Algorithms and Complexity
Solutions to Assignment 1

Erwan Lemonnier

November 9, 2000

1 Lunch tickets

1.1 Terms

Indata: Amount of tickets n, ticket values $v_1, \ldots, v_n$, amount of lunches m and lunch prices $l_1, \ldots, l_m$. Values and prices are integers.

Utdata: "no" if you can't pay any lunch exactly with the tickets, otherwise (j, k) where j is the lunch that uses the more tickets and k the amount of tickets used.

Design and analyse an algorithm that solves this problem in polynomial time, assuming that $\forall\ (i, j)$, $v_i \leq n^2$ and $l_j \leq m^2$.

1.2 Problem Analysis

This problem possesses 2 characteristics that point out *Dynamic Programming* as the technique that should be used to solve it:

overlapping subproblems This can be shown by drawing the graph of the "brut" search on a simple example. In such a graph, all the combinations of elements of $v_1, \ldots, v_n$ are represented, and not only the optimal ones. Furthermore, depending on the way the graph is built, different leaves may correspond to the same combination.

optimal substructure The problem at rank n can be divided into 2 subproblems of rank $n-1$ by taking the same problem in which one of the lunch ticket is used, and the same problem in which this ticket is not used. The optimal solution of the main problem is the "best" of the two solutions of these subproblems.

Furthermore, this problem shows a resemblance with a *0/1 Knapsack* problem for which the 'value' is 1 for all the tickets, and the 'weight' is the value of the ticket. The difference is that instead of searching solutions under an upper bound, we want to reach an exact sum, which is one (or more) of the l_i. In its general terms, *0/1 Knapsack* is NP complete, but when searching only the optimal weights, it can be solved in polynomial time. The *lunch ticket* problem has similar terms, and its solution suggested here is indeed a variant of the *0/1 Knapsack*'s solution.

1.3 Solution

The previous *optimal substructure* property is the path to understanding the algorithm presented here, which uses *Dynamic Programming* to build all the solutions of the problem for $\{v_1\}$, then $\{v_1, v_2\}$, and so forth until $\{v_1, \ldots, v_n\}$ is reached. To design this algorithm, let us first express the problem in other terms. Let's define the sequence V_n and L_m, and the integer l_{max} as follows:

$$\begin{array}{rl} V_n = & \{v_1, \ldots, v_n\} \\ L_m = & \{l_1, \ldots, l_m\} \\ l_{max} : & l_{max} \in L_m \text{ and } \forall\ l_i \in L_m,\ l_i \leq l_{max} \end{array}$$

l_{max} is thus the maximum value of L_m. We also define $K_{l_{max}}^{V_n} = \{k_1^n, \ldots, k_{l_{max}}^n\}$ by: given V_n, k_i^n is the size of the largest subset of V_n for which the sum of its elements equals i. That way, k_i^n is simply the solution to the *lunch ticket* problem for only one lunch of price i and n tickets V_n.

Our new goal is to design an algorithm that, for each lunch price i between 1 and l_{max} and for a sequence of tickets V_n, gives $K_{l_{max}}^n$, the set of all the optimal solutions for lunches of price between 1 and l_{max}. We will therefore build a recurrence over the tickets of V_n by just listing some properties of the problem:

At first, when there is no ticket, the maximum amount of tickets that can be used to pay one lunch is 0, hence:

$$V_n = \emptyset \quad \Rightarrow \quad \forall\ i \in [1, l_{max}], \quad K_i^0 = 0$$

Then, given the solution of the problem for all lunch prices between 1 and l_{max}, using $n-1$ tickets $v_1, \ldots, v_{n-1}$, let's suppose we receive one more ticket v_n and want to update the solutions to take it into account. For each lunch price i, there are three possibilities:

- $v_n > i$: the new ticket worths more than the lunch and cannot be used to pay the lunch. We have to keep the previous best solution obtained with $v_1, \ldots, v_{n-1}$.
- $v_n = i$: we have two options. We can pay with only v_n, or with the best combination of tickets among $v_1, \ldots, v_{n-1}$ whose sum is i. We should use the option that uses the more tickets.
- $v_n < i$: we again have two options. We can pay with the best combination of tickets among $v_1, \ldots, v_{n-1}$ whose sum is i, or we can pay with v_n and the tickets corresponding to the best combination among $v_1, \ldots, v_{n-1}$ whose sum equals $i - v_n$.

These properties are summarized in the following recursive relation:

- $K_{l_{max}}^{V_0} = \{0, 0, \ldots, 0\}$
- $K_{l_{max}}^{V_n} = \{k_1^n, \ldots, k_{l_{max}}^n\} : \forall\ i \in [1, l_{max}]$

$$\begin{cases} k_i^n = \max(k_i^{n-1}, 1 + k_{i-v_n}^{n-1}) & \text{if } i - v_n > 0 \\ k_i^n = \max(k_i^{n-1}, 1) & \text{if } i = v_n \\ k_i^n = k_i^{n-1} & \text{otherwise} \end{cases}$$

```
Solve(V_n,L_m)
(1)     l_max ← max(L_m)
(2)     for j from 1 to l_max
(3)         k_j^0 ← 0

(4)     for i from 1 to n
(5)         for j from 1 to l_max
(6)             if j > v_i and k_{j-v_i}^{i-1} ≠ 0
(7)                 k_j^i ← max(k_j^{i-1}, 1 + k_{j-v_i}^{i-1})
(8)             else if j = v_i
(9)                 k_j^i ← max(1, k_j^{i-1})
(10)            else
(11)                k_j^i ← k_j^{i-1}

(12)    lunch ← 0,  tickets ← 0
(13)    foreach l ∈ L_m
(14)        if k_l^n > k
(15)            lunch ← l,  tickets ← k_l^n
(16)    if tickets = 0
(17)        return "nej"
(18)    else
(19)        return (lunch, tickets)
```

Figure 1: Pseudo-code for an algorithm solving the *lunch ticket* problem

To get the solution (j, k) of the *lunch ticket* problem, we now just have to look in $K^n_{l_{max}}$ for the values corresponding to $l_1, l_2, \ldots l_m$, and select the highest one as k, and its corresponding lunch price as l. **Figure 1** gives the pseudo-code for such an algorithm, which solves the *lunch ticket* problem and uses the previous recursive relations. An example of a *java* implementation of this pseudo-code is given in Annexe on page 7.

This algorithm could be improved by using only one array to store the values of $K^i_{l_{max}}$, instead of $n+1$. It could then avoid copying k^i_j from one array to the next, thus removing the last case of the *if* blocks. This improved algorithm is illustrated in the *java* program given in Annexe.

1.4 Algorithm analysis

Let's call $T(n, m)$ the running time of 'Solve' with the input V_n and L_m. Line (1) is executed in time $O(m)$ and lines (2) and (3) in time $O(l_{max})$. Lines (4) to (11) are executed in time $O(n.l_{max})$, and lines (12) to (19) in $O(m)$. Therefore,

$$T(n, m) = O(m) + O(l_{max}) + O(n.l_{max})$$

$$\{\forall\ j \in [1,m],\ \ l_j \leq m^2\} \quad \Rightarrow \quad l_{max} \leq m^2$$

so,

$$\begin{aligned} T(n,m) &= O(m) + O(m^2) + O(n.m^2) \\ &= O(n.m^2) \end{aligned}$$

Thus, if we consider the running time of 'Solve' with n as the variable, it runs in time $O(n)$, and with m as the variable, it runs in time $O(m^2)$.

2 Unimodal sequence

2.1 Terms

A sequence $< a_1, a_2, \ldots, a_n >$ is **unimodal** if $\exists\ t$ so that $< a_1, a_2, \ldots, a_t >$ is strictly increasing and $< a_t, \ldots, a_n >$ strictly decreasing. A **circular unimodal** sequence is a sequence obtained by rotation (modulo n) of an unimodal sequence. Build an algorithm running in time $O(\log n)$ or better that finds the highest value in a circular unimodal sequence.

2.2 Problem analysis

There are algorithms finding the highest value in an unimodal sequence in time $O(\log n)$. Their principle is to reduce recursively the interval of values of the sequence in which to search for the maximum. At each step, 2 elements of the sequence are examined, and depending on their value, all the elements before the first element or after the last element are removed. And so on until only one element is left, which must be the maximum. A similar method can be used to solve the problem for circular unimodal sequences.

2.3 Solution

A unimodal sequence can be symbolically represented on a graph by a linear function that grows then decreases, as show on **figure 2**. Although this representation is not adequate for a sequence, it reflects the properties of strict growth and decrease of the sequence's elements. Therefore, in our case, we can base our reasoning on this representation.

We will build an algorithm that takes a circular unimodal sequence, and returns a circular unimodal sequence of half the length of the input, and which still contains the maximum element of the input. When applying recursively this algorithm, we will finally obtain a sequence with only one element: the maximum.

The circular unimodal sequence resulting of the rotation of a unimodal sequence can be of two types: strictly increasing, decreasing then increasing (case 1 in figure 2), or strictly decreasing, increasing then decreasing (case 2 in figure 2).

Let's $A_n =< a_1, a_2, \ldots a_n >$ be a unimodal sequence, and $i = \frac{n}{4}$, $j = \frac{3n}{4}$ (if $\frac{n}{4}$ is not an integer, take the biggest integer $< \frac{n}{4}$ instead). We assume that $n \geq 4$. If $a_i > a_{i+1}$ it means A_n is decreasing at i, which we will write: $(A_n)_i \searrow$. Similarly, $a_i < a_{i+1} \Rightarrow\ (A_n)_i \nearrow$. Let's define the 2 following subsequences of A_n:

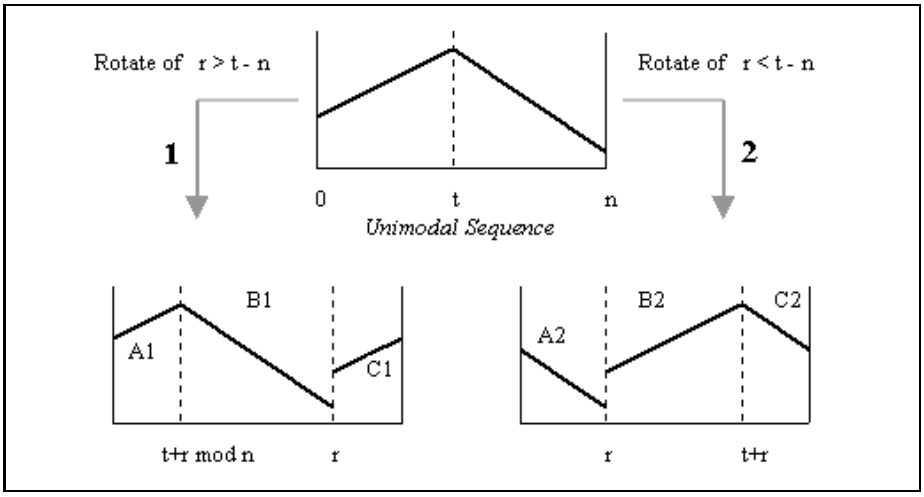

Figure 2: Two kind of circular unimodal sequences can be obtained by rotating a unimodal sequence.

$$A^{in}_{\frac{n}{2}} = \quad < a_i, a_{i+1}, \ldots, a_j >$$

$$A^{out}_{\frac{n}{2}} = \quad < a_0, \ldots, a_i, a_j, \ldots, a_n >$$

Depending on the values of a_i and a_j, and on whether $(A_n)_i$ increases or decreases, we can analyze 6 cases that define the algorithm we are looking for:

if $(A_n)_i \nearrow$ and $(A_n)_j \searrow$ **return** $A^{in}_{\frac{n}{2}}$

if $(A_n)_i \searrow$ and $(A_n)_j \nearrow$ **return** $A^{out}_{\frac{n}{2}}$

if $(A_n)_i \nearrow$ and $(A_n)_j \nearrow$
- **if** $a_i > a_j$ **return** $A^{out}_{\frac{n}{2}}$
- **if** $a_i < a_j$ **return** $A^{in}_{\frac{n}{2}}$

if $(A_n)_i \searrow$ and $(A_n)_j \searrow$
- **if** $a_i > a_j$ **return** $A^{in}_{\frac{n}{2}}$
- **if** $a_i < a_j$ **return** $A^{out}_{\frac{n}{2}}$

This algorithm returns a circular unimodal sequence of size $\frac{n}{2} \pm 1$ which contains the maximum element of A_n. We should prove this in each of the 6 cases, but since the demonstration is similar for all of them, we will only prove the first one. The demonstration is based on figure 2. a_{max} is the maximum in A_n.

$$(A_n)_i \nearrow \text{ and } (A_n)_j \searrow \quad \Rightarrow \quad \begin{cases} & a_i \in B1, \quad a_j \in C1 & \\ \text{or} & a_i \in B1, \quad a_j \in A1 & \rightarrow \text{ impossible since } i < j \\ \text{or} & a_i \in B2, \quad a_j \in C2 & \\ \text{or} & a_i \in D2, \quad a_j \in C2 & \rightarrow \text{ impossible since } i < j \end{cases}$$

$$\begin{array}{lll} a_i \in B1, \quad a_j \in C1 & \Rightarrow \; a_{max} \in < a_i, \ldots, a_j > & \textit{(see figure)} \\ a_i \in B2, \quad a_j \in C2 & \Rightarrow \; a_{max} \in < a_i, \ldots, a_j > & \textit{(see figure)} \end{array}$$

Finally, $< a_i, \ldots, a_j >$ is circular unimodal, since removing a subset of elements from a circular unimodal sequence does not affect its 'circular unimodality'.

We can now design an algorithm using this 6 case study to recursively divide by 2 the length of A_n. This recursion stops when only 3 elements are left, and returns the maximum of these 3 elements. The previous case study can not be made when there are 3 or less elements in the sequence, since we need 2 points around a_i and a_j to establish whether $(A_n)_i$ and $(A_n)_j$ increases or decreases. **Figure 3** gives the pseudo-code for the algorithm that returns the maximum of a circular unimodal sequence.

```
FindMax(< a_1, ..., a_n >)
(1)     if n ≤ 3
(2)         return max(a_1, ..., a_n)

(3)     i ← int(n/4), j ← int(3n/4)
(4)     if (a_i < a_{i+1}) and (a_j > a_{j+1})
(5)         return FindMax(< a_i, ..., a_j >)

(6)     if (a_i > a_{i+1}) and (a_j < a_{j+1})
(7)         return FindMax(< a_0, ..., a_i, a_j, ..., a_n >)

(8)     if (a_i < a_{i+1}) and (a_j < a_{j+1})
(9)         if a_i > a_j
(10)            return FindMax(< a_0, ..., a_i, a_j, ..., a_n >)
(11)        else
(12)            return FindMax(< a_i, ..., a_j >)

(13)    if a_i > a_j
(14)        return FindMax(< a_i, ..., a_j >)
(15)    else
(16)        return FindMax(< a_0, ..., a_i, a_j, ..., a_n >)
```

Figure 3: Pseudo-code for an algorithm returning the maximum element in a circular unimodal sequence

2.4 Algorithm analysis

Let's $T(n)$ be the running time of 'FindMax' with $< a_1, \ldots, a_n >$ in input. We assume $n > 3$. Since 'FindMax' calls itself recursively until the input has 3 elements, and that it divides by 2 the size of the input at each level of recursion, 'FindMax' calls itself k times where:

$$\text{int}(\frac{n}{2^k}) = 3$$

hence,

$$k = c.\log_2(n)$$

Since the time to execute lines (1) and (2), or to reach a call to 'FindMax' while executing the lines (3) to (16) is bounded by a constant independent of n, we have:

$$T(n) = O(\log n)$$

3 Annexe

The following program written in *java* is an example of implementation of the algorithm designed to solve the *lunch tickets* problem. It is optimized to use only 1 array and update only the necessary elements of the array when adding a new ticket.

```
public class alg1
{
    private static int[] ticket = {1, 5, 2, 4};     //the ticket values
    private static int[] lunch  = {8, 3, 7};        //the lunch prices

    private static int n = ticket.length;           //amount of tickets
    private static int m = lunch.length;            //amount of lunches

    private static int l_MAX;                       //the most expensive lunch

    //MAXs is an array containing at index i the size of the largest subset of
    //tickets for which there sum is equal to the lunch of price (i+1)
    private static int[] MAXs;


    public static void main(String[] arg)
    {
        //initialisation: compute l_MAX, build MAXs and fill it with zero

        l_MAX = 0;
        for(int i=0; i<lunch.length; i++)
        {
            l_MAX = Math.max(l_MAX, lunch[i]);
        }

        MAXs = new int[l_MAX];
```

```
        java.util.Arrays.fill(MAXs, 0);


        /* Build the set of the optimal solutions for all
         * lunch price between 1 and l_MAX
         */

        int value, sub;

        //add each 'ticket[i]' in turn to the set of solutions
        for(int i=0; i<n; i++)
        {
            value = ticket[i];

            //update the maximum sums that could be obtained without
            //ticket[i] to add ticket[i] to them
            for(int j=l_MAX-1; j>=value; j--)
            {
                sub = MAXs[j-value];
                if ( sub!=0 )
                    MAXs[j] = Math.max(sub+1, MAXs[j]);
            }

            MAXs[value-1] = Math.max(MAXs[value-1], 1);
        }


        //search the maximum in MAXs
        int j=0, k=0, l;

        for(int i=0; i<lunch.length; i++)
        {
            l = lunch[i];
            if (MAXs[l-1]>k)
            {
                j=l;
                k=MAXs[l-1];
            }
        }

        System.out.println("j= "+j+" k= "+k);
    }
}
```