Algorithmic techniques
— T.08 —

Summary:

- Divide-and-conquer (DaC) and the master theorem
- DaC examples
- Dynamic programming (DP)
- DP examples

Recommended bibliography for this lecture:

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Estruturas de Dados e Algoritmos em C**, António Adrego da Rocha, terceira edição, FCA.

Divide-and-conquer (DaC)

The **divide-and-conquer** algorithmic technique consists of solving a problem recursively using the following methodology:

Divide — If the problem is small enough, solve it directly. Otherwise, subdivide it into smaller instances (at least two, preferably of nearly equal size) of the same problem.

Conquer (recurse) — Solve each sub-problem using the same algorithm.

Combine — Construct the solution of the problem by combining the solutions of the sub-problems.

The Divide step can be trivial (almost nothing to do, just decide what subdivision to use), as in merge sort. The Combine step can also be trivial (nothing to do), as in quick sort. In general, both steps are nontrivial.

Let $T(n)$ be the effort required to solve a problem of size n, let $f(n)$ be the effort needed to perform steps Divide and Combine (again for a problem of size n), and let a be the number of sub-problems of size n/b that are done in the Conquer step. Assigning an effort of 1 to problems of size $n \leqslant n_0$, we have

$$T(n) = \begin{cases} 1 & \text{for } n \leqslant n_0; \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise.} \end{cases}$$

The so-called **master theorem** provides solutions to this recursion in some cases:

- If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
- If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some $k \geqslant 0$, then $T(n) = O(n^{\log_b a} \log^{k+1} n)$
- If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leqslant cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta\big(f(n)\big)$.

Divide-and-conquer (examples of application of the master theorem)

Merge sort is an example of a divide-and-conquer algorithm for which $f(n) = \Theta(n)$ and $a = b = 2$. The second case of the master theorem then says that $T(n) = O(n \log n)$.

In the improved multiplication method presented in lecture T.04, $f(n) = \Theta(n)$, because additions and subtractions are $O(n)$, $a = 3$, and $b = 2$. The first case of the master theorem then says that $T(n) = O(n^{\log_2 3})$.

In the improved matrix multiplication method also presented in lecture T.04, $f(n) = \Theta(n^2)$, because matrix additions and subtractions are $O(n^2)$, $a = 7$, and $b = 2$. The first case of the master theorem then says that $T(n) = O(n^{\log_2 7})$.

In the recursive exponentiation algorithms also presented in lecture T.04, and in the binary search algorithm presented in lecture T.06, $f(n) = \Theta(1)$, $a = 1$, and $b = 2$. The second case of the master theorem then says that $T(n) = O(\log n)$.

Homework: Apply the master theorem to each of the following recurrences:

$$T(n) = 2T\left(\tfrac{n}{2}\right) + n^4$$
$$T(n) = T\left(\tfrac{7n}{10}\right) + n$$
$$T(n) = 16T\left(\tfrac{n}{4}\right) + n^2$$
$$T(n) = 7T\left(\tfrac{n}{3}\right) + n^2$$
$$T(n) = 7T\left(\tfrac{n}{2}\right) + n^2$$
$$T(n) = 2T\left(\tfrac{n}{4}\right) + \sqrt{n}$$

Possible solution: In this case we have $a = 2$ — the factor before $T(\cdot)$ on the right hand side — $b = 2$ — the divisor of n inside $T(\cdot)$ — and $f(n) = n^4$. Since $\log_b a = \log_2 2 = 1$, we have $f(n) = \Theta(n^{1+3})$, that is, $\epsilon = 3$. Since $\epsilon > 0$, we are possibly in the last case of the master theorem. This will be so if and only if $af\left(\frac{n}{b}\right) \leqslant cf(n)$, for some $c < 1$. In this case $af\left(\frac{n}{b}\right) = \frac{1}{8}f(n)$, so any c larger than $\frac{1}{8}$ will do, and the third case of the master theorem can be applied. It follows that $T(n) = \Theta(n^4)$.

DaC examples (part 1a, selection)

Given an array of (unsorted) numbers, select the k-th smallest array element. For example, for $k = 0$ the smallest array element is the one selected, for k equal to the size of the array minus one the largest array element is the one selected, and for k equal to half the size of the array the median of the arrays elements is the one selected. By sorting the array in increasing order, a task that requires $O(n \log n)$ time, each selection takes $O(1)$ time.

If only one selection is needed, it is not necessary to sort the array. Using the divide-and-conquer paradigm, selection can be done in $O(n)$ average time. The following function does the job in $O(n^2)$ time (without sorting!), and so it should be used only for very small arrays:

```c
int select_v1(int *a,int n,int k)
{ // O(n^2) algorithm, could be improved to O(n log n)
  for(int i = 0;i < n;i++)
  {
    int rank = 0;
    for(int j = 0;j < n;j++)
      if(a[j] < a[i] || (a[j] == a[i] && j < i))
        rank++;
    if(rank == k)
      return a[i];
  }
  abort(); // if 0 <= k < n, this point cannot be reached
}
```

It will be used by the divide-and-conquer function of the next slide to deal with the terminal cases (i.e., the small ones). For that function, and assuming that the pivot splits the array in approximately equal-sized smaller and larger parts, we have $T(n) = T(n/2) + O(n)$, which, according to the master theorem, gives $T(n) = O(n)$.

DaC examples (part 1b, selection using DaC)

The following code solves the selection problem using the divide-and-conquer paradigm. Its main idea is the choose a pivot element at random, and to split the array into three parts: smaller than, equal to, and larger than the pivot (just like quick sort). If the desired selection lies on the smaller or larger parts, then the same problem is solved for the appropriate part (recursion!).

```
int select_v2(int *a,int n,int k)
{ // O(n) on average, at worst O(n^2)
  int i,j,pivot,n_smaller,n_larger,*aa;

  if(n <= 10)
    return select_v1(a,n,k);
  pivot = a[(int)rand() % n];                  // choose pivot at random
  for(n_smaller = n_larger = i = 0;i < n;i++) // count number of elements
    if(a[i] < pivot)      n_smaller++;        //   smaller than the pivot
    else if(a[i] > pivot) n_larger++;         //   larger than the pivot
  if(k >= n_smaller && k < n - n_larger)
    return pivot;                             // bingo!
  if(k < n_smaller)
  { // divide (keep only the elements smaller than the pivot) and conquer
    aa = (int *)malloc((size_t)n_smaller * sizeof(int));
    for(i = j = 0;i < n;i++) if(a[i] < pivot) aa[j++] = a[i];
    i = select_v2(aa,n_smaller,k);
  }
  else
  { // divide (keep only the elements larger than the pivot) and conquer
    aa = (int *)malloc((size_t)n_larger * sizeof(int));
    for(i = j = 0;i < n;i++) if(a[i] > pivot) aa[j++] = a[i];
    i = select_v2(aa,n_larger,k - (n - n_larger));
  }
  free(aa);
  return i;
}
```

DaC examples (part 2a, the maximum-sum sub-array problem)

Given an array of numbers, the maximum-sum sub-array problem consists of finding the maximum sum of consecutive elements of the array. Let $a[0], \ldots, a[n-1]$ be the elements of the array (of size n). Mathematically, the problem is to solve

$$\max_{0\leqslant i\leqslant j<n} \sum_{k=i}^{j} a[k] \qquad \text{and} \qquad \arg\max_{0\leqslant i\leqslant j<n} \sum_{k=i}^{j} a[k].$$

The notation $\arg\max$ denotes the argument where the maximum occurs, or one argument chosen arbitrarily if the maximum occurs in several places; in this case the argument is the pair (i, j), with $0 \leqslant i \leqslant j < n$. The following code solves this problem in $O(n^2)$ time:

```c
typedef struct { int first; int last; int sum; } ret_val;

ret_val max_sum_v1(int *a,int first,int one_after_last)
{
  ret_val r = { first,first,a[first] };

  for(int i = first;i < one_after_last;i++)
    for(int sum = 0,j = i;j < one_after_last;j++)
    {
      sum += a[j];
      if(sum > r.sum)
      {
        r.first = i;
        r.last = j;
        r.sum = sum;
      }
    }
  return r;
}
```

DaC examples (part 2b, the maximum-sum sub-array problem using DaC)

The maximum-sum sub-array either lies completely in the first half of the array, either lies completely in the second half of the array, or it must cross the middle of the array. This last case can be solved in $O(n)$ time, so the following code solves the problem in $O(n \log n)$ time using divide-and-conquer:

```
ret_val max_sum_v2(int *a,int first,int one_after_last)
{
  int i,sum,max,middle;
  ret_val r1,r2,r3;

  if(one_after_last - first < 20)
    return max_sum_v1(a,first,one_after_last);
  else
  {
    // divide
    middle = (first + one_after_last) / 2;
    // recurse
    r1 = max_sum_v2(a,first,middle);
    r2 = max_sum_v2(a,middle,one_after_last);
    // combine
    sum = max = a[r3.first = middle - 1];
    for(i = middle - 2;i >= first;i--) if((sum += a[i]) > max) { max = sum; r3.first = i; }
    r3.sum = max; // best left half (at least one element)
    sum = max = a[r3.last = middle];
    for(i = middle + 1;i < one_after_last;i++) if((sum += a[i]) > max) { max = sum; r3.last = i; }
    r3.sum += max; // add best right half (at least one element)
  }
  if(r2.sum > r1.sum)
    r1 = r2;
  if(r3.sum > r1.sum)
    r1 = r3;
  return r1;
}
```

DaC examples (part 2c, a better solution to the maximum-sum sub-array problem)

In the particular case of the maximum-sum sub-array problem the divide-and-conquer programming paradigm does not produce an algorithm with the best computational complexity. The following function solves the problem in only $O(n)$ time! This is achieved by scanning the array once, keeping track of the best (largest) sum of consecutive array elements ending at the scan location:

```
ret_val max_sum_v3(int *a,int first,int one_after_last)
{
  int i,max_sum_ending_here,first_max_sum_ending_here;
  ret_val r = { first,first,a[first] };

  max_sum_ending_here = a[first_max_sum_ending_here = first];
  for(i = first + 1;i < one_after_last;i++)
  {
    if(max_sum_ending_here > 0)
      max_sum_ending_here += a[i]; // max_sum_ending_here + a[i] is better than just a[i]
    else
      max_sum_ending_here = a[first_max_sum_ending_here = i]; // just a[i] is better
    if(max_sum_ending_here > r.sum)
    {
      r.first = first_max_sum_ending_here;
      r.last = i;
      r.sum = max_sum_ending_here;
    }
  }
  return r;
}
```

This solution can be considered to be a particular case of the application of the dynamic programming paradigm (this case does not require memoization!), to be discussed in the next slides.

Dynamic programming (DP)

The dynamic programming technique solves a problem by combining the solutions of smaller problems (solved recursively). It is useful when these sub-problems share sub-sub-problems. Time efficiency is gained, at the expense of extra storage space, by solving each sub-problem only once and by storing its output in an array, or other data structure, for later reuse (this is called **memoization**).

Dynamic programming is usually used to solve optimization problems for which the solution to an order n problem can be found by combining the solutions to one or more order $n-1$ problems. It can be applied with advantage:

- if an optimal solution to an order n problem contains within it optimal solutions to order $n-1$ problems, which in turn contain optimal solutions to order $n-2$ problems, and so on;
- when the solution of the sub-problems share sub-sub-problems.

If a problem does not have these two characteristics then the dynamic programming technique either cannot be applied or it will not solve the problem efficiently.

For example, consider a network of roads connecting several cities. The problem of finding the shortest route between two cities A and B can be solved using dynamic programming, because if that route passes through city C then we can be assured that the route from A to C is the shortest possible (if that were not the case we could produce a better solution by replacing the route from A to C by the shorter route). On the other hand, the problem of finding the longest route that does not visit a city **more than once** (that constraint is not present in the smallest route problem, because there a route with a loop cannot be optimal) cannot be solved by dynamic programming, because if that route passes through city C then the route from A to C might not be the longest route between these two cities (that route may pass through a city that appears later in the longest route from A to B).

DP examples (part 1, two simple dynamic programming examples)

The Fibonacci numbers are defined by the recursion

$$F_n = F_{n-1} + F_{n-2}, \qquad n > 1,$$

with initial conditions $F_0 = 0$ and $F_1 = 1$. They can be computed as follows:

```c
int F_v1(int n)
{
  return (n < 2) ? n : F_v1(n - 1) + F_v1(n - 2);
}
```

Using a dynamic programming approach, i.e., by storing and reusing previously computed results, the previous code becomes **much** faster at the expense of a small amount of memory:

```c
int F_v2(int n)
{
  static int Fv[50] = { 0,1 };

  if(n > 1 && Fv[n] == 0)
    Fv[n] = F_v2(n - 1) + F_v2(n - 2);
  return Fv[n];
}
```

The same trick can be used to compute the binomial coefficients (number of combinations of n objects taken k at a time), denoted by C_k^n, which are given by

$$C_k^n = \begin{cases} \dfrac{n!}{k!(n-k)!}, & \text{if } 0 \leqslant k \leqslant n, \\ 0, & \text{otherwise (by convention).} \end{cases}$$

Using this formula directly may lead to arithmetic overflow, so one can use the following recursive formula instead (Pascal's triangle):

$$C_k^n = \begin{cases} 0, & \text{if } k < 0 \text{ or } k > n, \\ 1, & \text{if } k = 0 \text{ or } k = n, \\ C_{k-1}^{n-1} + C_k^{n-1}, & \text{otherwise.} \end{cases}$$

This gives rise to the following function:

```c
int C(int n,int k)
{
  static int Cv[100][100];

  if(k < 0 || k > n)   return 0;
  if(k == 0 || k == n) return 1;
  if(Cv[n][k] == 0)
    Cv[n][k] = C(n - 1,k - 1) + C(n - 1,k);
  return Cv[n][k];
}
```

DP examples (part 2a, the maximum-sum scattered-sub-array problem)

Given an array of positive numbers, the maximum-sum scattered-sub-array problem consists of finding the maximum sum of elements of the array, with the restriction that adjacent array elements cannot both contribute to the sum. (Alternative problem formulation: given a line of n bottles with volumes $v_0, \ldots, v_{n-1}$, drink as much as you can, given that you cannot drink from adjacent bottles.)

This problem can be solved easily for size n if the solutions for the sizes $n-1$ and $n-2$ are known, as illustrated by the following recursive function:

```
int max_scattered_sum_v1(int *v,int n)
{
  if(n == 1)
    return v[0];                                  // one is better than nothing
  if(n == 2)
    return (v[0] > v[1]) ? v[0] : v[1];           // choose the larger of the two
  int t1 = max_scattered_sum_v1(v,n - 2) + v[n - 1]; // use v[n - 1], cannot use v[n - 2]
  int t2 = max_scattered_sum_v1(v,n - 1);         // do not use v[n - 1], can use v[n - 2]
  return (t1 > t2) ? t1 : t2;                     // choose the larger of the two
}
```

The number of function calls done by `max_scattered_sum_v1()` is exactly the same as the number of function calls done by `F_v1()` of the previous slide. As that number grows exponentially, this function is useless for n greater than, say, 40. The next slide shows how this severe handicap can be eliminated (memoization saves the day!).

DP examples (part 2b, the maximum-sum scattered-sub-array problem with DP)

To speed up `max_scattered_sum_v1()` enormously simply record and reuse its return value. This gives rise to the function `max_scattered_sum_v2()`, presented below on the left-hand side (notice that it is possible to reset the memoized data by using special values of the function arguments). In this case it is also possible to get rid of the memoized data entirely, as illustrated in `max_scattered_sum_v3()`, presented below on the right-hand side.

```c
int max_scattered_sum_v2(int *v,int n)
{ // O(n) on first call, O(1) on subsequent calls
  static int r[1001]; // n cannot be larger than 1000

  if(v == NULL || n <= 0 || n > 1000)
  { // invalidate memoized data
    for(int i = 0;i <= 1000;i++)
      r[i] = -1;
    return -1;
  }
  if(r[n] < 0)
  { // compute for the first time
    if(n == 1)
      r[n] = v[0];
    else if(n == 2)
      r[n] = (v[0] > v[1]) ? v[0] : v[1];
    else
    {
      int t1 = max_scattered_sum_v2(v,n - 2) + v[n - 1];
      int t2 = max_scattered_sum_v2(v,n - 1);
      r[n] = (t1 > t2) ? t1 : t2;
    }
  }
  return r[n];
}
```

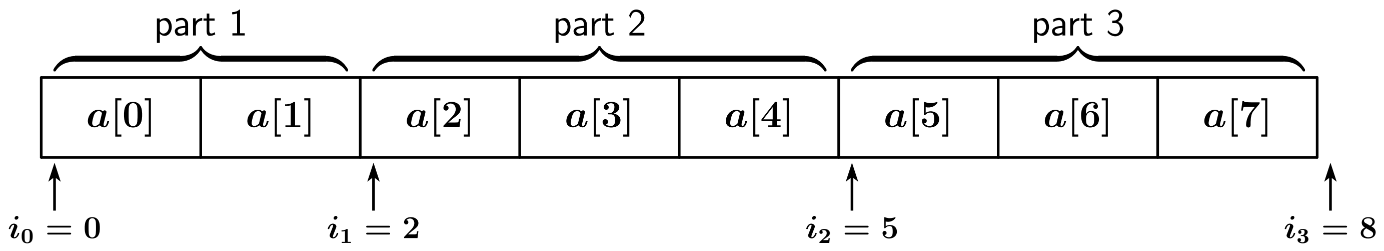
```c
int max_scattered_sum_v3(int *v,int n)
{ // always O(n) time and O(1) space
  if(n == 1)
    return v[0];
  int t1 = v[0];
  int t2 = (v[0] > v[1]) ? v[0] : v[1];
  for(int i = 2;i < n;i++)
  {
    int t3 = (t1 + v[i] > t2) ? t1 + v[i] : t2;
    t1 = t2;
    t2 = t3;
  }
  return t2;
}
```

DP examples (part 3a, best partition problem)

Given an array of $n > 0$ positive integers $a[0], a[1], \ldots, a[n-1]$, the best partition problem asks for the best way to partition these integers into $m \geqslant 1$ consecutive ranges covering the entire array so that the maximum sum of the array elements over each range is minimized. Mathematically, it asks for a set of indices $i_0, i_1, \ldots, i_m$, with $i_0 = 0$, $i_k \leqslant i_{k+1}$ for $k = 0, 1, \ldots, m-1$ and $i_m = n$, such that

$$\max_{0 \leqslant k < m} \sum_{i_k \leqslant i < i_{k+1}} a[i]$$

is minimized. By convention the sum is zero when the summation range is empty. For example, for $n = 8$ and $m = 3$ one possible partition of the array can be



Let $M[j, i]$ be the minimum of the maximum sums of the array elements $a[0], \ldots a[i-1]$ when they are subdivided into j ranges. We are interested in the value of $M[m, n]$ and in one partition that achieves it. The boundary cases gives us

$$M[1, i] = \sum_{0 \leqslant k < i} a[k], \quad 0 \leqslant i \leqslant n, \qquad \text{and} \qquad M[j, 0] = 0, \quad 2 \leqslant j \leqslant m,$$

and adding one more partition gives us (note that $\sum_{c \leqslant k < i} a[k] = M[1, i] - M[1, c]$)

$$M[j, i] = \min_{0 \leqslant c \leqslant i} \max\bigl(M[j-1, c], M[1, i] - M[1, c]\bigr), \quad 1 \leqslant i \leqslant n, 2 \leqslant j \leqslant m.$$

DP examples (part 3b, code for the best partition problem)

The recursive formula for $M[j,i]$ gives rise to an order $O(mn^2)$ algorithm to solve the best partition problem, as shown in the following code:

```
int best_partition(int *a,int n,int m,int show)
{
  int M[m + 1][n + 1]; // M[j][i] stores the best cost for the sub-problem with n=i and k=j
  int D[m + 1][n + 1]; // D[j][i] records the best partition point to get to M[j][i]
  int I[m + 1];        // partition indices
  int i,j,c,best_c,cost,best_cost;

  assert(n >= 1 && m >= 1);
  //
  // boundary cases
  //
  M[1][0] = 0;
  D[1][0] = 0;
  for(i = 1;i <= n;i++)
  {
    M[1][i] = M[1][i - 1] + a[i - 1]; // array elements in one partition
    D[1][i] = 0;
  }
  for(j = 2;j <= m;j++)
  {
    M[j][0] = 0; // zero array elements
    D[j][0] = 0;
  }
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
//
// apply dynamic programming to solve all sub-problems
//
for(j = 2;j <= m;j++)
  for(i = 1;i <= n;i++)
  {
    best_c = 0;
    best_cost = M[1][i] - M[1][0];
    for(c = 1;c <= i;c++)
    {
      cost = (M[j-1][c] > M[1][i] - M[1][c]) ? M[j-1][c] : M[1][i] - M[1][c];
      if(cost < best_cost)
      {
        best_cost = cost;
        best_c = c;
      }
    }
    M[j][i] = best_cost;
    D[j][i] = best_c;
  }
//
// construct best partition
//
I[m] = n;
for(i = n,j = m;j > 0;j--)
  i = I[j - 1] = D[j][i]; // partition separator
assert(I[0] == 0);
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
  //
  // show solution
  //
  if(show > 1)
    for(j = 1;j <= m;j++)
      for(i = 0;i <= n;i++)
        printf("%2d[%2d]%s",M[j][i],D[j][i],(i == n) ? "\n" : " ");
  if(show > 0)
  {
    printf("<%d> ",M[m][n]);
    for(i = j = 0;i <= n;i++)
    {
      if(i == I[j])
      {
        printf("|%s",(i == n) ? "\n" : " ");
        j++;
      }
      if(i < n)
        printf("%d ",a[i]);
    }
  }
  //
  // done (we should also return the best partition, but in C that is cumbersome)
  //
  return M[m][n];
}
```

DP examples (part 3c, best partition example)

Consider the problem of subdividing the array

8 4 6 1 2 7 2 1 1 1 7 9

into 4 parts so that the largest sum of the elements of each part is as small as possible. In this case the contents of the `M[j][i]` array of the function presented in the previous slides is

$j\backslash i$	0	1	2	3	4	5	6	7	8	9	10	11	12
1	0	8	12	18	19	21	28	30	31	32	33	40	49
2	0	8	8	10	11	12	16	18	18	18	18	21	28
3	0	8	8	8	8	9	10	11	12	12	12	16	18
4	0	8	8	8	8	8	9	9	10	10	10	11	**16**

and the contents of the `D[j][i]` array, useful to retrace the choices make by the algorithm and so find the partition boundaries, is

$j\backslash i$	0	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	**0**	0	0	0	0	0	0	0	0	0	0
2	0	0	1	1	1	**2**	2	2	3	3	3	4	5
3	0	0	1	2	2	2	3	4	4	5	**5**	6	8
4	0	0	1	2	2	3	4	5	5	6	6	7	**10**

From this information it is possible to extract (in bold in the `D[j][i]` array) the best partition

$|\,8\;\;4\,|\,6\;\;1\;\;2\,|\,7\;\;2\;\;1\;\;1\;\;1\,|\,7\;\;9\,|$

It has a maximum sum of 16 (in bold in the `M[j][i]` array).

DP examples (part 4a, smallest edit distance)

Given the strings $a[0], a[1], \ldots, a[n_a-1]$ and $b[0], b[1], \ldots, b[n_b-1]$, find the best sequence of insertions, deletions, and character changes, that transforms the first string into the second. The cost of an insertion or a deletion is I and the cost on a character change is X. (As insertion in $a[]$ can be considered a deletion in $b[]$, so the cost of an insertion should be the same as that of a deletion.)

Let $D[i,j]$ be the best (smallest) distance between the sub-strings $a[0], \ldots a[i-1]$ and $b[0], \ldots, b[j-1]$. We are interested in the value of $D[n_a, n_a]$, and in how one gets there in an optimal way starting from $D[0,0]$ (two empty strings). Clearly, we have $D[i,0] = iI$ and $D[0,j] = jI$, i.e., $D[i,j] = \max(i,j)I$ when $\min(i,j) = 0$. We also have

$$D[i,j] = \begin{cases} D[i-1,j-1], & \text{if a[i-1]=b[j-1],} \\ \min\bigl(D[i-1,j]+I, D[i,j-1]+I, D[i-1,j-1]+X\bigr), & \text{otherwise.} \end{cases}$$

This formula gives rise to the following function to partially solve the problem (just compute $D[n_a, n_b]$):

```
int distance_v1(char *a,char *b,int i,int j,int I,int X)
{
  if(i == 0 || j == 0)
    return ((i > j) ? i : j) * I;
  int d1 = distance_v1(a,b,i - 1,j,I,X) + I;                                // one more in a[]
  int d2 = distance_v1(a,b,i,j - 1,I,X) + I;                                // one more in b[]
  int d3 = distance_v1(a,b,i - 1,j - 1,I,X) + ((a[i -1] == b[j - 1]) ? 0 : X); // one more in a[] and in b[]
  if(d1 < d2 && d1 < d3)
    return d1;
  return (d2 < d3) ? d2 : d3;
}
```

DP examples (part 4b, smallest edit distance using dynamic programming)

The code of the previous slide suffers from two problems: the distance function may be called **more than once** with the same arguments (inefficient!), and, because the C language does not allow more than one return argument (if we need that, we need to return a structure), it is not trivial to retrieve the best sequence of insertions/deletions and of character exchanges. All this is solved if the values returned by the function are cached. The resultant algorithm is a dynamic programming algorithm. It uses $O(n_a n_b)$ time and space. It the best sequence of insertions/deletions and exchanges is not needed the space requirements can be lowered to $O\big(\min(n_a, n_b)\big)$.

```
int distance_v2(char *a,char *b,int na,int nb,int I,int X)
{
  int i,j,D[na + 1][nb + 1];

  for(i = 0;i <= na;i++)  D[i][0] = i * I;
  for(j = 0;j <= nb;j++)  D[0][j] = j * I;
  for(i = 1;i <= na;i++)
    for(j = 1;j <= nb;j++)
    {
      int d1 = ((D[i - 1][j] < D[i][j - 1]) ? D[i - 1][j] : D[i][j - 1]) + I;
      int d2 = D[i - 1][j - 1] + ((a[i - 1] == b[j - 1]) ? 0 : X);
      D[i][j] = (d1 < d2) ? d1 : d2;
    }
  return D[na][nb];
}
```

If the best sequence of insertions/deletions and exchanges were needed, one would only need to start at $D[n_a, n_b]$ and attempt to reach $D[0, 0]$ in the smallest number of moves to an adjacent position (either decreasing i by 1, j by 1, or i and j by 1), as illustrated in the next slide. When called with `a[] = "destruction"` $(n_a = 11)$ and `b[] = "construction"` $(n_b = 12)$, `distance_v1()` is **one million times slower** than `distance_v2()`!

DP examples (part 4c, smallest edit distance example)

The following table presents the values of $D[i,j]$ (`D[i][j]` in the C function) when `a[] = "master"` and `b[] = "small edit distance"`. One one the best ways to transform one string into the other is depicted in bold.

	b[j-1]		's'	'm'	'a'	'l'	'l'	' '	'e'	'd'	'i'	't'	' '	'd'	'i'	's'	't'	'a'	'n'	'c'	'e'
a[i-1]	$i\backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	0	**0**	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
'm'	1	1	1	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
'a'	2	2	2	2	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	12	13	14	15	16	17
's'	3	3	2	3	2	2	3	4	5	6	7	8	9	10	11	**11**	12	13	14	15	16
't'	4	4	3	3	3	3	3	4	5	6	7	7	8	9	10	11	**11**	**12**	**13**	14	15
'e'	5	5	4	4	4	4	4	4	4	5	6	7	8	9	10	11	12	12	13	**14**	14
'r'	6	6	5	5	5	5	5	5	5	5	6	7	8	9	10	11	12	13	13	14	**15**

This path was constructed by selecting, at position (i,j), its neighbor with the smallest value. Only the neighbors $(i-1,j-1)$, to the left and up, $(i-1,j)$, to the left, and $(i,j-1)$, up, need to be considered. In the case of a tie the choice of neighbor is arbitrary.