Design and Analysis of Algorithm

M. Sakthi Balan

Department of Computer Science & Engineering
LNMIIT, Jaipur

Contents

Greedy Approach

- Used in optimization problems
- Builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion
- One can often design many different greedy algorithms for the same problem, each one locally, incrementally optimizing some different measure on its way to a solution
- Does not guarantee optimal solutions in general but in most cases it produces a solution very close to an optimal one

Greedy Approach

Type 1 *Stays ahead:* It always *stays ahead* of any other algorithm

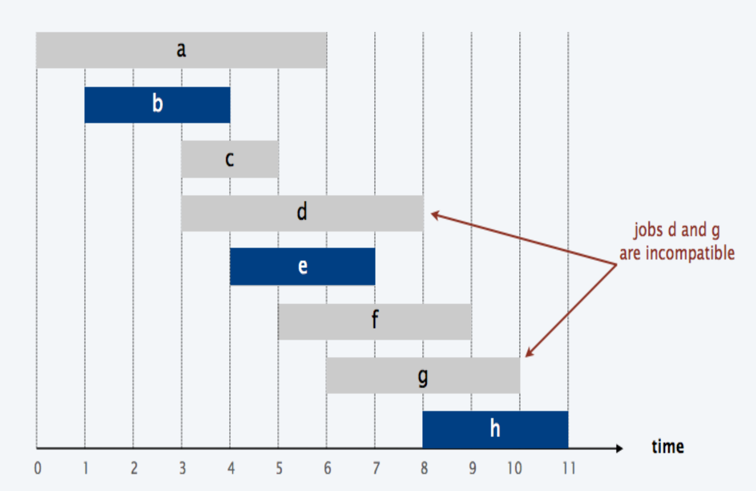
Type 2 *Exchange argument:* Considers any possible solution and slowly it transforms into an optimal one

Interval Scheduling

Problem

- Given n requests – $\{1, 2, \ldots, n\}$
- Each request i has a starting time $s(i)$ and finishing time $f(i)$
- Two requests are said to be compatible if their time does not overlap
- A set of requests is said to be compatible if no two pairs of request have time overlaps
- Problem is to find a subset of requests that is maximum (with respect to cardinality) and compatible

Interval Scheduling



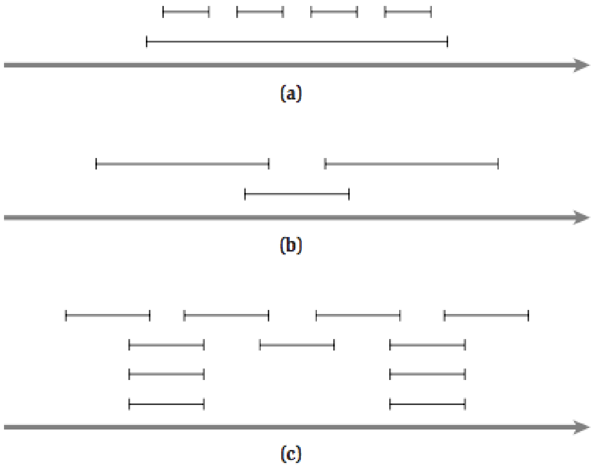
From Kleinberg and Eva Tardos

Interval Scheduling

Approaches

1. Select the available request that starts earliest, that is, the one with minimal start time $s(i)$
2. Accepting the request that requires the smallest interval of time, namely, the request for which $f(i) - s(i)$ is as small as possible
3. For each request, we count the number of other requests that are not compatible, and accept the request that has the fewest number of incompatible requests

Interval Scheduling



From Kleinberg and Eva Tardos

Interval Scheduling

Optimal Approach

- Accept first the request that finishes first, that is, the request i for which $f(i)$ is as small as possible.
- Now it seems natural! We ensure that our resource becomes free as soon as possible while still satisfying one request

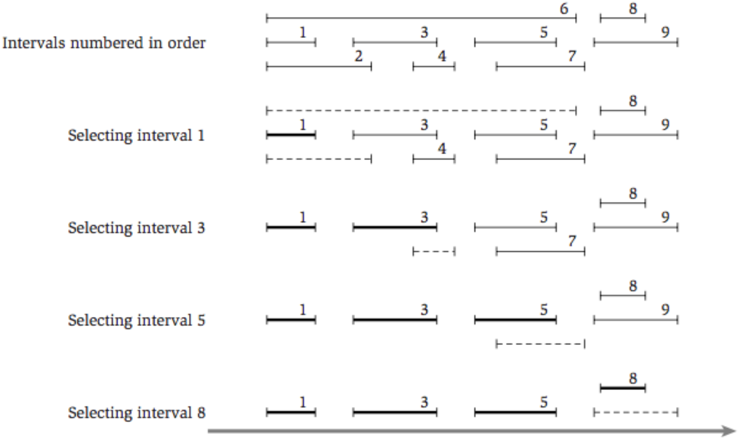
Interval Scheduling

```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request i∈R that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests
```

From Kleinberg and Eva Tardos

LNMIIT

Interval Scheduling



From Kleinberg and Eva Tardos

Interval Scheduling

Proof

- Suppose the optimal scheduling that our algorithm gives is $A = i_1, i_2 \ldots, i_k$
- Suppose there is another optimal scheduling that gives $\mathcal{O} = j_1, j_2 \ldots, j_m$
- We need to show $k = m$
- First we will show

$$\textit{For all indices } r \leq k \textit{ we have } f(i_r) \leq f(j_r)$$

Interval Scheduling

For all indices $r \leq k$ we have $f(i_r) \leq f(j_r)$
Prove by induction
$r = 1$ it is obvious
Assume the result for $r - 1$ and show for r

We can make this argument precise as follows. We know (since $\mathcal{O}$ consists of compatible intervals) that $f(j_{r-1}) \leq s(j_r)$. Combining this with the induction hypothesis $f(i_{r-1}) \leq f(j_{r-1})$, we get $f(i_{r-1}) \leq s(j_r)$. Thus the interval j_r is in the set R of available intervals at the time when the greedy algorithm selects i_r. The greedy algorithm selects the available interval with *smallest* finish time; since interval j_r is one of these available intervals, we have $f(i_r) \leq f(j_r)$. This completes the induction step. ■

Interval Scheduling

(4.3) *The greedy algorithm returns an optimal set A.*

Proof. We will prove the statement by contradiction. If A is not optimal, then an optimal set $\mathcal{O}$ must have more requests, that is, we must have $m > k$. Applying (4.2) with $r = k$, we get that $f(i_k) \leq f(j_k)$. Since $m > k$, there is a request j_{k+1} in $\mathcal{O}$. This request starts after request j_k ends, and hence after i_k ends. So after deleting all requests that are not compatible with requests $i_1, \ldots, i_k$, the set of possible requests R still contains j_{k+1}. But the greedy algorithm stops with request i_k, and it is only supposed to stop when R is empty—a contradiction. ∎

From Kleinberg and Eva Tardos

LNMIIT

Dynamic Programming

Dynamic programming. Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

fancy name for
caching away intermediate results
in a table for later reuse

From Kleinberg and Eva Tardos

Dynamic Programming

- First pioneered by Bellman (Mathematician) in 1950s
- Dynamic programming is nothing but planning over time
- Secretary of Defense was hostile to mathematical research and so Bellman sought an impressive name to avoid confrontation! (from Kleinberg and Eva Tardos)

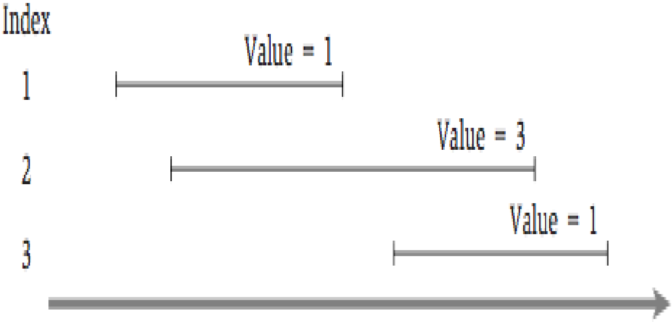
Weighted Interval Scheduling

Problem

- Given n requests – $\{1, 2, \ldots, n\}$
- Each request i has a starting time $s(i)$ and finishing time $f(i)$
- Each request has a value v_i associated with it
- Problem is to find a subset of requests S that has the maximum sum of weights and each pair of interval in S is compatible

Weighted Interval Scheduling

Same Greedy algorithm won't work!

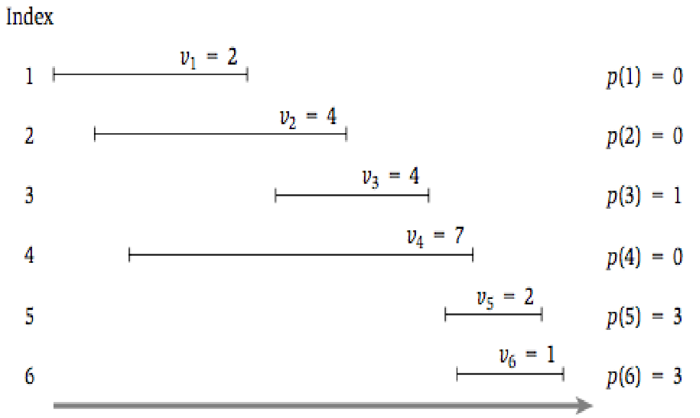


From Kleinberg and Eva Tardos

Weighted Interval Scheduling

- Suppose the intervals are sorted according to $f(i)$
- For an interval j calculate the value $p(j)$
- $p(j)$ is nothing but the largest i such that i is compatible with j

Weighted Interval Scheduling



From Kleinberg and Eva Tardos

Weighted Interval Scheduling

Suppose $\mathcal{O}$ is an optimal scheduling for the given instance of WIS problem

Case 1: $n \in \mathcal{O} \implies$ no intervals $p(n)+1, p(n)+2, \ldots, n-1$ can belong to $\mathcal{O}$

This means

$$OPT(n) = v_n + OPT(p(n))$$

optimal scheduling for the given instance of WIS problem

Case 2: $n \notin \mathcal{O} \implies$ means

$$OPT(n) = OPT(n-1)$$

This can be generalized as either

$$OPT(j) = v_n + OPT(p(j))$$

or

$$OPT(j) = OPT(j-1)$$

for any interval j

Now take

$$OPT(j) = Max(v_n + OPT(p(j)), OPT(j-1))$$

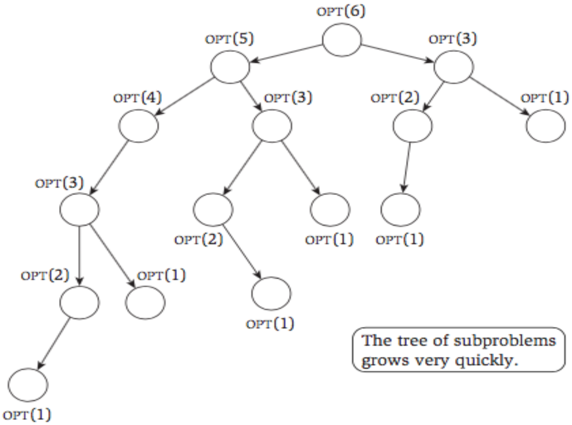
Weighted Interval Scheduling

```
Compute-Opt(j)
  If j=0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), Compute-Opt(j-1))
  Endif
```

The algorithm is not optimal! It computes many values again and again!

From Kleinberg and Eva Tardos

Weighted Interval Scheduling



From Kleinberg and Eva Tardos

Huffman Coding

- This was first proposed by Dr. David A. Huffman in 1952 through his paper on *A Method for the Construction of Minimum Redundancy Codes*
- Applications: data transmission, efficient storage of data and so on

Huffman Coding

Basic Idea

- All characters does not occur with same frequency
- But still all characters are given the same space
- Can we any savings in tailoring codes to frequency of character?
- Code word lengths can vary. It will be shorter for the more frequently used characters.

Huffman Coding

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If we assign each character a 3-bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

From Cormen et al

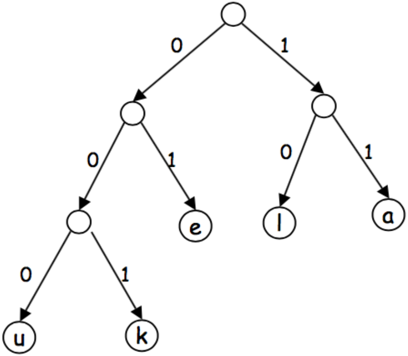
Huffman Coding

Prefix Codes

- A *prefix code* for a set S is a function c that maps each $x \in S$ to $1s$ and $0s$ in such a way that for $x, y \in S, x \neq y, c(x)$ is not a prefix of $c(y)$.
- For example:
  $c(a) = 11, c(e) = 01, c(k) = 001, c(l) = 10, c(u) = 000$. What is the meaning of 1001000001?

Huffman Coding

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000



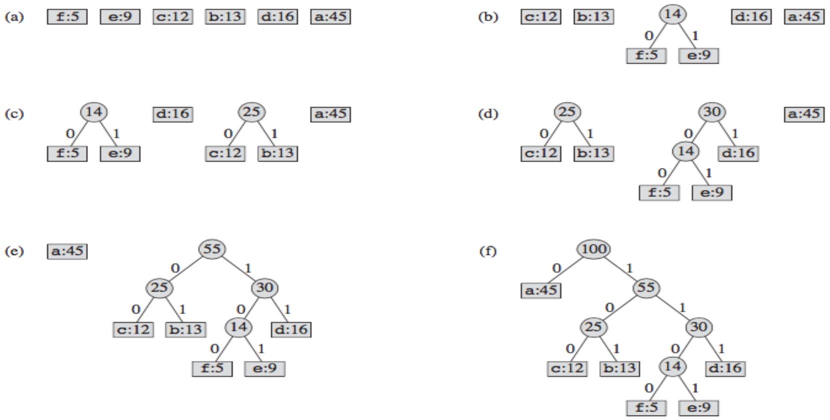
From Internet

Huffman Coding

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

From Cormen et al

Huffman Coding



From Cormen et al

Knapsack Problem

Knapsack Problem

A thief robbing a store finds n items. The i^{th} item is worth v_i dollars and weighs w_i kilogram, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W kilogram in his knapsack, for some integer W. Which items should he take?

Versions

- 0-1 Knapsack Problem
- Fractional Knapsack Problem

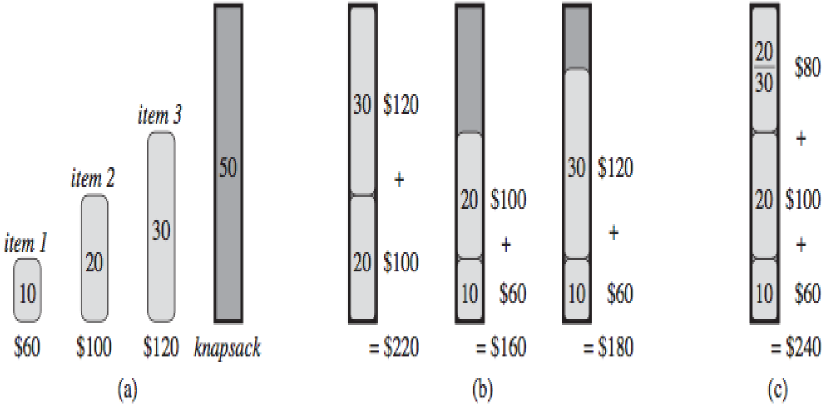
Knapsack Problem

- Greedy approach helps to solve fractional Knapsack
- For 0-1 knapsack greedy approach does not give an optimal solution

Fractional Knapsack Problem

- Find the f_i values where $f_i = v_i/w_i$
- Sort f_i in decreasing order
- Take the first item in the sorted list, say j, as much as possible
- For the remaining amount in the knaspsack repeat the same method as above with the remaining leftover things until the knapsack total weight becomes W

Knapsack Problem



From Cormen et al

0-1 Knapsack Problem

Problem

- There are n data files that we want to store
- We have available W bytes availability
- File i has size w_i bytes and takes v_i minutes to compute.
- We want to avoid recomputing as much as possible, so the idea is to find a subset of files to store such that the files have combined size at most W.
- The total computing time of the stored files is as large as possible.
- We cannot store parts of files, it is the whole file or nothing.
- How should we select the files in these circumstances?

0-1 Knapsack Problem

Problem

We are given two tuples of positive numbers:

$$(v_1, v_2, \ldots, v_n)$$

and

$$(w_1, w_2, \ldots, w_n)$$

with $W > 0$. The problem is to find a subset $S \subseteq \{1, 2, \ldots, n\}$ that maximises

$$\sum_{i \in T} v_i$$

subject to

$$\sum_{i \in T} w_i \leq W$$

0-1 Knapsack Problem

- Construct a table of values $V[0..n, 0..W]$
- $V[i, j]$ denotes maximum computing of files from $\{1, 2, \ldots, i\}$ with at most size j where $1 \leq i \leq n$ and $0 \leq j \leq W$
- $V[n, W]$ contains the maximum computing time of the files and that is the solution we are looking for

0-1 Knapsack Problem

Initial Settings: Set

$$
\begin{array}{lll}
V[0,w] = 0 & \text{for } 0 \le w \le W, & \text{no item} \\
V[i,w] = -\infty & \text{for } w < 0, & \text{illegal}
\end{array}
$$

Recursive Step: Use

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1, w-w_i])$$

for $1 \le i \le n$, $0 \le w \le W$.

0-1 Knapsack Problem

Lemma: For $1 \le i \le n$, $0 \le w \le W$,

$V[i,w] = \max(V[i-1,w], v_i + V[i-1,w-w_i])$.

Proof: To compute $V[i,w]$ we note that we have only two choices for file i:

Leave file i: The best we can do with files $\{1,2,\ldots,i-1\}$ and storage limit w is $V[i-1,w]$.

Take file i (only possible if $w_i \le w$): Then we gain v_i of computing time, but have spent w_i bytes of our storage. The best we can do with remaining files $\{1,2,\ldots,i-1\}$ and storage $(w-w_i)$ is $V[i-1,w-w_i]$.
Totally, we get $v_i + V[i-1,w-w_i]$.

Note that if $w_i > w$, then $v_i + V[i-1,w-w_i] = -\infty$ so the lemma is correct in any case.

0-1 Knapsack Problem

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

0-1 Knapsack Problem

The Dynamic Programming Algorithm

KnapSack(v, w, n, W)
{
  for ($w = 0$ to W) $V[0, w] = 0$;
  for ($i = 1$ to n)
    for ($w = 0$ to W)
      if ($w[i] \leq w$)
        $V[i, w] = \max\{V[i-1, w], v[i] + V[i-1, w - w[i]]\}$;
      else
        $V[i, w] = V[i-1, w]$;
  return $V[n, W]$;
}

Time complexity: Clearly, $O(nW)$.

0-1 Knapsack Problem

- Algorithm described does not find the subset of items that gives the optimal solution! How to find it?
- To compute the actual subset, we can add an auxiliary boolean array $keep[i, j]$ which is 1 if we decide to take the i^{th} file in $V[i, j]$ and 0 otherwise.

0-1 Knapsack Problem

- Algorithm described does not find the subset of items that gives the optimal solution! How to find it?
- To compute the actual subset, we can add an auxiliary boolean array $keep[i, j]$ which is 1 if we decide to take the i^{th} file in $V[i, j]$ and 0 otherwise.

0-1 Knapsack Problem

```
KnapSack(v, w, n, W)
{
    for (w = 0 to W) V[0, w] = 0;
    for (i = 1 to n)
        for (w = 0 to W)
            if ((w[i] ≤ w) and (v[i] + V[i − 1, w − w[i]] > V[i − 1, w]))
            {
                V[i, w] = v[i] + V[i − 1, w − w[i]];
                keep[i, w] = 1;
            }
            else
            {
                V[i, w] = V[i − 1, w];
                keep[i, w] = 0;
            }
    K = W;
    for (i = n downto 1)
        if (keep[i, K] == 1)
        {
            output i;
            K = K − w[i];
        }
    return V[n, W];
}
```

LCS Problem

Longest Common Subsequence Problem

We are given two sequences

$$X = \langle x_1, x_2 \ldots, x_m \rangle$$

and

$$Y = \langle y_1, y_2, \ldots, y_n \rangle$$

and wish to find a maximum length common subsequence of X and Y.

LCS Problem

LCS - Example

Let

$$X = \langle A, B, C, B, D, A, B \rangle$$

and

$$Y = \langle B, D, C, A, B, A \rangle$$

the sequence

$$\langle B, C, A \rangle$$

is a common subsequence but it is not a longest common subsequence (LCS) of X and Y.

$$\langle B, C, B, A \rangle$$

is an LCS of X and Y, as is the sequence

$$\langle B, D, A, B \rangle$$

LCS Problem

Theorem 15.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

From Cormen et al

LNMIIT

LCS Problem

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

From Cormen et al

LCS Problem

```
LCS-LENGTH(X, Y)
    m = X.length
    n = Y.length
    let b[1..m, 1..n] and c[0..m, 0..n] be new tables
    for i = 1 to m
        c[i, 0] = 0
    for j = 0 to n
        c[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            if x_i == y_j
                c[i, j] = c[i - 1, j - 1] + 1
                b[i, j] = "↖"
            elseif c[i - 1, j] ≥ c[i, j - 1]
                c[i, j] = c[i - 1, j]
                b[i, j] = "↑"
            else c[i, j] = c[i, j - 1]
                b[i, j] = "←"
    return c and b
```

From Cormen et al

LCS Problem

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

From Cormen et al

LCS Problem

```
PRINT-LCS(b, X, i, j)
    if i == 0 or j == 0
        return
    if b[i, j] == "↖"
        PRINT-LCS(b, X, i − 1, j − 1)
        print x_i
    elseif b[i, j] == "↑"
        PRINT-LCS(b, X, i − 1, j)
    else PRINT-LCS(b, X, i, j − 1)
```

From Cormen et al