (E_3\dots\ E_7)$ $6+41+32=79$
4				$4\dots3$ $W=0$ $T=0$ gap	$4\dots4$ $W=6$ $T=6$ $F_4=6$	$4\dots5$ $W=14$ $T=20$ $R=5$	$4\dots6$ $W=17$ $T=26$ $R=5$	$4\dots7$ $W=20$ $T=35$ $R=5$	For option $k=3$ $(E_1E_2)\ E_3\ (E_4\dots\ E_7)$ $14+35+32=81$
5					$5\dots4$ $W=0$ $T=0$ gap	$5\dots5$ $W=8$ $T=8$ $F_5=8$	$5\dots6$ $W=11$ $T=14$ $R=5$	$5\dots7$ $W=14$ $T=23$ $R=5$	For option $k=4$ $(E_1\dots\ E_3)\ E_4\ (E_5\dots E_7)$ $20+23+32=75$
6						$6\dots5$ $W=0$ $T=0$ gap	$6\dots6$ $W=3$ $T=3$ $F_6=3$	$6\dots7$ $W=6$ $T=9$ $R=6,7$	For option $k=5$ $(E_1\dots E_4)\ E_5\ (E_6E_7)$ $34+9+32=75$
7							$7\dots6$ $W=0$ $T=0$ gap	$7\dots7$ $W=3$ $T=3$ $F_7=3$	For option $k=6$ $(E_1\dots E_5)\ E_6\ (\ E_7)$ $54+3+32=89$
8								$8\dots7$ $W=0$ $T=0$ gap	For option $k=7$ $(E_1\dots E_6)\ E_7\ (\emptyset\ )$ $63+0+32=95$

4.3 OBST with partial coverage: GAPS

An **Optimal Binary Search Tree WITH gaps** (OBST) is a binary tree that is constructed and designed such that the average time complexity of a FIND-operation is minimal (lowest) from among all binary search trees that can possibly be constructed for these elements. Assume now that in addition to doing a FIND for elements that are present in the tree, there could also be FIND operations for an element that is not present in the search tree. That is the search ends up in a GAP. We number the Gaps from 0 t o n, so that the third element (say) is bordered by the second and third gap. Let the frequencies for the gaps be $G_0, G_1, \ldots G_n$.

Fortunately, there is only little that needs to be done to the algorithm to accommodate gaps. Let again $T[i, j]$ represent the minimal cost that can be attained. The driving equation for the minimal cost is essentially the same, except that the frequencies for gaps are now incorporated. Note, that there is always one more gaps than there are elements.

$$T[i\ ,\ j] \quad = \min_{k,\text{ with } i \leq k \leq j} \{\, T[i\ ,\ k-1] + T[k+1\ ,\ j] \,\} + \text{the sum of all frequencies within this search tree.}$$

where the “the sum of all frequencies within this search tree” now includes the gap frequencies. Adjust the definition of $W[\,..\,]$ to include gaps: ($W[i\ ,\ j] = 0$ for $j < i - 1$, whereas

$$W[i\ ,\ j] = \begin{cases} G_{i-1} & j = i-1 \\ W[i\ ,\ j-1] + F_j + G_j & j \geq i \end{cases} \tag{19}$$

Note, that $W[i\ ,\ i] = G_{i-1} + F_i + G_i$, since these are the single elements, but now have two neighboring gaps, and that $W[i\ ,\ i-1] = G_{i-1}$, since these represent gaps. With this adjusted definition, the driving equation for T remains

$$\boxed{T[i\ ,\ j] = \min_{k,\text{ with } i \leq k \leq j} \{\, T[i\ ,\ k-1] + T[k+1\ ,\ j] \,\} + W[i\ ,\ j]} \tag{20}$$

Similarly, the average time complexity of a FIND-operation on such an OBST for the elements $\langle E_1 \ldots\ E_n \rangle$ and the gaps $\langle G_0, G_1 \ldots\ G_n \rangle$ remains

$$\boxed{\mathsf{E}[1,\ n] = T^{\text{Average}}[1\ ,\ n] = \frac{T[1\ ,\ n]}{W[1\ ,\ n]}} \tag{21}$$

4.3.1 Example with gaps

For example, take the same seven elements and their frequencies as in the previous example, (that is $F_1 = 6$, $F_2 = 4$, $F_3 = 2$, $F_4 = 6$, $F_5 = 8$, $F_6 = 3$ and $F_7 = 3$). Now add gap- frequencies: $G_0 = 8$, $G_1 = 3$, $G_2 = 3$, $G_3 = 5$, $G_4 = 3$, $G_5 = 4$, $G_6 = 8$ and $G_7 = 2$.

$$\begin{array}{c} Element \\ Frequency \\ \\ Gap \\ Frequency \end{array} \left[\begin{array}{ccccccccccccccc} & E_1 & & E_2 & & E_3 & & E_4 & & E_5 & & E_6 & & E_7 & \\ & 6 & & 4 & & 2 & & 6 & & 8 & & 3 & & 3 & \\ \\ G_0 & & G_1 & & G_2 & & G_3 & & G_4 & & G_5 & & G_6 & & G_7 \\ 8 & & 3 & & 3 & & 5 & & 3 & & 4 & & 8 & & 2 \end{array} \right]$$

	0	1	2	3	4	5	6	7	
1	$1 \ldots 0$ $W = 8$ $T = 0$ $G_0 = 8$	$1 \ldots 1$ $W = 17$ $T = 17$ $F_1 = 6$	$1 \ldots 2$ $W = 24$ $T = 34$ $R = 1$	$1 \ldots 3$ $W = 31$ $T = 58$ $R = 1, 2$	$1 \ldots 4$ $W = 40$ $T = 86$ $R = 2$	$1 \ldots 5$ $W = 52$ $T = 125$ $R = 2, 4$	$1 \ldots 6$ $W = 63$ $T = 162$ $R = 4$	$1 \ldots 7$ $W =$ $T =$ $R =$	final answer $T =$ $R =$
2		$2 \ldots 1$ $W = 3$ $T = 0$ $G_1 = 3$	$2 \ldots 2$ $W = 10$ $T = 10$ $F_2 = 4$	$2 \ldots 3$ $W = 17$ $T = 27$ $R = 2, 3$	$2 \ldots 4$ $W = 26$ $T = 50$ $R = 3$	$2 \ldots 5$ $W = 38$ $T = 80$ $R = 4$	$2 \ldots 6$ $W = 49$ $T = 114$ $R = 5$	$2 \ldots 7$ $W = 54$ $T = 137$ $R = 5$	For option $k = 1$ $(\emptyset)\ E_1\ (E_2 \ldots\ E_7)$ $0 + 137 + 68 = 205$
3			$3 \ldots 2$ $W = 3$ $T = \emptyset$ $G_2 = 3$	$3 \ldots 3$ $W = 10$ $T = 10$ $F_3 = 2$	$3 \ldots 4$ $W = 19$ $T = 29$ $R = 4$	$3 \ldots 5$ $W = 31$ $T = 56$ $R = 4$	$3 \ldots 6$ $W = 42$ $T = 86$ $R = 5$	$3 \ldots 7$ $W = 47$ $T = 109$ $R = 5$	For option $k = 2$ $(E_1)\ E_2\ (E_3 \ldots\ E_7)$ $17 + 109 + 68 = 194$
4				$4 \ldots 3$ $W = 5$ $T = 0$ $G_3 = 5$	$4 \ldots 4$ $W = 14$ $T = 14$ $F_4 = 6$	$4 \ldots 5$ $W = 26$ $T = 40$ $R = 5$	$4 \ldots 6$ $W = 37$ $T = 66$ $R = 5$	$4 \ldots 7$ $W = 42$ $T = 89$ $R = 5$	For option $k = 3$ $(E_1 E_2)\ E_3\ (E_4 \ldots\ E_7)$ $34 + 89 + 68 = 191$
5					$5 \ldots 4$ $W = 3$ $T = 0$ $G_4 = 3$	$5 \ldots 5$ $W = 15$ $T = 15$ $F_5 = 8$	$5 \ldots 6$ $W = 26$ $T = 41$ $R = 5, 6$	$5 \ldots 7$ $W = 31$ $T = 59$ $R = 5$	For option $k = 4$ $(E_1 \ldots\ E_3)\ E_4\ (E_5 \ldots E_7)$ $58 + 59 + 68 = 185$
6						$6 \ldots 5$ $W = 4$ $T = 0$ $G_5 = 4$	$6 \ldots 6$ $W = 15$ $T = 15$ $F_6 = 3$	$6 \ldots 7$ $W = 20$ $T = 33$ $R = 6$	For option $k = 5$ $(E_1 \ldots E_4)\ E_5\ (E_6 E_7)$ $86 + 33 + 68 = 187$
7							$7 \ldots 6$ $W = 8$ $T = 0$ $G_6 = 8$	$7 \ldots 7$ $W = 13$ $T = 13$ $F_7 = 3$	For option $k = 6$ $(E_1 \ldots E_5)\ E_6\ (E_4 \ldots\ E_7)$ $125 + 13 + 68 = 206$
8								$8 \ldots 7$ $W = 2$ $T = 0$ $G_7 = 2$	For option $k = 7$ $(E_1 \ldots E_6)\ E_7\ (\emptyset)$ $162 + 0 + 68 = 230$

4.3.2 OBST: Suggested Exercises

1. (Graduate Students) For the example above, with $n = 7$ elements, no gaps, and frequencies $[F_1, \cdots F_7] = [6, 4, 2, 6, 8, 3, 3]$,

1. Draw the OBST.
2. Calculate the actual average number of comparisons, $T^A(n)$ and validate your answer using the formula

$$\sum_i (\text{depth of node } i) \times \frac{F_i}{\sum_k F_k}$$

3. The table indicates that the expression tree for $E_1\ E_2\ E_3$ is not unique. Draw all OBST's with minimal average costs, and validate the answers.
4. Change the frequency E_7 from 3 to 1, 2, 4, 5, and so on, till the structure of the OBST changes.

---

2. (Graduate Students) For the example above, with $n = 7$ elements, frequencies $[F_1, \cdots F_7] = [6, 4, 2, 6, 8, 3, 3]$ and gap frequencies $[G_0, G_1 \cdots G_7] = [8, 3, 3, 5, 3, 4, 8, 2]$

1. Draw the OBST.
2. Calculate the actual average number of comparisons, and validate your answer using the formula

$$\sum_i (\text{depth of node } i) \times \frac{F_i}{\sum_k F_k + \sum_\ell G_\ell} + \sum_i (\text{depth the parent of gap}_i) \times \frac{G_i}{\sum_k F_k + \sum_\ell G_\ell}$$

3. The table indicates that the expression tree for $E_1\ E_2 E_3$ is not unique. Draw all OBST's with minimal average costs, and validate the answers.
4. Change the frequency G_7 from $G_7 = 2$ to $G_7 = 10$ and repeat the earlier parts.

---

3. Write an algorithm in pseudo code for the OBST, implement it, and find both the minimum value for the average of a find, and present the pre-order traversal of the actual OBST with the 14 elements: $6, 4, 2, 6, 8, 3, 7, 6, 4, 2, 6, 8, 3, 7$. (Answer: $218/72 = 3.02$. The OBST-tree is not unique, there are 3 solutions, so there are three pre-order traversals).

---

4. (Graduate Students) A **Treap** is a binary search tree (without gaps for now) where elements have frequencies F_i, and where the Frequencies of the nodes satisfy the Max-Heap property: The Frequency of a node is larger than the frequency of either child. Is the OBST from the first example a treap? (i.e. with $n = 7$ elements, no gaps, and frequencies $[F_1, \cdots F_7] = [6, 4, 2, 6, 8, 3, 3]$)

---

5. (Graduate Students) There are several heuristic ideas that use a "greedy" approach to find a BST whose average FIND complexity may be optimal. One such idea is to create a Treap as follows: Sort the elements by frequencies, from high to low, and insert them into a binary search tree in that order. Use this approach to construct a treap from $[F_1, \cdots F_7] = [6, 4, 2, 6, 8, 3, 3]$, and determine the average time complexity for a FIND. What is the time complexity to construct this treap?

---

6. (Graduate Students) There are several heuristic ideas that use a "greedy" approach to find a BST whose average FIND complexity may be optimal. One such idea is to create a Treap as follows: Sort the elements by frequencies, from low to high this time, and insert them into a binary search tree in that order. You can restore the max-heap property by using an AVL-type rotation. Use this approach to construct a treap from $[F_1, \cdots F_7] = [6, 4, 2, 6, 8, 3, 3]$, and determine the average time complexity for a FIND. What is the time complexity to construct this treap?

---

7. (Graduate Students) The OBST-algorithms are based on the equation

$$\boxed{T[i,\ j] = W[i,\ j] + \min_{k,\text{ with } i \le k \le j} \{\ T[i,\ k-1] + T[k+1,\ j]\ \}} \tag{22}$$

What happens if you change the "min " to "max "? Does the resulting number indeed represent the most costly average FIND complexity? Take an example with only 3 elements (with their frequencies), and try to find a counterexample.

5 DP: Weighted Interval Scheduling – Telescope

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

Suppose you are manager of a telescope observatory and in charge of scheduling who gets to use the facility. You are given a list of telescope observation requests, $R_1, R_2, \ldots R_n$, each specified by a triple $R_i = \langle s_i, f_i, b_i \rangle$, representing the start times, finish times, and benefit amounts for request i. The telescope problem (also know as the '*weighted scheduling problem*') is to find the maximum benefit that can be attained with a schedule of conflict-free requests (i.e. the intervals of accepted requests are completely disjoint; these intervals do not overlap.) Let us use B_n to indicate this *maximal benefit*, even though all requests are included in the consideration and $B[1 \ldots n]$ would perhaps be more appropriate. But since "1" is always included, we use a shorter symbol.
Furthermore, an *optimal schedule* is a particular listing of a subset of these requests, such that this subset can be scheduled such that there is no scheduling conflict, and such that the optimum value is reached. The general problem is thus: determine both the Maximal Benefits B_i for the requests (in a first pass), and determine an optimal schedule (in a second pass). The algorithm has several steps:

1. **Sort** the requests by their finishing times. The time complexity for this step is, in the worst case, $\Theta(n \lg n)$ with merge sort, or $\Theta(n^2)$ using quick-sort or bubble sort.
2. For ease of discussion, we assume that the requests have been (re-)numbered, so that now $f_1 \leq f_2 \leq \ldots \leq f_n$.
3. For each transaction request, determine the highest numbered earlier request that does not conflict with the current request. Thus for request i, we are looking for highest numbered request j such that
   (a) $j < i$
   (b) $f_j < s_i < f_{j+1}$

   For each such index i, such a highest conflict-free earlier request is uniquely defined if it exists and we call it $pred(i)$. If there is no earlier conflict free request for i, such is the case for $i = 1$ e.g., then we define $pred(i) = 0$. In algorithmic terms:
   for i **from** 1 **to** n **do** determine pred(i) **end**
   There are several methods that can be used to determine these predecessors and stay within the time bound of $O(n \lg n)$ (e.g. binary search for element i: $\lg 1 + \lg 2 + \lg 3 + \ldots + \lg n = n \ \lg n$).
4. Next, realize that B_n can be determined by asking a simple question: Do we íinclude request n, (and thus exclude requests $pred(n) + 1$ through $n - 1$ because they cannot be scheduled without conflicting with request n), ór do we exclude request n? This same relation governs the computation of the previous values as well, and the governing relation for all maximum benefits is

$$\boxed{B_i \leftarrow \max_{\text{2 options}} \{B_{i-1}, \quad b_i + B_{\text{pred}(i)} \} \quad \text{for} \quad i = 1, \ldots n} \tag{23}$$

   This iteration can get started by either using using a **for-loop** that runs **from** $i = 1$ **to** n **do_end** and which uses $B_0 = 0$ as a sentinel value, or by considering $i = 1$ as a special boundary case, and start the **for-loop** with one index higher: **from** $i = 2$ **to** n **do_end**.
   The time complexity for this step is constant for each value of i, for a total of $\Theta(n)$.
5. If only the optimal benefit is desired, then there is nothing else to so. If however you also need to deliver an optimal schedule, then you need to track which way the maximal value is reached. This information can then be used in the second pass to construct an optimal schedule.
   If $B_{i-1} < b_i + B_{\text{pred(i)}}$, then the maximum is reached by including request i in the schedule: $Track(i) \leftarrow Yes$
   If $B_{i-1} > b_i + B_{\text{pred(i)}}$, then the maximum is reached by not including request i in the schedule: $Track(i) \leftarrow No$
   If $B_{i-1} = b_i + B_{\text{pred(i)}}$, then the maximum is reached either way: $Track(i) \leftarrow Tie$
6. An optimal schedule can then be produced by starting from request R_n and asking whether or not it should be included. The following idea could be used.

```
algorithm PrintSchedule ($MyTrack$, $n$)
    if $n > 0$ and $MyTrack[n] = “No''$ then PrintSchedule($MyTrack$, $n-1$) end-if
    if $n > 0$ and $MyTrack[n] \neq “No''$ then Concatenate(PrintSchedule($MyTrack$, pred($n$)) || “$n$”) end-if
end-alg PrintSchedule
```

7. The total time complexity for all steps is $\boxed{T^W(n) = \Theta(n \ \lg n \ )}$

5.1 Example

This is demonstrated by an example. In the first pass, the forward pass, the values for B_i and $Track_i$ $(Include_i)$ are determined. We also include a request that is called the zero request, and is used in the boundary equation.

i	s_i	f_i	b_i	pred(i)	$b_i + B_{\text{pred(i)}}$	$0 + B_{i-1}$	B_i	Track include? (Y / N / T)
0							0	
1	2	3	9	0	$9 + 0$	$0 + 0$	9	Yes
2	4	5	8	1	$8 + 9$	$0 + 9$	17	Yes
3	2	7	3	0	$3 + 0$	$0 + 17$	17	No
4	6	9	3	2	$3 + 17$	$0 + 17$	20	Yes
5	10	11	5	4	$5 + 20$	$0 + 20$	25	Yes
6	10	13	2	4	$2 + 20$	$0 + 25$	25	No
7	8	15	2	3	$2 + 17$	$0 + 25$	25	No
8	14	19	10	6	$10 + 25$	$0 + 25$	35	Yes
9	16	23	10	7	$10 + 25$	$0 + 35$	35	Tie
10	22	25	10	8	$10 + 35$	$0 + 35$	45	Yes
11	14	29	8	6	$8 + 25$	$0 + 45$	45	No

After completing the table, which specific requests are included to attain the value B_{11}? (A "Y" means it is included, a "N" indicates not included, but also no conflict with a higher number request that was included, and a "S" indicates that it was skipped over, because of a time conflict with a request that was included.

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$	$i = 8$	$i = 9$	$i = 10$	$i = 11$	Check
Included?	Y	Y	S	Y	Y	N	S	Y	S	Y	N	
	9	8		3	5			10		10		$B_{11} = 45$

5.2 Suggested Exercises

1. Repeat computation for the benefits $b_1 \dots b_{11}$ as $3, 3, 5, 4, 3, 3, 5, 4, 1, 1, 2$
2. Is it possible to extend the request list as shown in the example above by a 12th request, such that $B_{12} > B_{11}$, and there is exactly one request in the optimal schedule?
3. Is it possible to extend the request list as shown in the example above by a 12th request, such that $B_{12} > B_{11}$, and there are exactly two requests in the optimal schedule?
4. Is it possible to extend the request list as shown in the example above by a 12th request, such that $B_{12} > B_{11}$, and there are exactly three requests in the optimal schedule?
5. Repeat computation but now only for 5 requests with benefits $b_1 \dots b_5$ as $3, 3, 15, 9, 3$. Do you notice anything special? How many schedules are possible?
6. Is it possible to change the values of b_{10} and b_{11} so that request 9 will be part of the optimal schedule?
7. Is it possible to change the values of b_9 so that request 9 will be part of the optimal schedule?
8. Generate the optimal schedule for the first $n = 10$ requests only.
9. Implement the algorithm and test it on several situations.
10. **(Graduate students)** After you have posted the schedule, one of the requesters who has not been included wants to increase the benefit to be paid, but is not flexible with the start and finishing times. How would you find the minimal amount needed for a request that has not been included, such that the request now will be included in an updated schedule?
11. **(Graduate students)** After you have posted the schedule, one of the requesters who has been included wants to decrease the benefit to be paid, but is not flexible with the start and finishing times. How would you find the maximal amount of savings for a request that has been included, such that the request remains to be included in an updated schedule?
12. **(Graduate students)** After you have posted the schedule, one of the requesters is flexible with start and finishing times, and is willing to shift either up or down by no more than two days. The benefit amount cannot be changed. You think it may be possible to enter several dummy requests for each of the newly proposed start/finish times. If any of these dummies makes it in the updated schedule, then you say Yes. Please comment on this idea.

6 DP: Optimal Rod-cutting

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

These notes are very limited indeed. Please study from the book or find material on the internet. The setting: There is a rod of size n available for selling. You can either sell it in it's entirety, or cut it into pieces and sell these individual pieces. A piece of size ℓ sells for $p[\ell]$. So the object is to find the optimal partition of the rod that maximizes the total revenue, $R[n]$. So the question is: Do you make a cut, and if so, where? If you do not make a cut, then you sell the entire rod and get a profit of $p[n]$. If you do make a cut, then you separate a piece of rod of size ℓ from the "main rod" where size $n - \ell$ remains. So if you make this cut, then the profit could be $p[\ell] + R[n - \ell]$. If we agree to let $R[0] = 0$, then this can be captured in a single driving equation:

$$R[n] \leftarrow \max_{\ell=0\ldots n} \{ \quad p[\ell] + R[n-\ell] \quad \} \tag{24}$$

Reflection: If you need to actually perform the cutting, then you want to track the decisions you made. That is, we want to track all sizes ℓ such that the maximal value is reached. Here we have a situation where we sequentially make decisions, and we expect that several seemingly different decision sequences actually lead to the same configuration because we are actually generating a set. So actually the solution sequence is a representation of an optimal partition and there are several ways that partitions can be sequentially listed. Yet other sequences belong to a different partition. Such is the case in the example below: The tracking information stored for $n = 5$ appears different but lead to the same partition, whereas the information stored for $n = 7$ leads to two different partitions. So the rod-cutting tracking information provides a blueprint for a partition, and the partition itself which may or may not be unique. In the homework you are asked how to identify whether different sequences represent different partitions.

Length n	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$	$n=7$	$n=8$	$n=9$	$n=10$	$n=11$	$n=12$	$n=13$	$n=14$	$n=15$	$n=16$
Price per entire size $p[n]$	0	1	5	8	9	10	17	17	20	24	30	30	30	38	40	40	40

Max Revenue $R[n]$	0	1	5	8	10	13	17	18	22	25	30	31	35				
Track Sizes ℓ	0	1	2	3	2	$3,2$	6	$6,3,2,1$	$6,2$	$6,3$	10	$10,1$	$10,2$				

An optimal cutting blueprint can then be produced by starting from a size ℓ and recursively look at the tracking info for $n - \ell$ to generate the full partition recursively.

6.1 Suggested Exercises

1. Complete the computations.
2. Using the tracking information: How should a rod of size n be partitioned for $n = 3$? $n = 4$? $n = 10$? $n = 11$?
3. Suppose you need to pay 1 unit for each cut that you make. How does this change the method? Effect the change and complete the computations.
4. **(Graduate students)** Back to the original. Using the tracking information is interesting: You produce sequentially the sizes of the cuts that make up an optimal multi-set. Thus, you produce a sequence, whereas only a multi-set is needed. It is possible to produce multiple sequences that end up with the same multi-set. These are called partitions of the natural number $n, n = \ell_1 + \ell_2 + \ell_3 + \ldots$. Write an algorithm that generates a blueprint for the cutting procedure by producing a partition such that the ℓ's form a non-decreasing sequence, $\ell_i \leq \ell_{i+1}$.
5. **(Graduate students)** Back to the original. Change the problem so that now only 2 cuts are free, after which each cut will bring additional revenue of 3 units. Change the approach to accommodate for this change, and repeat the calculations. Demonstrate for prices $6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 9, 9, 9, 9$. I suggest you write a program for this.

6.2 Optimal Rod-cutting with added cutting cost

Consider now the situation where the standard rod-cutting is extended in the sense that there is possibly additional profit for the cuts that are made. Let $C[k]$ be the cutting cost/profit for the k^{th} cut. For instance, making the first two cuts "free", while charging 3 units for the third and subsequent cuts can be accomplished by $C[0] = C[1] = C[2] = 0$ and $C[3] = C[4] = \ldots = 3$.

But it *does* mean that the equations must be extended as well to include the number of cuts already made. Thus let $R[n, k]$ stand for the maximal profit that can be obtained from a rod of size n and making *exactly* k cuts. It should be noted that if the rod is of size n, then there can be at most $n - 1$ cuts and $R[n, k]$ is not defined for $n \leq k$. We use $R[n, k] =$ "$-''$ to indicate this in the table below. With this new notation, the driving equations are extended:

$$R[n,0] \quad \leftarrow \quad p[n] \tag{25}$$

$$R[n,k] \quad \leftarrow \quad C[k] + \max_{\ell=1\ldots n-k} \Big\{ R[n-\ell, k-1] + p[\ell] \Big\} \quad \text{for} \quad k = 0, \ldots, n-1. \tag{26}$$

Let us reflect on this equation for a minute: If you make the k^{th} cut, then you cut at least one unit, so that $\ell = 1$ is the smallest value to be considered for ℓ. Also, if you make a cut of size ℓ, then a rod of size $n - \ell$ remains and this size must be long enough to make $k - 1$ cuts, thus $n - \ell$ must be of size k or higher. This makes $\ell = n - k$ the largest value of ℓ that can be considered.Thus we now have a straightforward set of equations for $R[n, k]$, the optimal value from a rod with making exactly k cuts. Again, this is a set for which dynamic programming applies (why?) and a set of tables is readily generated, see below.

If instead the problem was to determine the optimal cost with any number of cuts, but maximally k cuts, which we denote with $R^\star[n]$, then is is the maximal value of $R[n, \ell]$, where n is kept fixed and ℓ varies between 0 and k:

$$R^\star[n] \leftarrow \max_{k=0\ldots n-1} \{ \quad R[n,k] \quad \}. \tag{27}$$

Consider the same example as in the earlier section, to which we now add cutting cost/profit information, but we do not incorporate the tracking info and leave this for homework.

Last Length	k^{st} cut	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$	$n=7$	$n=8$	$n=9$	$n=10$	$n=11$	$n=12$	$n=13$	$n=14$	$n=15$	$n=16$
Price per unit		0	1	5	8	9	10	17	17	20	24	30	30	30	38	40	40	40
With 0 cuts: $R[n,0]$	$C(0^{\text{th}}) = 0$	0	1	5	8	9	10	17	17	20	24	30	30	30	38	40	40	40
With 1 cut: $R[n,1]$	$C(1^{\text{th}}) = 0$	-	-	2	6	10	13	16	18	22	25	26	31	35	38	39	43	47
With 2 cuts: $R[n,2]$	$C(2^{\text{nd}}) = 0$	-	-	-	3	7	11	15	18	21	24	27	30	33	36	40	43	46
With 3 cuts: $R[n,3]$	$C(3^{\text{rd}}) = 3$	-	-	-	-	7	11	15	19	23	26	29	32	35	38	41	44	48
With 4 cuts: $R[n,4]$	$C(4^{\text{th}}) = 3$	-	-	-	-	-	11	15	19	23	27	31	34	37	40	43	46	49
With 5 cuts: $R[n,5]$	$C(5^{\text{th}}) = 3$	-	-	-	-	-	-	15	19	23	27	31	35	39	42	45	48	51
With 6 cuts: $R[n,6]$	$C(6^{\text{th}}) = 3$	-	-	-	-	-	-	-	19	23	27	31	35	39	43	47	50	53
With 7 cuts: $R[n,7]$	$C(7^{\text{th}}) = 3$	-	-	-	-	-	-	-	-	23	27	31	35	39	43	47	51	55
With 8 cuts: $R[n,8]$	$C(8^{\text{th}}) = 3$	-	-	-	-	-	-	-	-	-	27	31	35	39	43	47	51	55
Max Revenue $R^\star[n]$		-	1	5	8	10	13	17	19	23	27	31	35	39	43	47	51	55

6.2.1 Suggested Exercises

Take a rod of size $n = 5$ and use $ppu = 1, 3, 5, 7$. Compute by hand the two versions: Cutting is always free, and cutting with the added costs. Explain why the cutting cost *must* be incorporated in the optimization criterion.

Now use $ppu = 1, 3, 5, 7, 9, 11 \ldots 31$ and complete the work with rods up-to, and including $n = 16$.

7 DP: Various Shortest Path Problems

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

There are various problems with DP based algorithms in the application area of graphs and networks. The best known examples the algorithms by Floyd and Warchall to find the topological closure and the all-pairs-all-destination shortest paths. Also there is a DP based algorithm for the Traveling Salesman Problem. Even Dijkstra's algorithm is debatably based on DP, even though it is usually said to be Greedy.

These will be covered in depth in the chapter on Graphs.

8 DP: Knapsack Problems

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

The basic version of the *0-1 knapsack problem* is where you try to get the most benefit (or value) by selecting items that still fit in a knapsack. The knapsack can hold a maximum of W kilograms (kg), and you can choose from among n items, each having certain weight w_i (the cost) and each giving certain value v_i (the benefit). So on one side, you want as many valuable items as possible, and at the other you are limited by the maximal weight W that the knapsack can hold.

There are several versions of the knapsack problem, and in the most general version it is presented as

$$\text{Find values for } x_i \text{ such that } \sum_{i=1}^{n} x_i v_i \text{ is Maximized, and the selection is limited by } \sum_{j=1}^{n} x_j w_j \leq W \qquad (28)$$

where the different version are described by different limitations on the allowable values of x_i:

Fractional Knapsack Problem $x_i \in [0,\ 1]$, the full interval is available. This has an easy solution by sorting the items according to decreasing "value per weight": v_i/w_i which is then used to fill the knapsack with available items, and the last "hole" is filled up with a "partial" next item.

0-1 Knapsack Problem $x_i \in \{0,\ 1\}$, only the end-points of the interval: either zero or one. This is the version we will expand upon below.

Bounded Knapsack Problem $x_i \in \{0,\ 1,\ 2,\ \ldots c\}$, so that x_i is limited to being a nonnegative integer that is bounded by a common constant.

Unbounded Knapsack Problem $x_i \in \{0,\ 1,\ 2,\ \ldots\}$, so that x_i is still limited to being a nonnegative integer, but no longer bounded by a common constant.

Additional versions are described by different limitations on the allowable values of w_i: Are their values in $\mathbb{R}$ or in $\mathbb{N}$? In the most common version of the Knapsack Problem, the values of w_i are limited to the natural numbers, $\mathbb{N}$. The reason being that these weights (and their running totals, see below) can then be enumerated.

The 0-1 Knapsack Problem

This is the most commonly presented version of the knapsack problems, and we furthermore assume that all values $v_1, v_2 \ldots v_n$, all weights $w_1, w_2 \ldots w_n$ and the Knapsack limitation W are all positive integers. We also present the problem slightly different from the formulation above:

Find a subset T, $T \subseteq \{1, 2, \ldots n\}$, representing the selected items, such that

1. their combined value $\sum_{i \in T} v_i$ is maximized, **and**
2. their combined weight is limited by W, $\sum_{j \in T} w_j \leq W$

Introduce the following notation: $V[n, W]$ as the maximum value that can be obtained with these n items that are limited to having a combined weight to W. As usual, finding this maximum value is found in the first pass, while the items that can be maximally carried in the knapsack are determined in the second pass. Realize that $V[n, W]$ can be determined by asking a simple question: Do we ínclude item n, (and thus use up w_n of any available weight allowance), ór do we exclude item n (and thus the full allowance w is available for $n-1$ items)? The governing relation is

$$V[n, W] \leftarrow \max_{\text{2 options}} \{V[n-1, W], \quad v_n + V[n-1, W - w_n]\ \} \qquad (29)$$

and the decision is easily made if the value for $V[n-1, W-w_n]$ is readily available. If you allow this computation to be evaluated recursively, and evaluate two options for each item, then you effectively inspect all 2^n all possible subsets. So if you evaluate the value of these two options these in advance, then you need to precompute $V[n-1, W-w_n]$ and since w_n is not known in advance, you need to precompute $V[n-1, w]$ for all $w < W$. We are now ready to present the governing relation for all items:

$$\boxed{V[i,W] \leftarrow \begin{cases} 0 & \text{if } i=1 \quad \text{and} \quad w_1 > W \\ v_1 & \text{if } i=1 \quad \text{and} \quad w_1 \leq W \\ V[i-1,W] & \text{if } i>1 \quad \text{and} \quad w_i > W \\ \max\limits_{\text{2 options}} \{V[i-1,W], \quad v_i + V[i-1, W-w_i]\ \} & \text{if } i>1 \quad \text{and} \quad w_i \leq W \end{cases}} \tag{30}$$

Please note that this equation can be shortened by defining $V[0,W] = 0$ and $V[i,W] = -\infty$ for all $i = 1 \ldots n$ and $W < 0$. The above equation is however more transparent.

This is demonstrated by an example. In the first pass, the forward pass, the values for $V[i,W]$ and $MyTrack[i,W]$ (i.e. $Include\ i$?) are determined. We also include a request that is called the zero-th request, and is used in the boundary equation.

Add element	value–elt v	weight–elt w		$W=0$	$W=1$	$W=2$	$W=3$	$W=4$	$W=5$	$W=6$	$W=7$	$W=8$	$W=9$	$W=10$	$W=11$	$W=12$	$W=13$	$W=14$	$W=15$	$W=16$
1:	$v_1 = 18$	$w_1 = 8$	V-table	0	0	0	0	0	0	0	0	18	18	18	18	18	18	18	18	18
2:	$v_2 = 22$	$w_2 = 6$		0	0	0	0	0	0	22	22	22	22	22	22	22	22	40	40	40
3:	$v_3 = 5$	$w_3 = 7$		0	0	0	0	0	0	22	22	22	22	22	22	22	27	40	40	40
4:	$v_4 = 28$	$w_4 = 8$		0	0	0	0	0	0	22	22	28	28	28	28	28	28	50	50	50
5:	$v_5 = 3$	$w_5 = 3$		0	0	0	3	3	3	22	22	28	28	28	31	31	31	50	50	50
1:			$MyTrack$	0	N	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
2:				0	N	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
3:				0	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N
4:				0	N	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
5:				0	N	N	Y	Y	Y	N	N	N	N	N	Y	Y	Y	N	N	N

An optimal knapsack content can then be produced by starting from $MyTrack[n,W]$ where it is indicated whether or not it should be included. The following idea could be used, where we tacitly assume that $MyTrack$ is defined as a global variable .

```
**algorithm** PrintKnapsackSet ($n$, $W$)
    **if** $n > 0$ **and** $MyTrack[n,W] = $ “$No''$ **then** PrintKnapsackSet($n-1$, $W$) **end-if**
    **if** $n > 0$ **and** $MyTrack[n,W] \neq $ “$No''$ **then Concatenate**(PrintKnapsackSet($n-1$, $W-w_n$) || “$n$”) **end-if**
 **end-alg** PrintKnapsackSet
```

For the example above, this results in: (A “Y” means it is included, a “N” indicates not included.)

	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Check
Included?	N	Y	N	Y	N	
v_i		22		28		$V[n,W] = 50$
w_i		6		8		$6 + 8 \leq W$?: Yes

The total time complexity for all steps is $\boxed{T^W(n) = \Theta(n\,W).}$ This would appear to be linear, and so it is, but it does not only depend on the total number of inputs (i.e. n), which is what is normally done. But in this case, it also depends on the actual value of the input parameter W. This makes the time complexity a little different from before, and it has a different name: it is *pseudo linear*. Finding the time complexity of the Knapsack Problem, is complicated by this since now the value of W is represented by a string of length $\lg W$. This is true for the values v_i and w_i as well. Carrying out the full classification of the Knapsack Problem is not given here (yet?), but it is $\mathcal{NPH}ard$.

8.1 Suggested Exercises

1. You purchased a bigger knapsack, and can now hold up to $W = 17$ kg. Does anything change? Show all completed tables and conclusions.
2. Back to $W = 16$, but now add another request to the collection: $v_6 = 20$ and $w_6 = 4$. Show all completed tables and conclusions.
3. Sort the items by weight, and fit them in the knapsack from light to heavy. The answer does not change, but is the computation any easier or harder?
4. Sort the items by weight, and fit them in the knapsack from heavy to light. The answer does not change, but is the computation any easier or harder?
5. Sort the items by their relative values (the fractions of value per weight, v_i/w_i, and fit them in the knapsack from highest relative value to lowest relative value. The answer does not change, but is the computation any easier or harder?
6. **(Graduate students)** The Knapsack as presented above only indicates "a" solution, but does not indicate whether or not the set itself is unique. Indicate how you would change the above so that you can answer the question "Is the Knapsack solution set unique?"
7. **(Graduate students)** Suppose you have n items, and $v_i = w_i$ for all i, $1 \leq i \leq n$. Is the knapsack somewhat predictable?
8. **(Graduate students)** Consider the telescope example as before, but now change the starting and finishing days of the requests, and let the request only be for d_i days, where $d_i = f_i - (s_i - 1)$. Thus, for the sequence of requests given: $2, 2, 6, 4, 2, 4, 8, 6, 8, 4$ and 16. We dó require that all requests are completed by f_n. There is no reason anymore to sequence the requests and we now have a set of requests. A helpful friend thinks that the problem can perhaps be solved with the Knapsack algorithm. Do you agree? If you think it it can be solved his way, solve it and schedule the optimal knapsack set in numerical order: How does this compare with the optimal sequence when the start- and finish dates have been provided? (hint: you may have to write and implement the formulas, which should be about 10 lines of code). If you think that the problem cannot be solved this way, explain which crucial element is overlooked by your friend.
9. **(Graduate students)** Suppose you try to generalize the algorithm, and allow W (but not the W_i's) to be real valued, in stead of natural numbers. How does this complicate the solution?
10. **(Graduate students)** Suppose you try to generalize the algorithm, and allow the W_i's (but not W) to be real valued, in stead of natural numbers. How does this complicate the solution?
11. **(Graduate students)** Suppose you try to generalize the algorithm, and allow both the W_i's and W to be real valued, in stead of natural numbers. How does this complicate the solution?
12. **(Graduate students)** Suppose you purchase a second knapsack with the same maximal capacity. Can you design an approach to get the maximum benefit with these two knapsacks? What if you have to pay a penalty P_j for each knapsack you carry? (e.g. $50 for an additional suitcase).

9 DP: The Longest Common Subsequence, LCS

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

Suppose you are given two sequences, **X** $= \langle x_1, x_2, x_3 \cdots x_m \rangle$ and **Y** $= \langle y_1, y_2, y_3 \cdots\cdots y_n \rangle$. A common subsequence is a sequence $\boldsymbol{T} = \langle t_1, t_2, t_3 \cdots t_k \rangle$ such that each element t_i also occurs in both the sequences **X** and **Y**, say $t_i \equiv x_{m_i} \equiv y_{n_i}$ for increasing subsequences m_i and n_i. A *longest common subsequence* is a subsequence that is longer than (or at least as long as) any other common subsequence. The length of the longest common subsequence, $c[m, n]$, is unique, although the actual subsequence may not be unique. The length $c[m, n]$ itself is given by

$$\boxed{c[i,j] \leftarrow \begin{cases} 0 & \text{if } i = 0 \text{ or if } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\{c[i, j-1],\ c[i-1, j]\} & \text{otherwise} \end{cases}} \tag{31}$$

or equivalently,

$$\boxed{c[i,j] \leftarrow \max\{c[i, j-1],\ c[i-1, j],\ c[i-1, j-1] + 1 * \mathcal{I}(x_i = y_j)\},} \tag{32}$$

where $\mathcal{I}(\text{condition})$ is the indicator function, defined as $\mathcal{I}(\text{condition}) = 1$, if the condition is true, and $\mathcal{I}(\text{condition}) = 0$, if the condition is false. The actual longest common subsequence can be constructed by tracking the decisions. For example, in the table below, each table-entry contains $c[i, j]$, and at least one of the three symbols $\uparrow$, $\leftarrow$, or $\nwarrow$ (as appropriate, use multiple symbols if the choice is not unique).

Again, this situation one of dynamic programming and is best implemented by solving small problems and incorporating small solutions into larger solutions. In the table below, fill in the information in each table-entry for $c[i, j]$. Furthermore, to track decisions, add at least one of the three symbols $\uparrow$, $\leftarrow$, or $\nwarrow$ (as appropriate, use multiple symbols if the choice is not unique). I have used the symbol "*nil*" in the boundary boxes to indicate that either **X** or **Y** (or both) is $\varnothing$, and thus either $i = 0$ or $j = 0$ (or both 0).

For example, we could construct the table below for the following two sequences

X $= \langle x_1, x_2, x_3 \cdots x_6 \rangle = \langle A,\ B,\ A,\ B,\ A,\ B \rangle$ and

Y $= \langle y_1, y_2, y_3 \cdots\cdots y_8 \rangle = \langle A,\ A,\ A,\ A,\ B,\ B,\ B,\ B, \rangle$.

	$j = 0$ $y_0 = \varnothing$	$j = 1$ $y_1 = A$	$j = 2$ $y_2 = A$	$j = 3$ $y_3 = A$	$j = 4$ $y_4 = A$	$j = 5$ $y_5 = B$	$j = 6$ $y_6 = B$	$j = 7$ $y_7 = B$	$j = 8$ $y_8 = B$
$i = 0$ $x_0 = \varnothing$	*nil* 0	*nil* 0	*nil* 0	*nil* 0	*nil* 0	*nil* 0	*nil* 0	*nil* 0	*nil* 0
$i = 1$ $x_1 = A$	*nil* 0	$\nwarrow$ 1	$\nwarrow$ $\leftarrow$ 1	$\nwarrow$ $\leftarrow$ 1	$\nwarrow$ $\leftarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1
$i = 2$ $x_3 = B$	*nil* 0	$\uparrow$ 1	$\uparrow$ $\leftarrow$ 1	$\uparrow$ $\leftarrow$ 1	$\uparrow$ $\leftarrow$ 1	$\nwarrow$ 2	$\nwarrow$ $\leftarrow$ 2	$\nwarrow$ $\leftarrow$ 2	$\nwarrow$ $\leftarrow$ 2
$i = 3$ $x_3 = A$	*nil* 0	$\nwarrow\uparrow$ 1	$\nwarrow$ 2	$\nwarrow$ $\leftarrow$ 2	$\nwarrow$ $\leftarrow$ 2	$\uparrow$ $\leftarrow$ 2	$\uparrow$ $\leftarrow$ 2	$\uparrow$ $\leftarrow$ 2	$\uparrow$ $\leftarrow$ 2
$i = 4$ $x_4 = B$	*nil* 0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ $\leftarrow$ 2	$\uparrow$ $\leftarrow$ 2	$\nwarrow$ 3	$\nwarrow$ $\leftarrow$ 3	$\nwarrow$ $\leftarrow$ 3	$\nwarrow$ $\leftarrow$ 3
$i = 5$ $x_5 = A$	*nil* 0	$\nwarrow\uparrow$ 1	$\nwarrow\uparrow$ 2	$\nwarrow$ 3	$\nwarrow$ $\leftarrow$ 3	$\uparrow$ $\leftarrow$ 3	$\uparrow$ $\leftarrow$ 3	$\uparrow$ $\leftarrow$ 3	$\uparrow$ $\leftarrow$ 3
$i = 6$ $x_6 = B$	*nil* 0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ $\leftarrow$ 3	$\nwarrow$ 4	$\nwarrow$ $\leftarrow$ 4	$\nwarrow$ $\leftarrow$ 4	$\nwarrow$ $\leftarrow$ 4

This means that the length of the LCS is 4. Also, there is enough information stored in the table to construct an LCS. In fact, there are several actual sequences, all of them have length 4, that are longest common subsequences. Although the length of the LCS is unique, an actual LCS need not be unique. For instance, in the current example, the following are all actual LCS: $ABBB$, $AABB$, and $AAAB$. But even the subsequence $ABBB$ can be obtained by taking different indices of the X or Y-sequences. This is perhaps best illustrated by aligning them differently. I have bold-faced them for emphasis, and presented them under the heading "sequence alignment" below. *Note, there are several good youtube video's explaining these, the following have been recommended by students:* `https://youtu.be/P-mMvhfJhu8`

9.1 LCS: Suggested Exercises

1. Show a longest common subsequence **Z** for two sequences
X $= \langle x_1, x_2, x_3 \cdots x_6 \rangle = \langle A,\ B,\ B,\ A,\ A,\ B \rangle$ and
Y $= \langle y_1, y_2, y_3 \cdots\cdots y_8 \rangle = \langle A,\ A,\ A,\ B,\ B,\ B,\ B,\ A,\ \rangle$.

Is the location of the longest common subsequence unique, or are there multiple locations for the LCS in either the **X** or the **Y** sequence? Explain.

2. (Graduate Students) There is another classical problem that you are now asked to solve with two suggested approaches (neither of them is "the best known.")

Suppose you are given one sequence of natural numbers (positive integers), **Y** $= \langle y_1, y_2, \cdots\cdots y_n \rangle$. An increasing subsequence is a sequence **Y** $= \langle y_{n_1}, y_{n_2}, y_{n_3} \cdots\cdots y_{n_k} \rangle$, such that both $n_i < n_{i+1}$ and $y_{n_i} < y_{n_{i+1}}$. A longest increasing subsequence is a subsequence that is longer than (or at least as long as) any other increasing subsequence. Of course, this is a variation of the LCS, and since you are now an expert of the LCS, you are exploring two ideas:

2:a
Step 1: Find the $\max$ of the integers in the sequence **Y**. Let us call this maximal number m. That is, $m = \max\{y_1 \cdots y_n \}$.
Step 2: Create the sequence **X** $= \langle 1, 2, 3 \cdots m \rangle$
Step 3: Run the LCS on the two sequences.

2:b
Step 1: Sort a copy of the sequence **Y**, and store the result in the sequence **S**.
Step 2: Run the LCS on the two sequences.

Explain why these algorithms are correct and find their worst-case time complexities.

Note, neither of is these approaches is considered most efficient.The second one has optimal time-complexity, but does not have optimal space complexity, and furthermore, it is a "two-pass" algorithm, and can not be performed "on-line". For more information, see `https://en.wikipedia.org/wiki/Longest_increasing_subsequence`

3. (Graduate Students) Although the *length* of the LSC is unique, there could be more than one common sequences with that length. If you encounter choices on the path from table location $[m, n]$ back to location $[0, 0]$, then there is more than one maximal subsequence. But how many? Let $N[m, n]$ count the number of LCS's, then perhaps the appropriate expression is (loosely)

$$N[i,j] \leftarrow N[i,j-1] \cdot \mathcal{I}(``\leftarrow") \ + \ N[i-1,j] \cdot \mathcal{I}(``\uparrow") \ + \ N[i-1,j-1] \cdot \mathcal{I}(``\nwarrow"), \tag{33}$$

where the indicator function is used again to incorporate the various options. This expression is more accurately given as

$$N[i,j] \leftarrow N[i,j-1]\cdot\mathcal{I}(``\leftarrow \in Track[i,j]") + N[i-1,j]\cdot\mathcal{I}(``\uparrow \in Track[i,j]") + N[i-1,j-1]\cdot\mathcal{I}(``\nwarrow \in Track[i,j]"). \tag{34}$$

Could this possibly be correct? How would you initialize the table at the boundary? (i.e. whenever m, n or both are zero). Check your answer by comparing the answer both by using the above equation in a table, as well as by hand for the sequences **X** $= \langle x_1, x_2, x_3 \cdots x_7 \rangle = \langle A,\ A,\ A,\ A,\ B,\ B,\ B \rangle$ and **Y** $= \langle y_1, y_2, y_3 \rangle = \langle A,\ B,\ A \rangle$. Can you adjust the formula and make it work? (You may need to extend to an alphabet with more than two characters before you get the correct idea.)

4. (Graduate Students) The expression for $N[n,m]$ given above is nice, but is not correct: Compare e.g. the sequences $\mathbf{X} = \langle x_1, x_2 \rangle = \langle A,\ A \rangle$ and $\mathbf{Y} = \langle y_1, y_2 \rangle = \langle A,\ B \rangle$. The above formula gives 3, when the correct answer is 2. Consider now the following functions:
$\ell d[m,\ n] = [p,\ n]$, if $\nwarrow \in MyTrack[p,\ n]$ whereas $\nwarrow \notin MyTrack[i,\ n]$ for all $i = p+1 \ldots m-1$ or $\ell d[m,\ n] = [0,\ 0]$ if there is no more "$\nwarrow$" to the left. Also,
$ud[m,\ n] = [m,\ q]$, if $\nwarrow \in MyTrack[m,\ q]$ whereas $\nwarrow \notin MyTrack[m,\ j]$ for all $j = q+1 \ldots n-1$ or $ud[m,\ n] = [0,\ 0]$ if there is no more "$\nwarrow$" to the top. These are essentially pointing to the highest row-/column- values that have a diagonal arrow. With these new functions, the correct equation becomes:

$$N[i,j] \leftarrow N(\ell d[i,j]) \cdot \mathcal{I}(``\leftarrow \in Track[i,j]'') + N(ud[i,j]) \cdot \mathcal{I}(``\uparrow \in Track[i,j]'') + N[i-1,j-1] \cdot \mathcal{I}(``\nwarrow \in Track[i,j]''). \quad (35)$$

This looks pretty complicated, but it is not really that bad. Consider the table we have seen in the previous section, but now add the extra counter to it.

	$j=0$ $y_0=\oslash$	$j=1$ $y_1=A$	$j=2$ $y_2=A$	$j=3$ $y_3=A$	$j=4$ $y_4=A$	$j=5$ $y_5=B$	$j=6$ $y_6=B$	$j=7$ $y_7=B$	$j=8$ $y_8=B$
$i=0$ $x_0=\oslash$	*nil* $0,1$	*nil* $0,1$	*nil* $0,1$	*nil* $0,1$	*nil* $0,1$	*nil* $0,1$	*nil* $0,1$	*nil* $0,1$	*nil* $0,1$
$i=1$ $x_1=A$	*nil* $0,1$	$\nwarrow$ $1,1$	$\nwarrow$ $\leftarrow$ $1,2$	$\nwarrow$ $\leftarrow$ $1,3$	$\nwarrow$ $\leftarrow$ $1,4$	$\leftarrow$ $1,4$	$\leftarrow$ $1,4$	$\leftarrow$ $1,4$	$\leftarrow$ $1,4$
$i=2$ $x_3=B$	*nil* $0,1$	$\uparrow$ $1,1$	$\uparrow$ $\leftarrow$ $1,2$	$\uparrow$ $\leftarrow$ $1,3$	$\uparrow$ $\leftarrow$ $1,4$	$\nwarrow$ $2,4$	$\nwarrow$ $\leftarrow$ $2,8$	$\nwarrow$ $\leftarrow$ $2,12$	$\nwarrow$ $\leftarrow$ $2,16$
$i=3$ $x_3=A$	*nil* $0,1$	$\nwarrow\uparrow$ $1,2$	$\nwarrow$ $2,1$	$\nwarrow$ $\leftarrow$ $2,3$	$\nwarrow$ $\leftarrow$ $2,6$	$\uparrow$ $\leftarrow$ $2,10$	$\uparrow$ $\leftarrow$ $2,14$	$\uparrow$ $\leftarrow$ $2,18$	$\uparrow$ $\leftarrow$ $2,22$
$i=4$ $x_4=B$	*nil* $0,1$	$\uparrow$ $1,2$	$\uparrow$ $2,1$	$\uparrow$ $\leftarrow$ $2,3$	$\uparrow$ $\leftarrow$ $2,6$	$\nwarrow$ $3,6$	$\nwarrow$ $\leftarrow$ $3,16$	$\nwarrow$ $\leftarrow$ $3,30$	$\nwarrow$ $\leftarrow$ $3,48$
$i=5$ $x_5=A$	*nil* $0,1$	$\nwarrow\uparrow$ $1,3$	$\nwarrow\uparrow$ $2,3$	$\nwarrow$ $3,1$	$\nwarrow$ $\leftarrow$ $3,4$	$\uparrow$ $\leftarrow$ $3,10$	$\uparrow$ $\leftarrow$ $3,20$	$\uparrow$ $\leftarrow$ $3,34$	$\uparrow$ $\leftarrow$ $3,52$
$i=6$ $x_6=B$	*nil* $0,1$	$\uparrow$ $1,3$	$\uparrow$ $2,3$	$\uparrow$ $3,1$	$\uparrow$ $\leftarrow$ $3,4$	$\nwarrow$ $4,4$	$\nwarrow$ $\leftarrow$ $4,14$	$\nwarrow$ $\leftarrow$ $4,34$	$\nwarrow$ $\leftarrow$ $4,68$

Finally the question: Circle the table entires that were used to calculate $N[6,8] = 68$, $N[5,6] = 16$, $N[4,4] = 6$, and $N[2,2] = 2$.

10 DP: Sequence Alignment

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

This is pretty much a continuation of the LCS. Traditional presentations keep the sequences in place, and connect them using a straight line. This presentation moves towards the next problem with a similar structure, the smallest edit-distance, which will be the covered later. Here, we use vertical alignment between bold-faced elements.Like the situation before, and similar situations in the future, there are three, or perhaps even four cases to consider, and thus three, or potential four actions that can be taken.
For the *Sequence Alignment*, you are again given two sequences, **X** $= \langle x_1, x_2, x_3 \cdots x_m \rangle$ and **Y** $= \langle y_1, y_2, y_3 \cdots\cdots y_n \rangle$. For ease of explanation of the sequence alignment problem, we assume a longest common subsequence is now known in a first pass through the problem, and stored in $c[i, j]$ for all i and j in the correct range. We will explain the second pass, the construction pass here. In actuality you would want to implement them altogether, which is possible if you have a solid understand of strings and string-manipulation. Here we present the problem and their solution assuming that sequences are arrays. This is again a two-pass solution. In the sequence alignment problem, you need to construct two new sequences of equal length $\mathbf{X}^a = \langle x_1^a, x_2^a, x_3^a \cdots x_P^a \rangle$ and $\mathbf{Y}^a = \langle y_1^a, y_2^a, y_3^a \cdots\cdots y_P^a \rangle$, where $P \geq m$ and $P \geq n$, which are obtained from **X** and **Y** by inserting blank spaces if there is a mismatch, and such that elements of the longest common subsequence of **X** and **Y** now have the same index in the new sequences $\mathbf{X}^a$ and $\mathbf{Y}^a$. These sequences are easily constructed in the second phase of the algorithm, on the way "back".
First consider the sequence **X**: there are m elements total, of which $c[m, n]$ are in a longest common subsequence with **Y**, and thus there are $m - c[m, n]$ elements that are not matched. This means that the sequence $\mathbf{Y}^a$ contains all elements of **Y**, and that $m - c[m, n]$ new "blank spaces" inserted. Thus: $P = m + n - c[m, n]$.

It is easiest to understand the by making and following an example. We have assumed that information about $c[m, n]$ is known, and that we now are in the process of "identifying" an actual longest sequence. (Again, the length of the sequence is unique; it is a maximum, although the sequence itself may not be.)

$\boxed{\nwarrow}$ If you follow a "$\nwarrow$" at location m, n of the $c - table$, (which was possible because $x_m = y_n$), then both x_m and y_n are copied into the appropriate location of $\mathbf{X}^a$ and $\mathbf{Y}^a$:
Let $p = m + n - c[m, n]$, then $\mathbf{X}^a[p] \leftarrow x_m$ and $\mathbf{Y}^a[p] \leftarrow y_n$.

$\boxed{\uparrow}$ If you follow a "$\uparrow$" at location m, n of the $c - table$, (which was possible because of a mis-match $x_m \neq y_n$ and $c[m, n] == c[m - 1, n]$), then the element x_m is copied into the the appropriate location of $\mathbf{X}^a$, and a "blank space" is inserted as a "blank match" into the $\mathbf{Y}^a$:
Let $p = m + n - c[m, n]$, then $\mathbf{X}^a[p] \leftarrow x_m$ and $\mathbf{Y}^a[p] \leftarrow \square$.

$\boxed{\leftarrow}$ If you follow a "$\leftarrow$" at location m, n of the $c - table$, (which was possible because of a mis-match $x_m \neq y_n$ and $c[m, n] == c[m, n - 1]$), then the element y_n is copied into the the appropriate location of $\mathbf{Y}^a$, and a "blank space" is inserted as a "blank match" into the $\mathbf{X}^a$: Let $p = m + n - c[m, n]$, then $\mathbf{Y}^a[p] \leftarrow y_m$ and $\mathbf{X}^a[p] \leftarrow \square$.

When actually implementing this idea, you would in reality only have three iterators: i, j, p (initialized as $m, n, m + n - c[m, n]$ respectively), which are (or are not) decremented appropriately. Please look at the examples below, which are some of the aligned sequences for the example of the previous section.

$$\begin{array}{cc} & \begin{array}{cccccccccc} x_1^a & x_2^a & x_3^a & x_4^a & x_5^a & x_6^a & x_7^a & x_8^a & x_9^a & x_{10}^a \end{array} \\ 1 & \left[\begin{array}{cccccccccc} x_1 & \square & \square & \square & x_2 & x_3 & x_4 & x_5 & x_6 & \square \\ \mathbf{A} & & & & \mathbf{B} & A & \mathbf{B} & A & \mathbf{B} & \\ \mathbf{A} & A & A & A & \mathbf{B} & & \mathbf{B} & & \mathbf{B} & B \\ y_1 & y_2 & y_3 & y_4 & y_5 & \square & y_6 & \square & y_7 & y_8 \end{array}\right] \end{array}$$

$$\begin{array}{cc} & \begin{array}{cccccccccc} x_1^a & x_2^a & x_3^a & x_4^a & x_5^a & x_6^a & x_7^a & x_8^a & x_9^a & x_{10}^a \end{array} \\ 2 & \left[\begin{array}{cccccccccc} \square & x_1 & \square & \square & x_2 & x_3 & x_4 & x_5 & x_6 & \square \\ & \mathbf{A} & & & \mathbf{B} & A & \mathbf{B} & A & \mathbf{B} & \\ A & \mathbf{A} & A & A & \mathbf{B} & & \mathbf{B} & & \mathbf{B} & B \\ y_1 & y_2 & y_3 & y_4 & y_5 & \square & y_6 & \square & y_7 & y_8 \end{array}\right] \end{array}$$

$$\begin{array}{cc} & \begin{array}{cccccccccc} x_1^a & x_2^a & x_3^a & x_4^a & x_5^a & x_6^a & x_7^a & x_8^a & x_9^a & x_{10}^a \end{array} \\ 3 & \left[\begin{array}{cccccccccc} \square & \square & x_1 & \square & x_2 & x_3 & x_4 & x_5 & x_6 & \square \\ & & \mathbf{A} & & \mathbf{B} & A & \mathbf{B} & A & \mathbf{B} & \\ A & A & \mathbf{A} & A & \mathbf{B} & & \mathbf{B} & & \mathbf{B} & B \\ y_1 & y_2 & y_3 & y_4 & y_5 & \square & y_6 & \square & y_7 & y_8 \end{array}\right] \end{array}$$

$$\begin{array}{cc} & \begin{array}{cccccccccc} x_1^a & x_2^a & x_3^a & x_4^a & x_5^a & x_6^a & x_7^a & x_8^a & x_9^a & x_{10}^a \end{array} \\ 4 & \left[\begin{array}{cccccccccc} \square & \square & \square & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & \square \\ & & & \mathbf{A} & \mathbf{B} & A & \mathbf{B} & A & \mathbf{B} & \\ A & A & A & \mathbf{A} & \mathbf{B} & & \mathbf{B} & & \mathbf{B} & B \\ y_1 & y_2 & y_3 & y_4 & y_5 & \square & y_6 & \square & y_7 & y_8 \end{array}\right] \end{array}$$

and so on. You can continue till you have generated all alignment options for the **ABBB**-common sequence. The last one is probably

$$\begin{array}{ccccccccccc}
 & x_1^a & x_2^a & x_3^a & x_4^a & x_5^a & x_6^a & x_7^a & x_8^a & x_9^a & x_{10}^a \\
 & \square & \square & \square & x_1 & \square & x_2 & x_3 & x_4 & x_5 & x_6 \\
16 & & & & \mathbf{A} & & \mathbf{B} & A & \mathbf{B} & A & \mathbf{B} \\
 & A & A & A & \mathbf{A} & B & \mathbf{B} & & \mathbf{B} & & \mathbf{B} \\
 & y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & \square & y_7 & \square & y_8
\end{array}$$

and then you can get started on the **AABB** sequence, and so on. Each one of these is actually a different alignment, and thus should be counted differently. To get an even better feeling for the correspondence between an alignment and a path through the $c[i, j]$-table, we added arrow-notation in another example. The actual path construction uses back-tracking, so any path is generated from $c[m, n]$ back to $c[1, 1]$.

$$\begin{array}{cccccccccc}
x_1^a & x_2^a & x_3^a & x_4^a & x_5^a & x_6^a & x_7^a & x_8^a & x_9^a & x_{10}^a \\
\square & x_1 & \square & \square & \square & x_2 & x_3 & x_4 & x_5 & x_6 \\
 & \mathbf{A} & & & & \mathbf{B} & A & \mathbf{B} & A & \mathbf{B} \\
 & \nwarrow & \leftarrow & \leftarrow & \leftarrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow \\
A & \mathbf{A} & A & \mathbf{A} & B & \mathbf{B} & & \mathbf{B} & & \mathbf{B} \\
y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & \square & y_7 & \square & y_8
\end{array}$$

So in short: The first pass of the alignment problem is identical to the first pass of the longest common subsequence problem. The second pass is different, and rather than “merely“ identifying the actual elements in a common sequence, there are also actions involved in the non-matching elements: insert a blank symbol.

For more info, see the textbook, or `https://en.wikipedia.org/wiki/Longest_common_subsequence_problem` and its references.

10.1 Suggested Exercises

1. Show an alignment for the two sequences
X $= \langle x_1, x_2, x_3 \cdots x_6 \rangle = \langle A,\ B,\ B,\ A,\ A,\ A\ , A,\ B \rangle$ and
Y $= \langle y_1, y_2, y_3 \cdots\cdots y_8 \rangle = \langle A,\ A,\ A,\ B,\ B,\ A,\ A,\ B,\ B,\ A,\ \rangle$.
Is the alignment unique? Explain.

11 DP: The smallest (weighted) edit-distance

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

Suppose you are given two sequences, $\mathbf{X} = \langle x_1, x_2, x_3 \cdots x_m \rangle$ and $\mathbf{Y} = \langle y_1, y_2, y_3 \cdots\cdots y_n \rangle$. You want to change the sequence **X** into sequence **Y** by making the smallest number of edits. The term "edit" here is either: "Copy (C)" an element, "Substitute (S)" an element, "Delete (D)" an element, or finally "Insert (I)" an element. Each such "edits" comes with a certain editing cost. For this *smallest edit distance*, the various costs can be compared in advance, which allows us to fill up he table. We use the insertion-idea's that were used during the second pass of the alignment problem. The very first time that this problem was posed, they took that the cost of a "Copy" is zero, whereas the costs for the others were one. These costs were later generalized, and the problem is now to find the smallest weighted edit distance.

Thus, two sequences **X** and **Y** are given, and a third sequence is to be constructed, $\mathbf{X}^d$, such that $\mathbf{X}^d == \mathbf{Y}$. Again, we present the problems in the setting of arrays, whereas such sequences are better represented as strings and better explained with string manipulating operations. Also, there are two passes again: during the first phase, the minimum edit distance is computed, and an actual third sequence is constructed during the second pass, "on the way back" . Let $D[m, n]$ be the smallest editing cost to convert **X** (of length m) into $\mathbf{X}^d == \mathbf{Y}$ (of length n). In determining $D[m, n)$, we can consider $D[m-1, n-1)$, $D[m-1, n)$, and $D[m, n-1)$ as follows.

If $x_m = y_n$ then you do need to anything, just copy the element from **X** into $\mathbf{X}^d$. The cost for a Copy would normally be zero. Except that we use $\delta_{\text{copy}}(\ )$ to indicate the cost for a Copy, even though it could in fact be zero. The cost for this option, is $D[m-1, n-1] + \delta_{\text{copy}}(x_m, y_n) \times \mathcal{I}(x_m = y_n)$where the indicator function $\mathcal{I}$ is used again.

If $x_m \neq y_n$ then there are three options:

Substitute In this case, we still advance in both sequences, as long as the symbol x_m is substituted with the symbol y_n. The cost for this is $D[m-1, n-1] + \delta_{\text{subs}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n)$

Delete the symbol x_m from the **X**-sequence: $D[m-1, n] + \delta_{\text{delete}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n)$

Insert the symbol y_n into the **X**-sequence, $D[m, n-1] + \delta_{\text{insert}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n)$

So, the minimal editing costs is given by:

$$\boxed{D[m,n] \leftarrow \min_{\text{up to 4 options}} \begin{cases} D[m-1, n-1] + \delta_{\text{copy}}(x_m, y_n) \times \mathcal{I}(x_m = y_n) & \boxed{\nwarrow} \text{ Copy the element} \\ D[m-1, n-1] + \delta_{\text{subs}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n) & \boxed{\searrow} \text{ Substitute the element} \\ D[m-1, n] + \delta_{\text{delete}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n) & \boxed{\downarrow} \text{ Delete } x_m \\ D[m, n-1] + \delta_{\text{insert}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n) & \boxed{\uparrow} \text{ Insert } y_n \end{cases}} \quad (36)$$

where we used the arrows as tracking markers that allows the actual minimal edits to be made.

Editing Distance Measures: There are several different values for the δ-functions around. The Levenshtein distance is usually the first one mentioned. It has $\delta_{\text{copy}}(x_m, y_n) \times \mathcal{I}(x_m = y_n) = 0$, $\delta_{\text{subs}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n) = 1$, $\delta_{\text{delete}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n) = 1$ and $\delta_{\text{insert}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n) = 1$. The standard reference is however in Russian, and appears to be discussing the construction of error-correcting codes for binary strings that are robust for trains of 0's and 1's.

In the past, I have experimented with $\delta_{\text{copy}}(x_m, y_n) \times \mathcal{I}(x_m = y_n) = 0$, $\delta_{\text{subs}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n) = 3$, $\delta_{\text{delete}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n) = 1$ and $\delta_{\text{insert}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n) = 2$, which seem reasonable just by argument of distance and/or ease of reach for keys on the keyboard.

Several other measures of distance are possible, e.g. Needleman-Wunsch, and Smith-Waterman. We will not discuss these, but have a word of caution: Several textbooks combine the two functions $\delta_{\text{copy}}(x_m, y_n) \times \mathcal{I}(x_m = y_n)$ and $\delta_{\text{subs}}(x_m, y_n) \times \mathcal{I}(x_m \neq y_n)$ into one, and omit the conditions $\mathcal{I}(x_m = y_n)$ and $\mathcal{I}(x_m \neq y_n)$. This is a correct observation, since the "conditional" information can be incorporated into the $\delta_{\text{subs}}(\ )$ function. Keep this in mind when implementing a fully weighted edit distance program. The formulation we use keeps the conditional multiplication, even though it is superfluous, since it is easier to go back and forth between several weighting options.

11.1 Epilogue

You may think that this is the end of these kind of algorithms: Think again. Other algorithms incorporate a distance function that does not only depend on the pair (x_m, y_n), but also on the preceding symbols (note that it may be easier to delete (say) 5 elements in one shot, as compared to 5 individual deletions). Thus the δ-functions now incorporate the values of the indices as well. Before long, these algorithms can no longer be presented in the current framework, and must be presented in terms of patterns inside a text, and subsequently the related algorithms are classified as *pattern recognition* algorithms, with languages, language recognizers, parsers, and so on at it's core. The main tool in this area is finite state machines. Thus the current topic comes to an end.

11.2 Edit Distance, Suggested Exercises

1. (Graduate Students)
What is the minimal edit distance between the words DIFFICULT and SIMILAR if you use
$\delta_{\text{copy}}(x_m, y_n)\times \mathcal{I}(x_m = y_n) = 0,$
$\delta_{\text{subs}}(x_m, y_n)\times \mathcal{I}(x_m \neq y_n) = 1,$
$\delta_{\text{delete}}(x_m, y_n)\times \mathcal{I}(x_m \neq y_n) = 1$ and
$\delta_{\text{insert}}(x_m, y_n)\times \mathcal{I}(x_m \neq y_n) = 1.$

2. (Graduate Students)
What is the minimal edit distance between the words DIFFICULT and SIMILAR if you use
$\delta_{\text{copy}}(x_m, y_n)\times \mathcal{I}(x_m = y_n) = 0,$
$\delta_{\text{subs}}(x_m, y_n)\times \mathcal{I}(x_m \neq y_n) = 3,$
$\delta_{\text{delete}}(x_m, y_n)\times \mathcal{I}(x_m \neq y_n) = 1$ and
$\delta_{\text{insert}}(x_m, y_n)\times \mathcal{I}(x_m \neq y_n) = 2.$

3. (Graduate Students)
What is the minimal edit distance between the words ALGORITHM COURSE and COURSE ON ALGORITHMS if you use
$\delta_{\text{copy}}(x_m, y_n)\times \mathcal{I}(x_m = y_n) = 0,$
$\delta_{\text{subs}}(x_m, y_n)\times \mathcal{I}(x_m \neq y_n) = 1,$
$\delta_{\text{delete}}(x_m, y_n)\times \mathcal{I}(x_m \neq y_n) = 1$ and
$\delta_{\text{insert}}(x_m, y_n)\times \mathcal{I}(x_m \neq y_n) = 1.$

4. (Graduate Students)
Find common pictures: Two people are deciding on which pictures to use for a joint project: Each provides a sequence of pictures in order of preference: $x_1, x_2, x_3, \ldots x_m$ and $y_1, y_2, y_3, \ldots y_n$. You need to find the sequence of common pictures, but is there is a problem: two pictures that may appear in both sequences, may not appear in the same order. How would you solve this problem? Which algorithm would you use, and why? Does it matter if you switch the roles of **X** and **Y**?

5. (Graduate Students)
Given the following two sequences
$\mathbf{X} = \langle x_1, x_2, x_3 \cdots x_6 \rangle = \langle A,\ B,\ A,\ B,\ A,\ B \rangle$ and
$\mathbf{Y} = \langle y_1, y_2, y_3 \cdots\cdots y_8 \rangle = \langle A,\ A,\ A,\ A,\ B,\ B,\ B,\ B,\ \rangle.$

Using the $0, 1, 1, 1$ weights:

1. Determine the minimal edit distance,
2. Determine the sequence of editing instructions that changes **X** into **Y** (e.g. delete, delete, insert, ...)
3. Determine the sequence of editing instructions that changes **Y** into **X** (e.g. delete, delete, insert, ...)

Using the $0, 1, 2, 3$ weights:

1. Determine the minimal edit distance,
2. Determine the sequence of editing instructions that changes **X** into **Y** (e.g. delete, delete, insert, ...)
3. Determine the sequence of editing instructions that changes **Y** into **X** (e.g. delete, delete, insert, ...)

6. (Graduate Students)
Finding the number of different edit-instructions: It could very well be that there two different edit-sequences leading to the same target, and having the same cost.

12 DP: MIST: Maximum Independent Set of a Tree

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

(GRADUATE STUDENTS only)

So far, we have seen dynamic programming algorithms that could be implemented or structured with either a one or a two dimensional arrays. The following example shows a hierarchical structure. The tree in question can be anything, from binary tree, general tree, spanning tree, rooted tree, Shortest Path tree, and so on.

Suppose you have a graph $G = (V, E)$. A subset V_1 of V is said to be *an independent set* if two nodes in V_1 are not connected in E. A *maximal independent set* is an independent set of maximum size. Note that if an edge (u, v) is in E, then at most one of u and v, and perhaps neither, can be in V_1.

Determining the size of a maximal independent set is, in general, a very difficult problem. If however G has a special structure, then it may be easier. For instance, if G is a (free) tree, that is, it is connected and without cycles, then finding the MIST (*maximal independent set of a Tree*) is all of sudden a lot easier: Pick any node S as a *root*. This induces a hierarchy and the root has children, grandchildren, and so on. Each of these (grand-)children in turn becomes the root of it's own sub-tree, and all these trees are in different connected components, each of these components is a (free) tree, and each of these components has its own MIST. So the question is: Should the root node S be included, or not be included in the subset V_1? There are indeed just these two options:

1. If $Start$ is *not* included, then it's children have the option to be included.
2. If $Start$ *is* included, then it's children *do not* have an option, and they *cannot* be included. But the grandchildren of $Start$ have again the option, and can be considered for inclusion.

Before deriving the driving equation, introduce notation: Let $N(S)$ be the maximum number of nodes in the independent set of the Tree that is rooted by S. Yes, it is a mouthful, but needed: It counts the number of "special" nodes inside a tree, and as such the argument of the function is a Tree that happens to be rooted at S. So, $N(S)$ is the maximum of these two options:

1. If S is *not* to be included, then $N(S)$ is the sum of $N(c)$, where the sum goes over all subtrees whose roots are the children c of S.
2. If S *is* to be included, then it's children *do not* have an option, and they *cannot* be included. But the grandchildren of S have again the option, and can be considered for inclusion. Thus $N(S)$ is 1 (for S) PLUS the sum of $N(g)$, where the sum goes over all subtrees whose roots are the grandchildren g of S.

Thus, the maximum size of the independent set is:

$$\boxed{N(S) \leftarrow \max_{\text{2 options}} \begin{cases} \sum N(c) & \text{the sum taken over all subtrees rooted at children } c \text{ of } S \\ 1 + \sum N(g) & \text{the sum taken over all subtrees rooted at grandchildren } g \text{ of } S \end{cases}} \tag{37}$$

Note that the time complexity is linear, since a BFS traversal can be extended to track the values for $N(n)$ for all subtrees rooted at n. The edges are traveled forward to find the leaves, and then backward to report the findings on (grand)children and subsequent calculation. Also, there are other algorithms as well, but harder to prove they are correct: Take e.g. the standard DFS, where all leaves are taken as part of V_1, and other nodes are only taken if they have no children that are already taken.

12.1 MIST: Suggested Exercises

1. Draw a graph that is also a tree, with 9 or 10 nodes. Select a node as $Start$, and determine the MIST for this graph. Are all the leaves included in V_1?

2. Take the same graph as above, and select a different node as $Start$. Determine the MIST again. Is this the same as before? Are all the leaves included in V_1?

3. Given a graph that is also a free tree with n nodes. What are the upper and lower boundary values for $N(G)$ as a function of n? Draw graphs that are also free trees with $n = 10$ nodes such that these bounds are attained.

4. Now write the algorithm that determines the value of $N(G)$ for a general graph G that can be assumed to be a free tree as well. Find one or two friends that have done the same, and compare the solutions. Together make corrections if needed and make sure you all can explain why the algorithm is correct.

13 DP: Miscellaneous Other Problems

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

13.1 Taking turns with a coin at either end

Suppose you have a sequence of coins, each with a value. Let the number of coins be *even*. You play a game as follows: You and an opponent take turns taking a coins at either end, until no more coins are left. You start. The winner is the player with the higher sum total.

Notation: $V[i]$ is the value of coin i and $M[i,\ j]$ is the maximum you can have. We assume that the opponent is equally smart, and also wants to maximize the value. (And when your opponent picks the coin that $\max$imizes the score, then you are left with the $\min$. The driving equation:

$$\boxed{M[i,\ j] \leftarrow \begin{cases} 0 & \text{if } j < i \\ V[i] & \text{if } i == j \\ \max\{V[i],\ V[i+1]\} & \text{if } j = i+1 \\ \max\limits_{\text{2 options}} \left\{ \begin{array}{c} V[i] + \min\limits_{\text{2 options}} \{M[i+2,\ j],\ M[i+1, j-1]\} \\ \text{or} \\ V[j] + \min\limits_{\text{2 options}} \{M[i+1,\ j-1],\ M[i, j-2]\} \end{array} \right\} & \text{otherwise} \end{cases}} \tag{38}$$

This formula is pretty intimidating. It is however not too bad if you write it algorithmically as a nested **if** (_) **then** _ **else** _ **end-if**'s. Then put two **for-loop**s around it, and it becomes a CS101 type program which is hard to explain from iterative viewpoint.

Question: What is the maximal value you can get from the sequence $20, 25, 35, 28, 33, 23$?

(Graduate Students) Question: What is the maximal value you can get from the sequence $21, 27, 25, 48, 43, 23, 33, 12$?

13.2 Longest increasing subsequence

Let $L[n]$ be the length of the longest increasing subsequence, then driving equation:

$$\boxed{L[n] \leftarrow 1 + \max_{\substack{j=1\ldots n-1 \\ \text{with } A[j]<A[n]}} \{\ L[j]\ \}} \tag{39}$$

In class we discussed several other ways to incorporate conditional computations.

Below is the start of a template for taking turns with removing a coin game. Feel free to use it

	1	2	3	4	5	6	7	
1								
2								
3								
4								
5								
1								
7								

13.3 Maximum Subsequence Sum, aka Maximum Subarray Sum

Given a sequence (or array) of n integers (positive negative or zero), and the task is to find the maximal value that can be obtained by adding all elements in a consecutive subsequence (or subarray). Most often, an empty subsequence (sub-array) is also allowed. The sum of the elements in an empty sequence is zero. Let us present it as an array problem, and let the array $A[1] \dots A[n]$ be given, then a maximal subarray sum is the largest value of $A[\ell] + \dots + A[h]$ that can be obtained by adding the consecutive numbers $A[\ell] \dots A[h]$, for any choices of ℓ and h. If all elements are negative, then we can return an empty sub-array. If there is at least one positive integer in the array, then the maximal sub-array sum is positive and at least larger then that element.

Watch out: This is a very popular problem that can be found in many text books (and in many technical interview situations). The problem can be presented with several additional restrictions, and can be extended to several generalizations. Also, it is easily confused with the Maximum SubSet Sum, which relaxes the requirement that the chosen elements are consecutive. It is also particularly popular because it is a nontrivial problem for which there are two brute force algorithms that run in n^3 and n^2 time, then there is a divide-and-conquer algorithm that runs in $n \ \lg n$ time, whereas the Dynamic Programming approach is linear: it runs in n-time. We believe that the problem and the DP algorithm is easier to understand if you have not yet been exposed to the other solutions. Unfortunately, many textbooks present the dynamic programming solution as a program based on 'clever insight trick' and not first as an algorithmic design based on DP which is then implemented in an iterative fashion. Such a missed opportunity to explain and advocate DP occurs more often.

Let us use M_n to mean the largest sum of a consecutive subsequence anywhere between 1 and n. Let us try to build a recursive pattern like we have seen before:

If the previous value M_{n-1} is already available for some sub-interval that is anywhere between 1 and $n-1$, then we would try to determine the value of M_n by asking the simple question: do we ínclude or éxclude this last element? If we include the last element (numbered n) , then we can still not add the values of M_{n-1} and $A[n]$ together because the element $A[n]$ may not be contiguous to best interval between 1 and $n-1$. Thus we need to introduce another variable, an auxiliary variable, a helper variable, such as N_n, which will stand for the largest sum of a consecutive subsequence between 1 and n, and that includes the last element n. Some authors call this the prefix sum.

For this new variable, we can successfully write a driving equation. If $N_{n-1} + A[n]$ is positive, then this is the largest prefix sum N_n. If on the other hand $N_{n-1} + A[n]$ is zero or negative, then it is best to exclude element $A[n]$, so we make the prefix sum $N_n = 0$.

$$\boxed{N_i \leftarrow \max_{\text{2 options}} \{0, \quad N_{i-1} + A[i]\ \} \quad \text{for} \quad i = 1, \dots n} \tag{40}$$

The actual value we were looking for, M_n, is then simply the maximum value from among the pre-fix sums:

$$\boxed{M_n \leftarrow \max_{j=1\dots n} \ \{ \ N_j \ \}} \tag{41}$$

Actual programs based on this algorithm can get the iteration started by either using using a **for-loop** that runs **from** $i = 1$ **to** n **do**␣ ... ␣**end** and which uses $M_0 = 0$ and $N_0 = 0$ as a sentinel values, or by considering $i = 1$ as a special boundary case, and start the **for-loop** with one index higher: **from** $i = 2$ **to** n **do**␣ ... ␣**end**. Also, if only M_n is needed, and no tracking information is required, then you need only two variables that can be used over and over again. This is finally the tricky program that is often proudly presented. It is indeed correct, and it is indeed an iterative implementation of a dynamic program, and it runs in linear time.

The same problem can also be solved using divide-and-conquer: Find $M_{\text{left-half}}$, find $M_{\text{right-half}}$, find N_{middle} and post-fix sum P_{middle} (these last can be done in linear time), and return the $\max$ of three choices: $M_{\text{left-half}}$, $M_{\text{right-half}}$, or $N_{\text{middle}} + P_{\text{middle}}$. This entire algorithm can be shown to run in time $n \ \lg n$ with recurrence relation is $T(n) = 2\,T(n/2) + 2n$.