Applied Algorithms

CSCI-B505 / INFO-I500

Lecture 11.

Dynamic Programming - II

M. Oguzhan Kulekci

- Dynamic Programming
    - Longest Increasing Sequence
    - Subset Sum
    - Ordered Partitioning

Dynamic Programming

- Find the minimum or maximum of a combinatorial challenge (combinatorial opt.)
- Exhaustive search guarantees the optimum, but very expensive
- Greedy approach (!) is more reasonable, but no guarantees (*in general*)
- Dynamic programming aims to compute the optimum with a good complexity by storing the results of some prior computations for the sake of some others later.
- **DP is particularly useful when there is a reductive solution but with significant overlaps between the recursive steps.**

Longest Increasing Subsequence in an Array

$$S = \langle\, 2, 4, 3, 5, 1, 7, 6, 9, 8 \,\rangle$$

- **Not** longest increasing **run**, but **subsequence**, e.g., (2,4) is a **run**, (2,5,7,9) is a **subsequence**
- What do we need to decide on the current position? How?
    1. The longest increasing subsequence length of the previous position
    2. The last element information
- If we define L_i as the length of the longest increasing run of $\langle s_1, s_2, \ldots, s_i \rangle$ **ending** at s_i then (2) is automatically included.

$$L_0 = 0 \qquad\qquad L_i = 1 + \max_{\substack{0 \le j < i \\ s_j < s_i}} L_j,$$

Longest Increasing Subsequence in an Array

Index i	1	2	3	4	5	6	7	8	9
Sequence s_i	2	4	3	5	1	7	6	9	8
Length L_i	1	2	2	3	1	4	4	5	5
Predecessor p_i	–	1	1	2	–	4	4	6	6

$$\begin{aligned} L_0 &= 0 \\ L_i &= 1 + \max_{\substack{0 \le j < i \\ s_j < s_i}} L_j, \end{aligned}$$

- Computing L_i needs to investigate all previous positions. If they were cached, it will be a linear operation. However, the total process is **quadratic** as we need this linear operation on all positions.

- Reporting the sequence beyond its length requires also maintaining the predecessor array, which marks the j value in the equation of L_i above.

Longest Increasing Subsequence in an Array

- A second solution of DP for this problem is something akin to **longest common subsequence**.
- Align the input sequence with its sorted version.

		1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0
2	0	0	1	1	1	1	1	1	1	1
4	0	0	1	1	2	2	2	2	2	2
3	0	0	1	2	2	2	2	2	2	2
5	0	0	1	2	2	3	3	3	3	3
1	0	1	1	2	2	3	3	3	3	3
7	0	1	1	2	2	3	3	4	4	4
6	0	1	1	2	2	3	4	4	4	4
9	0	1	1	2	2	3	4	4	4	5
8	0	1	1	2	2	3	4	4	5	**5**

Subset Sum (Unordered Partitioning)

Let $S = \{\ s_1,\ s_2,\ s_3,\ \ldots,\ s_n\ \}$ be a set of integers. Is there a subset of S, whose elements sum up to a queried value k ?

The number of subsets is 2^n. Therefore, the exhaustive search is exponential.

The problem is NP-complete.

We will be examining the **pseudo-polinomial time (?)** dynamic programming solution.

Subset Sum

Let $S = \{ s_1, s_2, s_3, \ldots, s_n \}$ be a set of integers. Is there a subset of S, whose elements sum up to a queried value k ?

- Let $T_{n,k}$ denote whether there is such a subset or not.
  - If there is a subset of $\{ s_1, s_2, s_3, \ldots, s_{n-1} \}$ summing up to k, which we can show with $T_{n-1,k}$, then $T_{n,k}$ is true
  - **OR**, if there is a subset of $\{ s_1, s_2, s_3, \ldots, s_{n-1} \}$summing up to $k - s_n$, which we can show with $T_{n-1,k-s_n}$, then $T_{n,k}$ is true.
- Therefore, $T_{n,k} = T_{n-1,k} \vee T_{n-1,k-s_n}$.

Subset Sum

$S = \{\ 1, 2, 4, 8\ \}, k = 11$

```
bool sum[MAXN+1][MAXSUM+1];       /* table of realizable sums */
int parent[MAXN+1][MAXSUM+1];     /* table of parent pointers */

bool subset_sum(int s[], int n, int k) {
    int i, j;                          /* counters */

    sum[0][0] = true;
    parent[0][0] = NIL;

    for (i = 1; i <= k; i++) {
        sum[0][i] = false;
        parent[0][i] = NIL;
    }

    for (i = 1; i <= n; i++) {         /* build table */
        for (j = 0; j <= k; j++) {
          sum[i][j] = sum[i-1][j];
          parent[i][j] = NIL;

          if ((j >= s[i-1]) && (sum[i-1][j-s[i-1]]==true)) {
          sum[i][j] = true;
          parent[i][j] = j-s[i-1];
        }
    }
  }

  return(sum[n][k]);
}
```

	i	s_i	0	1	2	3	4	5	6	7	8	9	10	11
$\{\}$	0	$\varnothing$	T	F	F	F	F	F	F	F	F	F	F	F
$\{1\}$	1	1	T	T	F	F	F	F	F	F	F	F	F	F
$\{1, 2\}$	2	2	T	T	T	T	F	F	F	F	F	F	F	F
$\{1, 2, 4\}$	3	4	T	T	T	T	T	T	T	T	F	F	F	F
$\{1, 2, 4, 8\}$	4	8	T	T	T	T	T	T	T	T	T	T	T	$\boxed{\text{T}}$

The matrix shows which sums are possible and which are not.

$\mathbf{T_{2,4}}$ is FALSE. $\{\mathbf{s_1, s_2}\} = \{\mathbf{1, 2}\}$ does not have a subset whose elements sum up to $\mathbf{4}$.

$\mathbf{T_{4,11}}$ is TRUE. $\{\mathbf{s_1, s_2, s_3, s_4}\} = \{\mathbf{1, 2, 4, 8}\}$ has a subset whose elements sum up to $\mathbf{11}$.

$T_{i-1,j}$

$T_{i-1,j-s_n}$

$$T_{i,j} = T_{i-1,j} \vee T_{i-1,j-s_n}$$

$\mathbf{T_{i,j}}$ means the set $\{\mathbf{s_1, s_2, .., s_i}\}$ has a subset whose elements sum up to $\mathbf{j}$.

If $\mathbf{T_{i,j}}$ is true, then how can we detect the elements of the set $\{s_1, s_2, .. s_i\}$ that are in the subset whose elements sum up to j.

Subset Sum

If $T_{i,j}$ is true, then how can we detect the elements of the set $\{s_1, s_2, ..s_i\}$ that are in the subset whose elements sum up to j.

$S = \{\ 1, 2, 4, 8\ \}, k = 11$

	i	s_i	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	∅	T	F	F	F	F	F	F	F	F	F	F	F
{1}	1	1	T	T	F	F	F	F	F	F	F	F	F	F
{1, 2}	2	2	T	T	T	T	F	F	F	F	F	F	F	F
{1, 2, 4}	3	4	T	T	T	T	T	T	T	T	F	F	F	F
{1, 2, 4, 8}	4	8	T	T	T	T	T	T	T	T	T	T	T	T

```
bool sum[MAXN+1][MAXSUM+1];      /* table of realizable sums */
int parent[MAXN+1][MAXSUM+1];    /* table of parent pointers */

bool subset_sum(int s[], int n, int k) {
    int i, j;                        /* counters */

    sum[0][0] = true;
    parent[0][0] = NIL;

    for (i = 1; i <= k; i++) {
        sum[0][i] = false;
        parent[0][i] = NIL;
    }

    for (i = 1; i <= n; i++) {     /* build table */
        for (j = 0; j <= k; j++) {
            sum[i][j] = sum[i-1][j];
            parent[i][j] = NIL;

            if ((j >= s[i-1]) && (sum[i-1][j-s[i-1]]==true)) {
            sum[i][j] = true;
            parent[i][j] = j-s[i-1];
        }
      }
    }

    return(sum[n][k]);
}
```

The default parent of all $T_{i,j}$ values is $T_{i-1,j}$, which is marked by -1 in the parent matrix.

If $T_{i,j}$ is true not because of $T_{i-1,j}$, but due to $T_{i-1,j-s_i}$, then save $j - s_i$ in the matrix.

i	s_i	0	1	2	3	4	5	6	7	8	9	10	11
0	0	**-1**	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	1	-1	**0**	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	2	-1	-1	0	**1**	-1	-1	-1	-1	-1	-1	-1	-1
3	4	-1	-1	-1	**-1**	0	1	2	3	-1	-1	-1	-1
4	8	-1	-1	-1	-1	-1	-1	-1	-1	0	1	2	**3**

The parent matrix that marks the parent of each cell.

$T_{4,11}$ is true not because of $T_{3,11}$ (which is false), but due to $T_{3,3}$. then save $3 = 11 - 8$ in the matrix.

In the parent matrix if a cell$[i, j]$ is not -1, then it means the corresponding element s_i is in the subset of the set $\{s_1, s_2, ..s_i\}$ whose element sum up to j.

Subset Sum

If $\mathrm{T}_{\mathrm{i,j}}$ is true, then how can we detect the elements of the set $\{s_1, s_2, .. s_i\}$ that are in the subset whose elements sum up to j.

This can be solved by backtracking in the parent matrix

i	s_i	0	1	2	3	4	5	6	7	8	9	10	11
0	0	**-1**	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	1	-1	**0**	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	2	-1	-1	0	**1**	-1	-1	-1	-1	-1	-1	-1	-1
3	4	-1	-1	-1	**-1**	0	1	2	3	-1	-1	-1	-1
4	8	-1	-1	-1	-1	-1	-1	-1	-1	0	1	2	**3**

```
void report_subset(int n, int k) {
   if (k == 0) {
        return;
   }

   if (parent[n][k] == NIL) {
        report_subset(n-1,k);
   }
   else {
        report_subset(n-1,parent[n][k]);
           printf(" %d ",k-parent[n][k]);
   }
}
```

If parent value is -1, then s_i is not in the solution subset and the subset we are looking for is actually a subset of the previous elements $\{s_1, s_2, .. s_{i-1}\}$. Therefore, move to the up cell and keep backtracking.

If parent value is NOT -1, then s_i IS in the solution subset and the subset we are looking for is actually $\{s_i\}$ union the subset of the previous elements $\{s_1, s_2, .. s_{i-1}\}$ that sum up to $k - s_i$. Therefore, move up to the previous row $(i-1)$and column $k - s_i$. keep backtracking.

Subset Sum

Does $S = \{\ 1, 2, 4, 8\ \}$ **has a subset having sum** $k = 11$ **? Yes, since** $T_{4,11}$ **is TRUE.**

i	s_i	0	1	2	3	4	5	6	7	8	9	10	11
0	0	T	F	F	F	F	F	F	F	F	F	F	F
1	1	T	T	F	F	F	F	F	F	F	F	F	F
2	2	T	T	T	T	F	F	F	F	F	F	F	F
3	4	T	T	T	T	T	T	T	T	F	F	F	F
4	8	T	T	T	T	T	T	T	T	T	T	T	$\boxed{T}$

Then, what are the elements of this subset?

i	s_i	0	1	2	3	4	5	6	7	8	9	10	11
0	0	$\boxed{-1}$	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	1	-1	$\boxed{0}$	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	2	-1	-1	0	$\boxed{1}$	-1	-1	-1	-1	-1	-1	-1	-1
3	4	-1	-1	-1	$\boxed{-1}$	0	1	2	3	-1	-1	-1	-1
4	8	-1	-1	-1	-1	-1	-1	-1	-1	0	1	2	$\boxed{3}$

$Subset = \{s_4 = 8, s_2 = 2, s_1 = 1\}$

Row 1, Column 1 is 0 in the parent matrix. So $s_1 = 1$ IS in the solution subset. We move to column 0(=1-s_1) on the previous row 0(=1-1). Once we reach column 0, we stop the backtracking.

$Subset = \{s_4 = 8, s_2 = 2\}$

Row 2, Column 3 is 1 in the parent matrix. So $s_2 = 2$ IS in the solution subset. We move to column 1(=3-s_2) on the previous row 1(=2-1).

$Subset = \{s_4 = 8\}$

Row 3, Column 3 is -1 in the parent matrix. So $s_3 = 4$ is NOT in the solution subset. We move to up cell at column 3 row 2(=3-1).

$Subset = \{s_4 = 8\}$

Row 4, Column 11 is 3 in the parent matrix. So $s_4 = 8$ IS in the solution subset. We move to column 3(=11-s_4) on the previous row 3(=4-1).

Subset Sum

Does $S = \{\ 1, 2, 4\}$ has a subset having sum $k = 6$? Yes, since $T_{3,6}$ is TRUE.

i	s_i	0	1	2	3	4	5	6	7	8	9	10	11
0	0	T	F	F	F	F	F	F	F	F	F	F	F
1	1	T	T	F	F	F	F	F	F	F	F	F	F
2	2	T	T	T	T	F	F	F	F	F	F	F	F
3	4	T	T	T	T	T	T	$\boxed{\text{T}}$	T	F	F	F	F
4	8	T	T	T	T	T	T	T	T	T	T	T	T

Then, what are the elements of this subset?

i	s_i	0	1	2	3	4	5	6	7	8	9	10	11
0	0	**-1**	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	1	$\boxed{-1}$	**0**	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	2	-1	-1	$\boxed{0}$	**1**	-1	-1	-1	-1	-1	-1	-1	-1
3	4	-1	-1	-1	**-1**	0	1	$\boxed{2}$	3	-1	-1	-1	-1
4	8	-1	-1	-1	-1	-1	-1	-1	-1	0	1	2	**3**

$Subset = \{s_3 = 4, s_2 = 2\}$

Once we have reached column 0, we stop the backtracking.

$Subset = \{s_3 = 4, s_2 = 2\}$

Row 2, Column 2 is 0 in the parent matrix. So $s_2 = 2$ IS in the solution subset. We move to column 0(=3-s_2) on the previous row 1(=2-1).

$Subset = \{s_3 = 4\}$

Row 3, Column 6 is 2in the parent matrix. So $s_3 = 4$ IS in the solution subset. We move to column 2(=6-s_3) on the previous row 2(=3-1).

Do we need to maintain two matrices, one for the true/false and the other for the parent information?

Not necessarily, since we can compute the parent value once we have the true false matrix and the given set elements.

Subset Sum

Question: What happens if we change the position of the elements in the input subset ? For instance, instead of $S = \{1,2,4,8\}$, what if we assume $S = \{2,1,8,4\}$ and query if there is a subset with sum 11?

We will fill the True/False matrix with the new order of elements in the set. Therefore, will have a different matrix, BUT the value at position $T_{4,11}$ will still evaluate to true, and the backtracking will give us the correct set of elements? On the other hand, notice that the elements in the matrix now will tell us different subset sum queries.

$$S = \{\ 1,\ 2,\ 4,\ 8\ \},\ k = 11$$

	i	s_i	0	1	2	3	4	5	6	7	8	9	10	11
$\{\}$	0	$\varnothing$	T	F	F	F	F	F	F	F	F	F	F	F
$\{1\}$	1	1	T	T	F	F	F	F	F	F	F	F	F	F
$\{1, 2\}$	2	2	T	T	T	T	F	F	F	F	F	F	F	F
$\{1, 2, 4\}$	3	4	T	T	T	T	T	T	T	T	F	F	F	F
$\{1, 2, 4, 8\}$	4	8	T	T	T	T	T	T	T	T	T	T	T	T

There is a subset of $\{1,2,4\}$ that sum up to 5

$$S = \{\ 2,\ 1,\ 8,\ 4\ \},\ k = 11$$

	i	s_i	0	1	2	3	4	5	6	7	8	9	10	11
$\{\}$	0	$\varnothing$	T	F	F	F	F	F	F	F	F	F	F	F
$\{2\}$	1	2	T	F	F	F	F	F	F	F	F	F	F	F
$\{2, 1\}$	2	1	T	T	T	T	F	F	F	F	F	F	F	F
$\{2, 1, 8\}$	3	8	T	T	T	T	F	F	F	F	T	T	T	T
$\{2, 1, 8, 4\}$	4	4	T	T	T	T	T	T	T	T	T	T	T	T

There is NO subset of $\{2,1,8\}$ that sum up to 5

Ordered Partitioning Problem

Given a **sorted** array of n positive integers as $\langle s_1, s_2, \ldots, s_n \rangle$, split this array into k partitions such that the sum of the integers in the partitions will be as balanced as possible, **which can be stated as the largest sum of integers in those partitions will be minimum.**

$k = 3$ $\quad S = \langle 100, 200, 300, 400, 500, 600, 700, 800, 900 \rangle$

300	2500	1700	Maximum is 2500
600	1500	2400	Maximum is 2400, so better than 2500

What are the best positions for the (k-1) dividers so that we get the most balanced solution.

Important notice: Rearrangement is not allowed !!!

Ordered Partitioning Problem

s_1 s_i | s_{i+1} s_n

The minimal cost of partitioning $\langle s_1, s_2, \ldots, s_i \rangle$ into $(k-1)$ pieces

A

$$\sum_{t=1+1}^{n} s_t$$

B

- The cost of placing the **(k-1)th** divider between i and $(i+1)$ is the maximum of A and B.
- Notice that A is indeed the same problem as partitioning $\langle s_1, s_2, \ldots, s_i \rangle$ into $(k-1)$ pieces
- If $M[n,k]$ is the minimum cost of partitioning $\langle s_1, s_2, \ldots, s_n \rangle$ into k pieces, then we can formulate

$$M[n,k] = \min_{i=1}^{n} \left(\max(M[i,k-1], \sum_{j=i+1}^{n} s_j) \right)$$

$$\boxed{\begin{aligned} M[1,k] &= s_1, \text{ for all } k > 0 \\ M[n,1] &= \sum_{i=1}^{n} s_i \end{aligned}}$$

Ordered Partitioning Problem

$$M[n,k] = \min_{i=1}^{n}\left(\max(M[i,k-1], \sum_{j=i+1}^{n} s_j)\right)$$

$S = \langle s_1, s_2, \ldots, s_9 \rangle = \langle 1,2,3,4,5,6,7,8,9 \rangle, \quad k = 3$

	1	2	3
1	1	1	1
2	3	2	2
3	6	3	3
4	10	6	4
5	15	9	6
6	21	11	9
7	28	15	11
8	36	21	15
9	45	24	17

$M[1,k] = s_1$

$$M[n,1] = \sum_{i=1}^{n} s_i$$

$$M[9,3] = min\Big(max(M[1,2], \sum_{j=2}^{9} s_j),$$

$$max(M[2,2], \sum_{j=3}^{9} s_j),$$

$$\vdots$$

$$max(M[7,2], \sum_{j=7}^{9} s_j),$$

$$\vdots$$

$$max(M[9,2], \sum_{j=10}^{9} s_j) \Big)$$

$max(15,8+9) = 17$

Means $\langle s_1, .. s_7 \rangle$ is divided into 2 with a cost 15, and $\langle s_8, s_9 \rangle$ is the last partition

Ordered Partitioning Problem

$S = \langle s_1, s_2, \ldots, s_9 \rangle = \langle 1,2,3,4,5,6,7,8,9 \rangle, \quad k = 3$

```
void partition(int s[], int n, int k) {
    int p[MAXN+1];              /* prefix sums array */
    int m[MAXN+1][MAXK+1];      /* DP table for values */
    int d[MAXN+1][MAXK+1];      /* DP table for dividers */
    int cost;                   /* test split cost */
    int i,j,x;                  /* counters */

    p[0] = 0;                   /* construct prefix sums */
    for (i = 1; i <= n; i++) {
        p[i] = p[i-1] + s[i];
    }

    for (i = 1; i <= n; i++) {
        m[i][1] = p[i];     /* initialize boundaries */
    }

    for (j = 1; j <= k; j++) {
        m[1][j] = s[1];
    }

    for (i = 2; i <= n; i++) {   /* evaluate main recurrence */
        for (j = 2; j <= k; j++) {
            m[i][j] = MAXINT;
            for (x = 1; x <= (i-1); x++) {
                cost = max(m[x][j-1], p[i]-p[x]);
                if (m[i][j] > cost) {
                    m[i][j] = cost;
                    d[i][j] = x;
                }
            }
        }
    }
    reconstruct_partition(s, d, n, k);    /* print book partition */
}
```

M	k		
s	1	2	3
1	1	1	1
2	3	2	2
3	6	3	3
4	10	6	4
5	15	9	6
6	21	11	9
7	28	15	11
8	36	21	15
9	45	24	17

D	k		
s	1	2	3
1	–	–	–
2	–	1	1
3	–	2	2
4	–	3	3
5	–	3	4
6	–	4	5
7	–	**5**	6
8	–	5	6
9	–	6	**7**

$O(kn^2)$-time, $O(kn)$-space,

Ordered Partitioning Problem

M	k				D	k		
s	1	2	3		s	1	2	3
1	1	1	1		1	–	–	–
2	3	2	2		2	–	1	1
3	6	3	3		3	–	2	2
4	10	6	4		4	–	3	3
5	15	9	6		5	–	3	4
6	21	11	9		6	–	4	5
7	28	15	11		7	–	5	6
8	36	21	15		8	–	5	6
9	45	24	17		9	–	6	7

$s_1, s_2, s_3, s_4, s_5 \;|\; s_6, s_7 \;|\; s_8, s_9$

```c
void reconstruct_partition(int s[],int d[MAXN+1][MAXK+1], int n, int k) {
    if (k == 1) {
        print_books(s, 1, n);
    } else {
        reconstruct_partition(s, d, d[n][k], k-1);
        print_books(s, d[n][k]+1, n);
    }
}


void print_books(int s[], int start, int end) {
    int i;    /* counter */

    printf("\{");
    for (i = start; i <= end; i++) {
        printf(" %d ", s[i]);
    }
    printf("}\n");
}
```

If we want to construct the partitions, we need to save the divider information for each cell, and then backtrack the optimum solution.

Reading assignment

- Read the Dynamic Programming chapters from the text books, particularly from Cormen and Skiena.