Dynamic Programming Notes

- The Algorithm Design Manual Steven S. Skiena

1. Greedy algorithms that make the best local decision at each step are typically efficient but usually do not guarantee global optimality.
2. Exhaustive search algorithms that try all possibilities and select the best always produce the optimum result, but usually at a prohibitive cost in terms of time complexity.
3. Dynamic Programming gives us a way to design custom algorithms that systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency).
4. Dynamic programming is a technique for efficiently implementing a recursivealgorithm by storing partial results.
5. Dynamic programming is generally the right method for optimization problems on combinatorial objects that have an inherent left to right order among components. Left-to-right objects includes: character strings, rooted trees, polygons, and integer sequences.
6. Dynamic programming is best learned by carefully studying examples until things start to click.

8.1 Caching vs. Computation

1. Dynamic programming is essentially a tradeoff of space for time.

8.1.1 Fibonacci Numbers by Recursion

1. $F_n = F_{n-1} + F_{n-2}$

   with basis cases $F_0 = 0$ and $F_1 = 1$. Thus, $F_2 = 1$, $F_3 = 2$, and the series continues {3,5,8,13,21,34,55,89,144,...}.

2. A recursive function algorithm written in C looks like this:

```
long fib_r(int n)
{
    if (n == 0) return(0);
    if (n == 1) return(1);
    return(fib_r(n-1) + fib_r(n-2));
}
```

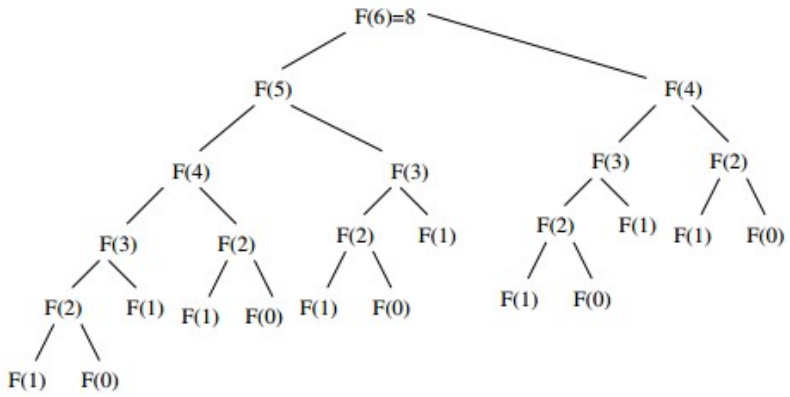

Figure 8.1: The computation tree for computing Fibonacci numbers recursively

3. This tree is evaluated in a depth-first fashion, as are all recursive algorithms.
4. Note that F(4) is computed on both sides of the recursion tree.

8.1.2 Fibonacci Numbers by Caching

1. We can explicitly store (or cache) the results of each Fibonacci computation F(k) in a table data structure indexed by the parameter k.
2. The key to avoiding recomputation is to explicitly check for the value before trying to compute it:

```
#define MAXN 45 /* largest interesting n */
#define UNKNOWN -1 /* contents denote an empty cell */
long f[MAXN+1]; /* array for caching computed fib values */

long fib_c(int n)
{
    if (f[n] == UNKNOWN)
    f[n] = fib_c(n-1) + fib_c(n-2);
    return(f[n]);
}

long fib_c_driver(int n)
{
    int i; /* counter */
    f[0] = 0;
    f[1] = 1;
    for (i=2; i<=n; i++) f[i] = UNKNOWN;
    return(fib_c(n));
```

```
}
```

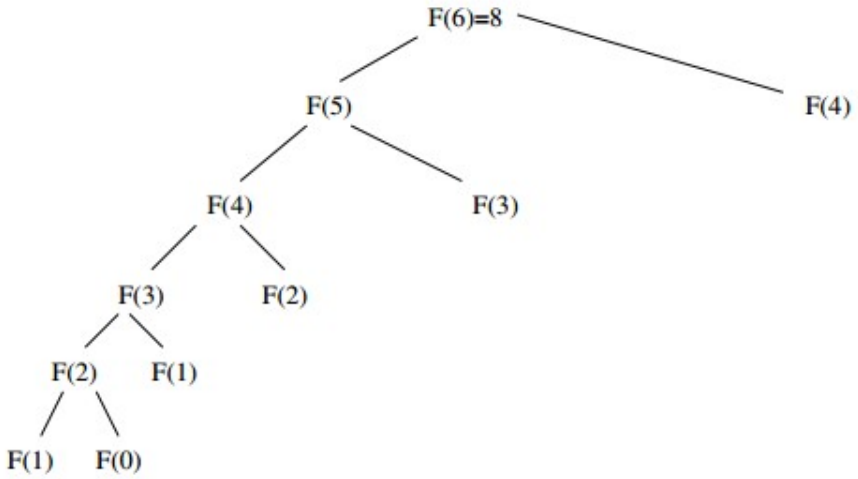

Figure 8.2: The Fibonacci computation tree when caching values

3. To compute *F(n)*, we call `fib_c_driver(n)`. This initializes our cache to the two values we initially know (*F(0)* and *F(1)*) as well as the `UNKNOWN` flag for all the rest we don't.
4. This cached version runs instantly up to the largest value that can fit in a long integer.
5. Tt computes *F(n)* in linear time (in other words, *O(n)* time) because the recursive function `fib_c(k)` is called exactly twice for each value $0 \leq k \leq n$.
6. Storing partial results would have done absolutely no good for such recursive algorithms as *quicksort*, *backtracking*, and *depth-first search* because all the recursive calls made in these algorithms have *distinct parameter* values. It doesn't pay to store something you will never refer to again.
7. Caching makes sense only when the space of distinct parameter values is modest enough that we can afford the cost of storage.

8.1.3 Fibonacci Numbers by Dynamic Programming

1. We can calculate Fn in linear time more easily by explicitly specifying the order of evaluation of the recurrence relation:

```
long fib_dp(int n)
{
    int i; /* counter */
    long f[MAXN+1]; /* array to cache computed fib values */
    f[0] = 0;
    f[1] = 1;
    for (i=2; i<=n; i++) f[i] = f[i-1]+f[i-2];
```

```
    return(f[n]);
}
```

2. Each of the n values is computed as the simple sum of two integers in total *O(n)* time and space.
3. More careful study shows that we do not need to store all the intermediate values for the entire period of execution. Because the recurrence depends on two arguments, we only need to retain the last two values we have seen:

```
long fib_ultimate(int n)
{
    int i; /* counter */
    long back2=0, back1=1; /* last two values of f[n] */
    long next; /* placeholder for sum */

    if (n == 0) return (0);

    for (i=2; i<n; i++) {
        next = back1+back2;
        back2 = back1;
        back1 = next;
    }
    return(back1+back2);
}
```

8.1.4 Binomial Coefficients

1. The binomial coefficients are the most important class of counting numbers, where $\binom{n}{k}$ counts the number of ways to choose *k* things out of *n* possibilities.
2. $\binom{n}{k} = n!/((n-k)!k!)$
3. A more stable way to compute binomial coefficients is using the recurrence relation implicit in the construction of Pascal's triangle:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

4. Each number is the sum of the two numbers directly above it. The recurrence relation implicit in this is that

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

```
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] );
}
```

8.2 Approximate String Matching

1. To deal with inexact string matching, we must first define a cost function telling us how far apart two strings are—i.e., a distance measure between pairs of strings.
2. A reasonable distance measure reflects the number of changes that must be made to convert one string to another.
3. There are three natural types of changes:
    1. *Substitution* – Replace a single character from pattern P with a different character in text T, such as changing “shot” to “spot.”
    2. *Insertion* – Insert a single character into pattern P to help it match text T, such as changing “ago” to “agog.”
    3. *Deletion* – Delete a single character from pattern P to help it match text T, such as changing “hour” to “our.

8.2.1 Edit Distance by Recursion

1. Let *D[i,j]* be the minimum number of differences between P_1,P_2, . . . , P_i and the segment of *T* ending at *j*.
2. *D[i,j]* is the *minimum* of the three possible ways to extend smaller strings:

1. If ($P_i = T_j$), then $D[i-1,j-1]$, else $D[i-1,j-1]+1$. This means we either match or substitute the *ith* and *jth* characters, depending upon whether the tail characters are the same.
2. $D[i-1,j]+1$. This means that there is an extra character in the pattern to account for, so we do not advance the text pointer and pay the cost of an insertion.
3. $D[i,j-1]+1$. This means that there is an extra character in the text to remove, so we do not advance the pattern pointer and pay the cost of a deletion.

```
#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 );
    }
```

3. It takes exponential time because it recomputes values again and again and again.

8.2.2 Edit Distance by Dynamic Programming

1. There can only be $|P| \cdot |T|$ possible unique recursive calls, since there are only that many distinct (i,j) pairs to serve as the argument parameters of recursive calls.
2. By storing the values for each of these (i,j) pairs in a table, we just look them up as needed and avoid recomputing them.
3. Our dynamic programming implementation has three differences from the recursive version. First, it gets its intermediate values using table lookup instead of recursive calls.

4. Second, it updates the parent field of each cell, which will enable us to reconstruct the edit sequence later.
5. Third, it is implemented using a more general goal_cell() function instead of just returning m[|P|][|T|].cost. This will enable us to apply this routine to a wider class of problems.

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

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

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 );
}
```

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

Figure 8.4: Example of a dynamic programming matrix for editing distance computation, with the optimal alignment path highlighted in bold

8.2.3 Reconstructing the Path

1. The string comparison function returns the cost of the optimal alignment, but not the alignment itself.
2. The possible solutions to a given dynamic programming problem are described by paths through the dynamic programming matrix, starting from the initial configuration (the pair of empty strings (0,0)) down to the final goal state (the pair of full strings (|P|,|T|))
3. Reconstructing these decisions is done by walking backward from the goal state,following the parent pointer back to an earlier cell. We repeat this process untilwe arrive back at the initial cell.
4. The parent field for m[i,j] tells us whether the operation at *(i,j)* was MATCH, INSERT, or DELETE.
5. Tracing back through the parent matrix in Figure 8.5 yields the edit sequence DSMMMMMISMSMMMM from “thou-shaltnot” to “you-should-not”—meaning delete the first “t”, replace the “h” with “y”, match the next five characters before inserting an “o”, replace “a” with “u”, and finally replace the “t” with a “d.

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

Figure 8.5: Parent matrix for edit distance computation, with the optimal alignment path highlighted in bold

```
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;
    }
}
```

8.2.4 Varieties of Edit Distance

1. The string_compare and path reconstruction routines reference several functions that we have not yet defined. These fall into four categories:

    1. *Table Initialization* – The functions row_init and column_init initialize the zeroth row and column of the dynamic programming table, respectively. For the string edit distance problem, cells *(i,0)* and *(0,i)* correspond to matching length-*i* strings against the empty string. This requires exactly *i* insertions/deletions, so the definition of these functions is clear:

```
 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[i][0].parent = -1;
}
```

2. *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 should cost nothing if the characters are identical, and 1 otherwise; while indel returns 1 regardless of what the argument is.

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

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

3. *Goal Cell Identification* – The function goal_cell returns the indices of the cell marking the endpoint of the solution.

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

4. *Traceback Actions* – The functions match_out, insert_out, and delete_out perform the appropriate actions for each edit operation during traceback.

```
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");
}
```

2. Several important problems can now be solved as special cases of edit distance using only minor changes to some of these stub functions:

3. *Substring Matching* – Suppose that we want to find where a short pattern*P* best occurs within a long text *T*—say, searching for “Skiena” in all itsmisspellings (Skienna, Skena, Skina, .. .) within a long file.

    1. Matching any scattered $\cdots S \cdots k \cdots i \cdots e \cdots n \cdots a$ and deleting the rest yields an optimal solution.

    2. 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.

    3. Now 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;
}
```

4. *Longest Common Subsequence* - A common subsequence is defined by all the identical-character matches in an edit trace. To maximize the number of such matches, we must prevent substitution of nonidentical characters.

    1. With substitution forbidden, the only way to get rid of the noncommon subsequence is through insertion and deletion.

    2. The minimum cost alignment has the fewest such “in-dels”, so it must preserve the longest common substring.

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

5. *Maximum Monotone Subsequence* – A numerical sequence is *monotonically increasing* if the *i*th element is at least as big as the *(i − 1)*st element.

    1. The *maximum monotone subsequence* problem seeks to delete the fewest number of elements from an input string *S* to leave a monotonically increasing subsequence. A longest increasing subsequence of 243517698 is 23568.

    2. In fact, this is just a longest common subsequence problem, where the second string is the elements of S sorted in increasing order.

8.3 Longest Increasing Sequence

1. We distinguish an increasing sequence from a run, where the elements must be physical neighbors of each other. The selected elements of both must be sorted in increasing order from left to right.
2. For example, consider the sequence S = {2, 4, 3,5,1, 7, 6, 9, 8} The longest increasing subsequence of S has length 5, including {2,3,5,6,8}. In fact, there are eight of this length (can you enumerate them?). There are four longest increasing runs of length 2: (2, 4), (3, 5), (1, 7), and (6, 9).
3. Define l_i to be the length of the longest sequence ending with s_i.
4. The longest increasing sequence containing the *n*th number will be formed by appending it to the longest increasing sequence to the left of *n* that ends on a number smaller than $s_{n.}$.
5. $l_i = \max_{0<j<i} l_j + 1$, where $(s_j < s_i)$,

   $l_0 = 0$

   Here is the table associated with our previous example:

Sequence s_i	2	4	3	5	1	7	6	9	8
Length l_i	1	2	3	3	1	4	4	5	5
Predecessor p_i	–	1	1	2	–	4	4	6	6

6. For each element s_i, we will store its *predecessor*— the index p_i of the element that appears immediately before s_i in the longest increasing sequence ending at s_i.
7. Since all of these pointers go towards the left, it is a simple matter to start from the last value of the longest sequence and follow the pointers so as to reconstruct the other items in the sequence.
8.

8.4 War Story: Evolution of the Lobster

8.5 The Partition Problem

1. *Problem*: Integer Partition without Rearrangement

   Input: An arrangement *S* of nonnegative numbers $\{s_1, \ldots, s_n\}$ and an integer *k*.

   Output: Partition *S* into *k* or fewer ranges, to minimize the maximum sum over all the ranges, without reordering any of the numbers.
2. This so-called *linear partition* problem arises often in parallel process. We seek to balance the work done across processors to minimize the total elapsed run time.
3. Let us define *M[n,k]* to be the minimum possible cost over all partitionings of $\{s_1, \ldots, s_n\}$ into k ranges, where the cost of a partition is the largest sum of elements in one of its parts.

$$M[n,k] = \min_{i=1}^{n} \max(M[i,k-1], \sum_{j=i+1}^{n} s_j)$$

4. For this problem, the smallest reasonable value of the first argument is n = 1, meaning that the first partition consists of a single element.
5. The smallest reasonable value of the second argument is k = 1, implying that we do not partition S at all.

$$M[1,k] = s_1, \text{ for all } k > 0 \text{ and,}$$

$$M[n,1] = \sum_{i=1}^{n} s_i$$

6. A total of $k \cdot n$ cells exist in the table. If filling each of *kn* boxes takes at most n^2 time per box, the total recurrence can be computed in $O(kn^3)$ time.

   `partition(int s[], int n, int k)`

```
{
    int m[MAXN+1][MAXK+1]; /* DP table for values */
    int d[MAXN+1][MAXK+1]; /* DP table for dividers */
    int p[MAXN+1]; /* prefix sums array */
    int cost; /* test split cost */
    int i,j,x; /* counters */

    p[0] = 0; /* construct prefix sums */
    for (i=1; i<=n; i++) p[i]=p[i-1]+s[i];

    for (i=1; i<=n; i++) m[i][1] = p[i]; /* initialize boundaries */
    for (j=1; j<=k; j++) m[1][j] = s[1];

    for (i=2; i<=n; i++) /* evaluate main recurrence */
        for (j=2; j<=k; j++) {
            m[i][j] = MAXINT;
            for (x=1; x<=(i-1); x++) {
                cost = max(m[x][j-1], p[i]-p[x]);
                if (m[i][j] > cost) {
                    m[i][j] = cost;
                    d[i][j] = x;
                }
            }
        }

    reconstruct_partition(s,d,n,k); /* print book partition */
}
```

M	k		
n	1	2	3
1	1	1	1
1	2	1	1
1	3	2	1
1	4	2	2
1	5	3	2
1	6	3	2
1	7	4	3
1	8	4	3
1	9	5	3

D	k		
n	1	2	3
1	–	–	–
1	–	1	1
1	–	1	2
1	–	2	2
1	–	2	3
1	–	3	4
1	–	3	4
1	–	4	5
1	–	4	6

M	k		
n	1	2	3
1	1	1	1
2	3	2	2
3	6	3	3
4	10	6	4
5	15	9	6
6	21	11	9
7	28	15	11
8	36	21	15
9	45	24	17

D	k		
n	1	2	3
1	–	–	–
2	–	1	1
3	–	2	2
4	–	3	3
5	–	3	4
6	–	4	5
7	–	5	6
8	–	5	6
9	–	6	7

Figure 8.8: Dynamic programming matrices M and D for two input instances. Partitioning $\{1,1,1,1,1,1,1,1,1\}$ into $\{\{1,1,1\},\{1,1,1\},\{1,1,1\}\}$ (l) . Partitioning $\{1,2,3,4,5,6,7,8,9\}$ into $\{\{1,2,3,4,5\},\{6,7\},\{8,9\}\}$ (r).

7. The second matrix, D, is used to reconstruct the optimal partition.
8. Whenever we update the value of *M[i,j]*, we record which divider position was required to achieve that value.

9. To reconstruct the path used to get to the optimal solution, we work backward from *D[n,k]* and add a divider at each specified position. This backwards walking is best achieved by a recursive subroutine:

```
reconstruct_partition(int s[],int d[MAXN+1][MAXK+1], int n, int k)
{
    if (k==1)
        print_books(s,1,n);
    else {
        reconstruct_partition(s,d,d[n][k],k-1);
        print_books(s,d[n][k]+1,n);
    }
}

print_books(int s[], int start, int end)
{
    int i; /* counter */

    for (i=start; i<=end; i++) printf(" %d ",s[i]);
    printf("\n");
}
```

8.6 Parsing Context-Free Grammars

1. We assume that the text string S has length n while the grammar G itself is of constant size.
2. This is fair, since the grammar defining a particular programming language (say C or Java) is of fixed length regardless of the size of the program we are trying to compile.
3. Further, we assume that the definitions of each rule are in *Chomsky normal form.* This means that the right sides of every nontrivial rule consists of (a) exactly two nonterminals, i.e. $X \rightarrow Y\,Z$, or (b) exactly one terminal symbol, $X \rightarrow \alpha$.
4. The key observation is that the rule applied at the root of the parse tree (say $X \rightarrow Y\,Z$) splits *S* at some position *i* such that the left part of the string (*S[1,i]*) must be generated by nonterminal *Y* , and the right part (*S[i + 1,n]*) generated by *Z*.
5. Define *M[i,j,X]* to be a boolean function that is true iff substring *S[i,j]* is generated by nonterminal *X*.
6. This is true if there exists a production $X \rightarrow Y\,Z$ and breaking point *k* between *i* and j such that the left part generates *Y* and the right part *Z*.

$$M[i,j,X] = \bigvee_{(X \rightarrow Y\,Z) \in G} \left(\bigvee_{i=k}^{j} M[i,k,Y] \cdot M[k+1,j,Z] \right)$$

where ∨ denotes the logical *or* over all productions and split positions, and · denotes the logical *and* of two boolean values.

7. The one-character terminal symbols define the boundary conditions of the recurrence.
8. In particular, *M[i, i, X]* is true iff there exists a production $X \rightarrow \alpha$ such that $S[i] = \alpha$.

8.6.1 Minimum Weight Triangulation

1. A triangulation of a polygon $P = \{v_1, \ldots, v_n, v_1\}$ is a set of nonintersecting diagonals that partitions the polygon into triangles.
2. We say that the *weight* of a triangulation is the sum of the lengths of its diagonals.
3. We seek to find its minimum weight triangulation for a given polygon *p*.
4. Let *T[i,j]* be the cost of triangulating from vertex v_i to vertex v_j, ignoring the length of the chord d_{ij} from v_i to v_j.

$$T[i,j] = \min_{k=i+1}^{j-1} (T[i,k] + T[k,j] + d_{ik} + d_{kj})$$

5. The basis condition applies when *i* and *j* are immediate neighbors, as *T[i,i+1]* = 0

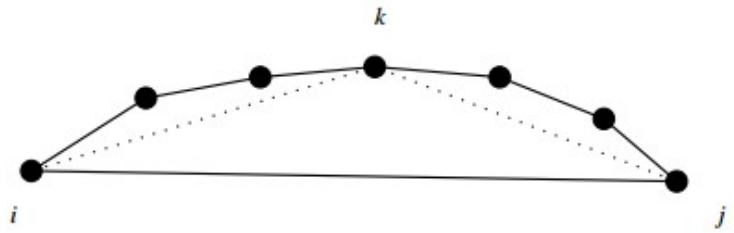

Figure 8.11: Selecting the vertex k to pair with an edge (i, j) of the polygon

```
Minimum-Weight-Triangulation(P)
    for i = 1 to n − 1 do T[i, i + 1] = 0
    for gap = 2 to n − 1
        for i = 1 to n − gap do
            j = i + gap
            T[i, j] = min_{k=i+1}^{j−1} (T[i, k] + T[k, j] + d_{P_i,P_k} + d_{P_k,P_j})
    return T[1, n]
```

6. There are $\binom{n}{2}$ values of T, each of which takes $O(j - i)$ time if we evaluate the sections in order of increasing size. Since $j - i = O(n)$, complete evaluation takes $O(n^3)$ time and $O(n^2)$ space.

8.7 Limitations of Dynamic Programming: TSP

8.7.1 When are Dynamic Programming Algorithms Correct?

8.7.2 When are Dynamic Programming Algorithms Efficient?

8.8 War Story: What's Past is Prolog

8.9 War Story: Text Compression for Bar Codes

Reference: - The Algorithm Design Manual

Steven S. Skiena

https://programmercave0.github.io/

https://github.com/programmercave0

https://www.facebook.com/programmercave/

https://t.me/programmercave