Dynamic Programming

Anna Brandenberger

February 2, 2020

This is the augmented transcript of lectures given by Luc Devroye on the week of the 25th of January 2018 for the Honours Data Structures and Algorithms class (COMP 252, McGill University). The subject was Dynamic Programming.

The Principle: in dynamic programming, to find a solution of a problem of a given size, we solve all the necessary sub-problems.

1 *Binomial Coefficient*

We would like to compute the binomial coefficient defined as

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$$

Direct computation: if $k < n/2$, this can be done in $2k$ multiplications. So we have RAM model complexity $\Theta(\min(k, n-k))$, since one of k or $n-k$ will be $\leq n/2$ and $\binom{n}{k} = \binom{n}{n-k}$.

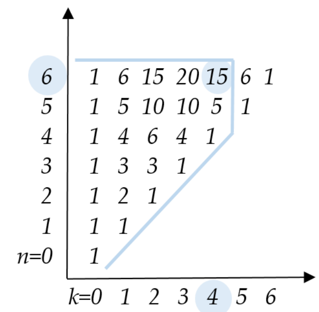


Figure 1: To compute $\binom{6}{4}$, compute all entries in the matrix below this element.

Recurrence relation: the binomial coefficient $\binom{n}{k}$ is the number of ways of choosing k out of n integers. Recall the recursive formula:[1]

$$\binom{n}{0} = \binom{n}{n} = 1$$
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

[1] Why is this true?

Proof. Fix some integer a. It is either part of the k or not.

- If the set of k numbers contains a, then for the remaining we must pick $k-1$ out of $n-1$ integers.
- If the set doesn't contain a, we must pick k out of $n-1$ integers. □

Using this, we can form a Pascal Triangle as shown in Figure 1. The following algorithm will, for a given element (N, K), compute the matrix below that element, obtaining $\binom{N}{K}$.

Compute-Binomial-Coefficient(N, K)

```
for n = 0 to N            // rows
    for k = 0 to K        // columns
        if k = 0 or k = n then (n k) = 1
        else (n k) = (n-1 k-1) + (n-1 k)
```

This algorithm has time complexity $O(NK)$.

Exercise 1. Improve the code to get complexity $O(K \cdot (N-K))$.
Hint: compute only a strip as shown in Figure 2.

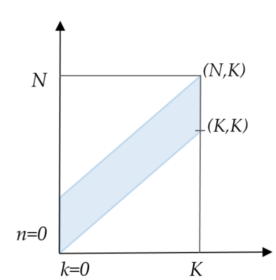


Figure 2: Hint for Exercise 1.

2 *Partitions of $\{1,\ldots,n\}$ into k non-empty sets*

Given $\{1,\ldots,n\}$, we want to compute the number of possible partitions of these integers into k non-empty sets. Indeed, observe the recursion:

$$P(\{1,\ldots,n\},k) = \underbrace{P(\{1,\ldots,n-1\},k)\cdot k}_{\text{element } n \text{ joins an existing set}} + \underbrace{P(\{1,\ldots,n-1\},k-1)}_{\text{starts a new set by itself}}$$

$$P_{n,k} = k \cdot P_{n-1,k} + P_{n-1,k-1}$$

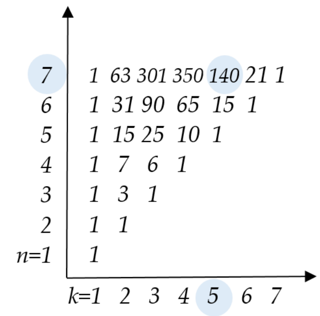


Figure 3: We compute, for example, 140 via: $140 = 5 \times 15 + 65$

We see that we can use exactly the same matrix-filling algorithm as the last section with only minor changes. Note that here, in the initialization, $P_{n,n} = P_{n,1} = 1$.

The complexity is therefore the same as Section 1: $O(nk)$.

3 *Travelling Salesman Problem (TSP)*

Input: Matrix of distances $dist[i,j]$ between all cities $1 \le i,j \le n$.

Objective: Find the tour through all cities of smallest total length.

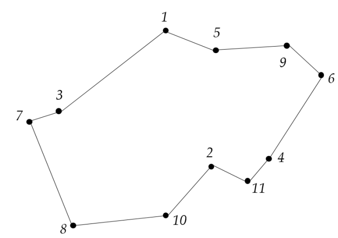


Figure 4: Example of a tour through $n = 11$ cities.

3.1 *Naive Algorithm*

Consider all $(n-1)!$ permutations of $\{2,\ldots,n\}$ and compute the lengths of all tours that start and end at "1".

With this approach, we obtain complexity $T(n) = n \times (n-1)! = n!$ where n comes from summing the lengths and $(n-1)!$ is the number of tours.

3.2 *Dynamic Programming Approach: Finding $L[1,S,j]$*

Definition 2. Consider $L(1,S,j)$, the length of the shortest path between 1 and j via all of S, where $S \subseteq \{1,\ldots,n\}$ is the set of all cities with 1 and j removed, i.e., $S = \{1,\ldots,n\} - \{1\} - \{j\}$.

In this algorithm, we will store $L[1,S,j]$ for all j and subsets S in a large matrix. Figure 5 illustrates line 5 of the algorithm. Once this matrix is found, computing the TSP tour will only require $\Theta(n)$ time.

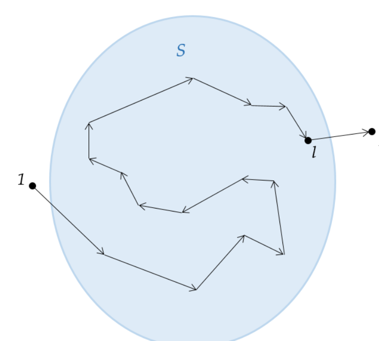


Figure 5: Illustration of innermost algorithm loop: recall that

$$S = \{1,\ldots,n\} - \{1\} - \{j\}.$$

The length of the path between city 1 and city j is equal to the length of the shortest path between 1 and some city $\ell \in S$, plus the distance between ℓ and j. To find the *shortest* path between 1 and j, we must choose the city ℓ which minimizes this quantity.

TSP-DP-Algorithm($dist[i,j]\ \forall i,j$)

1 **for** all $j \neq 1$: $L[1,\emptyset,j] = dist[1,j]$ // initialization
2 **for** $k = 1$ **to** $n-2$ // k: size of S
3 **for** all S with $|S| = k$, $S \subseteq \{2,\ldots,n\}$
4 **for** all $j \notin S$
5 $L[1,S,j] = \min_{\ell \in S} \left(L[1,S-\{\ell\},\ell] + dist[\ell,j] \right)$

To analyze the time complexity, we must consider three parts:

- all subsets of S are considered at most once:[2] contribution $\leq 2^n$
- for every set S, we consider at most n values of j: contribution $\leq n$
- for each (S, j) pair, we calculate a minimum over at most n choices of ℓ: contribution $\leq n$

[2] Recall that a set of size n has 2^n subsets

So the total complexity of the dynamic programming algorithm to find L, the length of the shortest path between 1 and any other city j, is $T(n) \leq n^2 \cdot 2^n$ in the RAM model. Note that this is $\ll n!$. Storage of order $\Theta(n \cdot 2^n)$ is needed.

3.3 *Finding the TSP Tour*

Once we have $L[1, S, j]$ for all $S \subseteq \{1, \ldots, n\}$ and $j \in S$, the length of the TSP tour is, as shown in Figure 6

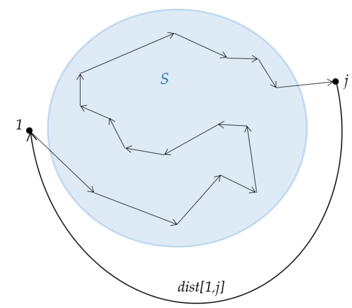


Figure 6: Completing the TSP tour.

$$TSPLen = \min_{j \neq 1} \left(L[1, S, j] + dist[1, j] \right)$$

where we can read all $L[1, S, j]$ off our table. The time complexity of this search (over $n - 1$ possibilities of j) is just $\Theta(n)$, which is added to the time needed to build L. Therefore the total algorithm time complexity remains $T(n) = O(n^2 2^n)$.

Exercise 3. Use additional storage so that you also output the optimal tour as a sequence of vertices.[3]

[3] HINT: think about pointers from j to the last vertex in S visited for $L[1, S, j]$.

4 *Knapsack Problem*

INPUT: Items of sizes $x_1, \ldots, x_N \in \mathbb{Z}$; and a knapsack of size $K \in \mathbb{Z}$.

OBJECTIVE: Determine if there exists a subset $S \subseteq \{1, \ldots, N\}$ for the input sizes such that $\sum_{i \in S} x_i = K$.

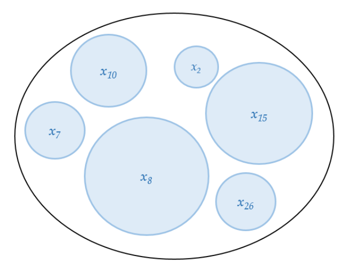


Figure 7: Example of a solution: a set of items x_i, $i \in S \subseteq \{1, \ldots, N\}$, which fill a knapsack of size K

Notation 4. Define the matrices $P[n, k]$ and $S[n, k]$ respectively as

$$P[n, k] = \begin{cases} 1 \text{ if } \textsc{Knapsack}\left(\{x_1, \ldots, x_n\}, k\right) \text{ has a solution} \\ 0 \text{ else} \end{cases}$$

$$S[n, k] = \begin{cases} 1 \text{ if } x_n \text{ belongs to a solution of } P[n, k] \\ 0 \text{ else} \end{cases}$$

such that P tells us, for a knapsack of capacity k, whether or not there exists a solution with elements up to element n; and S tells us, given a knapsack of capacity k, if we should select element n or not.

4.1 *Solving for possibility matrix $P[N, K]$*

To find the matrix P, we will be computing all values $P[i, j]$ for $i \leq n$ and $j \leq k$. As defined in the inputs, let the knapsack size be K and number of elements be N.

KNAPSACK-COMPUTE-$P(N, K)$

for $n = 0$ **to** N
  for $k = 0$ **to** K
    if $n = k = 0$ **then** $P[n, k] = 1$
    else if $n = 0, k > 0$ **then** $P[0, k] = 0$ // no elements
    else if $n > 0, k = 0$ **then** $P[n, 0] = 1$ // can choose the
      // empty set S to fill a knapsack of capacity 0
    else

$$P[n,k] = \begin{cases} P[n-1,k] & \text{if } x_n > k \\ 1 & \text{if } x_n = k \\ \max\Big(P[n-1,k], P[n-1,k-x_n]\Big) & \text{if } x_n < k \end{cases}$$

$x_n > k$: the element x_n is greater than the remaining capacity: no solution

$x_n = k$: there definitely exists a solution containing x_n

$x_n < k$: we either (1st option) don't put x_n into the set or (2nd option) put it in

This algorithm has complexity $T(N) = \Theta(NK)$.

4.2 *Computing a solution via $S[N, K]$*

We can easily modify the previous algorithm[4] (by simply adding a few lines) to also fill the matrix S, which tells us which items x_i, $i \in S \subseteq \{1, \ldots, N\}$ are used to fill the knapsack.

[4] The previous algorithm gave us whether it is possible to solve the knapsack problem for N elements $x_1, \ldots, x_N$ and a knapsack of size K.

KNAPSACK-ALSO-COMPUTE-$S(N, K)$

for $n = 0$ **to** N
  for $k = 0$ **to** K
    if $n = k = 0$ **then** $P[n, k] = 1$, $S[n, k] = 0$
    else if $n = 0, k > 0$ **then** $P[0, k] = 0$, $S[0, k] = 0$
    else if $n > 0, k = 0$ **then** $P[n, 0] = 1$, $S[n, 0] = 0$
      // we choose the empty set so no elements are selected
    else
      if $x_n > k$ **then** $P[n, k] = P[n-1, k]$, $S[n, k] = 0$
        // don't select this element
      else if $x_n = k$ **then** $P[n, k] = 1$, $S[n, k] = 1$
        // we add this element to the solution
      else if $x_n < k$
        $P[n, k] = \max\Big(P[n-1, k], P[n-1, k-x_n]\Big)$
        if $P[n, k] = 0$ **then** $S[n, k] = 0$
          // neither was possible
        else if $P[n-1, k-x_n] = 1$ **then** $S[n, k] = 1$
          // choosing to put in x_n worked
        else $S[n, k] = 0$ // solution without x_n worked

This will have the same time complexity as the previous algorithm.

Exercise 5. Write a program that also outputs a knapsack solution if it exists. Assume $P[N,K] = 1$ and all entries $P[n,k]$ and $S[n,k]$ for $n \le N$, $k \le K$ are known.

Solution (4).

$k \leftarrow K$
for $n = N$ **down to** 1
    if $S[n,k] = 1$
        output x_n
        $k \leftarrow k - x_n$

Exercise 6. Modify the dynamic program for the case that there is an unlimited supply of items of each of the sizes $x_1, \dots, x_n$.

5 *Assignment Problem*

INPUT: An $n \times n$ matrix, as shown in figure 8, which describes matches $\delta_{ij} \ge 0$.

OBJECTIVE: Find the permutation $(\sigma_1, \dots, \sigma_n)$ of $(1, \dots, n)$ that maximizes $\sum_{i=1}^{n} \delta_{i\sigma_j}$

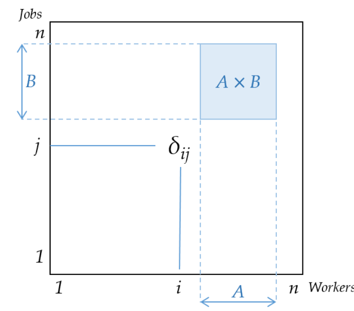


Figure 8: The assignment matrix can be thought of as matching n workers to n jobs. Entries δ_{ij} represent how well worker i and job j 'match'.

Naively, this can be done in time $O(n! \cdot n)$. We will use dynamic programming to reduce this, by computing sub-solutions for all submatrices $A \times B$ where $A, B \subseteq \{1, \dots, n\}$, as shown in Figure 8.

Definition 7. Denote the best assignment for this sub-matrix $A \times B$ as $\textsc{Best}[A,B]$. The goal is to compute $\textsc{Best}[A,B]$ for all sets A and B such that $|A| = |B| = k$, for k running from 1 to n.

FIND-BEST-ASSIGNMENT$(\delta_{ij}\ \forall i,j)$

for $k = 0$ **to** n
    for all sets $A, B \subseteq \{1, \dots, n\}$ with $|A| = |B| = k$ // of size k
        if $k = 0$ **then** $\textsc{Best}(\emptyset, \emptyset) = 0$
        else
            $\textsc{Best}[A,B] = \max\limits_{x \in A,\, y \in B} \left(\delta_{xy} + \textsc{Best}[A - \{x\}, B - \{y\}] \right)$

For the time complexity of this algorithm, we again consider different parts of the algorithm. We consider all sets A, B of size k, so since there are 2^k subsets of size k and we consider k running up to n, we can upper bound this by $2^n \cdot 2^n$. In the else loop, we compute the minimum over all $x \in A$ and $y \in B$ which both have size k: we can therefore upper bound this cost by n^2.

We therefore have total cost $T(n) \le 2^n \cdot 2^n \cdot n^2 = O(n^2 4^n)$.

6 *Job Scheduling*

INPUT: jobs $J_1, \dots, J_n$ requiring times $\tau_1, \dots, \tau_n$ to complete; and $c_i(t)$ = cost incurred if job i ends at time t.

OBJECTIVE: Find a permutation $(\sigma_1, \dots, \sigma_n)$ of $(1, \dots, n)$ such that the total cost is minimal if jobs are sequenced as job J_{σ_1} first, then job J_{σ_2}, etc.

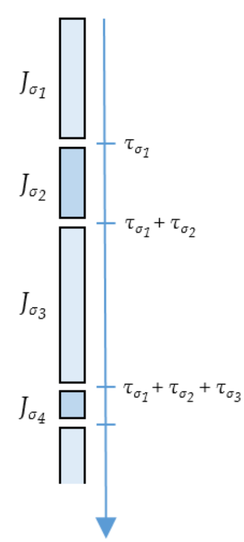


Figure 9: Job scheduling problem visualization

This total cost will be

$$Cost = c_{\sigma_1}(T_{\sigma_1}) + c_{\sigma_2}(T_{\sigma_1} + T_{\sigma_2}) + \cdots + c_{\sigma_n}(T_{\sigma_1} + \cdots + T_{\sigma_n})$$

For our dynamic programming algorithm, let $S \subseteq \{1, \ldots, n\}$ be a subset of the jobs and set $C(S)$ be the optimal cost for that subset.

JOB-SCHEDULING$(J_i, \tau_i, c_i(t)\ \forall i)$

for $k = 0$ **to** N
  for all $S \subseteq \{1, \ldots, n\}$ of size k do:
    $$C(s) = \min_{i \in S} \left(C(S - \{i\}) + c_i \left(\sum_{j \in S} \tau_j \right) \right)$$
    // i is the last job: find the one that minimizes total cost

$C(S - \{i\})$: cost of all jobs without job i

$c_i \left(\sum_{j \in S} \tau_j \right)$: cost of job i

By a similar analysis as in Section 5, this algorithm has complexity $T(n) = O(n \cdot 2^n)$.

7 *Longest Common Subsequence*

The next two sections (7 and 8) are adapted from Ruo Yu Tao and Sitong Chen's 2018 scribed notes[5], with a few adjustments.

[5] R. Y. Tao and S. Chen. *Dynamic Programming (2)*. McGill University, January 2018

INPUT: two ordered sequences: $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ where all elements x_i and y_i come from a finite alphabet A, (for example $\{0, 1\}$ or $\{A, C, G, T\}$).

OBJECTIVE: Find the longest common subsequence: that is, the longest sequences $1 \le i_1 < i_2 \cdots < i_k \le n$, $1 \le j_1 < \cdots < j_k \le m$ such that $x_{i_1} = y_{j_1}, \ldots, x_{i_k} = y_{j_k}$. See Figure 10 for an example.

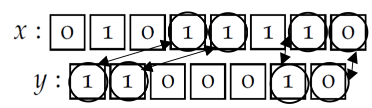


Figure 10: Example of a two sequences with longest common subsequence 1110.

Let the matrix element $L[i, j]$ be the length of the longest common subsequence of $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$. The following dynamic program will fill the matrix L.

The last element in the matrix L would be:

$$L[i,j] = \begin{cases} 1 + L[i-1, j-1] & \text{if } x_i = x_j, \\ \max(L[i-1, j], L[i, j-1]) & \text{if } x_i \neq x_j. \end{cases}$$

COMPUTE-LCS-LENGTH(n, m)

for all $i = 0$ **to** n
  for all $j = 0$ **to** m
    if $i = 0$ or $j = 0$ **then** $L[i, j] = 0$ // initialize
    else
      $$L[i,j] = \begin{cases} 1 + L[i-1, j-1] & \textbf{if} \quad x_i = x_j \\ \max\left(L[i-1, j], L[i, j-1]\right) & \textbf{if} \quad x_i \neq x_j \end{cases}$$

The entry $L[n, m]$ will be the length of the Longest Common Subsequence for the given input $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$. The construction of this matrix L takes time $\Theta(nm)$.

We can now define an algorithm that takes in the matrix defined above and returns the longest common subsequence:

Compute-LCS

let $i = n$, $j = m$, $r =$ empty list for results.
// We start at the cell of the last row of the last column.
while $i \geq 0$ and $j \geq 0$ // repeat this until out of matrix bounds
    if $x_i = y_j$
        append x_i to r.
        $i = i - 1, j = j - 1$ // go North West (NW) one cell
    else // else, if $x_i \neq y_j$, choose the maximum between the
        // the numbers in the West and North cells.
        if $L[i-1, j] \geq L[i, j-1]$ **then** $i = i - 1$
        else $j = j - 1$
return r

This algorithm is illustrated in Figure 7, by the circles and arrows.

Sequence x, with $x_i \in x, i = 1, \ldots, n$

Sequence y, with $y_j \in y, j = 1, \ldots, m$

	0	1	0	1	1	1	1	0
1	0	1	1	1	1	1	1	1
1	0	1	1	2	2	2	2	2
0	1	1	2	2	2	2	2	3
0	1	1	2	2	2	2	2	3
0	1	1	2	2	2	2	2	3
1	0	2	2	3	3	3	3	3
0	1	2	3	3	3	3	3	4

Figure 11: Example of two sequences with a longest common subsequence of length 4.

8 Optimal Binary Search Tree

Once again, this section is adapted from Ruo Yu Tao and Sitong Chen's 2018 scribed notes[6].

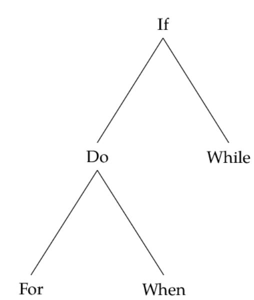
[6] R. Y. Tao and S. Chen. *Dynamic Programming (2)*. McGill University, January 2018

8.1 Background

Suppose that we are designing a compiler for a language, in which there are n syntactic keywords with corresponding semantics. For each occurrence of a keyword, we would want to perform a lookup operation by building a static binary search tree with n syntactic words as keys and their semantics as data stored in corresponding nodes. For the efficiency of the compiler, we would like to design a static binary search tree that minimizes total search time.[7]

[7]

Key words	Frequency(w)
If	w_1
Do	w_2
While	w_3
For	w_4
When	w_5
...	...
$Keyword_n$	w_n



We know that for a balanced tree, we can ensure an $O(\log n)$ search time per occurrence; however those syntactic words can appear with different frequencies. For example, if a frequently used word such as "if" is placed at the leave of this tree, it will greatly increase the total search time and hence the compiling time, vice versa. Therefore, given that we know the frequency of each key word appearing, we would like to organize a binary search tree in a way that minimizes the overall number of nodes visited. Such a tree is known as an optimal binary search tree. Moreover, it may be intuitive to consider a tree with smallest depth and key words of highest frequency at the root as an optimal binary search tree. However neither condition is necessary.[8]

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 1989

8.2 *Algorithm*

INPUT: sorted (key, weight) pairs $(k_1, w_1), \ldots (k_n, w_n)$, where the weights denote frequency or popularity.

OBJECTIVE: construct a binary search tree (BST) of minimal total weight $\sum_{i=1}^{n} d_i w_i$, where key k_i is at depth d_i.

Here $d_i = 1$ if k_i is the root.

Observe that any subtree of a binary search tree contains keys in a contiguous range $k_i \ldots k_j$, for some $1 \le i \le j \le n$. If an optimal binary search tree T has a subtree T' containing keys $k_i \ldots k_j$, then this subtree T' must be optimal as well for the subproblem with key $k_i \ldots k_j$. If there were a subtree T'' whose expected cost of searching is lower than that of T', then we could replace T' with T''.

Therefore, given a binary tree with keys $k_i \ldots k_j$, say k_r where $i \le r \le j$ is the root of an optimal subtree, then the left subtree contains keys $k_i \ldots k_{r-1}$, while the right subtree contains keys $k_{r+1} \ldots k_j$. If we check all possible candidate roots k_r, and identify the left and right subtree with minimum cost of searching, we are guaranteed to find an optimal binary search tree.

Definition 8.

- Let $C[i,j] = \sum_{k=i}^{j} w_k d_k$ denote the optimal cost for the tree containing $(k_i, w_i), \ldots, (k_j, w_j)$.
- Let $W[i,j] = \sum_{k=i}^{j} w_k$ denote the total weight of all keywords with indices $i, \ldots, j$.

We will compute $C[1,n]$ by computing all $C[i,j]$ for $1 \le i \le j \le n$.

Computing W

From Definition 8, we have that

$$W[i,j] = \begin{cases} w_i & \text{if } i = j, \\ W[i,j-1] + w_j & \text{if } i < j. \end{cases}$$

COMPUTE-$W((k_i, w_i)\ \forall i)$

```
for i = 1 to n: W[i,i] = w_i    // initialization
for k = 1 to n − 1
    for i = 1 to n
        if i + k ≤ n then W[i, i+k] = W[i, i+k−1] + w_{i+k}
return W
```

COMPUTE-W has time complexity $T(n) = \Theta(n^2)$.

Computing C

If $k_r, r \in [i,j]$ is the root of the optimal subtree for the tree containing k_i to k_j, we can consider subtrees as in Figure 8.2 where the left and right subtrees are both optimal and respectively contain $(k_i, w_i), \ldots, (k_{r-1}, w_{r-1})$ and $(k_{r+1}, w_{r+1}) \ldots, (k_j, w_j)$. Taking the root that yields minimum total cost thus yields the following formula

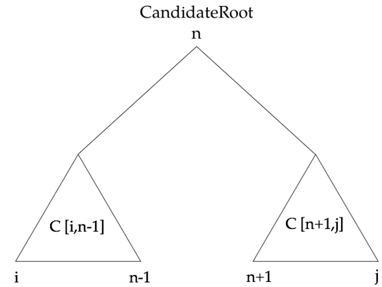


Figure 12: The above is an tree view of our recursive formula for computing total cost of searching given a keyword is chosen as the root.

$$C[i,j] = \min_{i \le r \le j} \left\{ \begin{array}{c} C[i, r-1] + W[i, r-1] \\ + C[r+1, j] + W[r+1, j] \\ + w_r \end{array} \right\}$$

which we can rewrite using $W[i,j] = W[i,r-1] + W[r+1,j] + w_r$, to

$$C[i,j] = \min_{i \le r \le j} \Big(C[i, r-1] + C[r+1, j] + W[i,j] \Big)$$

where we have $C[i,j] = 0$ if $i = j$.
Having computed the matrix W in time $\Theta(n^2)$, we can now find C:

Compute-C($(k_i, w_i)\ \forall i$)

```
for i = 1 to n : C[i,i] = w_i
for sizeofSubtree = 2 to n
    for i = 1 to n
        j = i + sizeofSubtree − 1
        if j ≤ n
            C[i,j] = min_{i≤r≤j} C[i,r−1] + C[r+1,j] + W[i,j]
            root[i,j] = one of the r's that minimizes C[i,j]
return C, root
```

Compute-C has time complexity $T(n) = \Theta(n^3)$, i.e., the algorithm takes time $\Theta(n^3)$ in total.

Remark 9. Knuth[9] has shown that there are always roots of an optimal subtree such that $root[i, j-1] \le root[i,j] \le root[i+1, j]$ for all $1 \le i < j \le n$. Hence we can reduce the running time of Compute-C to $\Theta(n^2)$ by replacing the innermost for loop `for` $r = i$ `to` j with `for` $r = root[i, j-1]$ `to` $root[i+1, j]$.

[9] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1998

9 *Matrix Multiplication*

Input: matrices $M_1, M_2, \ldots, M_n$ of dimensions $r_1 \times c_1, r_2 \times c_2, \ldots, r_n \times c_n$. r_i stands for the number of rows and c_i the number of columns. For the matrix multiplication to make sense, we require $c_1 = r_2, c_2 = r_3, \ldots, c_{n-1} = r_n$.

Objective: compute $M_1 \times M_2 \times \cdots \times M_n$ using standard matrix multiplication, such that the total number of operations is smallest in the RAM model.

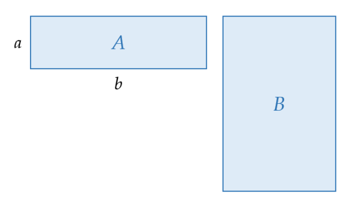


Figure 13: The number of operations needed to multiply matrices A and B of sizes $a \times b$ and $b \times c$ is $a \cdot b \cdot c$.

For the algorithm, we store the following subproblems:

Definition 10.

- Let $C[i, j]$ be the optimal number of operations for multiplying $M_i \times \cdots \times M_j$, $1 \leq i \leq j \leq n$.
- Let $B[i, j]$ be the index of best split when multiplying $M_i \times \cdots \times M_j$, say ℓ where $i \leq \ell \leq j$, so that we first do $M_i \times \cdots \times M_\ell$, then $M_{\ell+1} \times \cdots \times M_j$, and then $(M_i \times \cdots \times M_\ell) \times (M_{\ell+1} \times \cdots \times M_j)$. $B[i, j]$ is needed if we want to output the best schedule.

Matrix-Multiply(M_i $\forall i$)

1 **for** $i = 1$ **to** n **do** $C[i, i] = 0$ // initialization: multiplying no // matrices takes no operations

2 **for** $k = 1$ **to** $n - 1$ // iterate over the number of matrices to be

3 $\quad j = i + k$ // multiplied together

4 $\quad$ **if** $j \leq n$ **then** // j can't go over the number of matrices

5 $\quad\quad C[i, j] = \min\limits_{i \leq \ell \leq j} \left(C[i, \ell] + C[\ell + 1, j] + r_i r_{\ell+1} c_j \right)$

$\quad\quad$ // find the index of best split for $M_i \times \cdots \times M_j$

6 $\quad\quad B[i, j] = \arg\min\limits_{i \leq \ell \leq j} \left(C[i, \ell] + C[\ell + 1, j] + r_i r_{\ell+1} c_j \right)$

The term in red on line 5, $r_i r_{\ell+1} c_j$, comes from the fact that this line is splitting $M_i \times \cdots \times M_j$ into

$$\underbrace{(M_i \times \cdots \times M_\ell)}_{r_i \times r_{\ell+1} \text{ matrix}} \times \underbrace{(M_{\ell+1} \times \cdots \times M_j)}_{r_{\ell+1} \times c_j \text{ matrix}}$$

and counting the total number of operations needed to get the answer. $C[i, \ell]$ and $C[\ell + 1, j]$ respectively count the number of operations needed to perform $M_i \times \cdots \times M_\ell$ and $M_{\ell+1} \times \cdots \times M_j$. The middle multiplication requires $r_i r_{\ell+1} c_j$ operations, as explained in Figure 13.

Exercise 11. The tree view of this algorithm is shown in Figure 14. Given the $B[\cdot, \cdot]$ matrix, write an algorithm to construct this optimal tree.

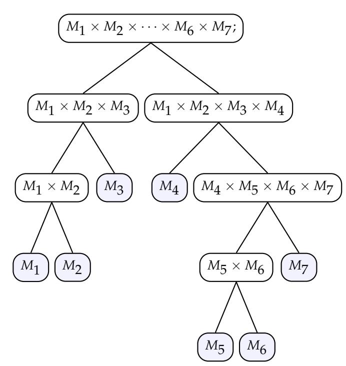


Figure 14: Tree view of the matrix multiplication algorithm.

References

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 1989.

D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1998.

R. Y. Tao and S. Chen. *Dynamic Programming (2)*. McGill University, January 2018.