Lecture 11
Dynamic Programming

Euiseong Seo
(euiseong@skku.edu)

Dynamic Programming?

- Very powerful and general tool for solving optimization problems on left-right-ordered items
- Little bit difficult to understand
- Once mastered, it would be your favorite weapon
- Examples
  - Binomial coefficients
  - Floyd's all-pairs shortest path algorithm

Greedy VS Dyn. Programming

- Greedy algorithms focus on making the best local choice at each decision point
  - Heuristic
  - Very likely to fail
- Dynamic programming systematically searches all possibilities while storing partial results to avoid recomputing
  - Always correct
  - Efficient

Backtracking VS Dyn. Prog.

- Both search all of solution space
- Both usually produce recursive algorithms
- Backtracking
  - Gradually build a solution candidate
  - Abandon it and retry another when the solution building meets a dead end
- Dynamic programming
  - Break the problem down into small sub-problems
  - Obtain the solution from the sub-problem solutions
  - Never do the same thing twice or more

Edit Distance

- Approximate pattern matching
  - Persistent to misspelling and changes in word usage
  - Example
    - Thou shalt not kill => You should not murder
- Edit distance problem
  - To obtain the cost of changes that have to be made to convert one string to another
  - Types of changes
    - Substitution
    - Insertion
    - Deletion

Edit Distance Algorithm

- The last character in the string must be matched, substituted, inserted, or deleted
- If we knew the cost of editing the three pairs of the remaining strings, we could decide which option leads to the best solution

Backtracking Version

```
#define MATCH     0       /* enumerated type symbol for match */
#define INSERT    1       /* enumerated type symbol for insert */
#define DELETE    2       /* enumerated type symbol for delete */

int string_compare(char *s, char *t, int i, int j)
{
        int k;                  /* counter */
        int opt[3];             /* cost of the three options */
        int lowest_cost;        /* lowest cost */

        if (i == 0) return(j * indel(' '));
        if (j == 0) return(i * indel(' '));

        opt[MATCH] = string_compare(s,t,i-1,j-1) + match(s[i],t[j]);
        opt[INSERT] = string_compare(s,t,i,j-1) + indel(t[j]);
        opt[DELETE] = string_compare(s,t,i-1,j) + indel(s[i]);

        lowest_cost = opt[MATCH];
        for (k=INSERT; k<=DELETE; k++)
                if (opt[k] < lowest_cost) lowest_cost = opt[k];

        return( lowest_cost );
}
```

Perf. Analysis of Edit Dist.

- How many calls are required for strings, of which sizes are |s| and |t|, respectively?
- How many character pairs are available for comparisons?
- Some pairs are calculated multiple times in the proposed algorithm
- Get rid of the redundant calculations by adopting a two dimensional matrix m of which size is |s|*|t|
  - Each cell contains the cost of the optimal solution of this subproblem, as well as a parent pointer

Dyn. Programming Version

```
typedef struct {
        int cost;          /* cost of reaching this cell */
        int parent;        /* parent cell */
} cell;

cell m[MAXLEN+1][MAXLEN+1];  /* dynamic programming table */
```

		y	o	u	-	s	h	o	u	l	d	-	n	o	t
:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	1	2	3	4	5	6	7	8	9	10	11	12	13	13
h:	2	2	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	2	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	3	2	3	4	5	6	5	6	7	8	9	10	11
-:	5	5	4	3	2	3	4	5	6	6	7	7	8	9	10
s:	6	6	5	4	3	2	3	4	5	6	7	8	8	9	10
h:	7	7	6	5	4	3	2	3	4	5	6	7	8	9	10
a:	8	8	7	6	5	4	3	3	4	5	6	7	8	9	10
l:	9	9	8	7	6	5	4	4	4	4	5	6	7	8	9
t:	10	10	9	8	7	6	5	5	5	5	5	6	7	8	8
-:	11	11	10	9	8	7	6	6	6	6	6	5	6	7	8
n:	12	12	11	10	9	8	7	7	7	7	7	6	5	6	7
o:	13	13	12	11	10	9	8	7	8	8	8	7	6	5	6
t:	14	14	13	12	11	10	9	8	8	9	9	8	7	6	5

		y	o	u	-	s	h	o	u	l	d	-	n	o	t
:	-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
t:	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h:	2	0	0	0	0	0	0	1	1	1	1	1	1	1	1
o:	2	0	0	0	0	0	0	0	1	1	1	1	1	0	1
u:	2	0	2	0	1	1	1	1	0	1	1	1	1	1	1
-:	2	0	2	2	0	1	1	1	1	0	0	0	1	1	1
s:	2	0	2	2	2	0	1	1	1	1	0	0	0	0	0
h:	2	0	2	2	2	2	0	1	1	1	1	1	1	0	0
a:	2	0	2	2	2	2	2	0	0	0	0	0	0	0	0
l:	2	0	2	2	2	2	2	0	0	0	1	1	1	1	1
t:	2	0	2	2	2	2	2	0	0	0	0	0	0	0	0
-:	2	0	2	2	0	2	2	0	0	0	0	0	1	1	1
n:	2	0	2	2	2	2	2	0	0	0	0	2	0	1	1
o:	2	0	0	2	2	2	2	0	0	0	0	2	2	0	1
t:	2	0	2	2	2	2	2	2	0	0	0	2	2	2	0

Dyn. Programming Version

```
int string_compare(char *s, char *t)
{
     int i,j,k;              /* counters */
     int opt[3];             /* cost of the three options */

     for (i=0; i<MAXLEN; i++) {
          row_init(i);
          column_init(i);
     }

     for (i=1; i<strlen(s); i++)
          for (j=1; j<strlen(t); j++) {
                  opt[MATCH] = m[i-1][j-1].cost + match(s[i],t[j]);
                  opt[INSERT] = m[i][j-1].cost + indel(t[j]);
                  opt[DELETE] = m[i-1][j].cost + indel(s[i]);

                  m[i][j].cost = opt[MATCH];
                  m[i][j].parent = MATCH;
                  for (k=INSERT; k<=DELETE; k++)
                       if (opt[k] < m[i][j].cost) {
                               m[i][j].cost = opt[k];
                               m[i][j].parent = k;
                       }
          }

     goal_cell(s,t,&i,&j);
     return( m[i][j].cost );
}
```

Reconstruction the Path

- Walking backward from the goal state
- Parent [i,j] tells us whether the transform at (i,j) was MATCH (or SUBSTITUTION), INSERT or DELETE

```
reconstruct_path(char *s, char *t, int i, int j)
{
        if (m[i][j].parent == -1) return;

        if (m[i][j].parent == MATCH) {
                reconstruct_path(s,t,i-1,j-1);
                match_out(s, t, i, j);
                return;
        }
        if (m[i][j].parent == INSERT) {
                reconstruct_path(s,t,i,j-1);
                insert_out(t,j);
                return;
        }
        if (m[i][j].parent == DELETE) {
                reconstruct_path(s,t,i-1,j);
                delete_out(s,i);
                return;
        }
}
```

Elevator Optimization

- We work in a tall building with a slow elevator
  - The lobby is at the 0^{th} floor
  - Offices are located from the 1^{st} floor
- The riders all enter their intended destinations at the beginning of the trip
- Then, the elevator decides which floors it will stop at along the way
- You limit the elevator to making at most k stops on any given run, but select the floors so as to minimize the total number of floors people have to walk either up or down

Elevator Optimization

- Let m[i][j] denote the minimum cost of serving all the riders using exactly j stops, the last of which is at floor i
- $m_{i,j+1} = \min_{k=0 \text{ to } i}(m_{k,j} - \text{floors-walked}(k, \infty) + \text{floors-walked}(k,i) + \text{floors-walked}(i, \infty))$

Is Bigger Smarter?

Some people think that the bigger an elephant is, the smarter it is. To disprove this, you want to analyze a collection of elephants and place as large a subset of elephants as possible into a sequence whose weights are increasing but IQ's are decreasing.

Input

The input will consist of data for a bunch of elephants, at one elephant per line terminated by the end-of-file. The data for each particular elephant will consist of a pair of integers: the first representing its size in kilograms and the second representing its IQ in hundredths of IQ points. Both integers are between 1 and 10,000. The data contains information on at most 1,000 elephants. Two elephants may have the same weight, the same IQ, or even the same weight and IQ.

Output

The first output line should contain an integer n, the length of elephant sequence found. The remaining n lines should each contain a single positive integer representing an elephant. Denote the numbers on the ith data line as $W[i]$ and $S[i]$. If these sequence of n elephants are $a[1]$, $a[2]$,..., $a[n]$ then it must be the case that

$$W[a[1]] < W[a[2]] < ... < W[a[n]] \quad \text{and} \quad S[a[1]] > S[a[2]] > ... > S[a[n]]i$$

In order for the answer to be correct, n must be as large as possible. All inequalities are strict: weights must be strictly increasing, and IQs must be strictly decreasing.

Your program can report any correct answer for a given input.

Is Bigger Smarter?

Sample Input

```
6008 1300
6000 2100
500 2000
1000 4000
1100 3000
6000 2000
8000 1400
6000 1200
2000 1900
```

Sample Output

```
4
4
5
9
7
```