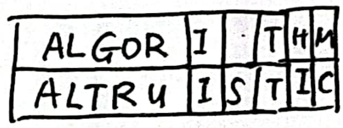
Jeff. Erickson : Algorithms

Chapter 3. Dynamic Programming (Continued, part 2).

Example 4. Edit Distance

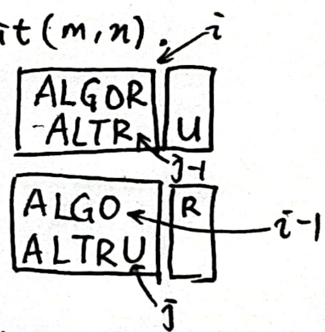
The <u>edit distance</u> of 2 strs is the minimum number of insertions, deletions, substitutions required to transform one string into another.



- Recursive Structure : If we remove the last column, the remaining cols must represent shortest edit seq for remaining prefixes.

  Proof by contradiction. If the prefix had shorter edit seq, glue back → more optimal

- State: Let $Edit(i,j)$ denote the Edit distance between $A[1..i]$, $B[1..j]$. Compute $Edit(m,n)$.



  ▷ Insertion : $Edit(i,j) = Edit(i,j-1)+1$

  ▷ Deletion : $Edit(i,j) = Edit(i-1,j)+1$

  ▷ Substitution:

  If $A[i]$ and $B[j]$ are different, $Edit(i,j) = Edit(i-1,j-1)+1$

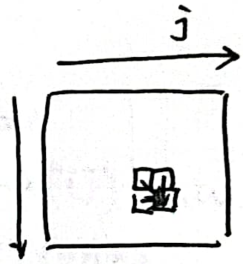
  otherwise, no need for substitution $Edit(i,j) = Edit(i-1,j-1)$.

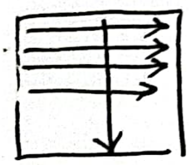
  ▷ Boundary : $Edit(i,j) = i$ (if $j=0$)

- Recursive Expression

$$Edit(i,j) = \begin{cases} i & \text{if } j=0 \\ j & \text{if } i=0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A_i \neq B_j] \end{cases} & \text{o.w.} \end{cases}$$

- Dynamic Programming
  - ▷ Subproblems. Every recursive subproblem is identified by 2 idencies $0 \le i \le m$ and $0 \le j \le m$.
  - ▷ Memoization Structure. 2D Array Edit[0..m , 0..n]
  - ▷ Dependencies. Edit[i,j] depends on Edit[i-1,j], Edit[i,j-1] and Edit[i-1,j-1].
  - ▷ Evaluation Order
  - ▷ Space & Time $O(mn)$ space time.





```
EDIT DISTANCE
    for j←0 to n
        Edit[0,j] = j
    for i←1 to n
        Edit[i,0] = i
        for j←1 to n
            ins ← Edit[i,j-1]+1
            del ← Edit[i-1,j]+1
            if A[i]=B[j]
                rep ← Edit[i-1,j-1]
            else
                rep ← Edit[i-1,j-1]+1
            Edit[i,j] ← min{ins, del, rep}.
    return Edit[m,n].
```

Example 5. Subset Sum : Whether any subset of a given array $X[1..n]$ of positive integers sums to a given integer T.

We defined $\boxed{SS(i,t) = \text{True if and only if some subset of } X[i..n] \text{ sums to } t.}$

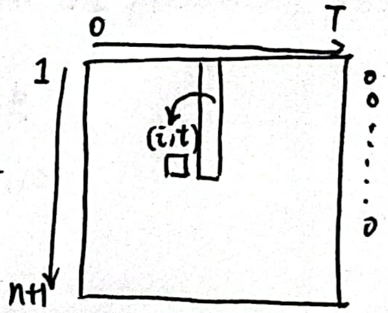
and

$$SS(i,t) = \begin{cases} \text{TRUE} & \text{if } t=0 \\ \text{FALSE} & \text{if } t<0 \text{ or } t>n \\ SS(i+1,t) \vee SS(i+1, t-X[i]) & \text{o.w.} \end{cases}$$

- Subproblems : described by i, t s.t. $1 \leq i < n+1$, $t < T$.

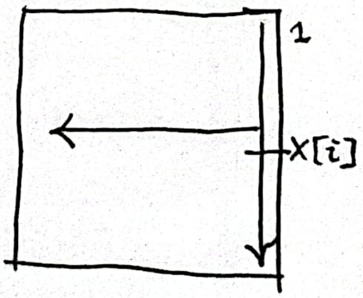
$$SS(i,t) = \begin{cases} \text{TRUE} & \text{if } t=0 \\ \text{FALSE} & \text{if } i>n \\ SS(i+1, t) & \text{if } t<X[i] \\ SS(i+1,t) \vee SS(i+1, t-X[i]) & \text{o.w.} \end{cases}$$

- Data structure. Memorize by 2D array $S[1..n+1, 0..T]$
- Evaluation Order.
- Space: $O(nT)$ Time: $O(nT)$




```
FAST SUBSET SUM (X[1..n], T):
    S[n+1, 0] ← True
    for t ← 1 to T
        S[n+1, t] ← False
    for i ← n downto 1
        S[i, 0] = TRUE
        for t ← 1 to X[i]-1
            S[i,t] ← S[i+1, t]
        for t ← X[i] to T
            S[i,t] ← S[i+1, t] ∨ S[i+1, t-X[i]].

    return S[1,T].
```



Example 6. Optimal BST

Input. a sorted array $A[1..n]$ of search keys

an array $f[1..n]$ of freq. counts

$f[i] :=$ # we will search for $A[i]$.

Objective: Construct BST s.t. total costs ~~of~~ is as small as possible.

Idea: Knowing where to split.

$$\underset{\text{Constr. Optimal BST for subarr } A[i..k]}{\underline{OptCost(i,k)}} = \begin{cases} 0 & , \text{if } i>k \\ \sum_{j=i}^{k} f[j] + \min_{i \le r \le k} \begin{Bmatrix} OPTCOST(i, r-1) \\ + \\ OPTCOST(r+1, k) \end{Bmatrix}, & \text{o.w.} \end{cases}$$

▷ For any pairs of idencies $i \le k$, let $F(i,k)$ denote total frequency count for all keys in $A[i..k]$.

$F(i,k) = \sum_{j=i}^{k} f[j]$, with the following recr.

$$F(i,k) = \begin{cases} f[i] & , \text{if } i=k \\ F(i,k-1) + f[k], & \text{o.w.} \end{cases}$$

We may init F first

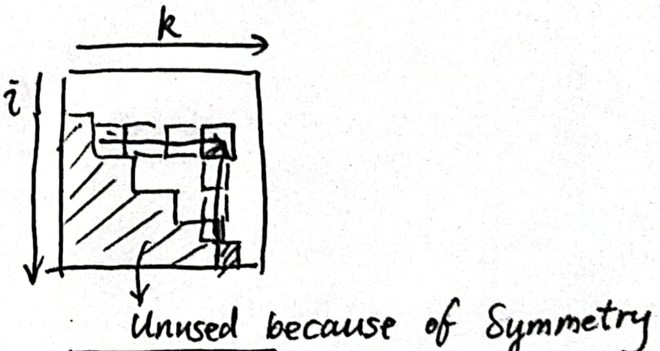
```
INITF (f[1..n])
    for i←1 to n
        F[i,i-1]←0
        for k← i to n
            F[i,k] ← F[i,k-1]+F[k]
```

We will get

$$OptCost(i,k) = \begin{cases} 0 & i>k \\ F[i,k] + \min_{i \le r \le k} \begin{Bmatrix} OptCost(i, r-1) \\ + OptCost(r+1, k) \end{Bmatrix} & , \text{o.w} \end{cases}$$

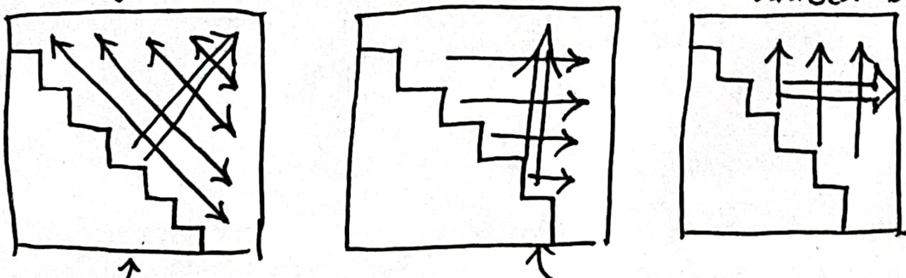
- Subproblems: Each recur. prob. is specified by 2 ints i and k, s.t. $1 \le i \le n+1$, $0 \le k \le n$
- Memoization. In 2-D array OptCost[1..n+1, 0..n].
- Dependencies: OptCost[i,k] depends on OptCost[i, j-1]
  Optcost[j+1, k], $\forall j$, s.t. $i \le j \le k$.

  ⇒ directly left, directly down.



- Evaluation Order

  Vary.



```
for i ← 1 to n+1
    OptCost[i, i-1] ← 0
for d ← 0 to n-1
    for i ← 1 to n-d
        ComputeOptCost(i, i+d)
return OptCost[1,n]
```

```
for i ← n+1 downto 1
    OptCost[i, i-1] ← 0
    for j ← i to n
        ComputeOptCost(i, j)
return OptCost[1,n]
```

```
for j ← 0 to n+1
    Optcost[j+1, j] ← 0
    for i ← j downto
        ComputeOptCost(i
return OptCost[1,n
```

- Time and Space: Space: $O(n^2)$
  Time: $O(n^3)$