Programming and Data Structures with Python

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

15 March, 2021

Inductive definitions

- Factorial
  - $f(0) = 1$
  - $f(n) = n \times f(n-1)$
- Insertion sort
  - isort([ ]) = [ ]
  - isort([x1,x2,..,xn]) = insert(x1,isort([x2,…,xn]))

… Recursive programs

```
def factorial(n):
  if n <= 0:
    return(1)
  else:
    return(n*factorial(n-1))
```

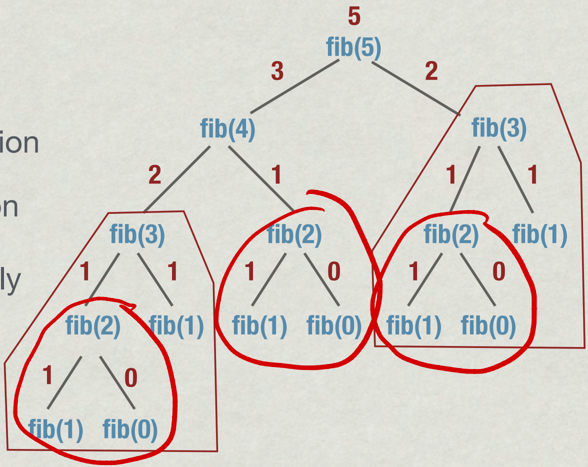
Sub problems

- factorial(n-1) is a subproblem of factorial(n)
  - So are factorial(n-2), factorial(n-3), …, factorial(0)
- isort([x2,…,xn]) is a subproblem of isort([x1,x2,…,xn])
  - So is isort([xi,…,xj]) for any $1 \leq i \leq j \leq n$
- Solution of f(y) can be derived by combining solutions to subproblems

Computing fib(5)

Overlapping subproblems

- Wasteful recomputation
- Computation tree grows exponentially



Never re-evaluate a subproblem

- Build a table of values already computed
  - Memory table
- Memoization
  - Remind yourself that this value has already been seen before

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

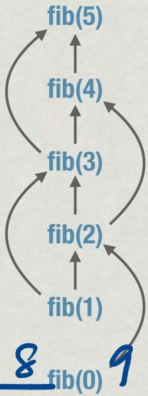
fib(5)

k	fib(k)

Dynamic programming

Bottom up

* Anticipate what the memory table looks like
  * Subproblems are known from problem structure
* Solve subproblems in order of dependencies
  * Must be acyclic



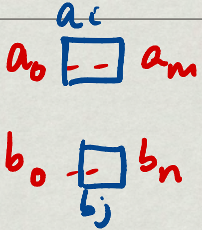
Fib(n)

0	1	2	3	4	5	6	7	8	9	10
0	1	1	2	3	5	8	13	21	34	55

Longest common subword

- Given two strings, find the (length of the) longest common subword
  - “secret”, “secretary” — “secret”, length 6
  - “bisect”, “trisect” — “isect”, length ~~4~~ 5
  - “bisect”, “secret” — “sec”, length 3
  - “director”, “secretary” — “ec”, “re”, length 2

More formally ...

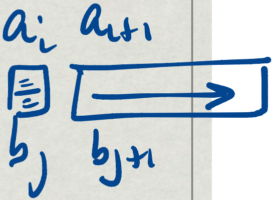


- Let $u = a_0a_1\ldots a_m$ and $v = b_0b_1\ldots b_n$ be two strings
- If we can find i, j such that $\underline{a_ia_{i+1}\ldots a_{i+k-1}} = \underline{b_jb_{j+1}\ldots b_{j+k-1}}$, u and v have a common subword of length k
- Aim is to find the length of the longest common subword of u and v

Brute force

- Let $u = a_0a_1\ldots a_m$ and $v = b_0b_1\ldots b_n$
- Try every pair of starting positions i in u, j in v] $m \times n$
  - Match $(a_i, b_j), (a_{i+1},b_{j+1}),\ldots$ as far as possible
  - Keep track of the length of the longest match
- Assuming $m > n$, this is $O(mn^2)$
  - mn pairs of positions
  - From each starting point, scan can be $O(n)$

Inductive structure



- Let $u = a_0a_1 \ldots a_m$ and $v = b_0b_1 \ldots b_n$
- $a_ia_{i+1} \ldots a_{i+k-1} = b_jb_{j+1} \ldots b_{j+k-1}$ is a common subword of length k at (i,j) iff $a_{i+1} \ldots a_{i+k-1} = b_{j+1} \ldots b_{j+k-1}$ is a common subword of length k-1 at (i+1,j+1)
- LCW(i,j): length of the longest common subword starting at a_i and b_j
  - If $a_i \neq b_j$, LCW(i,j) is 0, otherwise 1+LCW(i+1,j+1)
  - Boundary condition: when we have reached the end of one of the words

Inductive structure

0 .. m | m+1

a_0 a_m | a_{m+1} ?

- Consider positions 0 to m+1 in u, 0 to n+1 in v
  - m+1, n+1 means we have reached the end of the word

m × n subproblems

- LCW(m+1,j) = 0 for all j
- LCW(i,n+1) = 0 for all i
- LCW(i,j) = 0, if $a_i \neq b_j$,

  1+ LCW(i+1,j+1), if $a_i = b_j$

Subproblem dependency

* LCW(i,j) depends on LCW(i+1,j+1)
* Last row and column have no dependencies
* Start at bottom right corner and fill by row or by column

0 m+1

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

0

n+1

Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Reading off the solution

- Find (i,j) with largest entry
  - LCW(2,0) = 3
- Read off the actual subword diagonally

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

LCW(u,v), DP

```
function LCW(u,v) # u[0..m], v[0..n]

for r = 0,1,…,m+1  { LCW[r][n+1] = 0 } # r for row

for c = 0,1,…,m+1  { LCW[m+1][c] = 0 } # c for col

maxLCW = 0

for c = n,n-1,…,0
  for r = m,m-1,…0
    if (u[r] == v[c])
      LCW[r][c] = 1 + LCW[r+1][c+1]
    else
      LCW[r][c] = 0
    if (LCW[r][c] > maxLCW)
      maxLCW = LCW[r][c]

return(maxLCW)
```

← Track r,c as well

Complexity

- Recall that the brute force approach was $O(mn^2)$
- The inductive solution is $O(mn)$ if we use dynamic programming (or memoization)
  - Need to fill an $O(mn)$ size table
  - Each table entry takes constant time to compute

Longest common subsequence

- Subsequence: can drop some letters in between
- Given two strings, find the (length of the) longest common subsequence
  - "secret", "secretary" — "secret", length 6
  - "bisect", "trisect" — "isect", length 5
  - "bisect", "secret" — "sect", length 4
  - "director", "secretary" — "ectr", "retr", length 4

LCS

director

secretary

* LCS is longest path we can find between non-zero LCW entries, moving right and down

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Applications

- Analyzing genes
  - DNA is a long string over A,T,G,C
  - Two species are closer if their DNA has longer common subsequence
- UNIX diff command
  - Compares text files
  - Find longest matching subsequence of lines

Inductive structure

a_0 in soln

b_0 in soln

u	a_0	a_1	a_2	…	…	a_{m-1}	a_m
v	b_0	b_1	b_2	…	b_{n-1}	b_n	

- If $a_0 = b_0$,

  $LCS(a_0a_1\ldots a_m,\ b_0b_1\ldots b_n) = 1 + LCS(a_1a_2\ldots a_m,\ b_1b_2\ldots b_n)$

  - Can force (a_0,b_0) to be part of LCS
- If not, a_0 and b_0 cannot both be part of LCS
  - Not sure which one to drop
  - Solve both subproblems $LCS(a_1a_2\ldots a_m,\ b_0b_1\ldots b_n)$ and $LCS(a_0a_1\ldots a_m, b_1b_2\ldots b_n)$ and take the maximum

$a_0 - a_m$ | $a_1 - a_m$

$b_1 - b_n$ | $b_0 - b_n$

Inductive structure

u	a_i	a_{i+1}	a_{i+2}	…	…	a_{m-1}	a_m
v	b_j	b_{j+1}	b_{j+2}	…	b_{n-1}	b_n	

- LCS(i,j) stands for LCS($a_i a_{i+1} \ldots a_m$, $b_j b_{j+1} \ldots b_n$)
- If $a_i = b_j$, LCS(i,j) = 1 + LCS(i+1,j+1)
- If $a_i \neq b_j$, LCS(i,j) = max(LCS(i+1,j), LCS(i,j+1))
- As with LCW, extend positions to m+1, n+1
  - LCS(m+1,j) = 0 for all j
  - LCS(i,n+1) = 0 for all i

Subproblem dependency

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

$a_i = b_j$ $\quad 1 + LCS(i+1, j+1)$

Subproblem dependency

$a_i \neq b_j$ $\quad \max\left(\begin{array}{l} i+1, j \\ i, j+1 \end{array} \right)$

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	2	2	1	0
1	i	4	3	2	2	2	1	0
2	s	4	3	2	2	2	1	0
3	e	3	3	2	2	2	1	0
4	c	2	2	2	1	1	1	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Recovering the sequence

- Trace back the path by which each entry was filled
- Each diagonal step is an element of the LCS
  - “sect”

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

	c	a	t	
b	1	1	1	0
e	1	1	1	0
t	1	1	1	0
	0	0	0	0

d i r e c t o r y

s

e

$f(u, w)$ depends on all subsequences of u & w

↓

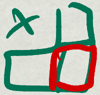
exponentially many subproblems

DP <u>cannot</u> be better than exponentially

LCS(u,v), DP

```
function LCS(u,v) # u[0..m], v[0..n]

for r = 0,1,…,m+1  { LCS[r][n+1] = 0 }

for c = 0,1,…,m+1  { LCS[m+1][c] = 0 }

for c = n,n-1,…,0
  for r = m,m-1,…0
    if (u[r] == v[c])
      LCS[r][c] = 1 + LCS[r+1][c+1]
    else
      LCS[r][c] = max(LCS[r+1][c],
                      LCS[r][c+1])

return(LCS[0][0])
```

Complexity



- Again O(mn) using dynamic programming (or memoization)
  - Need to fill an O(mn) size table
  - Each table entry takes constant time to compute

Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”
- “The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructures ~~property~~ can~~d~~ ~~it~~b~~s~~e used in designing algorithms”
- 28 characters inserted, 18 ~~deleted~~, 2 ~~substituted~~

Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute a character by another one
- In our example,
  28 characters inserted, 18 ~~deleted~~, 2 ~~substituted~~
- Edit distance is at most 48

Edit distance

- Also called Levenshtein distance
  - First proposed by Vladimir Levenshtein
- Applications
  - Suggest spelling corrections in word processor, search engine queries
  - Another way of comparing genetic similarity across species

Edit distance and LCS

bi Sec~~re~~t

bisect

- Longest common subsequence of u and v
  - What remains after minimum number of deletes to make them equal
- Deleting a letter in u equivalent to inserting it in v
  - “secret”, “bisect” — LCS is “sect”
    - delete “r”, “e” in “secret”, “b”, “i” in “bisect”
    - delete “r”, “e” then insert “b”, “i” in “secret”
- LCS is equivalent to edit distance without substitution

Inductive structure for edit distance

u	a_i	a_{i+1}	a_{i+2}	…	…	a_{m-1}	a_m
v	b_j	b_{j+1}	b_{j+2}	…	b_{n-1}	b_n	

- Recall LCS
  - If $a_i = b_j$, $LCS(i,j) = 1 + LCS(i+1,j+1)$
  - If $a_i \neq b_j$, $LCS(i,j) = \max(LCS(i+1,j), LCS(i,j+1))$
  - Boundary condition when one of the words is empty

Edit distance…

u	a_i	a_{i+1}	a_{i+2}	…	…	a_{m-1}	a_m
v	b_j	b_{j+1}	b_{j+2}	…	b_{n-1}	b_n	

a_i

- Aim is to transform u into v
  - If $a_i = b_j$, ED(i,j) = ED(i+1,j+1) — nothing to be done at (a_i,b_j)
  - If $a_i \neq b_j$, can do one of three things
    - Substitute a_i by b_j: 1 + ED(i+1,j+1)
    - Delete a_i: 1 + ED(i+1,j)
    - Insert b_j before a_i: 1 + ED(i,j+1)
    - Take the minimum of these

m+1, j

Inductive structure

u	a_i	a_{i+1}	a_{i+2}	...	...	a_{m-1}	a_m
v	b_j	b_{j+1}	b_{j+2}	...	b_{n-1}	b_n	

- ED(i,j) stands for ED($a_i a_{i+1} \ldots a_m$, $b_j b_{j+1} \ldots b_n$)
- If $a_i = b_j$, ED(i,j) = ED(i+1,j+1)
- If $a_i \neq b_j$, ~~LCS~~(i,j) =

  ED 1 + min(ED(i+1,j+1),ED(i+1,j), ED(i,j+1))
- As with LCS/LCW, extend positions to m+1, n+1
  - ED(m+1,j) = n-j+1 for all j # Insert $b_j b_{j+1} \ldots b_n$ in u
  - ED(i,n+1) = m-i+1 for all i, # Insert $a_i a_{i+1} \ldots a_m$ in v

Subproblem dependency

- Like LCS, ED(i,j) depends on ED(i+1,j+1), ED(i+1,j) and ED(i,j+1)
- Dependencies for ED(m,n) are known
- Start at ED(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

Subproblem dependency

m+1 →

xysect

ysect

secret•

bisect •

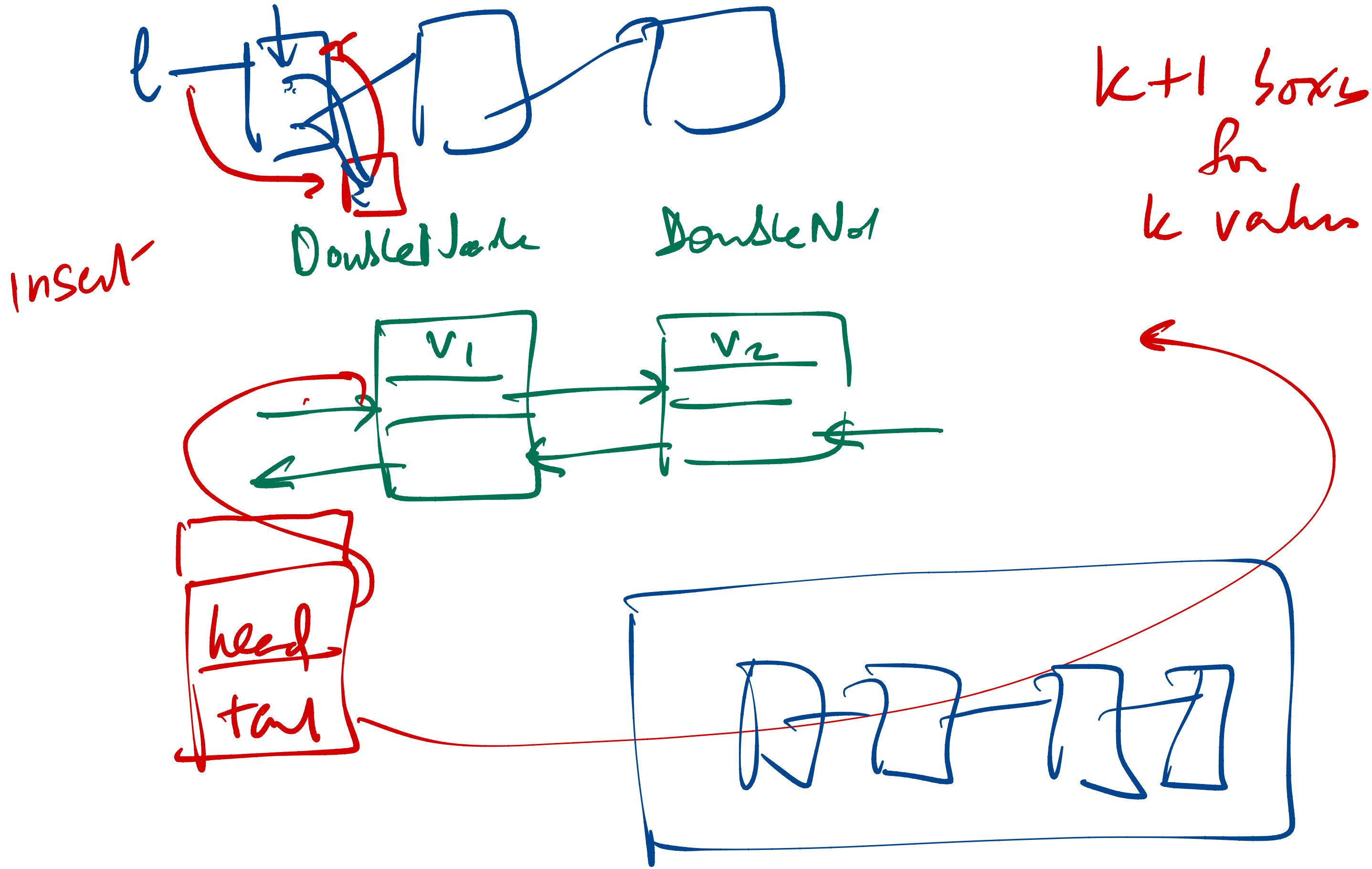
2 →

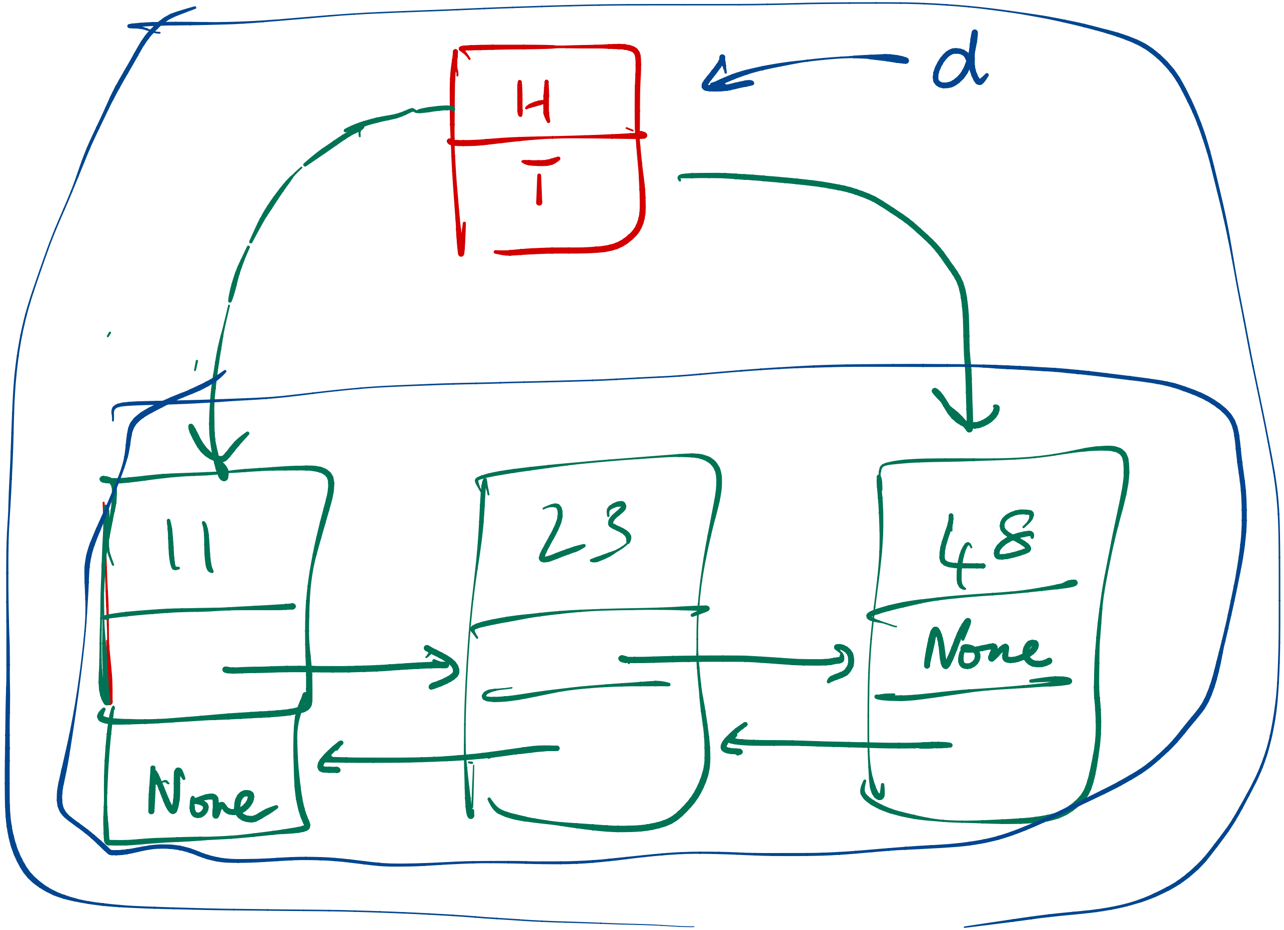
- Like LCS, ED(i,j) depends on ED(i+1,j+1), ED(i+1,j) and ED(i,j+1)
- Dependencies for ED(m,n) are known
- Start at ED(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	4	4	4	4	5	6
1	i	3	4	3	3	3	4	5
2	s	2	3	3	2	2	3	4
3	e	3	2	3	2	1	2	3
4	c	4	3	2	2	1	1	2
5	t	5	4	3	2	1	0	1
6	•	6	5	4	3	2	1	0

Insert sect in u

Delete sect in v


l
DoubleNode
DoubleNode
insert
k+1 boxes
for
k values
v1
v2
head
tail



d
H
T
11
None
23
48
None