University of Zurich

Department of Informatics
Database Technology Group
Prof. Dr. Anton Dignös

Informatics II
Exercise 11

May 10, 2021

Goals:

- Solve dynamic programming problems on 2-D array.
- Practice writing programs with dynamic programming algorithms.
- Compare programs in dynamic programming and recursive approaches.

Dynamic programming

Task 1 Longest Palindromic Subsequence (LPS) Problem A palindromic sequence is a sequence that reads the same backwards as forwards, e.g., “racecar” or “lagerregal”. Given a string `A[]` with n characters, a subsequence is a sequence that can be derived from `A` by deleting some or no elements without changing the order of the remaining elements. Given a string `A[]`, LPS finds the length of the longest subsequence of `A` that is a palindrome. The idea for a dynamic programming solution is to compute an auxiliary 2-D matrix `L` with the dimension n × n. The value of element `L[i, j]` is the length of the longest LPS from the ith character to the jth character, both included. Note that $j \geq i$, and that the 0th character is the first character of `A`.

1. Determine the length of the LPS of the sequence “bbcab” using `L`.

   We use a 2-D matrix of the dimension 5×5 `L` in which `L[i, j]` is the length of the LPS from the ith characters to the jth characters, both included. The matrix `L` has the following two properties :

   - The `L[i, j]` has a value if $j \geq i$.
   - The `L[i, i]` $= 1$.

   Then we fill other items with three cases:

   - If $j = i+1$, then there are only two characters. If they are the same, then we let `L[i, j]` to be 2, otherwise we let it to be 1.
   - If the ith and jth characters are not the same, the LPS comes either from (1) the subarry from $i+1$ th character to j th character or (2) the subarry from ith character to $j-1$th character. For example, for the `L[0, 2]` item where i $= 0$, j $= 2$, since `A[0] = b` and `A[2] = c` are not the same, `L[0, 2]` will be the larger one of `L[0, 1]` and `L[1, 2]`, which is 2.
   - If the i th and jth characters are the same,`L[i, j]` will be the `L[i + 1, j - 1]` + 2. For example, for `L[0, 4]`, since `A[0] = b` and `A[4] = b` are the same, `L[0, 4]` = `L[1, 3]` + 2. In other words, we select the LPS the from $i+1$ th character to $j-1$ th character, and combine this LPS with ith and jth characters.

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

Hence, the length of its LPS will be 3.

2. Give a recursive solution to find out the length of the LPS of a given sequence `A`

There are 2 termination conditions:

- The length of the sequence is 1, then the length of its LPS is 1.
- The length of the sequence is 2 and the two characters in the sequence are the same. In this case, the length of its LPS is 2.

The recursion relations can be divided into two cases:

- If the first and last characters are the same, and by recursion we can calculate the LPS of a subsequence which removes the first and last characters. If the length of this LPS is k, then the length of LPS of the full sequence (with the first and last characters) will be $k + 2$. Formally, we have

  `LPS(A, i, j)` $=$ `LPS(A,i+1,j-1)+2`.

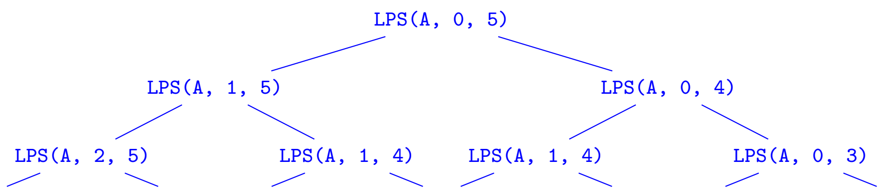
  where i and j specify the first and the last character.

- If the first and last characters are not the same, then the length of its LPS is the larger one of
  - `LPS(A, i+1, j)`.
  - `LPS(A, i, j-1)`.

```
int lps(char A[], int i, int j) {
  // If there is only one character.
  if (i == j) {
    return 1;
  }
  // If there are only two characters
  // and both are the same.
  if (i + 1 == j && A[i] == A[j]) {
    return 2;
  }
  // If the first and the last characters
  // are the same. Then the length of the LPS will be
  // lps(s, i+1, j−1) + 2
  if (A[i]==A[j]) {
      return lps(A, i+1, j−1)+2;
  }
  // Otherwise, the lps is the larger one of
  // lps(s, i+1) and lps(s, j−1)
  return max(lps(A, i+1, j), lps(A, i, j−1));
}
```

3. Is a dynamic programming solution more efficient than a recursive solution? Explain.

Yes, it will be more efficient. For example, if the given sequence is `abcdef`, then our recursion tree will be like:



In the above recursion tree, we notice that `LPS(A, 1, 4)` has been calculated twice. If we continue to draw the complete recursion tree, there will be more subproblems that are solved multiple times. Hence, the LPS problem has overlapping subproblems and we can avoid recomputations by using dynamic programming approach.

4. Provide a recursive problem formulation for determining `L[i, j]` that is the length of the longest LPS from from $i-$th character to $j-$th character, both included.

$$L(i,j) = \begin{cases} 1 & \texttt{i = j} \\ 2 & \texttt{j = i + 1, and A[i] = A[j]} \\ \texttt{L[i + 1, j - 1]} + 2 & \texttt{j > i + 1, and A[i] = A[j]} \\ max(\texttt{L[i + 1, j]}, \texttt{L[i, j - 1]}) & \texttt{A[i]} \neq \texttt{A[j]} \end{cases}$$

5. Implement the dynamic programming solution using `L` in C.

```c
int lps(char s[], int n) {
  int i, j;
  // n is the length of substring
  int k;
  int L[n][n];
  // The diagonal of L denotes the string of length 1
  // They are of course palindrome of length 1
  for (i = 0; i < n; i++) {
    L[i][i] = 1;
  }
  for (k = 2; k <=n; k++) {
    for (i = 0; i < n - k + 1; i++) {
      j = i + k - 1;
      if (s[i]==s[j] && k == 2) {
          L[i][j] = 2;
      } else if (s[i]==s[j]) {
          L[i][j] = L[i+1][j-1]+2;
      } else {
          L[i][j] = max(L[i][j-1], L[i+1][j]);
      }
    }
  }
  return L[0][n-1];
}
```

6. Analyse the time complexity of the program in (5).

The time complexity of this program is $O(n^2)$ where n is the length of the sequence. In the nested loop, the outer loop takes $O(n)$ time and the inner loop also takes $O(n)$ times. As a result, $O(n^2)$.

Task 2 Knapsack Problem Given n items, each with a weight `w[i]` and a value `v[i]`. Suppose we want to put some items in a knapsack of capacity W, i.e. the sum of weights of all the selected items must be less than or equal to W. Assume that each item can be used only once. Our goal is to maximise the value in the knapsack.

The idea is to create a two dimensional matrix `K` of $n+1$ rows and $W+1$ columns. As for `K[i][j]`, j means that the allocated weight in the knapsack is j, and i means that we can put first i items into the knapsack; `K[i][j]` means the maximum value that we can get by putting first i items with the allocated weight j. We can't put items in the knapsack if their weights exceeds the allocated weight.

1. Suppose we have a knapsack which can hold $W = 5$ and there are $n = 3$ items to choose from. The values of these items are `v={10,20,30}` and the weights of these items are `w={1,2,3}`. Determine the maximal value in the knapsack.

In the above example, we create a 4×6 matrix.

In this matrix, the first column means we do not select any item, so the entries in the first column will all be 0. Similarly, the first row means the weight of selected items is 0, so the entries in the first row will all be 0.

Then we fill other entries. For `K[i][j]`, there are two possibilities:

- If we cannot put the ith item to the knapsack, i.e. $w[i] > j$, then `K[i][j]=K[i-1][j]`.
- If we can put the ith item to the knapsack, then there are two possibilities:
  - If we do not put the ith item to the knapsack, then `K[i][j]=K[i-1][j]`.
  - If we put the ith item to the knapsack, then `K[i][j]=K[i-1][j-w[i]]+v[i]`. It means that we need to find the maximum value under the $j - w[i]$ capacity and add it with $v[i]$.
    We choose the larger `K[i][j]` between these two possibilities.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	10	10	10	10	10
2	0	10	20	30	30	30
3	0	10	20	30	40	50

Then in order to get the list of selected items, we start from the `K[n][W]`, which is the maximum value we could get. This value is either from `K[n-1][W]` or `K[n-1][j-w[n]]+v[n]`. If it is the latter one, then the nth item is selected. Then we remove the nth item from the knapsack and check `K[n-1][W-w[n]]`. After we remove the nth item, the maximum value becomes `K[n][W]-v[n]`.

2. Write a C program that determines which items should be put into the knapsack and the corresponding value of selected items. Your program should work for general cases as well (not only the first sub-task).

```c
void Knapsack(int n, int capacity, int w[], int v[]) {
  int max_v[n + 1][capacity + 1];
  int i, j;
  // the entries in first column and the first row are always 0
  for (i = 0; i <= n; i++) {
    max_v[i][0] = 0;
  }
  for (j = 0; j <= capacity; j++) {
    max_v[0][j] = 0;
  }
  for (i = 1; i <= n; i++) {
    for (j = 1; j <= capacity; j++) {
      if (w[i-1] > j) {
        max_v[i][j] = max_v[i - 1][j];
      } else {
        max_v[i][j] = max(max_v[i - 1][j - w[i-1]] + v[i-1], max_v[i - 1][j]);
      }
    }
  }
  printf("Max_value: %d\n", max_v[n][capacity]);
  int max_value = max_v[n][capacity];
  int selected[n];
  j = capacity;

  /*
  print the table
  for (i=0;i<=n;i++) {
    for (j=0;j<=capacity;j++) {
      printf("%d ", max_v[i][j]);
    }
    printf("\n");
  }
  */
  // Now finds the selected items
```

```
    for (i = n; i >=1; i--) {
        if (max_value==max_v[i-1][capacity]) {
            selected[i-1] = 0;
        } else {
            selected[i-1] = 1;
            max_value = max_value - v[i-1];
            capacity = capacity - w[i-1];
        }
    }
    printf("The selected items are (0 for not selected, 1 for selected): ");
    for (i = 0; i < n; i++) {
        printf("%d ", selected[i]);
    }
    printf("\n");
}
```

More exercises: dynamic programming tasks in the exercises and exams in the past.