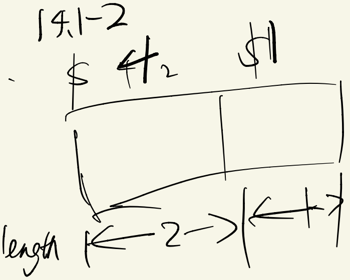
14.1-2



$4+1=5$

$p_2/2 = 4/2 = 2 > p_3/3 = 1.8$

$p_3 = 5.4 > 4+1$

14.1-3

```
EXTENDED-ROD-CUTTING(p, n)
let r[0:n] and s[0:n] be new arrays.
r[0]=0
for j=1 to n
    q=+∞
    for i=0 to j
        if q < p[i] + r[n-i] - c
            q = p[i] + r[n-i] - c
            s[j]=i
    r[j]=q
return r and s
```

14.1-4

the constant factor will change.

But the $O(n^2)$ stays the same.

for $i = 1$ to $\lceil \frac{n}{2} \rceil$
  $q = \max\{q,$ $p[i] +$ ----$(p, n-i, r)$,
                $p[n-i] +$ ---$(p, i, r)\}$

14.1-5

```
EXTENDED-MEMOIZED-CUT-ROD(p, n)
let r[0:n], s[0:n] be a new array
for i = 0 to n
  r[i] = -∞
return EXTENDED-MEMOIZED-CUT-ROD-AUX(p, n, r, s)

EXTENDED-MEMOIZED-CUT-ROD-AUX(p, n, r, s)
 if r[n] ≥ 0
   return r, s
 if n == 0
   r[n] = 0
   s[0] = 0
 else
   for i = 1 to n
     r, s = EXTENDED-MEMOIZED-CUT-ROD-AUX(p, n-i, r, s)
     if r[n] < p[i] + r[n-i]
       r[n] = p[i] + r[n-i]
       s[n] = i
return r, s
```

```
PRINT-MEMOIZED-CUT-ROD(p, n)
r, s = EXTENDED-MEMOIZED-CUT-ROD(p, n)
while n > 0
  print s[n]
  n = n - s[n]
```

141-6

$$F_i = \begin{cases} 0 & i=0 \\ 1 & i=1 \\ F_{i-1}+F_{i-2} & i \geq 2 \end{cases}$$

BOTTOM-UP-FIBONACCI(n)

a = 0

b = 1

for i = 2 to n   O(n)

    $F_i = a + b$

    a = b

    $b = F_i$

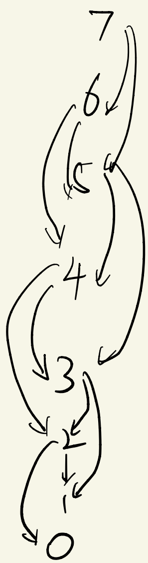
return F_i

Vertices

O(n)

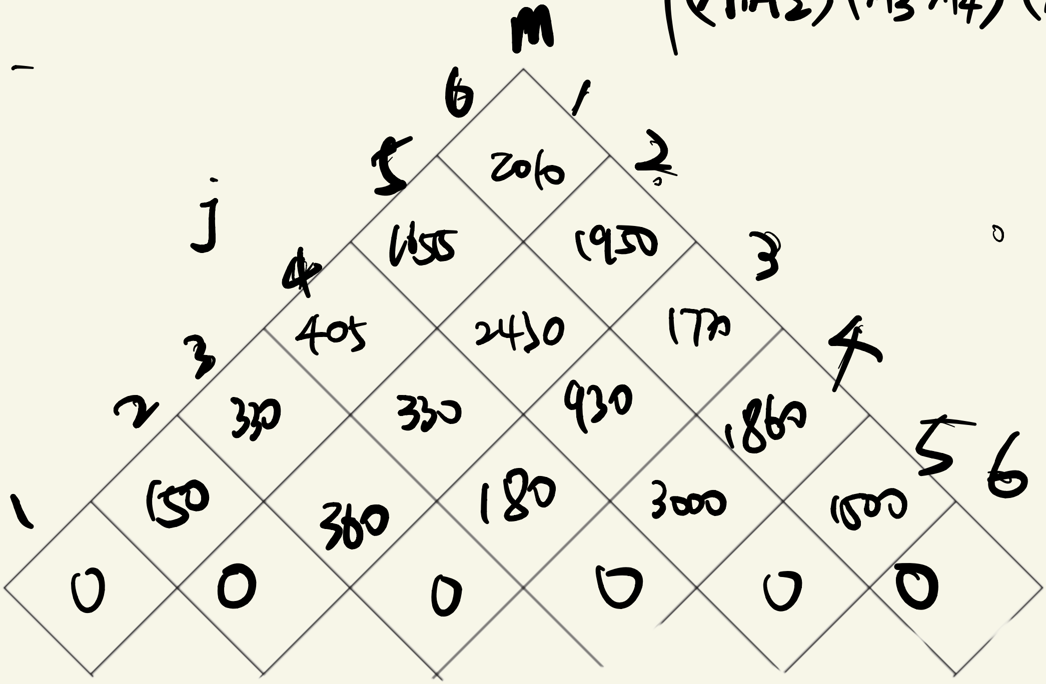
edges

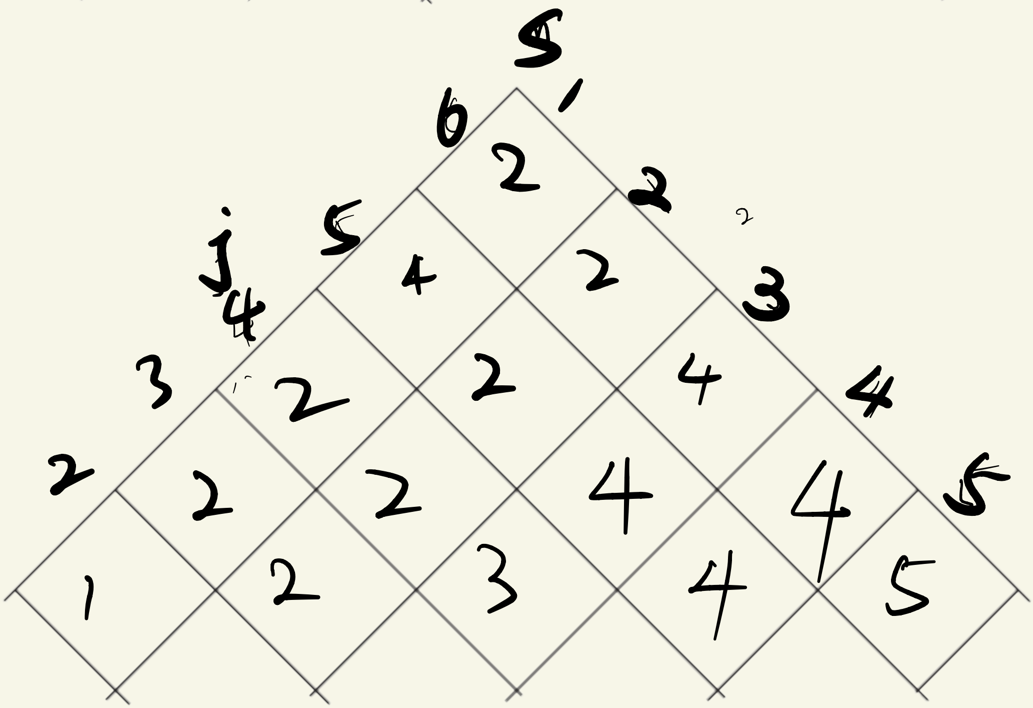
O(2n)



14.2-1 $n=6$ $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$

$((A_1A_2)(A_3A_4)(A_5A_6))$





```
package main

import (
        "fmt"
  "math"
)

func main() {
  p := []int{5, 10, 3, 12, 5, 50, 6}
  m, s := matrixChainOrder(p)
  printOptimalParens(s, 1, 6)
  printMatrixSubCost(m, len(p)-1)
  printK(s, len(p) - 1)
}

func printOptimalParens(s [][]int, i int , j int){
  if i == j{
    fmt.Printf("A%d", i)
  } else{
    fmt.Printf("(")
    printOptimalParens(s, i, s[i][j])
    printOptimalParens(s, s[i][j] + 1, j)
    fmt.Printf(")")
  }
}

func printMatrixSubCost(m [][]int, n int){

  for l:=7; l >0; l --{
    fmt.Printf("\n")
      for i := 1; i < n - l + 2; i ++ {
      j := i + l - 1
        fmt.Printf("%d ", m[i][j])
    }
  }
}

func printK(s [][]int, n int){

  for l:=7; l >1; l --{
    fmt.Printf("\n")
      for i := 1; i < n - l + 2; i ++ {
      j := i + l - 1
        fmt.Printf("%d ", s[i][j])
    }
  }
}

func matrixChainOrder(p []int) ([][]int, [][]int){
  n := len(p) -1 // the number of matrix
  m := make([][]int, n+1)
  for i := range m {
    m[i] = make([]int, n+1)
    m[i][i] = 0
  }
  s := make([][]int, n+1)
  for i := range s {
    s[i] = make([]int, n+1)
  }

  for l := 2; l < n + 1; l ++ {
    for i := 1; i < n - l + 2; i ++ {
      j := i + l - 1
      m[i][j] = math.MaxInt
        for k := i; k < j; k ++ {
          q := m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j]
          // the starting demension of ith matrix, the ending demension of kth matrix and the ending demension of jth matrix
          if q < m[i][j] {
            m[i][j] = q
            s[i][j] = k
          }
        }
    }
  }
  return m,s
}
```

14.2-2

MATRIX-CHAIN-MULTIPLY(A, S, i, j)

if j == i+1

  return RECTANGULAR-MATRIX-MULTIPLY(A[i], A[j])

else

  left = MATRIX-CHAIN-MULTIPLY(A, S, i, S[i, j])

  right = MATRIX-CHAIN-MULTIPLY(A, S, S[i, j]+1, j)

  return RECTANGULAR-MATRIX-MULTIPLY(left, right)

14.2-3

14.6 Recurrence

$$p(n) = \begin{cases} 1 & \text{if } n=1 \\ \sum_{k=1}^{n} p(k)\,p(n-k) & \text{if } n \geq 2 \end{cases}$$

$p(n) = \Omega(2^n)$

Suppose $p(n) \geq c\,2^n$

$$p(n) = \sum_{k=1}^{n} c \cdot 2^k \cdot c\,2^{n-k}$$ as long as $n > \frac{1}{c}$

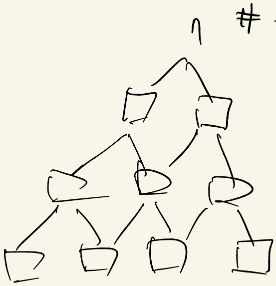
$$= \sum_{k=1}^{n} c^2\, 2^n = n c^2 2^n \geq c 2^n$$

14.2-4

The subproblem graph is like a pyramid.

$$\# \text{ vertices: } 1+2+3+4 \cdots + n = \frac{(n+1)n}{2}$$

$$\# \text{ edges: } 2+4+6+ \cdots +2(n-1) = 2 \cdot \frac{(n-1+1)(n-1)}{2} = n(n-1)$$



14.2-5

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

line 9 has n access for table m

$$\sum_{i=1}^{n}\sum_{j=1}^{n} R(i,j) = 2\sum_{l=2}^{n} (l-1)(n-l+1) = 2 \cdot \sum_{l=1}^{n-1} l \cdot (n-l)$$

Genius!

$$= 2 \sum_{l=2}^{n} (ln - l^2 + l - n + l - 1)$$

$$= 2n \frac{(n+2)(n-1)}{2} - 2 \cdot \left(\frac{n(n+1)(2n+1)}{6} + 1\right) + 2 \cdot 2 \cdot \frac{(n+2)(n-1)}{2}$$

$$- 2 \cdot (n-1) \cdot n - 2(n-1)$$

$$= n^3\left(1 - \frac{2}{3}\right) + n^2(1 - 1 + 2 - 2) + n\left(-2 - \frac{1}{3} + 2 + 2 - 2\right)$$

$$+ (-4 + 2 + 2)$$

$$= \frac{n^3 - n}{3}$$

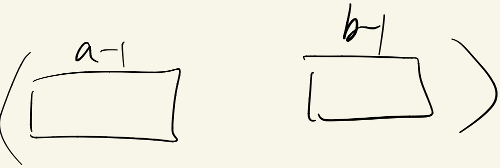
q.e.d

14.2-6

$n=1$ holds.

$n=2$

induction

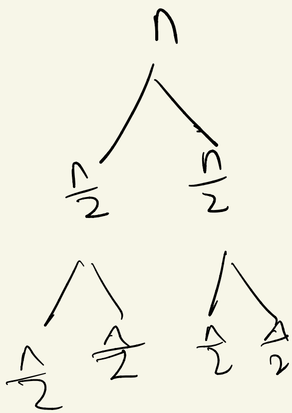


$$a-1+b-1+1 = \underbrace{a+b}_{n} - 1 = n-1$$

q.e.d.

14.3-1 RECURSIVE-MATRIX-CHAIN is asymptotically the same with enumerating all the ways of parenthesing. However RECURSIVE-MATRIX-CHAIN may have a bigger constant factor overhead due to it recursiveness.

14.3-2 no overlapping subproblems.



14.3-3 Yes.

14.3-4 1000 100 20 10 1000

14.3-5 The subproblems are not independent.

14.4-1

		0	1	2	3	4	5	6	7	8	9
	y_i		0	1	0	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	1	0	↑0	↖1	↑0	↖1	↖1	↑0	↖1	↖1	↑0
2	0	0	↖1	↑1	↖2	←2	←2	↖2	←2	←2	↖2
3	0	0	↖1	↑1	↖2	↑2	↑2	↖3	↑2	↑2	↖3
4	1	0	↑1	↖2	↑2	↖3	↖3	↑3	↖4	↖3	↑3
5	0	0	↖1	↑2	↖3	↑3	↑3	↖4	↑4	←3	↖4
6	1	0	↑1	↖2	↑3	↖4	↖4	↑4	↖5	↖4	↑4
7	0	0	↖1	↑2	↑3	↑4	↖5	←5	↑5	←5	↖5
8	1	0	↑1	↖2	↑3	↖4	↖5	↑5	↖6	↖6	←6

LCS = ⟨1,0,1,0,1,1⟩

14.4-2

PRINT-LCS-C(c, x, y, i, j)

```
if i==0 or j==0
    return

else if x[i] == y[j]
    PRINT-LCS-C (c, x, i-1, j-1)
    print x[i]

else if c[i-1, j] ≥ c[i, j-1]
    PRINT-LCS-C(c, x, i-1, j)

else
    PRINT-LSN-C(c, x, i, j-1)
```

14.4-3

```
LCS-LENGTH-MEMOIZED(x, y, m, n)
    let c[0:m, 0:n] be a new tables
    for i = 1 to m
        c[i, 0] = 0
    for j = 1 to n
        c[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            c[i, j] = -1        // the token for unsolved subproblems
    LCS-LENGTH-MEMOIZED-AUX(x, y, m, n, c)
    return c

LCS-LENGTH-MEMOIZED-AUX(x, y, i, j, c)
    if c[i, j] ≠ -1
        return c[i, j]
    else if x[i] == y[i]
        c[i, j] = 1 + LCS-LENGTH-MEMOIZED-AUX(x, y, i-1, j-1, c)
    else
        up = LCS-LENGTH-MEMOIZED-AUX(x, y, i-1, j, c)
        left = LCS-LENGTH-MEMOIZED-AUX(x, y, i, j-1, c)
        if up ≥ left
            c[i, j] = up
        else
            c[i, j] = down
    return c[i, j]
```

14.4-4

Space

$2\min(m,n)+O(1)$

```
LCS-LENGTH(x, y, m, n)
  if m ≥ n          // row-major order
    let C[0:1, 0:n] be a new table
    for i = 0 to 1
      for j = 0 to n
        C[i, j] = 0
    for i = 1 to m
      for k = 1:n                       // Copy the 1st to the 0th row
        C[0, k] = C[1, k]
      for j = 1 to n
        if x[i] == y[j]
          C[1, j] = C[0, j-1] + 1
        else if  C[0, j] ≥ C[1, j-1]
                 C[1, j] = C[0, j]
        else
                 C[1, j] = C[1, j-1]
  else symmetric with respect to m and n, or row and column
  return C
```

Space

$\min(m,n)+O(1)$

```
LCS-LENGTH(x, y, m, n)
  if m ≥ n          // row-major order
    let C[0:n] be a new arrary
    for j = 0 to n
      C[j] = 0
    for i = 1 to m
      for j = 1 to n
        if x[i] == y[j]
          C[1, j] = C[1, j-1] + 1
        else if C[1, j] ≥ C[1, j-1]
                continue
        else    C[1, j] = C[1, j-1]
  else symmetric with m and n
  return C
```

14.4-5 Brutal force: the permutations of all subsequence

$O(2^n)$ check $\begin{cases} \text{whether they are monotonically increasing} \\ \text{keep track of the length.} \end{cases}$

Dynamic programming $O(n^2)$

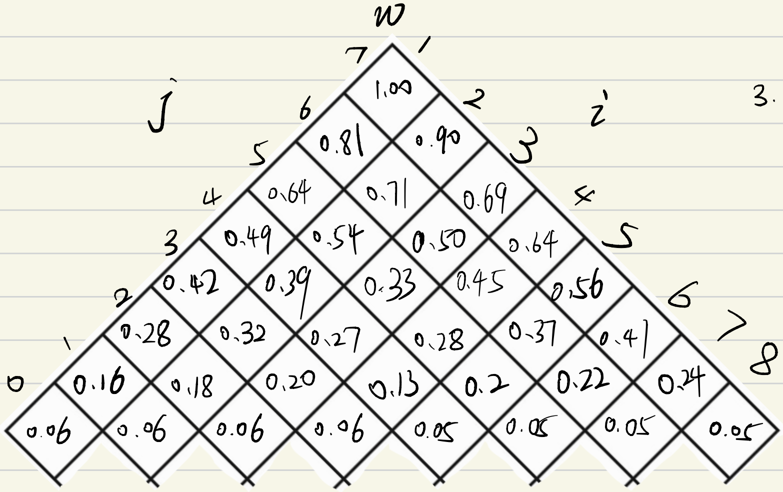
14.5

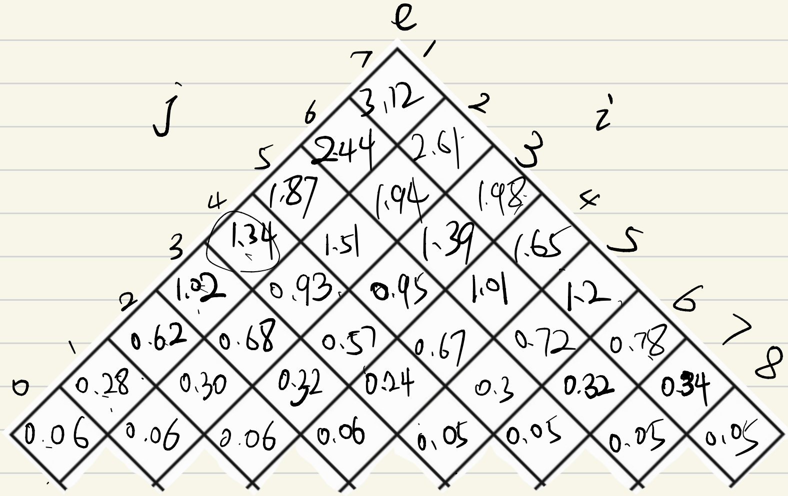
```
CONSTRUCT-OPTIMAL-BST(root, n)
    m = root[i, j]
    print "k_{m} is the root"
    CONSTRUCT-OPIMAL-LEFT-BTS(1, m-1)
    CONSTRUCT-OPTIMAL-RIGHT-BTS(m+1, n)


CONSTRUCT-OPTIMAL-LEFT-BST(root, i, j)
    if i > j
        print "d_j is the left child of k_{j+1}"
        return
    else
        m = root[i, j]
        print "k_m is the left child of k_{j+1}"
        CONSTRUCT-OPTIMAL-LEFT-BST(root, i, m-1)
        CONSTRUCT-OPTIMAL-RIGHT-BST(root, m+1, j)


CONSTRUCT-OPTIMAL-RIGHT-BST(root, i, j)
    if i > j
        print "d_j is the right child of k_{i-1}"
        return
    else
        m = root[i, j]
        print "k_m is the right child of k_{i-1}"
        CONSTRUCT-OPTIMAL-LEFT-BST(root, i, m-1)
        CONSTRUCT-OPTIMAL-RIGHT-BST(root, m+1, j)
```

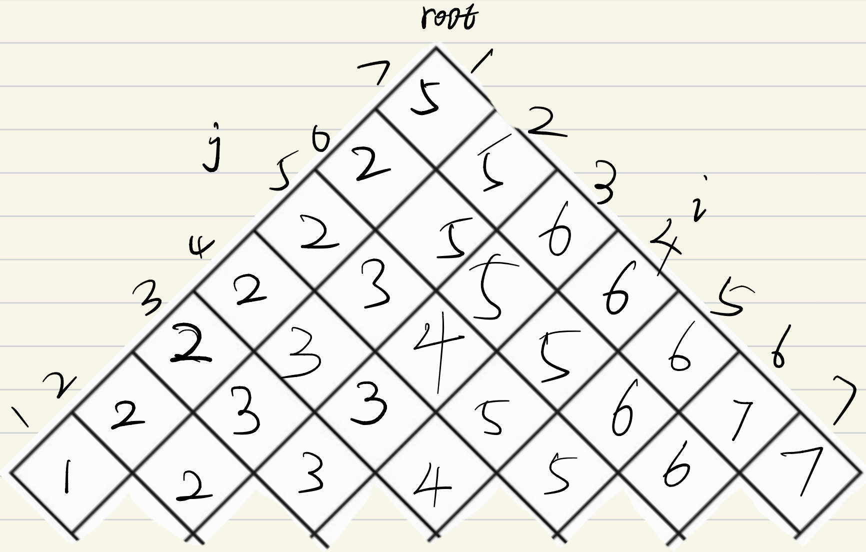
14.5-2

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.1	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05
		0.1	0.12	0.14	0.07	0.15	0.17	0.19

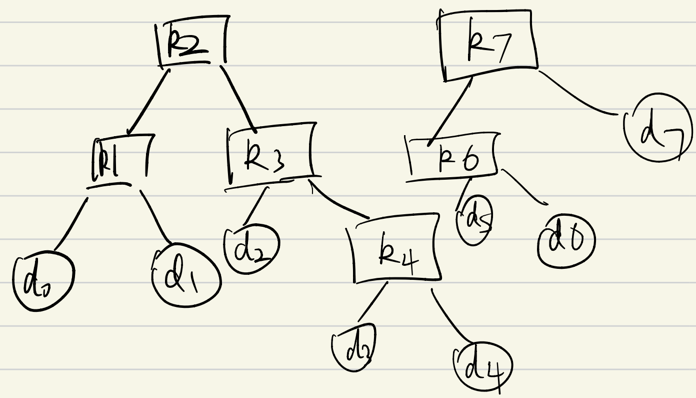
3.4







Cost: 3.12
structure:



14.5-3 The running time of OPTIMAL-BST will become $O(n^3)$
For every (i,j), the will be $O(n)$ additions for $w[i,j]$ there is $O(n^2)$ (i,j).

14.5-4 Change the line 10 of OPTIMAL-BST, "$r = i$ to j",
to "$r =$ root$[i, j-1]$ to root$[i+1, j]$".

proof of $O(n^2)$ of OPIMAL-BST

ith iteration

$r =$ root$[i, j-1]$ to root$[i+1, j]$

$i+1$ th iteration

$r =$ root$[i+1, j+1-1]$ to root$[i+2, j+1]$

$=$ root$[i+1, j]$ to root$[i+1, j+1]$

There is nearly no overlapping r between 2 consecutive iterations.
Thus the loop in line 6 is $O(n)$.
Thus the nested loop in line 5 is $O(n^2)$.

14-1 Dynamic programming

```
STRUCT EDGE
  a VERTEX
  b VERTEX
  weight number

LONGEST-SIMPLE-PATA(edges, s, t)
  map = {points : {edges}}
  for edge in edges
      if edge.a is not in map.keys
          map[edge.a] = {edge]
      else
          map[edge.a].append(edge)
  dis = {point: number}
  longestPath = {point: edge}
  point = S
  points = {S]
  while points is not empty
      for edge in map[point]
          if edge.b is not in dis.keys
              dis[edge.b] = edge.weight
              longestPath[edge.b] = edge
              if edge.b != t
                  points.append(edge.b)
              else
                  if edge.weight > dis[edge.b]
                      dis[edge.b] = edge.weight
                      longestPath[edge.b] = edge.
      points.delete(point)
      point = points.next
```

$O(E)$