CMPUT 498/501: Advanced Algorithms **Fall 2025**

Lecture 2 (Sep 8, 2025): Dynamic Programming

Lecturer: Mohammad R. Salavatipour *Scribe: Parsa Zarezadeh, Mohsen Mohammadi*

This lecture continues the discussion on Dynamic Programming (DP) by exploring several classic problems that can be solved efficiently using this technique.

2.1 Maximum Subarray Problem

Given a sequence of numbers $A[1], A[2], \ldots, A[n]$, find a contiguous subarray that has the maximum sum. In other words find the indexes i and j that $\underset{1 \le i \le j \le n}{\operatorname{argmax}} \sum_{k=i}^{j} A[k]$. To give an example, look at Figure 2.1:

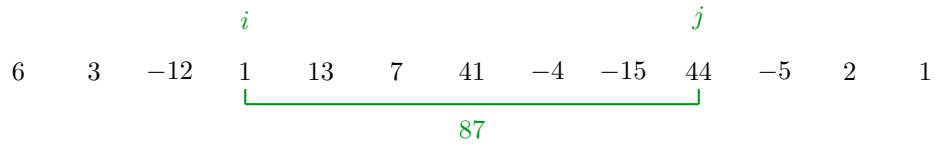


Figure 2.1: A diagram representing the maximum subarray and its sum.

2.1.1 Simple Approach

A naive approach would be to compute the sum for every possible pair of indices (i, j). This results in a time complexity of $\Theta(n^3)$. We can improve this by first computing prefix sums. For all $1 \le i \le n$, let $S[i] = \sum_{j=1}^{i} A[j]$. Then, for each pair (i, j), the sum of the subarray $A[i..j]$ can be found in $O(1)$ by computing $S[j] - S[i-1]$. Computing the prefix sum takes $\Theta(n)$, and the whole prefix sums technique improves the total time complexity to $\Theta(n^2)$.

2.1.2 Linear Time Solution (Kadane's Algorithm)

A more efficient DP solution exists that runs in linear time. It was developed by Jay Kadane in 1978.

Let $B[j]$ be the maximum sum of a subarray that ends at index j. The overall goal is to find $\max_j B[j]$. The recurrence relation is as follows:

$$B[j] = \begin{cases} A[1] & \text{if } j = 1 \\ \max\{A[j], A[j] + B[j-1]\} & \text{if } j > 1 \end{cases}$$

For the base case, $B[1]$, the only possible subarray ending at the first position is the element $A[1]$ itself. To calculate $B[j]$ for any index j, there are two possibilities for the maximum subarray ending at that position. The first is a new subarray starting at j, which consists of only the element $A[j]$. The second possibility is to extend the maximum subarray that ends at the previous index, $j-1$, by adding $A[j]$. The value of this extended subarray would be $A[j] + B[j-1]$. Therefore, $B[j]$ correctly finds the maximum subarray sum by taking the greater of these two options. From this recurrence, it is easy to derive that Kadane's algorithm runs in $\Theta(n)$ time, as it requires only a single pass through the array.

2.1.3 Extension to Matrices

The Maximum Submatrix problem is a direct extension of the Maximum Subarray problem. Given a matrix of numbers $A_{n\times n}$, the goal is to find a contiguous rectangular submatrix whose elements have the largest possible sum. To give an example, look at Figure 2.2:

$$A = \begin{bmatrix} 5 & 3 & -9 & -3 & 7 \\ -3 & 6 & -9 & -8 & 3 \\ 2 & -1 & 8 & 5 & -7 \\ -4 & 2 & -3 & 4 & 3 \\ -8 & 7 & 2 & 9 & -6 \end{bmatrix}$$

33

Figure 2.2: An example of 5×5 matrix and it's maximum submatrix sum

A naive brute-force algorithm would be to calculate the sum for every possible submatrix. In an $n \times n$ grid, there are $\Theta(n^4)$ possible submatrices, and this approach has a complexity of $\Theta(n^4)$. However, by building upon the solution to the previous problem, we can achieve a much more efficient algorithm with a time complexity of $\Theta(n^3)$.

To do this, we first implement a simple idea from the previous problem: prefix sums, but now applied to each row. For all $1 \le i, j \le n$, we define the row-prefix sum as $S_{i,j} = \sum_{k=1}^{j} A_{i,k}$. These values can be computed in $\Theta(n^2)$ time.

The next key idea is to reduce the 2D problem to a series of 1D problems. This is done by iterating through all possible pairs of left and right columns that could form the vertical boundaries of our optimal submatrix. For each fixed pair of columns j and j' (where $j \le j'$), we create a temporary 1D array, which we will call L. For each row i (where $1 \le i \le n$), the corresponding element in this array is calculated as $L_i = S_{i,j'} - S_{i,j-1}$.To give an example, look at Figure 2.3:

$$A = \begin{array}{c} \begin{matrix} & j & & j' & \end{matrix} \\ \begin{bmatrix} 5 & 3 & -9 & -3 & 7 \\ -3 & 6 & -9 & -8 & 3 \\ 2 & -1 & 8 & 5 & -7 \\ -4 & 2 & -3 & 4 & 3 \\ -8 & 7 & 2 & 9 & -6 \end{bmatrix} \end{array} \qquad L = \begin{bmatrix} -9 \\ -11 \\ 12 \\ 3 \\ 18 \end{bmatrix}$$

Figure 2.3: Fixing columns $j = 2$ and $j' = 4$ to create a temporary 1D array L of row sums.

Once the left and right column boundaries are fixed, finding the optimal submatrix for that selection becomes a maximum subarray problem on the temporary array L. This 1D problem can be solved efficiently using Kadane's algorithm. Since the overall algorithm iterates through all possible pairs of left and right boundaries, it is guaranteed to find the maximum submatrix sum.

The complexity analysis is as follows. Iterating over all possible pairs of left and right boundaries requires $\Theta(n^2)$ iterations. Within each iteration, generating the temporary array L and solving the maximum subarray problem on it both take $\Theta(n)$ time. Therefore, the total running time for the entire algorithm is $\Theta(n^3)$.

2.2 Segmented Least Squares

2.2.1 Single Straight Line

Given a set of n points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, sorted by their x-coordinates ($x_1 \leq x_2 \leq \cdots \leq x_n$), the initial problem is to find a single straight line that best fits these points. Let the line be L, defined by the equation $y = ax + b$, where a is the slope and b is the y-intercept. To find the best fit, we must choose the line L that minimizes the sum of the squared vertical distances from each point to the line (An example is given in Figure 2.4). This error metric is often called the Sum of Squared Distances (SSD). In other words, we want to find the line L that minimizes the following error function:

$$\text{SSD}(L, P) = \sum_{i=1}^{n} (y_i - (ax_i + b))^2$$

This is a classic calculus problem that can be solved by taking partial derivatives of the error function with respect to a and b and setting them to zero. The solution gives the following formulas for the optimal slope and y-intercept:

$$a = \frac{n(\sum x_i y_i) - (\sum x_i)(\sum y_i)}{n(\sum x_i^2) - (\sum x_i)^2} \qquad b = \frac{\sum y_i - a(\sum x_i)}{n}$$

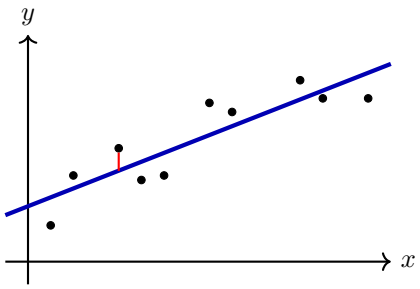


Figure 2.4: A set of points with a best-fit line. The red vertical segment shows the error for one point.

2.2.2 Multiple Straight Lines

A complication with the simple least squares method is that the data points may not lie on a single straight line, but rather on multiple consecutive line segments. The objective is to partition the points and find a sequence of lines that best fits the data. An example of this is shown in Figure 2.5.

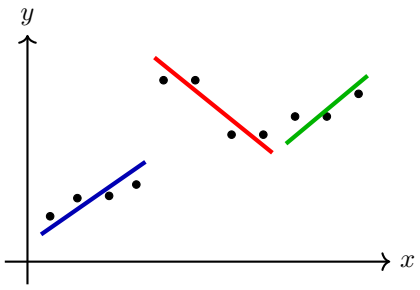


Figure 2.5: A set of points best described by three distinct line segments.

The function to optimize is a trade-off between the fit of the lines and the number of lines used. We want to minimize the total error, which is defined as: (Sum of SSD for all segments)$+C\times$(number of line segments), where C is a penalty factor for each new segment introduced.

A dynamic programming approach can solve this problem efficiently. To do so, we first define our subproblems. Let $Opt[j]$ be the minimum cost for the optimal solution covering points $P_1, \ldots, P_j$. The ultimate goal is to compute $Opt[n]$. For any set of points from P_i to P_j (where $1 \leq i \leq j \leq n$), let e_{ij} denote the minimum SSD for a single best-fit line through just those points. To compute $Opt[j]$, we consider all possible final segments that could end at point P_j. If the last segment in an optimal solution covering points $P_i, \ldots, P_j$, then the total cost of this solution is the cost for that specific segment $(e_{ij} + C)$ plus the optimal cost for all points before it, which is $Opt[i-1]$. We must find the minimum cost over all possible start points i for this final segment. This logic gives us the following recurrence relation:

$$Opt[j] = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j}\{e_{ij} + C + Opt[i-1]\} & \text{if } j > 0 \end{cases}$$

A naive implementation of this recurrence would take $O(n^3)$ time, as there are $O(n^2)$ pairs of (i, j) to consider, and calculating each e_{ij} can take $O(n)$ time. However, the e_{ij} values can be computed more efficiently, reducing the overall time complexity to $O(n^2)$.

2.3 Maximum Independent Set on Trees

Given a graph $G = (V, E)$ and a weight function on its vertices, $w : V \to \mathbb{R}^+$, the Maximum Weighted Independent Set problem is to find a subset of vertices $V' \subseteq V$ such that no two vertices in V' are adjacent, and the sum of the weights of the vertices in V' is maximized. This problem is extremely difficult on general graphs, but can be solved efficiently with dynamic programming if the graph is a tree. An example is shown in Figure 2.6:

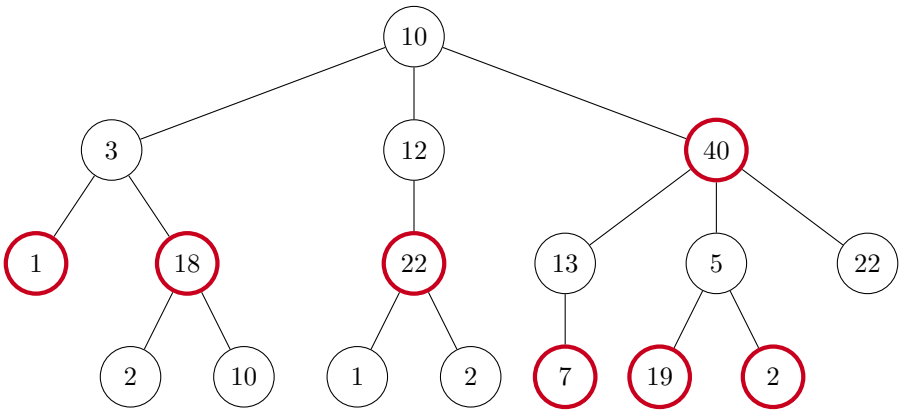


Figure 2.6: A weighted tree. The nodes circled in red form a maximum independent set.

To apply dynamic programming, we must first establish a structure for our subproblems. First, we root the given tree T at an arbitrary node. For each vertex $v \in V$, let C_v denote the set of its children, and let T_v be the subtree rooted at v. The subproblem is then defined as follows: let $M[v]$ be the maximum weight of an independent set within the subtree T_v. The recurrence relation for $M[v]$ is based on two cases:

1. If vertex v is not included in the independent set, then we are free to take the optimal solutions for all of its children. The total weight in this case is: $\sum_{u \in C_v} M[u]$.

2. If vertex v is included in the independent set, then none of its children can be included. Therefore, we take the weight of v, and add the sum of the optimal solutions for all of its grandchildren (where grandchildren are the children of v's children). The total weight is: $w_v + \sum_{u \in C_v} \sum_{u' \in C_u} M[u']$.

The value of $M[v]$ is the maximum of these two cases. The base case is for a leaf node v, where the only independent set is the node itself, so $M[v] = w_v$. This can be formulated more neatly as:

$$M[v] = \begin{cases} w_v & \text{if } v \text{ is a leaf} \\ \max\left\{\sum_{u \in C_v} M[u], \quad w_v + \sum_{u \in C_v} \sum_{u' \in C_u} M[u']\right\} & \text{otherwise} \end{cases}$$

To analyze the running time, it is easy to see that each subproblem $M[v]$ is computed only once. Since the calculation for each node involves summing values from its children and grandchildren, and each node appears as a child or grandchild at most twice, the work is amortized across the tree. This results in a total running time of $\Theta(n)$.

2.4 Sequence Alignment

Sequence alignment is a method used to measure the similarity between two strings, with common applications in spell checking and bioinformatics. The Edit Distance provides a quantitative measure of this similarity, defined as the minimum cost required to transform one string into another (by introducing gaps and mismatches). Before we dig deeper, let us see an example in Figure 2.7:

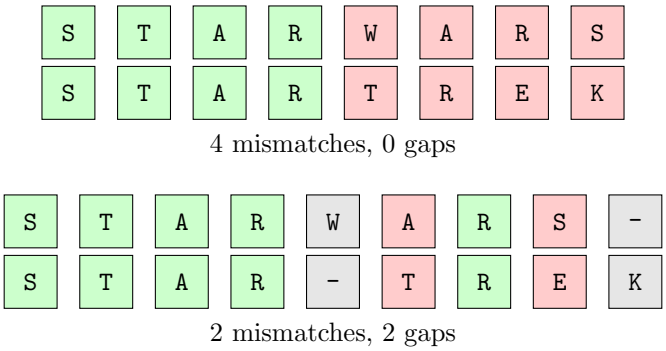


Figure 2.7: Two possible alignments for the strings STARWARS and STARTREK.

To define the edit distance formally, let's consider two strings, $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$, over some alphabet Σ. An alignment of these strings is a set M containing pairs of indices (i, j), which represents that character x_i is matched with character y_j. A valid alignment must satisfy two conditions:

1. Each character (represented by its index) can appear in at most one pair in M.
2. There are no crossings. This means that if we match x_i with y_j and $x_{i'}$ with $y_{j'}$, and we have $i < i'$, then it must also be that $j < j'$. It is not permitted to have $j' < j$, as illustrated in Figure 2.8:

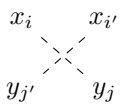
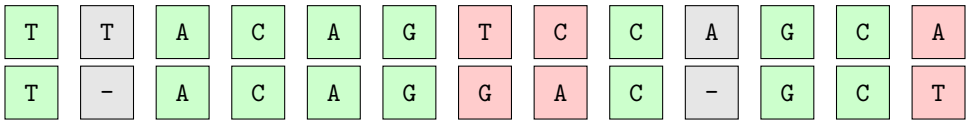


Figure 2.8: A crossing in a sequence alignment. Where $i < i'$, $j' < j$ and $(i, j), (i', j') \in M$

The cost of an alignment is determined by two types of operations. The first one is a mismatch, which means matching a character p with a different character q. This incurs a mismatch penalty, denoted α_{pq} (We consider that for all $p \in \Sigma$, $\alpha_{pp} = 0$, meaning that a match has no penalty). The second one is a gap that happens when a character is not matched to any other character. This incurs a fixed gap penalty, δ. The total cost of an alignment M, denoted $\text{Cost}(M)$, is the sum of the costs of all mismatches and all gaps. This is formally defined as:

$$\text{Cost}(M) = \sum_{(i,j)\in M} \alpha_{x_i y_j} + \sum_{x_i \text{ is unmatched}} \delta + \sum_{y_j \text{ is unmatched}} \delta$$

The goal of the sequence alignment problem is to find an alignment M that minimizes this total cost. To give an example of the best alignment, look at Figure 2.9:



$$\textbf{Cost} = 2\delta + \alpha_{TG} + \alpha_{CA} + \alpha_{AT}$$

Figure 2.9: Two DNA sequences and the best alignment for them

2.4.1 Dynamic Programing Approach

To solve this problem using dynamic programming, we first define our subproblems. Let $A[i, j]$ represent the minimum cost of aligning the prefix $X_i = x_1 \dots x_i$ with the prefix $Y_j = y_1 \dots y_j$. Our goal is to compute $A[m, n]$. To compute $A[i, j]$, we consider three possibilities for the alignment of the final characters:

1. x_i is matched with y_j: The cost is the penalty for matching x_i with y_j plus the cost of the best alignment for X_{i-1} and Y_{j-1}.
2. x_i is aligned with a gap: The cost is the penalty for a gap plus the cost of the best alignment for X_{i-1} and Y_j.
3. y_j is aligned with a gap: The cost is the penalty for a gap plus the cost of the best alignment for X_i and Y_{j-1}.

The value of $A[i, j]$ is the minimum of these three options. The base cases correspond to aligning a sequence with an empty string, which yields $A[i, 0] = i \cdot \delta$ and $A[0, j] = j \cdot \delta$. Therefore, the complete recurrence relation can be formulated as:

$$A[i, j] = \min \begin{cases} \alpha_{x_i y_j} + A[i-1, j-1] & \text{(match/mismatch)} \\ \delta + A[i-1, j] & \text{(gap in Y)} \\ \delta + A[i, j-1] & \text{(gap in X)} \end{cases}$$

This algorithm requires filling an $m \times n$ table, and each element of the table takes $\Theta(1)$ to compute, resulting in a time and space complexity of $\Theta(mn)$.

2.4.2 From Recurrence to Graph

The Sequence Alignment problem can be solved using dynamic programming by visualizing the solution space as a grid or a directed acyclic graph (DAG). We can construct an $(m + 1) \times (n + 1)$ grid where each node (i, j) represents the state of having aligned the prefix X_i with Y_j.

The goal is to find the minimum cost path from node $(0, 0)$ to node (m, n). Each possible move between adjacent nodes in the grid corresponds to one of the three alignment operations, and each move has an associated cost (or edge weight), as shown in Figure 2.10:

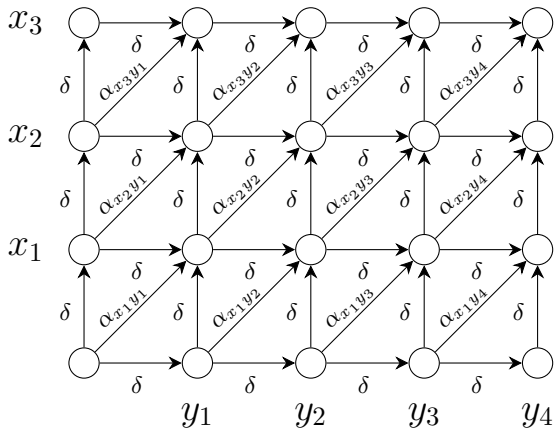


Figure 2.10: The grid representation for sequence alignment, in which the horizontal and vertical moves cost δ, while diagonal moves cost $\alpha_{x_i y_j}$.

2.4.3 Can We Improve the Complexity?

The short answer for improving the time complexity is "no", and it is because of the following theorem proved by Backurs and Indyk:

Theorem 1 ([BI15]) *If we can solve the Edit Distance problem in $O(n^{2-\epsilon})$ time for any constant $\epsilon > 0$, then we can also solve SAT instances with N variables and M clauses in $M^{O(1)}2^{(1-\delta)N}$ time for some constant $\delta > 0$.*

Such an algorithm for SAT would refute the Strong Exponential Time Hypothesis (SETH), which conjectures that no such algorithm exists. Therefore, under the assumption that SETH is true, we cannot solve the Edit Distance problem in truly sub-quadratic time. This makes the standard $\Theta(nm)$ dynamic programming solution essentially optimal.

But fortunately, the $\Theta(mn)$ space complexity can be improved. A clever idea combining dynamic programming with divide and conquer, known as Hirschberg's algorithm, can reduce the space requirement to $\Theta(m + n)$. To understand how, let's first consider the problem of finding only the value of the optimal alignment, not the alignment itself.

Observation 1 *To compute the value $f(i, j)$ in our DP grid, we only need the values from the previous row and column.*

From the observation mentioned, we can compute the entire grid, by only maintaining the current and previous column at each step. This requires only $\Theta(m)$ space, because at each step we only keep $f(i, j)$ for two columns at a time. However, this method only finds the value of the optimal path and does not store enough information to reconstruct the optimal alignment path. To find the alignment path within linear space, more ideas are needed.

Let $g(i, j)$ denote the length of the shortest path from node (i, j) to the end node (m, n). This is equivalent to the cost of the best alignment for the suffixes $x_i, \ldots, x_m$ and $y_j, \ldots, y_n$. A similar DP formulation (a backward DP) can be used to compute these $g(i, j)$ values.

Observation 2 *The length of the shortest path from* $(0,0)$ *to* (m,n) *that passes through an intermediate node* (i,j) *is exactly* $f(i,j)+g(i,j)$.

These observations lead to a divide-and-conquer strategy.

Lemma 1 *Let* k *be a column index, typically* $k \approx n/2$. *Let* q *be the row index that minimizes the sum* $f(q,k)+g(q,k)$ *over all possible row indexes. Then there exists a minimum-cost path from* $(0,0)$ *to* (m,n) *that passes through the node* (q,k).

The D&Q+DP algorithm is as follows: to find the path, we first identify this midpoint $(q, n/2)$. For all $0 \le i \le m$, we can compute all $f(i,n/2)$ values using the space-efficient forward DP and all $g(i,n/2)$ values using the space-efficient backward DP. We then find the index q that minimizes their sum. Once $(q,n/2)$ is known to be on an optimal path, we recursively find the optimal path from $(0,0)$ to $(q,n/2)$ and from $(q,n/2)$ to (m,n).

Lemma 2 *The space complexity of this algorithm is* $\Theta(m+n)$.

Proof. Each recursive call needs linear space to compute the $f(i,n/2)$ and $g(i,n/2)$ values for the relevant subproblem. The space required to store the coordinates of the midpoints found is also linear, as the number of recursive calls is $O(n)$. ■

Lemma 3 *The running time of this algorithm is* $\Theta(mn)$.

Proof. Let $T(m,n)$ be the time complexity. The work done at the top level involves two passes over a grid of size $m \times n/2$, which takes $c \cdot mn$ time for some constant c. This leads to a recurrence:

$$T(m,n) \le c \cdot mn + T(q,n/2) + T(m-q,n/2)$$

We can prove $T(m,n) \le k \cdot mn$ by strong induction for some constant k.

$$T(m,n) \le c \cdot mn + k \cdot q(n/2) + k(m-q)(n/2)$$

$$T(m,n) \le c \cdot mn + k \cdot m(n/2) = (c + k/2)mn$$

This is less than or equal to kmn if we choose $k \ge 2c$. Thus, the total time remains $\Theta(mn)$. ■

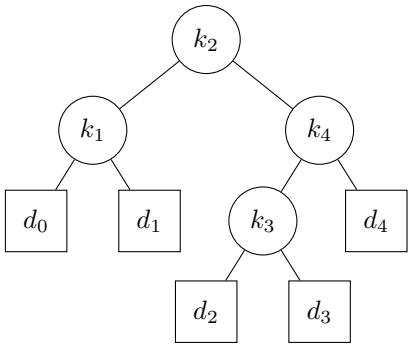
2.5 Optimal Binary Search Tree

Suppose we have a sequence of n distinct keys, sorted as $k_1 < k_2 < \cdots < k_n$, that we want to store in a Binary Search Tree (BST). Each key k_i has an associated probability or frequency p_i of being searched for. Additionally, we may search for values that are not in the set of keys. These are represented by $n+1$ "dummy keys," $d_0, d_1, \ldots, d_n$, where $d_0 < k_1 < d_1 < k_2 < d_2 < \cdots < k_n < d_n$. Each dummy key d_i has a search frequency q_i. The sum of all probabilities must be 1: $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$.

The expected cost of a search in a given tree T depends on the depth of the nodes. Let $d_T(x)$ be the depth of a node x (with the root at depth 0). The expected cost is:

$$E[\text{cost of } T] = \sum_{i=1}^{n}(d_T(k_i)+1)p_i + \sum_{i=0}^{n}(d_T(d_i)+1)q_i = 1 + \sum_{i=1}^{n} d_T(k_i)p_i + \sum_{i=0}^{n} d_T(d_i)q_i$$

The goal is to build a tree T that minimizes this expected search cost. An example is given in Figure 2.11:



i	0	1	2	3	4
p_i		0.18	0.22	0.03	0.12
q_i	0.01	0.11	0.09	0.05	0.19

Figure 2.11: An example of an Optimal Binary Search Tree with the corresponding frequencies for its keys and dummy keys shown in the table.

2.5.1 Dynamic Programming Approach

We can observe that, this problem exhibits optimal substructure, meaning that in an optimal BST a subtree T that contains keys $k_i, \ldots, k_j$ then this subtree optimum for those keys. Based on observation, we can use dynamic programming.

Let $O[i, j]$ be the minimum expected search cost for a BST over the keys $k_i, \ldots, k_j$. Let $f[i, j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$ be the sum of frequencies in this subproblem. If k_r is the root, the cost is:

$$O[i, j] = p_r + (O[i, r-1] + f[i, r-1]) + (O[r+1, j] + f[r+1, j]) = O[i, r-1] + O[r+1, j] + f[i, j]$$

Since we do not know the optimal root r, we must try all possibilities:

$$O[i, j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min_{i \le r \le j}\{O[i, r-1] + O[r+1, j] + f[i, j]\} & \text{if } j \ge i \end{cases}$$

The values for $f[i, j]$ can be pre-computed in $O(n^2)$ time. A DP algorithm can then fill the table for $O[i, j]$. A pseudo-code of the algorithm is given below:

Algorithm 1 Opt-BST-DP(p, q, n)

```
1:  for i ← 1 to n + 1 do
2:      O[i, i − 1] ← q_{i−1}
3:      f[i, i − 1] ← q_{i−1}
4:  for l ← 1 to n do
5:      for i ← 1 to n − l + 1 do
6:          j ← i + l − 1
7:          O[i, j] ← ∞
8:          f[i, j] ← f[i, j − 1] + p_j + q_j
9:          for  dor ← i to j
10:             temp ← O[i, r − 1] + O[r + 1, j] + f[i, j]
11:             if temp < O[i, j] then
12:                 O[i, j] ← temp
    return O[1, n]
```

A memoized, recursive version can also be implemented:

Algorithm 2 Memoized-Opt-BST(i, j)

```
1: if O[i, j] is already calculated then return O[i, j]
2: O[i, j] ← ∞
3: for r ← i to j do
4:     temp ← Memoized-Opt-BST(i, r − 1) + Memoized-Opt-BST(r + 1, j) + f[i, j]
5:     if temp < O[i, j] then
6:         O[i, j] ← temp
   return O[i, j]
```

Assuming the $f[i, j]$ values have been pre-computed, a step required for memoized versions, the time complexity of the presented algorithms is $\Theta(n^3)$. This naturally leads to the question of whether a faster approach is possible, a topic that will be explored in the following lecture.

References

[1] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, pages 51–58, 2015.