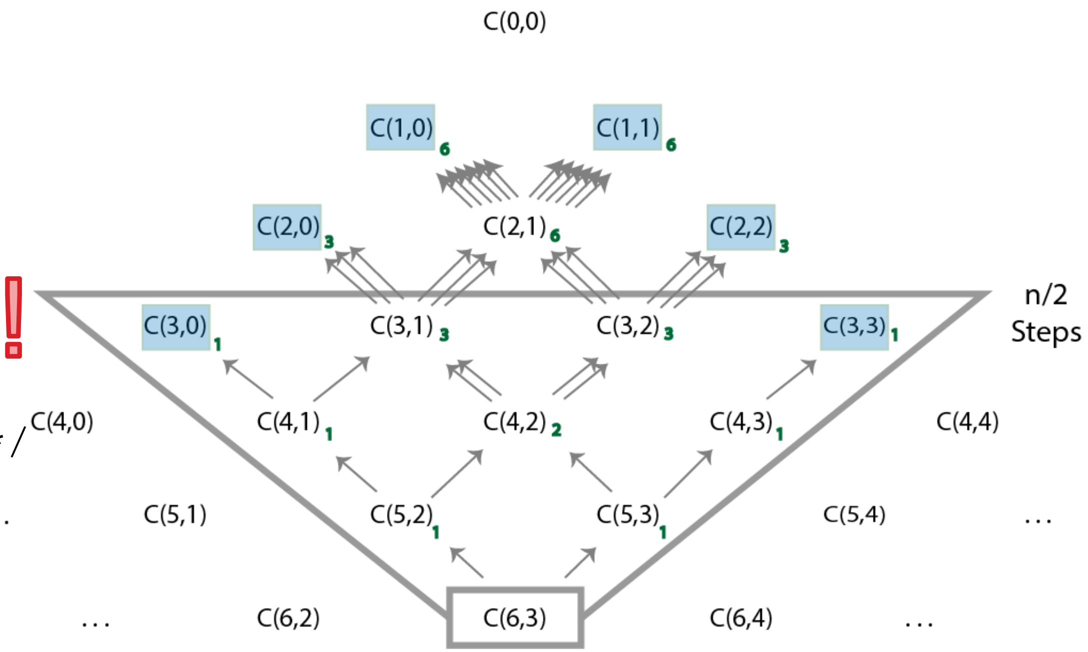
Problem Solving
Dynamic Programming

Review: Computing Binomial Coefficient

Store Partial Results!!!




```
#define MAXN 100 /* largest n or m */
long binomial_coefficient(n,m)
int n,m; /* computer n choose m */
{
   int i,j; /* counters */
   long bc[MAXN][MAXN]; /* table of binomial coefficients */

   for (i=0; i<=n; i++) bc[i][0] = 1;
   for (j=0; j<=n; j++) bc[j][j] = 1;

   for (i=1; i<=n; i++)
       for (j=1; j<i; j++)
        bc[i][j] = bc[i-1][j-1] + bc[i-1][j];

   return( bc[n][m] );
}
```

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Dynamic Programming

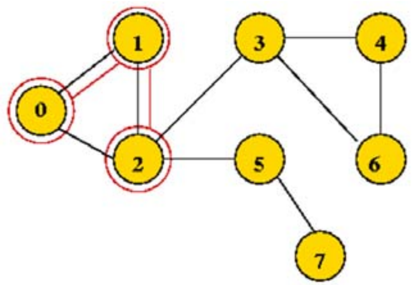
- Dynamic programming is a very powerful, general tool for solving optimization problems on left-right-ordered items such as character strings.
- Once understood it is relatively easy to apply, but many people have trouble understanding it.
- Start by reviewing the binomial coefficient function in the combinatorics section, as an example of how we stored partial results to help us compute what we were looking for.
- Floyd's all-pairs shortest-path algorithm discussed in the graph algorithms chapter is another example of dynamic programming.

Greedy Algorithms

- *Greedy* algorithms focus on making the best local choice at each decision point.
- In the absence of a correctness proof greedy algorithms are very likely to fail.
- Example: A natural way to compute a shortest path from x to y might be to walk out of x, repeatedly following the cheapest edge until we get to y. Natural, but wrong!
- Dynamic programming gives us a way to design custom algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency).

Evaluating Recurrence Relations

- Dynamic programming algorithms are defined by recursive algorithms/functions that describe the solution to the entire problem in terms of solutions to smaller problems.
- Backtracking is one such recursive procedure we have seen, as is depth-first search in graphs.
- Efficiency in any such recursive algorithm requires storing enough information to avoid repeating computations we have done before.
- Depth-first search in graphs is efficient because we mark the vertices we have visited so we don't visit them again.

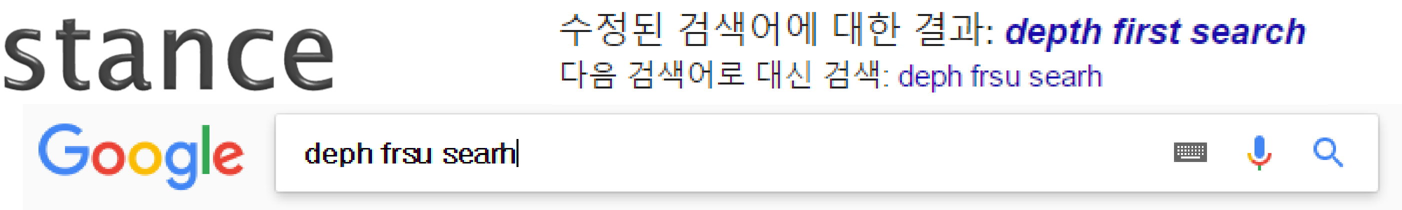


- After visiting 0, 1, and 2
- Red circles mark the visited nodes

Dynamic Programming

- Dynamic programming is a technique for efficiently implementing a recursive algorithm by storing partial results.
- The trick is to see that the naive recursive algorithm repeatedly computes the same subproblems over and over and over again.
- If so, storing the answers to them in a table instead of recomputing can lead to an efficient algorithm.
- Thus we must first hunt for a correct recursive algorithm – later we can worry about speeding it up by using a results matrix.

Edit Distance



- Misspellings and changes in word usage (“Thou shalt not kill” morphs into “You should not murder.”) make *approximate pattern matching* an important problem.
- A reasonable distance measure minimizes the cost of the *changes* which have to be made to convert one string to another.
- There are three natural types of changes:
  - *Substitution* – Change a single character from pattern s to a different character in text t, such as changing “shot” to “spot”.
  - *Insertion* – Insert a single character into pattern s to help it match text t, such as changing “ago” to “agog”.
  - *Deletion* – Delete a single character from pattern s to help it match text t, such as changing “hour” to “our”.

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

General Definition of Edit Distance

$$b = b_1 \ldots b_m \longrightarrow a = a_1 \ldots a_n$$

The edit distance from b to a is given by d_{mn}, defined by the recurrence

$$d_{i0} = \sum_{k=1}^{i} w_{\mathrm{del}}(b_k), \qquad \text{for } 1 \le i \le m$$

$$d_{0j} = \sum_{k=1}^{j} w_{\mathrm{ins}}(a_k), \qquad \text{for } 1 \le j \le n$$

$$d_{ij} = \begin{cases} d_{i-1,j-1} & \text{for } a_j = b_i \\ \min \begin{cases} d_{i-1,j} + w_{\mathrm{del}}(b_i) \\ d_{i,j-1} + w_{\mathrm{ins}}(a_j) \\ d_{i-1,j-1} + w_{\mathrm{sub}}(a_j, b_i) \end{cases} & \text{for } a_j \ne b_i \end{cases} \qquad \text{for } 1 \le i \le m, 1 \le j \le n.$$

Edit operations
with minimum cost

H	O	M	*	E
|	|	|	|	|
H	O	U	S	E
		s	i	

vs.

H	O	M	*	*	E
|	|	|	|	|	|
H	O	*	U	S	E
		d	i	i	

- Properly posing the question of string similarity requires us to set the cost of each of these string transform operations.
- Setting each operation to cost one step defines the *edit distance* between two strings.

Example

b1	b2	b3	b4
H	O	M	E

a1	a2	a3	a4	a5
H	O	U	S	E

- b:HOME => a:HOUSE
- Goal: compute $d_{4,5}$
- $d_{4,5}$: $b_4 = a_5 \rightarrow d_{3,4}$?
- $d_{3,4}$: $b_3 \neq a_4$
  - Delete: $d_{2,4}+w_{del}$('M')
  - Insert: $d_{3,3}+w_{ins}$('S')
  - Substitute: $d_{2,3}+w_{sub}$('M','S')

  minimum
- $d_{2,4}$: $b_2 \neq a_4$
  - Delete: $d_{1,4}+w_{del}$('O')
  - Insert: $d_{2,3}+w_{ins}$('U')
  - Substitute: $d_{1,3}+w_{sub}$('O','U')

  minimum
- …

Recursive Algorithm

- We can compute the edit distance with recursive algorithm using the observation that the last character in the string must either be matched, substituted, inserted, or deleted.
- *If* we knew the cost of editing the three pairs of smaller strings, we could decide which option leads to the best solution and choose that option accordingly.
- We *can* learn this cost, through the magic of recursion:

```
#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 );
}
```

Speeding it Up

- This program is absolutely correct but impossibly slow to compare two 12-character strings! It takes exponential time because it recomputes values again and again.
- But there can only be $|s| \cdot |t|$ possible unique recursive calls, since there are only that many distinct (i, j) pairs to serve as the parameters of recursive calls.
- By storing the values for each of these (i, j) pairs in a table, we can just look them up as needed.
- The table is a two-dimensional matrix m where each of the $|s| \cdot |t|$ cells contains the cost of the optimal solution of this subproblem, as well as a parent pointer explaining how we got to this location:

```
typedef struct {
    int cost; /* cost of reaching this cell */
    int parent; /* parent cell */
} cell;
cell m[MAXLEN+1][MAXLEN+1]; /* dynamic programming table */
```

- The dynamic programming version has three differences from the recursive version:
  - It gets its intermediate values using table lookup instead of recursive calls.
  - It updates the parent field of each cell, which will enable us to reconstruct the edit-sequence later.
  - It is instrumented using a more general goal cell() function instead of just returning m[$|s|$][$|t|$].cost. This will enable us to apply this routine to a wider class of problems.

- Be aware that we adhere to certain unusual string and index conventions in the following routines.
- In particular, we assume that each string has been padded with an initial blank character, so the first real character of string s sits in $s[1]$.

S[0]	S[1]	S[2]	S[3]
H	O	M	E

S[0]	S[1]	S[2]	S[3]	S[4]
~~b~~	H	O	M	E

```
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 );
}
```

Example

- To determine the value of cell (i, j) we need three values sitting and waiting for us, namely, the cells (i–1, j–1), (i, j–1), and (i–1, j).
- Any evaluation order with this property will do, including row-major order. “thou shalt not” goes to “you should not” in 5 moves: DSMMMMMISMSMMMM

cost matrix

		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

parent matrix

		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

Reconstructing the Path

- The possible solutions are described by paths through the dynamic programming matrix, starting from the initial configuration (0, 0) to the final goal state ($|s|$, $|t|$).
- Reconstructing these decisions is done by walking backward from the goal state, following the parent pointer to an earlier cell.
- The parent field for m[i,j] tells us whether the transform at (i, j) was MATCH, INSERT, or DELETE.

Walking backward reconstructs the solution in reverse order. However, clever use of recursion can do the reversing for us:

```
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;
    }
}
```

Customizing Edit Distance

- *Table Initialization* – The functions *row_init*() and *column_init*() initialize the zeroth row and column of the dynamic programming table, respectively.

```
row_init(int i)
{
  m[0][i].cost = i;
  if (i>0)
        m[0][i].parent =  INSERT;
  else
        m[0][i].parent = -1;
}
```

```
column_init(int i)
{
  m[i][0].cost = i;
  if (i>0)
        m[i][0].parent = DELETE;
  else
        m[0][i].parent = -1;
}
```

- *Penalty Costs* – The functions *match*(c,d) and *indel*(c) present the costs for transforming character c to d and inserting/deleting character c.
  - For standard edit distance, match costs 0 for matching characters, and 1 otherwise, while indel returns 1.

```
int match(char c, char d)
{
  if (c == d) return(0);
  else return(1);
}
```

```
int indel(char c)
{
  return(1);
}
```

- *Goal Cell Identification* – The function goal cell returns the indices of the cell marking the endpoint of the solution.
  - For edit distance, this is defined by the length of the two input strings.

```
goal_cell(char *s, char *t, int *i, int *j)
{
  *i = strlen(s) - 1;
  *j = strlen(t) - 1;
}
```

- *Traceback Actions* – The functions match out, insert out, and delete out perform the appropriate actions for each edit-operation during traceback.
  - For edit distance, this might mean printing out the name of the operation or character involved, as determined by the needs of the application.

```
insert_out(char *t, int j)
{
        printf("I");
}

delete_out(char *s, int i)
{
        printf("D");
}
```

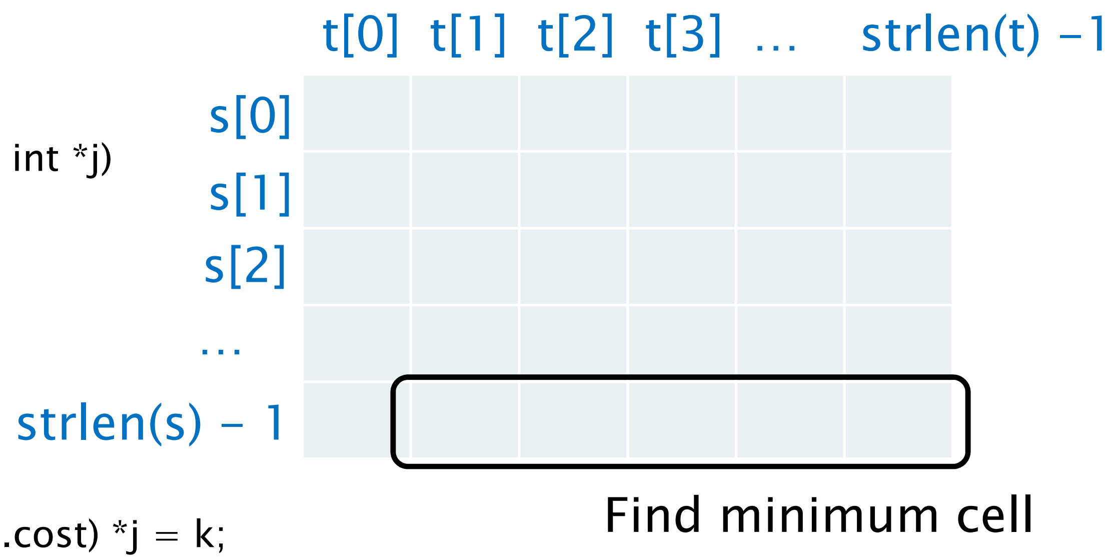
```
match_out(char *s, char *t,
          int i, int j)
{
    if (s[i]==t[j]) printf("M");
    else printf("S");
}
```

Substring Matching

- Suppose that we want to find where a short pattern *s* best occurs within a long text *t*, say, searching for “Skiena” in all its misspellings (Skienna, Skena, Skina, . . .).
- We want an edit distance search where the cost of starting the match is independent of the position in the text, so that a match in the middle is not prejudiced against.
- Likewise, the goal state is not necessarily at the end of both strings, but the cheapest place to match the entire pattern somewhere in the text.
- Modifying these two functions gives us the correct solution:

```
row_init(int i)
{
    m[0][i].cost = 0; /* note change */
    m[0][i].parent = -1; /* note change */
}

goal_cell(char *s, char *t, int *i, int *j)
{
    int k; /* counter */
    *i = strlen(s) - 1;
    *j = 0;
    for (k=1; k<strlen(t); k++)
        if (m[*i][k].cost < m[*i][*j].cost) *j = k;
}
```



Longest Common Subsequence

- Often we are interested in finding the longest scattered string of characters which is included within both words.
- The *longest common subsequence* (LCS) between "d<u>e</u>mo<u>c</u>r<u>a</u>t" and "r<u>e</u>publi<u>ca</u>n" is *eca*.
- A common subsequence is defined by identical-character matches in an edit trace.
- To maximize such traces, we must prevent substitution of non-identical characters by changing the match-cost function:

```
int match(char c, char d)
{
   if (c == d) return(0);
   else return(MAXLEN);
}
```

Maximum Monotone Subsequence

- A numerical sequence is *monotonically increasing* if the ith element is at least as big as the $(i-1)$st element.
- The *maximum monotone subsequence* problem seeks to delete the fewest number of elements from an input string S to leave a monotonically increasing subsequence.
- Thus a longest increasing subsequence of "243517698" is "23568."
- In fact, this is just a longest common subsequence problem, where the second string is the elements of S sorted in increasing order.
- The trick to using edit distance is observing that your problem is just a special case of approximate string matching.