15.4 Longest common subsequence

- Biological applications often need to compare the DNA of two (or more) different organisms.
- A strand of DNA consists of a string of molecules called **bases**, where the possible bases are **Adenine**, **Guanine**, **Cytosine**, and **Thymine**.
- We express a strand of DNA as a string over the alphabet \{**A, C, G, T**\}.
- E.g., the DNA of two organisms may be:
  - \(S_1 = \text{ACCGTCGAGTGCGCGGAAGCCGGCCGAA}\)
  - \(S_2 = \text{GTCGTTCGGAAATGCGTTGCTCTGTA}\)

- By comparing two strands of DNA we determine how “similar” they are, as some measure of how closely related the two organisms are.
- We can define similarity in many different ways.
  - E.g., we can say that two DNA strands are similar if one is a substring of the other;
    - Neither \(S_1\) nor \(S_2\) is a substring of the other.
  - Alternatively, we could say that two strands are similar if the number of changes needed to turn one into the other is small.
Yet another way to measure the similarity of $S_1$ and $S_2$ is by finding a third strand $S_3$ in which the bases in $S_3$ appear in each of $S_1$ and $S_2$

- these bases must appear in the same order, but not necessarily consecutively

- The longer the strand $S_3$ we can find, the more similar $S_1$ and $S_2$ are

In our example, the longest strand $S_3$ is

- $S_1 = \text{ACCGTTCGAGTCCGTAAGCCGGCCGAA}$
- $S_2 = \text{GTCGTTCCGGAATGCCGTCTGCTGTAAA}$
- $S_3 = \text{GTCGTCGGAAGCCGGCCGAA}$

Formalize this notion of similarity as the longest-common-subsequence problem

A subsequence is just the given sequence with zero or more elements left out

Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of $X$ if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of $X$ such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$

For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$
A sequence $Z$ is a **common subsequence** of $X$ and $Y$ if $Z$ is a subsequence of both $X$ and $Y$.

For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both $X$ and $Y$.

It is not a **longest common subsequence** (LCS) of $X$ and $Y$.

The sequence $\langle B, C, B, A \rangle$ is also common to both $X$ and $Y$ and has length 4.

This sequence is an LCS of $X$ and $Y$, as is $\langle B, D, A, B \rangle$; $X$ and $Y$ have no common subsequence of length 5 or greater.

In **longest-common-subsequence problem**, we are given $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a max-length common subsequence of $X$ and $Y$.

**Step 1: Characterizing a longest common subsequence**

In a brute-force approach, we would enumerate all subsequences of $X$ and check each of them to see whether it is also a subsequence of $Y$, keeping track of the longest subsequence we find.

Each subsequence of $X$ corresponds to a subset of the indices $\langle 1, 2, \ldots, m \rangle$ of $X$.

Because $X$ has $2^m$ subsequences, this approach requires exponential time, making it impractical for long sequences.
The LCS problem has an optimal-substructure property, however, as the following theorem shows:

The natural classes of subproblems correspond to pairs of “prefixes” of the two input sequences.

Precisely, given a sequence $X = (x_1, x_2, \ldots, x_m)$, we define the $i$th prefix of $X$, for $i = 0, 1, \ldots, m$, as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$.

For example, if $X = A, B, C, B, D, A, B$, then $X_4 = \langle A, B, C, B \rangle$ and $X_0$ is the empty sequence.

**Theorem 15.1** (Optimal substructure of LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of $X$ and $Y$.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X$ and $Y_{n-1}$.
Proof (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to $Z$ to obtain a common subsequence of $X$ and $Y$ of length $k + 1$, contradicting the supposition that $Z$ is a LCS of $X$ and $Y$. Thus, we must have $z_k = x_m = y_n$. Now, the prefix $Z_{k-1}$ is a length-$(k-1)$ common subsequence of $X_{m-1}$ and $Y_{n-1}$. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence $W$ of $X_{m-1}$ and $Y_{n-1}$ with length greater than $k-1$. Then, appending $x_m = y_n$ to $W$ produces a common subsequence of $X$ and $Y$ whose length is greater than $k$, which is a contradiction.

(2) If $z_k \neq x_m$, then $Z$ is a common subsequence of $X_{m-1}$ and $Y$. If there were a common subsequence $W$ with length greater than $k$, then $W$ would also be a common subsequence of $X_m$ and $Y$, contradicting the assumption that $Z$ is an LCS of $X$ and $Y$.

(3) The proof is symmetric to (2).

- Theorem 15.1 tells us that an LCS of two sequences contains within it an LCS of prefixes of the two sequences.
- Thus, the LCS problem has an optimal-substructure property.
Step 2: A recursive solution

- We examine either one or two subproblems when finding an LCS of $X$ and $Y$
- If $x_m = y_n$, we find an LCS of $X_{m-1}$ and $Y_{n-1}$
- Appending $x_m = y_n$ yields an LCS of $X$ and $Y$
- If $x_m \neq y_n$, then we (1) find an LCS of $X_{m-1}$ and $Y$ and (2) find an LCS of $X$ and $Y_{n-1}$
- Whichever of these two LCSs is longer is an LCS of $X$ and $Y$
- These cases exhaust all possibilities, and we know that one of the optimal subproblem solutions must appear within an LCS of $X$ and $Y$

To find an LCS of $X$ and $Y$, we may need to find the LCSs of $X$ and $Y_{n-1}$ and of $X_{m-1}$ and $Y$

Each subproblem has the subsubproblem of finding an LCS of $X_{m-1}$ and $Y_{n-1}$

Many other subproblems share subsubproblems

As in the matrix-chain multiplication, recursive solution to the LCS problem involves a recurrence for the value of an optimal solution

Let us define $c[i,j]$ to be the length of an LCS of the sequences $X_i$ and $Y_j$

If either $i = 0$ or $j = 0$, one of the sequences has length 0, and so the LCS has length 0
The optimal substructure of the LCS problem gives

\[
c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\
\max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j
\end{cases}
\]

Observe that a condition in the problem restricts which subproblems we may consider:

- When \(x_i = y_j\), we consider finding an LCS of \(X_{i-1}\) and \(Y_{j-1}\).
- Otherwise, we instead consider the two subproblems of finding an LCS of \(X_i\) and \(Y_{j-1}\) and of \(X_{i-1}\) and \(Y_j\).

In the previous dynamic-programming algorithms — for rod cutting and matrix-chain multiplication — we ruled out no subproblems due to conditions in the problem.

**Step 3: Computing the length of an LCS**

- Since the LCS problem has only \(\Theta(mn)\) distinct subproblems, we can use dynamic programming to compute the solutions bottom up.
- LCS-LENGTH stores the \(c[i, j]\) values in \(c[0..m, 0..n]\), and it computes the entries in **row-major** order.
  - I.e., the procedure fills in the first row of \(c\) from left to right, then the second row, and so on.
- The procedure also maintains the table \(b[1..m, 1..n]\).
- Intuitively, \(b[i, j]\) points to the table entry corresponding to the optimal subproblem solution chosen when computing \(c[i, j]\).
- \(c[m, n]\) contains the length of an LCS of \(X\) and \(Y\).
LCS-LENGTH($X,Y$)

1. $m \leftarrow X.\text{length}$
2. $n \leftarrow Y.\text{length}$
3. let $b[1..m,1..n]$ and $c[0..m,0..n]$ be new tables
4. for $i \leftarrow 1$ to $m$
5. $c[i,0] \leftarrow 0$
6. for $j \leftarrow 0$ to $n$
7. $c[0,j] \leftarrow 0$
8. for $i \leftarrow 1$ to $m$
9. for $j \leftarrow 1$ to $n$
10. if $x_i = y_j$
11. $c[i,j] \leftarrow c[i-1,j-1] + 1$
12. $b[i,j] \leftarrow \times$
13. elseif $c[i-1,j] \geq c[i,j-1]$
14. $c[i,j] \leftarrow c[i-1,j]$
15. $b[i,j] \leftarrow \uparrow$
16. else $c[i,j] \leftarrow c[i,j-1]$
17. $b[i,j] \leftarrow \leftarrow$
18. return $c$ and $b$

Running time: $\Theta(mn)$

The $c$ and $b$ tables computed by LCS-LENGTH on $X = \langle A,B,C,B,D,A,B \rangle$ and $Y = \langle B,D,C,A,B,A \rangle$
Step 4: Constructing an LCS

- The $b$ table returned by LCS-LENGTH enables us to quickly construct an LCS of $X$ and $Y$
- We simply begin at $b[m,n]$ and trace through the table by following the arrows
- Whenever we encounter a “$\rightarrow$” in entry $b[i,j]$, it implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH found
- With this method, we encounter the elements of this LCS in reverse order
- A recursive procedure prints out an LCS of $X$ and $Y$ in the proper, forward order

The square in row $i$ and column $j$ contains the value of $c[i,j]$ and the appropriate arrow for the value of $b[i,j]$

- The entry 4 in $c[7,6]$ — the lower right-hand corner of the table — is the length of an LCS $\langle B, C, B, A \rangle$
- For $i, j > 0$, entry $c[i,j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1,j]$, $c[i,j-1]$, and $c[i-1,j-1]$, which are computed before $c[i,j]$
- To reconstruct the elements of an LCS, follow the $b[i,j]$ arrows from the lower right-hand corner
- Each “$\rightarrow$” on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS
Improving the code

- Each $c[i,j]$ entry depends on only 3 other $c$ table entries: $c[i-1,j]$, $c[i,j-1]$, and $c[i-1,j-1]$
- Given the value of $c[i,j]$, we can determine in $O(1)$ time which of these three values was used to compute $c[i,j]$, without inspecting table $b$
- We can reconstruct an LCS in $O(m+n)$ time
- The auxiliary space requirement for computing an LCS does not asymptotically decrease, since we need $\Theta(mn)$ space for the $c$ table anyway

We can, however, reduce the asymptotic space requirements for LCS-LENGTH, since it needs only two rows of table $c$ at a time
- the row being computed and the previous row
- This improvement works if we need only the length of an LCS
  - if we need to reconstruct the elements of an LCS, the smaller table does not keep enough information to retrace our steps in $O(m+n)$ time
15.5 Optimal binary search trees

- We are designing a program to translate text
- Perform lookup operations by building a BST with $n$ words as keys and their equivalents as satellite data
- We can ensure an $O(\log n)$ search time per occurrence by using a RBT or any other balanced BST
- A frequently used word may appear far from the root while a rarely used word appears near the root
- We want frequent words to be placed nearer the root
- How do we organize a BST so as to minimize the number of nodes visited in all searches, given that we know how often each word occurs?

What we need is an **optimal binary search tree**

- Formally, given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of $n$ distinct sorted keys ($k_1 < k_2 < \cdots < k_n$), we wish to build a BST from these keys
- For each key $k_i$, we have a probability $p_i$ that a search will be for $k_i$
- Some searches may be for values not in $K$, so we also have $n + 1$ “dummy keys” $d_0, d_1, \ldots, d_n$ representing values not in $K$
- In particular, $d_0$ represents all values less than $k_1$, $d_n$ represents all values greater than $k_n$
For $i = 1, 2, \ldots, n - 1$, the dummy key $d_i$ represents all values between $k_i$ and $k_{i+1}$.

For each dummy key $d_i$, we have a probability $q_i$ that a search will correspond to $d_i$.

Each key $k_i$ is an internal node, and each dummy key $d_i$ is a leaf.

Every search is either successful (finds a key $k_i$) or unsuccessful (finds a dummy key $d_i$), and so we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Because we have probabilities of searches for each key and each dummy key, we can determine the expected cost of a search in a given BST $T$.
Let us assume that the actual cost of a search equals the number of nodes examined, i.e., the depth of the node found by the search in $T + 1$.

Then the expected cost of a search in $T$,

$$E[\text{search cost in } T] = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i,$$

where $\text{depth}_T$ denotes a node’s depth in tree $T$.
For a given set of probabilities, we wish to construct a BST whose expected search cost is smallest.

- We call such a tree an **optimal binary search tree**.
- An optimal BST for the probabilities given has expected cost 2.75.

An optimal BST is not necessarily a tree whose overall height is smallest.

Nor can we necessarily construct an optimal BST by always putting the key with the greatest probability at the root.

- The lowest expected cost of any BST with $k_5$ at the root is 2.85.
Step 1: The structure of an optimal BST

- Consider any subtree of a BST
- It must contain keys in a contiguous range $k_i, \ldots, k_j$, for some $1 \leq i \leq j \leq n$
- In addition, a subtree that contains keys $k_i, \ldots, k_j$ must also have as its leaves the dummy keys $d_{i-1}, \ldots, d_j$
- If an optimal BST $T$ has a subtree $T'$ containing keys $k_i, \ldots, k_j$, then this subtree $T'$ must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$.

Given keys $k_i, \ldots, k_j$, one of them, say $k_r$, is the root of an optimal subtree containing these keys.

- The left subtree of the root $k_r$ contains the keys $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$)
- The right subtree contains the keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$)
- As long as we
  - examine all candidate roots $k_r$, where $i \leq r \leq j$,
  - and determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and those containing $k_{r+1}, \ldots, k_j$,
we are guaranteed to find an optimal BST.
Suppose that in a subtree with keys \( k_i, \ldots, k_j \), we select \( k_i \) as the root
- \( k_i \)'s left subtree contains the keys \( k_i, \ldots, k_{i-1} \)
- Interpret this sequence as containing no keys
- Subtrees, however, also contain dummy keys
- Adopt the convention that a subtree containing keys \( k_i, \ldots, k_{i-1} \) has no actual keys but does contain the single dummy key \( d_{i-1} \)
- Symmetrically, if we select \( k_j \) as the root, then \( k_j \)'s right subtree contains no actual keys, but it does contain the dummy key \( d_j \)

**Step 2: A recursive solution**
- We pick our subproblem domain as finding an optimal BST containing the keys \( k_i, \ldots, k_j \), where \( i \geq 1, j \leq n, \) and \( j \geq i - 1 \)
- Let us define \( e[i, j] \) as the expected cost of searching an optimal BST containing the keys \( k_i, \ldots, k_j \)
- Ultimately, we wish to compute \( e[1, n] \)
- The easy case occurs when \( j = i - 1 \)
- Then we have just the dummy key \( d_{i-1} \)
- The expected search cost is \( e[i, i - 1] = q_{i-1} \)
When \( j > i \), we need to select a root \( k_r \) from among \( k_i, \ldots, k_j \) and make an optimal BST with keys \( k_i, \ldots, k_{r-1} \) as its left subtree and an optimal BST with keys \( k_{r+1}, \ldots, k_j \) as its right subtree.

- What happens to the expected search cost of a subtree when it becomes a subtree of a node?
  - Depth of each node increases by 1
  - Expected search cost of this subtree increases by the sum of all the probabilities in it

For a subtree with keys \( k_i, \ldots, k_j \), let us denote this sum of probabilities as
\[
\omega(\cdot, \cdot) = \sum_{t=i}^{j} p_t + \sum_{t=i-1}^{j} q_t
\]

Thus, if \( k_r \) is the root of an optimal subtree containing keys \( k_i, \ldots, k_j \), we have
\[
e[i, j] = p_r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j))
\]

Noting that
\[
w(i, j) = w(i, r - 1) + p_r + w(r + 1, j)
\]
we rewrite
\[
e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j)
\]

We choose the root \( k_r \) that gives the lowest expected search cost:

\[
\begin{align*}
e[i, j] &= \begin{cases} 
q_{i-1} & \text{if } j = i - 1 \\
\min_{i \leq r \leq j} e[i, r - 1] + e[r + 1, j] + w(i, j) & \text{if } i \leq j
\end{cases}
\end{align*}
\]
• The $e[i,j]$ values give the expected search costs in optimal BSTs
• To help us keep track of the structure of optimal BSTs, we define $\text{root}[i,j]$, for $1 \leq i \leq j \leq n$, to be the index $r$ for which $k_r$ is the root of an optimal BST containing keys $k_i, \ldots, k_j$
• Although we will see how to compute the values of $\text{root}[i,j]$, we leave the construction of an optimal binary search tree from these values as an exercise

Step 3: Computing the expected search cost of an optimal BST
• We store $e[i,j]$ values in a table $e[1..n+1,0..n]$
• The first index needs to run to $n + 1$ because to have a subtree containing only the dummy key $d_n$, we need to compute and store $e[n+1,n]$
• The second index needs to start from 0 because to have a subtree containing only the dummy key $d_0$, we need to compute and store $e[1,0]$
• We use only the entries $e[i,j]$ for which $j \geq i - 1$
• We also use a table $root[i,j]$, for recording the root of the subtree containing keys $k_i, \ldots, k_j$
• This table uses only the entries $1 \leq i \leq j \leq n$
• We also store the $w(i,j)$ values in a table $w[1..n + 1,0..n]$
• For the base case, we compute $w[i, i - 1] = q_i$
• For $j \geq i$, we compute $w[i,j] = w[i,j - 1] + p_j + q_j$
• Thus, we can compute the $\Theta(n^2)$ values of $w[i,j]$ in $\Theta(1)$ time each

**OPTIMAL-BST(p, q, n)**
1. let $e[1..n + 1,0..n], w[1..n + 1,0..n], \ root[1..n,1..n]$ be new tables
2. for $i = 1$ to $n + 1$
3. $e[i,i - 1] = q_{i-1}$
4. $w[i,i - 1] = q_{i-1}$
5. for $l = 1$ to $n$
6. for $i = 1$ to $n - l + 1$
7. $j = i + l - 1$
8. $e[i,j] = \infty$
9. $w[i,j] = w[i,j - 1] + p_j + q_j$
10. for $r = i$ to $j$
11. $t = e[i,r - 1] + e[r + 1,j] + w[i,j]$
12. if $t < e[i,j]$
13. $e[i,j] = t$
14. $\ root[i,j] = r$
15. return $e$ and $\ root$
The OPTIMAL-BST procedure takes $\Theta(n^3)$ time, just like MATRIX-CHAIN-ORDER.

Its running time is $O(n^3)$, since its for loops are nested three deep and each loop index takes on at most $n$ values.

The loop indices in OPTIMAL-BST do not have exactly the same bounds as those in MATRIX-CHAIN-ORDER, but they are within $\leq 1$ in all directions.

Thus, like MATRIX-CHAIN-ORDER, the OPTIMAL-BST procedure takes $\Omega(n^3)$ time.
16 Greedy Algorithms

- Optimization algorithms typically go through a sequence of steps, with a set of choices at each step.
- For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do.
- A greedy algorithm always makes the choice that looks best at the moment.
- That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

16.1 An activity-selection problem

- Suppose we have a set \( S = \{a_1, a_2, ..., a_n\} \) of \( n \) proposed activities that wish to use a resource (e.g., a lecture hall), which can serve only one activity at a time.
- Each activity \( a_i \) has a start time \( s_i \) and a finish time \( f_i \), where \( 0 \leq s_i < f_i < \infty \).
- If selected, activity \( a_i \) takes place during the half-open time interval \( [s_i, f_i) \).
Activities $a_i$ and $a_j$ are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

I.e., $a_i$ and $a_j$ are compatible if $s_i \geq f_j$ or $s_j \geq f_i$.

We wish to select a maximum-size subset of mutually compatible activities.

We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n$$

Consider, e.g., the following set $S$ of activities:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>$f_j$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities.

It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger.

In fact, it is a largest subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.
The optimal substructure of the activity-selection problem

- Let $S_{ij}$ be the set of activities that start after $a_i$ finishes and that finish before $a_j$ starts.
- We wish to find a maximum set of mutually compatible activities in $S_{ij}$.
- Suppose that such a maximum set is $A_{ij}$, which includes some activity $a_k$.
- By including $a_k$ in an optimal solution, we are left with two subproblems: finding mutually compatible activities in the set $S_{ik}$ and finding mutually compatible activities in the set $S_{kj}$.

Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, so that
- $A_{ik}$ contains the activities in $A_{ij}$ that finish before $a_k$ starts and
- $A_{kj}$ contains the activities in $A_{ij}$ that start after $a_k$ finishes.
- Thus, $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum-size set $A_{ij}$ in $S_{ij}$ consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.
- The usual cut-and-paste argument shows that the optimal solution $A_{ij}$ must also include optimal solutions for $S_{ik}$ and $S_{kj}$.
• This suggests that we might solve the activity-selection problem by dynamic programming
• If we denote the size of an optimal solution for the set $S_{ij}$ by $c[i,j]$, then we would have the recurrence
  \[ c[i, j] = c[i, k] + c[k, j] + 1 \]
• Of course, if we did not know that an optimal solution for the set $S_{ij}$ includes activity $a_k$, we would have to examine all activities in $S_{ij}$ to find which one to choose, so that

\[
c[i, j] = \begin{cases} 
0 & \text{if } S_{ij} = \emptyset \\
\max_{a_k \in S_{ij}} \{ c[i, j] = c[i, k] + c[k, j] + 1 \} & \text{if } S_{ij} \neq \emptyset 
\end{cases}
\]

Making the greedy choice
• For the activity-selection problem, we need consider only the greedy choice
• We choose an activity that leaves the resource available for as many other activities as possible
• Now, of the activities we end up choosing, one of them must be the first one to finish
• Choose the activity in $S$ with the earliest finish time, since that leaves the resource available for as many of the activities that follow it as possible
• Activities are sorted in monotonically increasing order by finish time; greedy choice is activity $a_1$
If we make the greedy choice, we only have to find activities that start after $a_1$ finishes.

- $s_1 < f_1$ and $f_1$ is the earliest finish time of any activity $\Rightarrow$ no activity can have a finish time $\leq s_1$
- Thus, all activities that are compatible with activity $a_1$ must start after $a_1$ finishes.
- Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after $a_k$ finishes.
- Optimal substructure: if $a_1$ is in the optimal solution, then an optimal solution to the original problem consists of $a_1$ and all the activities in an optimal solution to the subproblem $S_1$

**Theorem 16.1** Consider any nonempty subproblem $S_k$, and let $a_m$ be an activity in $S_k$ with the earliest finish time. Then $a_m$ is included in some maximum-size subset of mutually compatible activities of $S_k$.

**Proof** Let $A_k$ be a max-size subset of mutually compatible activities in $S_k$, and let $a_j$ be the activity in $A_k$ with the earliest finish time. If $a_j = a_m$, we are done, since $a_m$ is in a max-size subset of mutually compatible activities of $S_k$.

If $a_j \neq a_m$, let the set $A'_k = A_k \setminus \{a_j\} \cup \{a_m\}$. The activities in $A'_k$ are disjoint because the activities in $A_k$ are disjoint, $a_j$ is the first activity in $A_k$ to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that $A'_k$ is a maximum-size subset of mutually compatible activities of $S_k$ and includes $a_m$. ■
We can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain.

Moreover, because we always choose the activity with the earliest finish time, the finish times of the activities we choose must strictly increase.

We can consider each activity just once overall, in monotonically increasing order of finish times.

### A recursive greedy algorithm

**RECURSIVE-ACTIVITY-SELECTOR**(\(s, f, k, n\))

1. \(m \leftarrow k + 1\)
2. while \(m \leq n\) and \(s[m] < f[k]\) // find the first // activity in \(S_k\) to finish
3. \(m \leftarrow m + 1\)
4. if \(m \leq n\)
5. return \(\{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m, n)\)
6. else return \(\emptyset\)
An iterative greedy algorithm

**GREEDY-ACTIVITY-SELECTOR**(s, f)

1. \( n \leftarrow s.\text{length} \)
2. \( A \leftarrow \{a_1\} \)
3. \( k \leftarrow 1 \)
4. **for** \( m \leftarrow 2 \) **to** \( n \)
5.  **if** \( s[m] \geq f[k] \)
6.  \( A \leftarrow A \cup \{a_m\} \)
7.  \( k \leftarrow m \)
8. **return** \( A \)
- The set \( A \) returned by the call
  \[ \text{GREEDY-ACTIVITY-SELECTOR}(s,f) \]
  is precisely the set returned by the call
  \[ \text{RECURSIVE-ACTIVITY-SELECTOR}(s,f,k,n) \]

- Both the recursive version and the iterative algorithm schedule a set of \( n \) activities in \( \Theta(n) \) time, assuming that the activities were already sorted initially by their finish times.