```
INSERTION-SORT($A$)
1  for $j = 2$ to $A.length$
2      $key = A[j]$
3      // Insert $A[j]$ into the sorted sequence $A[1 \dots j-1]$
4      $i = j - 1$
5      while $i > 0$ and $A[i] > key$
6          $A[i+1] = A[i]$
7          $i = i - 1$
8      $A[i+1] = key$
```

```
MERGE($A, p, q, r$)
 1  $n_1 = q - p + 1$
 2  $n_2 = r - q$
 3  let $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$ be new arrays
 4  for $i = 1$ to $n_1$
 5      $L[i] = A[p + i - 1]$
 6  for $j = 1$ to $n_2$
 7      $R[j] = A[q + j]$
 8  $L[n_1 + 1] = \infty$
 9  $R[n_2 + 1] = \infty$
10  $i = 1$
11  $j = 1$
12  for $k = p$ to $r$
13      if $L[i] \leq R[j]$
14          $A[k] = L[i++]$
15      else $A[k] = R[j++]$
```

```
MERGE-SORT($A, p, r$)
1  if $p < r$
2      $q = \lfloor (p + r)/2 \rfloor$
3      MERGE-SORT($A, p, q$)
4      MERGE-SORT($A, q + 1, r$)
5      MERGE($A, p, q, r$)
```

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c \end{aligned}$$

Master Theorem says that if a, b, c, and k are all constants, then…

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k, \\ \Theta(n^k \lg(n)) & \text{if} a = b^k, \\ \Theta(n^{\log_b a}) & \text{if} a > b^k \end{cases}$$

Stirling's Approximation: $\lg(n!) \approx n \lg n$

```
PARENT$(A, i)$ **return** $\lfloor i/2 \rfloor$    LEFT$(A, i)$ **return** $2i$    RIGHT$(A, i)$ **return** $2i + 1$
```

```
MAX-HEAPIFY$(A, i)$
1  $l = $ LEFT$(i)$
2  $r = $ RIGHT$(i)$
3  if $l \leq A.heap\text{-}size$ and $A[l] > A[i]$
4      $largest = l$
5  if $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$
6      $largest = r$
7  if $larget \neq i$
8      exchange $A[i]$ with $A[largest]$
9      MAX-HEAPIFY$(A, largest)$
```

```
BUILD-MAX-HEAP$(A)$
1  $A.heap\text{-}size = A.length$
2  for $i = \lfloor A.length/2 \rfloor$ downto 1
3      MAX-HEAPIFY$(A, i)$
```

```
HEAP-SORT$(A)$
1  BUILD-MAX-HEAP$(A)$
2  for $i = A.length$ downto 2
3      exchange $A[1]$ with $A[i]$
4      $A.heap\text{-}size = A.heap\text{-}size - 1$
5      MAX-HEAPIFY$(A, 1)$
```

```
HEAP-MAXIMUM$(A)$
1  return $A[1]$
```

```
HEAP-EXTRACT-MAX$(A)$
1  if $A.heap\text{-}size] < 1$
2      error "heap underflow"
3  $max = A[1]$
4  $A[1] = A[A.heap\text{-}size]$
5  $A.heap\text{-}size = A.heap\text{-}size - 1$
6  MAX-HEAPIFY$(A, 1)$
7  return $max$
```

```
HEAP-INCREASE-KEY$(A, i, key)$
1  if $key \leq A[i]$
2      error "key isn't larger"
3  $A[i] = key$
4  while $i > 1$ and $A[$PARENT$(i)] < A[i]$
5      exchange $A[i]$ with $A[$PARENT$(i)]$
6      $i = $ PARENT$(i)$
```

```
HEAP-INSERT$(A, key)$
1  $A.heap\text{-}size = A.heap\text{-}size + 1$
2  $A[A.heap\text{-}size] = -\infty$
3  HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$
```

```
Quick-Sort($A, p, r$)
if $p < r$
    $q =$ Partition($A, p, r$)
    Quick-Sort($A, p, q - 1$)
    Quick-Sort($A, q + 1, r$)
```

```
Partition($A, p, r$)
$x = A[r]$
$i = p - 1$
for $j = p$ to $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
Return $i + 1$
```

```
Randomized-Partition($A, p, r$)
$i =$ Random($p, r$)
exchange $A[i]$ with $A[r]$
return Partition($A, p, r$)
```

```
Randomized-Select($A, p, r, i$)
if $p == r$
    return $A[p]$
$q =$ Randomized-Partiton($A, p, r$)
$k = q - p + 1$
if $i == k$ // pivot value is answer
    return $A[q]$
elseif $i < k$
    return Randomized-Select($A, p, q - 1, i$)
else return Randomized-Select($A, q + 1, r, i - k$)
```

$\Theta(n \lg n)$ sorting algorithm complexity analysis

$$\begin{aligned}
& & T(n) &= 2T\left(\frac{n}{2}\right) + n & \\
&\Rightarrow & T(2^k) &= 2T(2^{k-1}) + 2^k & (n = 2^k) \\
&\Rightarrow & u_k &= 2u_{k-1} + 2^k & (T(2^k) = u_k) \\
&\Rightarrow & 2^k v_k &= 2 \cdot 2^{k-1} v_{k-1} + 2^k & (u_k = 2^k v_k) \\
&\Rightarrow & v_k - v_{k-1} &= 1 & \\
&\Rightarrow & \sum_{i=1}^{k} v_i - v_{i-1} &= \sum_{i=1}^{k} 1 & \\
&\Rightarrow & v_k - v_0 &= k & \text{(telescoping sum)} \\
&\Rightarrow & v_k &= v_0 + k & \\
&\Rightarrow & u_k &= 2^k(v_0 + k) = v_0 2^k + k2^k & (u_k = 2^k v_k) \\
&\Rightarrow & T(2^k) &= v_0 2^k + k2^k & (T(2^k) = u_k) \\
&\Rightarrow & T(n) &= v_0 n + n \lg n & (n = 2^k) \\
&\Rightarrow & T(n) &= \Theta(n \lg n) &
\end{aligned}$$

Showing Randomized-Select has an expected running time of $\Theta(n)$

$$\begin{aligned}
& & T(n) &= T\left(\frac{n}{2}\right) + n & \\
&\Rightarrow & T(2^k) &= T(2^{k-1}) + 2^k & (n = 2^k) \\
&\Rightarrow & u_k &= u_{k-1} + 2^k & (T(2^k) = u_k) \\
&\Rightarrow & 2^k v_k &= 2^{k-1} v_{k-1} + 2^k & (u_k = 2^k v_k) \\
&\Rightarrow & 2v_k - v_{k-1} &= 2 &
\end{aligned}$$

$2v_k - v_{k-1} = 2$ has homogenous part $2v_k - v_{k-1} = 0$ yielding characteristic polynomial $2x - 1$ with root $x = \frac{1}{2}$. So the homogenous solution is $v_k = A \cdot (\frac{1}{2})^k$. This gives us a general solution to our recurrance $v_k = A \cdot (\frac{1}{2})^k + B$. We can now substitute back to find our Randomized-Select expected running time...

$$\begin{aligned}
& & v_k &= A \cdot \left(\frac{1}{2}\right)^k + B & \\
&\Rightarrow & u_k &= 2^k \left[A \cdot \left(\frac{1}{2}\right)^k + B\right] & (u_k = 2^k v_k) \\
& & &= A + B \cdot 2^k & \\
&\Rightarrow & T(2^k) &= A + B \cdot 2^k & (T(2^k) = u_k) \\
&\Rightarrow & T(n) &= A + Bn & (n = 2^k) \\
&\Rightarrow & T(n) &= \Theta(n) &
\end{aligned}$$

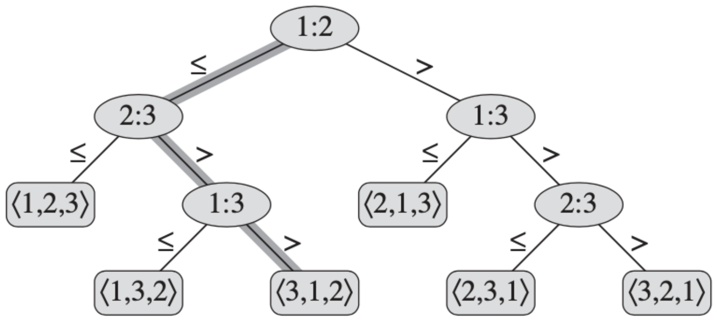

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i{:}j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

Theorem 8.1
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height h has no more than 2^h leaves, we have

$$n! \leq l \leq 2^h ,$$

which, by taking logarithms, implies

$$\begin{aligned} h &\geq \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.19))} . \end{aligned}$$
■

Corollary 8.2
Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof The $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from Theorem 8.1. ■

Matrix Chain Multiplication

Step 1: Characterize Structure of an Optimal Solution

Take $i \leq j$ and let $A_{i \ldots j}$ denote the product $A_i A_{i+1} \cdots A_j$. If $i = j$ then we already have an optimal solution for this case, because it requires no more computation. If $i < j$, then we can take some k where $i \leq k < j$, and compute the multiplication $A_{i \ldots k}$ and $A_{k+1 \ldots j}$, then multiply them together. The cost of this parenthesization is the cost to compute $A_{i \ldots k}$ plus the cost to compute $A_{k+1 \ldots j}$, plus the cost to multiply them together.

Suppose, to optimally parenthesize $A_{i \ldots j}$, we split the product between A_k and A_{k+1}. The multiplication $A_i \ldots A_k$ must have an optimal parenthesization, otherwise we could take a less costly parenthesization of $A_i \ldots A_k$, which would mean there'se a less costly way to parenthesize $A_{i \ldots j}$, but this is a contradiction, thus any subchain of $A_{i \ldots j}$ must be an optimal parenthesization without loss of generality.

Step 2: Recursively define the value of an optimal solution

Now we define the cost of an optimal solution recursively in terms of the cost of it's subproblems. Let $m[i,j]$ denote the minimum number of scalar multiplications to compute the matrix $A_{i \ldots j}$, so the lowest cost of the whole problem is $m[1,n]$. If $i = j$ then no scalar multiplications, so $m[i,j] = 0$. If $i < j$ then we can use the structure of our optimal solution. Assuming the optimal split of our multiplication is between A_k and A_{k+1}, then we get that $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$. $p_{i-1}p_k p_j$ represents the number of scalar multiplications for $A_{i \ldots k} A_{k \ldots j}$. Since we don't actually know k we get the following recurrance definition...

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}, & \text{if } i < j. \end{cases}$$

We must also define $s[i,j]$ to be the value of k at which the split at A_k, A_{k+1} yields the optimal parenthesization. Without this, we won't actually be able to find our solution.

Step 3: Compute the value of an optimal solution

Dynamic programming is meant for two specific cases. The first is optimal substructure, and the second is overlapping subproblems, which this problem has a lot of. Instead of recursion, we can use the tabular, bottom-up approach.

Assuming matrix A_i has dimensions $p_{i-1} \times p_i$, the input of our Matrix-Chain-Order formula will be a sequence $p = \langle p_0, \ldots, p_n \rangle$. The algorithm will use auxiliary table $m[i,j]$ for storing costs, and auxiliary table $s[i,j]$ for actually constructing the optimal solution.

To actually implement the bottom-up approach, we must determine entries of the table we refer to when computing $m[i,j]$. We know the cost of computing $A_{i \ldots j}$ is dependent on the cost of computing $A_{i \ldots k}$ and $A_{k+1 \ldots j}$ for some $i \leq k < j$. This means the cost of $A_{i \ldots j}$ depends only on the cost of matrix-chain-products of fewer matrices. So we must fill in table m in a manner that corresponds to increasing length of matrix chain multiplications.

```
MATRIX-CHAIN-ORDER($p$)
n = $p.length$ - 1
let $m[1 \ldots n, 1 \ldots n]$ and $s[1 \ldots n, 1 \ldots n]$ be new tables
**for** $i = 1$ **to** $n$
    $m[i,i] = 0$
**for** $l = 2$ **to** $n$ // $l$ is the chain length
    **for** $i = 1$ **to** $n - l + 1$
        $j = i + l - 1$
        $m[i,j] = \infty$
        **for** $k = i$ **to** $j - 1$
            $q = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
            **if** $q < m[i,j]$
                $m[i,j] = q$
                $s[i,j] = k$
**return** $m$ and $s$
```

This is bottom up, because we first assign 0 to all 1-length chains, then take a new chain length of 2, compute every possible multiplication and assign them to their corresponding locations in m. We keep repeating this loop, increasing the chain length until we find the solution for our entire procedure.

This is done in $\Theta(n^3)$ time, with $\Theta(n^2)$ space, which is much better than the exponential brute-force approach.

Step 4: Construct an optimal solution from computed information

We made sure to also keep track of an extra auxiliary array s to store the actual locations of our splits for an optimal parenthesization. $s[i,j]$ records the value of k such that an optimal parenthesization of $A_iA_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}. So to compute $A_{1\ldots n}$ optimally, we must take $A_{1\ldots k}A_{k+1\ldots n} = A_{1\ldots s[i,j]}A_{s[i,j]+1\ldots n}$. Then to find the value where the two subproblems are split, we check $s[1,k] = s[1,s[1,n]]$ and $s[k+1,n] = s[s[1,n]+1,n]$.

The following algorithm will print the optimal parenthesization of $\langle A_i, A_{i+1}, \ldots A_j\rangle$. The initial call to PRINT-OPTIMAL-PARENS should be $i = 1$ and $j = n$.

```
PRINT-OPTIMAL-PARENS($s, i, j$)
If $i == j$
    print "$A$"$_i$
**else** print "("
    PRINT-OPTIMAL-PARENS($s, i, s[i,j]$)
    PRINT-OPTIMAL-PARENS($s, s[i,j] + 1, j$)
    print ")"
```

In order for us to use optimal substructure for a problem, our subproblems must always be independent. Meaning that the solution to one subproblem must not affect another. Shorts paths for vertices has independent subproblems, but not longest paths. Dynamic programming solutions also need overlapping subproblems so it can solve it once, then store the answe in an auxiliary array where it can be looked up when needed, using constant-time lookup.

Using regular recursion to solve the matrix chain multiplication problem actually takes $\Theta(2^n)$ time because subproblems are called multiple times, and subsequent calling of subproblems isn't linear like our above bottom-up approach. Also when reconstructing the optimal solution, it's crucial to keep the $s[i,j]$ table to keep track of the actuall values, without this table it becomes a bit slower to recall back with only the $m[i,j]$ table.

Memoization takes the top-down approach of regular recursion, but also implements a table like in the more efficient bottom-up approach. Each table entry initially contains a value to indicate that it hasn't been filled out yet. When the subproblem is first encountered as the recursive algorithm unfolds, it's solution is computed, then stored in the table, then any subsequent lookup can just be found in the table from then on.

```
MEMOIZED-MATRIX-CHAIN($p$)
$n = p.length - 1$
let $m[1 \ldots n, 1 \ldots n]$ be a new table
**for** $i = 1$ **to** $n$
    **for** $j = 1$ **to** $n$
        $m[i,j] = \infty$
**return** LOOKUP-CHAIN($m, p, 1, n$)
```

```
LOOKUP-CHAIN($m, p, i, j$)
**if** $m[i,j] < \infty$
    **return** $m[i,j]$
**if** $i == j$
    $m[i,j] = 0$
**else for** $k = i$ **to** $j - 1$
        $q$ = LOOKUP-CHAIN($m, p, i, k$)
            +LOOKUP-CHAIN($m, p, k+1, j$)
            $+p_{i-1}p_kp_j$
        **if** $q < m[i,j]$
            $m[i,j] = q$
**return** $m[i,j]$
```

It first checks to see if the value is less than infinity, if so, then it just returns the value because this part of the table has already been computed. Then it goes through and recursively calls LOOKUP-CHAIN which will recurse all the way to the cases where $i == j$, and plop down a 0. Then the recursion works it's way back, adding in new values to m as the call stack reduces.

You can think of the bottom-up tabular approach as a sort of breadth-first-search, and the top-down memoization approach as a sort of depth-first search.

Longest Common Subsequence

Step 1: Characterizing a longest common subsequence

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ and sequence $Y = \langle y_1, y_2, \ldots, x_n \rangle$. We will let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

Theorem (Optimal Substructure of an LCS)

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 Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies Z is an LCS of X and Y_{n-1}.

Proof

For (1), if $z_k \neq x_m$ then we could append $x_m = y_n$ to Z to get a solution of length $k+1$, this contradicts Z being the longest common subsequence. Now we have that Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}. Suppose that theres a subsequence W_{k-1} that's longer than Z_{k-1}, then appending $x_m = y_n$ gives us a longer subsequence which is again a contradiction.

Without loss of generality, for (2) and (3), if $z_k \neq x_m$ then Z is a common subsequence of X_{m-1} and Y. Suppose there's a subsequence W of X_{m-1} and Y with length greater than k, then W is also a subsequence of X but this is a contradiction because the LCS of X and Y is of length k. Therefore, since an LCS of two sequences contains an LCS of the prefixes of the two sequences, this problem has optimal substructure.

Step 2: A recursive solution

When $x_m = y_n$, we must find the LCS of X_{m-1} and Y_{n-1}, and appending $x_m = y_n$ gives us the LCS for X and Y. And when $x_m \neq y_n$, there are possibilities we must cover. We find the LCS of X_{m-1} and Y and we find the LCS of X and Y_{n-1}. Whichever one is longer is the one we use for the LCS of X and Y.

We also have overlapping subproblems, because if we must find LCS of X_{m-1} and Y and LCS of X and Y_{n-1}, then they may share an LCS subproblem of X_{m-1} and Y_{n-1}. Lots of other subproblems also share subproblems. Our recursive solution to LCS involves establishing a recurrance for the value of an optimal solution. Let $c[i,j]$ be the lenghth of the LCS of X_i and Y_j, this gives us the following recurrence...

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

LCS is actually a problem that rules out subproblems depending on the conditions of the problem. For example, if $x_i = y_j$ then we only check the subproblem of finding the LCS of X_{i-1} and Y_{j-1}. But if $x_i \neq y_j$, then we actually have two subproblems.

Step 3: Computing the length of an LCS

The LCS problem has very few distinct subproblems, so we can use dynamic programming to compute the solutoin bottom up. LCS-Length takes two inputs, $X = \langle x_1, x_2, \ldots, x_m \rangle$, and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Stores the LCS values for X_i and Y_j in $c[i, j]$. It will also use $b[1 \ldots m, 1 \ldots n]$ to help construct our optimal solution.

$b[i, j]$ will point to the entry containing the optimal subproblem solution, so if

```
LCS-Length($X, Y$)
m = $X.length$
n = $Y.length$
let $b[1 \ldots m, 1 \ldots n]$ and $c[0 \ldots m, 0 \ldots n]$ be new tables
for $i = 0$ to $m$
    $c[i, 0] = 0$
for $j = 0$ to $n$
    $c[0, j] = 0$
for $i = 1$ to $m$
    for $j = 1$ to $n$
        if $x_i == y_j$
            $c[i, j] = c[i-1, j-1] + 1$
            $b[i, j] =$ "$\nwarrow$"
        elseif $c[i-1, j] > c[i, j-1]$
            $c[i, j] = c[i-1, j]$
            $b[i, j] =$ "$\uparrow$"
        else $c[i, j] = c[i, j-1]$
            $b[i, j] =$ "$\leftarrow$"
return $c$ and $b$
```

Step 4: Constructing an LCS

Now that we have our auxiliary table b all we need to do is start at $b[m, n]$ and work our way back, printing a letter every time we encounter a "$\nwarrow$". But we don't want to print our LCS in reverse order, so we'll use tail recursion, actually adding the letter after the recursive call.

```
Print-LCS($b, X, i, j$)
if $i == 0$ or $j == 0$
    return
if $b[i, j] ==$ "$\nwarrow$"
    Print-LCS($b, X, i-1, j-1$)
    print $x_i$
elseif $b[i, j] ==$ "$\uparrow$"
    Print-LCS($b, X, i-1, j$)
else Print-LCS($b, X, i, j-1$)
```

We only need X for this because we're finding a substring, so either one would work. Also to get the whole LCS with the above function, we would call Print-LCS($b, X, X.length, Y.length$).

Activity Selection Problem

Given a set of activities $S = \{a_1, a_2, \ldots, a_n\}$ with a_i's corresponding start and finish times s_i and f_i, we want to find the largest numbers of non-overlapping activities. Take $[s_i, f_i)$ and $[s_j, f_j)$, they're compatible if $s_j \geq f_i$ or if $s_i \geq f_j$.

Optimal substructure of activity-selection

Let S_{ij} denote the set of activities that start during or after f_i and finish before s_j. Also let A_{ij} be the maximum non-overlapping activity set for S_{ij}. If A_{ij} has activity a_k, then we can split into two subproblems, giving us $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and the maximum sized set A_{ij} has $|A_{ik}| + |A_{kj}| + 1$ activities.

If we find an A'_{ik} where $|A'_{ik}| > |A_{ik}|$ then we could could substitute it in to find (A'_{ij} where $|A'_{ij}| > |A_{ij}|$ which is a contradiction, because we assume A_{ij} is a maximal set of activities. This means that dynamic programming may be a good idea.

Letting $c[i, j]$ represent the size of the optimal solution for S_{ij} we get that...

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max\limits_{a_k \in S_{ij}} \big(c[i,k] + c[k,j] + 1\big) & \text{if } S_{ij} \neq \emptyset. \end{cases}$$

Making the greedy choice

We must choose the activity that leaves resources available for as many other activities as possible. Let's choose our greedy choice to be the activity with the earliest finish time. Since the activities are sorted in monotonically increasing order by finish time, our greedy choice is a_1.

Making the greedy choice leaves one subproblem, finding activities starting after a_1 finishes. Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities starting after a_k finishes. Greedy choice a_1 leaves subproblem S_1. Now we must see if this greedy choice does actually yield an optimal solution.

Theorem

Consider non-empty subproblem S_k, and let a_m be the activity with the earliest finish time in S_k. Then a_m is in a maximum mutually-exclusive subset of S_k.

Proof

Let A_k be a maximum size subset of S_k and let a_j be the activity with the earliest finish time in A_k. If $a_k = a_m$ then the optimal solution contains our greedy choice, and we are done. Otherwise, we can let $A'_k = A_k - \{a_j\} \cup \{a_m\}$. The activities in A'_k are disjoint because A_k is disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Therefore, since $|A'_k| = |A_k|$, we have a new solution to the problem which now contains our greedy choice.

Finding a greedy algorithm approach can work using the top-down method, because we can simply choose an activity to put into our optimal solution, then recursively find the next activity. Greedy algorithms usually work top-down, where you make a choice, then solve the subproblem, as opposed to the bottom-up approach.

Recursive greedy algorithm

we can now write a straightforward recursive algorithm to find the maximum disjoint set of activities. Recursive-Activity-Selector takes arrays s and f for start and finish times, index k representing the S_k subproblem, and size n of the original problem. We assum the n activities are sorted by finish time. To start the recursive call, we create activity a_0 with finish time $f_0 = 0$ so S_0 contains all the activities. So our call is Recursive-Activity-Selector$(s, f, 0, n)$.

```
Recursive-Activity-Selector($s, f, k, n$)
1  $m = k + 1$
2  while $m \leq n$ and $s[m] < f[k]$ // find first activity in $S_k$ to finish
3      $m = m + 1$
4  if $m \leq n$
5      return $\{a_m\} \cup$ Recursive-Activity-Selector($s, f, m, n$)
6  else return $\emptyset$
```

This algorithm has a running time of $\Theta(n)$ because every element of S is checked once. If the while look stops at some element k, then the next recursive call picks up right where the previous one left off, thus iterating through every activity once.

Iterative greedy algorithm

Since the recursive call in the above greedy algorithm is called at the end of the function, we can actually use an iterative approach for this problem as well. This type of recursion is called tail recursion, and is actually quite easy to turn into an iterative algorithm. Some compilers will actually automatically turn a tail-recursive algorithm into an iterative one. Greedy-Activity-Selector assumes activities are sorted and creates a set A containing the activities, returning it at the end.

```
Greedy-Activity-Selector($s, f$)
1  $n = s.length$
2  $A = \{a_1\}$
3  $k = 1$
4  for $m = 2$ to $n$
5      if $s[m] \geq f[k]$
6          $A = A \cup \{a_m\}$
7          $k = m$
8  return $A$
```

This algorithm essentially does the same thing as calling Recursive-Activity-Selector$(s, f, 0, n)$. And You can now, more obviousely, see that the running time is $\Theta(n)$.

Elements of a Greedy Strategy

The heuristic greedy approach doesn't always work, but there are specific cases that it does. If you want to fashion a solution to a problem with a greedy choice in mind, then you can follow these steps. . .

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the problem that contains the greedy choice, thus making the greedy choice safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that combining the optimal solution to the subproblem with our greedy choice yields an optimal solution. Inductively, this shows that making the greedy choice at every inductive step also yields an optimal solution.

Greedy-choice propety

In dynamic programming, we usually make a choice at every step, but that choice usually depends on solutions to subproblems. This is why dynamic programming usually works bottom-up. Unless we want to use memoization, which technically works top-down, but still must first answer subproblems. With greedy algorithms, we just make a choice that seems right, then solve the remaining subproblem. Greedy algorithm choices may depend on previous choices, but never depend on solutions to subproblems. Therefore, greedy algorithms actually do work top-down. Proving that greedy algorithms work is usually done by taking a global optimal solution, replacing a choice with the greedy choice, and showing that it's still an optimal solution.

Optimal substructure

A problem exhibits optimal substructure if an optimal solution to a problem contains within it optimal solutions to subproblems. For greedy algorithms, all we really need to prove optimality is to prove that an optimal solution to the subproblem, combined with the greedy choice yields an optimal solution to the original problem.

Greedy vs dynamic programming

Not all problems can be solved using a greedy algorithm. For example, take the knapsack problem. For the 0-1 knapsack problem and the fractional knapsack problem, they both contain optimal substructure, but only one can be solved using a greedy algorithm.

For the fractional knapsack problem, if we take our greedy choice to be the item with the most valuable density, then taking as much of that before running out (or running out of space in the knapsack), then this solution is optimal.

For the 0-1 knapsack problem though, take items o_1, o_2, o_3 with $w_1 = 10, w_2 = 20, w_3 = 30$ and $c_1 = 60, c_2 = 100, c_3 = 120$. Also assume knapsack holds 50 lbs. Greedy strategy would be to take objects with highest money density, which is o_1 and o_2. If we also put o_3 in the sack, we have too much weight, so we can only carry the first two. This gives us a price of \$160. But the actual optimal solution is to take o_2 and o_3, this hits our knapsack weight perfectly, and gives us a price of \$220. So not all problems can be solved with greedy algorithms.

Huffman Codes

Huffman's greedy algorithm takes in a table of character counts and produces a variable length bit sequence for each character. It does this with ***prefix codes***: no codeword is a prefix of another. Prefix codes help simplify decoding. We just identify a codeword, convert it to it's corresponding character, then move on to the rest of the prefix code.

To optimally represent the prefix code, we must use a full binary tree, where every internal node has two children. Given a tree T corresponding to a prefix code, we can compute the number of bits needed to encode a file. For any leter c in alphabet C, we can let $d_T(c)$ denote the depth of c's leaf in T. So to find the number of bit's needed to encode a file, we use this equation...

$$\sum_{c \in C} c.\mathit{freq} \cdot d_T(c)$$

Constructing a Huffman code

Assuming C is a set of n characters, the algorithm Huffman(C) builds the tree T corresponding to the optimal code bottom-up. Starts with $|C|$ leaves, and merges them $|C|-1$ times to create the final tree. We must use a min-priority-queue Q sorted by $c.\mathit{freq}$. We take the two least frequent characters and merge them, we work upwards until everything is merged, and more frequent characters have less depth.

```
HUFFMAN($C$)
1  $n = |C|$
2  $Q = C$
3  for $i = 1$ to $n - 1$
4      allocate new node $z$
5      $z.left = x =$ EXTRACT-MIN($Q$)
6      $z.right = y =$ EXTRACT-MIN($Q$)
7      $z.freq = y.freq + y.freq$
8      INSERT($Q, z$)
9  return EXTRACT-MIN($Q$) // return root of tree
```

Initializing the min-heap Q is $O(n)$ with the Build-Min-Heap procedure. Then we iterate through the for-loop which is $O(n)$. Inside each for loop iteration, we call Extract-Min twice and Insert once, all of these are $O(\lg n)$ operations, so this makes the whole Huffman algorithm run in $O(n \lg n)$ time.

Correctness of Huffman's algorithm

To prove that this Huffman's algorithm is correct we must show that the problem of an optimal prefix code exhibits a greedy-choice and optimal substructure property.

Greedy Algorithm Lemma. Let x and y be two characters in C having lowest frequencies. Then there exists an optimal prefix code where x and y have the same length and only differ in the last bit. Proof on the next page.

Proof. Let a and b be two sibling leaves of max depth on tree T. WLOG, we assum $a.\mathit{freq} \leq b.\mathit{freq}$ and $x.\mathit{freq} \leq y.\mathit{freq}$. Since x and y are the lowest frequency leaves, we know $x.\mathit{freq} \leq a.\mathit{freq}$ and $y.\mathit{freq} \leq b.\mathit{freq}$. If they're equal, then lour lemma holds, as our greedy choice already does yield an optimal solution.

So let's assume $x \neq b$. Now we can take tree T replacing x with a making a new tree T'. Then replacing y with b giving us T''. We can take size of a file from T and subtract the size of a file from T' to show T' cannot be greater.

$$\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} c.\mathit{freq} \cdot d_T(c) - \sum_{c \in C} c.\mathit{freq} \cdot d_{T'}(c) \\
&= x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_{T'}(x) - a.\mathit{freq} \cdot d_{T'}(a) \\
&= x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_T(a) - a.\mathit{freq} \cdot d_T(x) \\
&= (a.\mathit{freq} - x.\mathit{freq}) \cdot (d_T(a) - d_T(x)) \\
&\geq 0
\end{aligned}$$

So since $B(T') \leq B(T)$, and WLOG, $B(T'') \leq B(T')$, we have $B(T'') \leq B(T)$, And since T is an optimal tree, $B(T'') \geq B(T)$, so it must also be the case that T'' is also an optimal tree. □

Optimal Substructure Lemma. Let C' be the alphabet with characters x and y removed and new character z added where $z.\mathit{freq} = y.\mathit{freq} + y.\mathit{freq}$. Let T' be any tree representing optimal prefix code for C'. Then the tree T obtained from T' by replacing the leaf node for z with an internal node containing x and y as children, represents an optimal prefix code for C.

Proof. We first must express the cost $B(T)$ in terms of $B(T')$. First note that for any $c \in C - \{x, y\}$, $d_T(c) = d_{T'}(c)$. Also $d_T(x) = d_T(y) = d_{T'}(z) + 1$. So we get $B(T) = B(T') + x.\mathit{freq} + y.\mathit{freq}$.

Now, suppose to the contrary, that T is not an optimal prefix code for C. Then there exists tree T'' such that $B(T'') < B(T)$. x and y are siblings in this tree because of the previous lemma, so if we take T''' wheren x and y are replaced by node z then we get the following relation...

$$\begin{aligned}
B(T''') &= B(T'') - y.\mathit{freq} - y.\mathit{freq} \\
&\leq B(T) - y.\mathit{freq} - y.\mathit{freq} \\
&= B(T')
\end{aligned}$$

But since we assumed that T' represents an optimal prefix code, we have a contradiction, so it must be the case that T represents an optimal prefix code of C. □

Huffman Algorithm Theorem. The HUFFMAN algorithm produces optimal prefix code.

Proof. look at previous lemmas. □

Disjoint set forests

```
MAKE-SET($x$)
1  $a.p = x$
2  $x.rank = 0$
```

```
UNION($x, y$)
1  LINK(FIND-SET($x$), FIND-SET($y$))
```

We use union by rank to ensure our sets don't get unnecessarily long. We take the longer of the two sets and use their root as the root of the new set.

```
LINK($x, y$)
1  if $x.rank > y.rank$
2        $y.p = x$
3  else $x.p = y$
4        if $x.rank == y.rank$
5              $y.rank = y.rank + 1$
```

We can use path compression for the FIND-SET operation, so every time we traverse down a set, we update the parents of each node to be the root of the set. This makes subsequent calls to FIND-SET faster, thus also making UNION faster.

```
FIND-SET($x$)
1  if $x.p \neq x$
2        $x.p = $ FIND-SET($x.p$)
3  return $x.p$
```

Both the union by rank heuristic and the path compression heuristic improve our data structures slightly. When implementing both, we get a running time of $O(m\alpha(n))$, where α is a very slowly growing function. This basically means running time is linear.

Representation of graphs

You can represent graphs as a list or a matrix. For sparce graphs (less edges), a list is a better representation, as it takes up much less space. But all-pairs shortest path algorithm uses matrices because edges are more dense.

Adjacency lists are represented as an array of $|V|$ lists (one for each vertex), and each adjacency list $Adj[u]$ contains nodes v st edge $(u, v) \in E$. if G is directed, then sum of adjacency list lengths are $|E|$, and $2|E|$ for undirected graphs. Space complexity is $\Theta(|V| + |E|)$. For storing weights, we can add a weight attribute $w(u, v)$ with vertex v in u's adjacency-list.

Adjacency matrices on the other hand have $\Theta(|V^2|)$ space complexity but they do have constant-time lookup for edges, where the list representation requires searching through the $Adj[u]$ list. Then for each element of the matrix, a_{ij} recieves a 1 if there exists an edge from i to j and a 0 otherwise. This makes adding weight easy because we can instead just add $w(i, j)$ to the entry.

Minimum Spanning Trees

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a minimum spanning tree
3      find an edge (u, v) that's safe for A
4      A = A ∪ {(u, v)}
5  return A
```

cut: $(S, V - S)$ is a cut of G if it partitions G into two sets of vertices.
cross: an edge (u, v) crosses the cut if $u \in S$ and $v \in V - S$ or vice versa.
respect: a cut respects a set A of edges, if no edge in A crosses said cut.
light edge: an edge is a light edge if it's the minimum weight of any edge that crosses the cut.
safe edge: an ege that can be added to A to keep a as a subset of some MST.

Safety of an Edge Theorem. If $A \subseteq E$ in some minimum spanning tree for G. Then take cut $(S, V - S)$ of G that respects A, if we append light edge (u, v) crossing our cut to A then A is still a subset of our MST G, hence (u, v) is safe for A.

Proof. let T be an MST that includes A and doesn't include edge (u, v). We create new MST T' with $A \cup \{(u, v)\}$, thus showing (u, v) is safe for A. Since T is an MST, if we add (u, v) to T, it would create a cycle with another edge (x, y) crossing cut $(S, V - s)$.

Since the cut respects A, (x, y) cannot be in A. If we remove (x, y) from T, and add in (u, v) we still have a tree because they cross the same cut. So take $T' = (T - \{(x, y)\}) \cup (u, v)$, now we show T' is a minimum spanning tree. $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$. Therefore T' must also be an MST, thus (u, v) is safe for A. □

Now we go over two algorithms for making minimum spanning trees. The first is Kruskal's algorithm, where the safe edge added is always the least-weight edge in the graph that connects two distinct components. This basically creates tons of mini MST's before finally connecting everything into one big MST.

Prim's algorithm is the other. In this case our set A forms a single tree, and the safe edge added to A is always a least-weight edge connecting the existing tree to a vertex not in the tree.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort edges of G.V in increasing order by weight w
5  for each edge (u, v) ∈ G.E
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

Assuming we're using the disjoint set forest data structure with our heuristics, we get the following complextiy. Sorting the edges is done in $O(E \lg E)$ time. The loop does $O(E)$ FIND-SET and UNION operations. Along with the $|V|$ MAKE-SET calls, we get a running time of $O(V+E)\alpha(V)$. Since $|E| \geq |V| - 1$, we get $E\alpha(V)$, and since $\alpha(V) = O(\lg V)$, we get that the overall running time is $O(E \lg V)$.

For Prim's algorithm, we'll need a fast way to select a new edge for the tree, so we will need two new variables. We also keep a min-priority-queue queue for all the vertices that aren't in the tree yet. The queue is sorted by the vertices *key* attribute, which represents the weight of the minimum weight edge connecting our vertex to the tree. If there is no edge currently connecting an edge v to the tree, then we set $v.key = \infty$, and $v.\pi =$ NIL. π represents the penultimate node with the property that $(v, v.\pi) = v.key$. Finally when the algorithm terminates, we have an empty queue, because every vertex is in our tree.

```
MST-PRIM($G, w, r$)
 1  for each $u \in G.V$
 2      $u.key = \infty$
 3      $u.\pi =$ NIL
 4  $r.key = 0$
 5  $Q = G.V$
 6  while $Q \neq \emptyset$
 7      $u =$ EXTRACT-MIN($Q$)
 8      for each $v \in G.Adj[u]$
 9          if $v \in Q$ and $w(u, v) < v.key$
10              $v.\pi = u$
11              $v.key = w(u, v)$
```

For the running time, assuming we're using a min-heap for Q, we must use the BUILD-MIN-HEAP operation which is $O(V)$ time. The while loop executes $|V|$ times, and EXTRACT-MIN takes $O(\lg V)$ time, so we get $O(V \lg V)$ for that. The for loop is executed exactly $|E|$ times, and since we must call DECREASE-KEY every time the *key* value is adjusted, that's another $O(\lg V)$ time. So we get the the overall running time is $O(V \lg V + E \lg V) = O(E \lg V)$, this is the same as for Kruskal's algorithm.

Single Source Shortest Path

```
INITIALIZE-SINGLE-SOURCE($G, s$)
1  for each vertex $v \in G.V$
2      $v.d = \infty$
3      $v.\pi =$ NIL
4  $s.d = 0$
```

```
RELAX($u, v, w$)
1  if $v.d > u.d + w(u, v)$
2      $v.d = u.d + w(u, v)$
3      $v.\pi = u$
```

We will implement Dijkstra's algorithm on the next page.

```
DIJKSTRA$(G, w, s)$
1  INITIALIZE-SINGLE-SOURCE$(G, s)$
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN$(Q)$
6      $S = S \cup \{u\}$
7      for each $v \in G.Adj[u]$
8          RELAX$(u, v, w)$
```

This is a greedy strategy because the algorithm chooses the ligthest weight vertex in $V - S$.

Now we can attempt to calculate the running time. The for loop is called $|E|$ times as it covers every edge for every vertex. So DECREASE-KEY is called at most $|E|$ times. The while loop iterates $|V|$ times, and EXTRACT-MIN is $O\lg V$. Also building our heap is $O(V)$, and each DECREASE-KEY takes $O(\lg V)$. So we get a running time of $O(V \lg V) + O(E \lg V) = O(E \lg V)$.

We can use a couple different data structures to achieve slightly better running times but this is essentially it.

All Pairs Shortest Path

This algorithm takes the shortest $1 \dots k$ length path, and computes the next shortest path up to $k + 1$ length

```
EXTEND-SHORTEST-PATHS$(L, W)$
1  $n = L.rows$
2  let $L' = (l'_{ij})$ be a new $n \times n$ matrix
3  for $i = 1$ to $n$
4      for $j = 1$ to $n$
5          $(l'_{ij}) = \infty$
6          for $k = 1$ to $n$
7              $(l'_{ij}) = \min\{l'_{ij}, l_{ik} + w_{kj}\}$
8  return $L'$
```

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \quad \Rightarrow \quad l_{ij}^{(m-1)} = \min_{1 \le k \le n} l_{ik}^{(m)} + w_{kj}$$

$$a \to l^{(m-1)} \qquad b \to w \qquad c \to l^{(m)} \qquad + \to \min \qquad \cdot \to +$$

We can start with the W matrix and call EXTEND-SHORTEST-PATHS $n - 1$ times to give us our all-pairs shortest paths matrix, and this can be done in $O(V^4)$ time. We can actually improve this by instead doing matrix multiplication on two W matrices, then multiple them together, and keep doing this until we reach a matrix power greater than or equal to $n - 1$. This makes the algorithm $O(V^3 \lg V)$ and I'll show it on the next page.

```
FASTER-ALL-PAIRS-SHORTEST-PATHS($W$)
1  $n = W.rows$
2  $L^{(1)} = W$
3  $m = 1$
4  while $m < n - 1$
5      let $L^{(2m)}$ be a new $n \times n$ matrix
6      $L^{(2m)} =$ EXTEND-SHORTEST-PATHS($L^{(m)}, L^{(m)}$)
7      $m = 2m$
8  return $L^{(m)}$
```

We can also delete the previously created matrix through each iteration to keep $O(V^2)$ space complexity.

Floyd-Warshall Algorithm

Floyd-Warshall actually runs in $\Theta(V^3)$ time, which is the best so far to solve the all-pairs shortest paths problem. The Floyd-Warshall algorithm considers intermediate vertices. One-by-one, we pick a vertex, then update the all shortest paths containing our vertex as an intermediary. Now, when we pick vertex k, we already looked at vertices $1, \ldots, k-1$, so for every path (i, j), there are 2 possible cases...

- k is not an intermediate vertex on the path from i to j. So the shortest path with intermediate vertices in the set $\{1, \ldots, k-1\}$ is also the shortest path with intermediate verticies in the set $\{1, \ldots, k-1\}$.
- k is an intermediate vertex of the shortest path, so we can now denote our path p as $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$. And since subpaths of optimal paths are optimal p_1 and p_2 are shortest paths. And since our intermediate node k was just introduced, The intermeiate vertices of p_1 are in $\{1, \ldots, k-1\}$, and same for p_2.

Let $d_{ij}^{(k)}$ be the weight of a shortest path from i to j for which all the intermediate vertices are in the set $\{1, \ldots, k-1\}$. When $k = 0$ there are no intermediate vertices, so paths are at most length 1. This means $d_{ij}^{(0)} = w_{ij}$, now we can define $d_{ij}^{(k)}$ recursively...

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1. \end{cases}$$

Now we can use a bottom-up procedure to compute $d_{ij}^{(k)}$ in order of increasing k. The following algorithm takes in our edge weight matrix W and calculates $D^{(n)}$ like so (algorithm on the next page). Also it's pretty easy to see it's $\Theta(V^3)$.

```
FLOYD-WARSHALL($W$)
1  $n = W.rows$
2  $D^{(0)} = W$
3  for $k = 1$ to $n$
4      let $D^{(k)}$ be a new $n \times n$ matrix
5      for $i = 1$ to $n$
6          for $j = 1$ to $n$
7              $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
8  return $D^{(n)}$
```

If we wanted to actually construct the shortest paths, we can create a predecessor matrix Π, We can use this recurrence to solve for $\pi_{ij}^{(k)}$...

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Transitive Closure of a Directed Graph

We might want to know, if for every pair of vertices i, j there exists a path. We define transitive closure of G as $G* = (V, E*)$, $E* = \{(i, j) : \text{there is a path from } i \text{ to } j\}$

We can compute transitive closure by letting all the edge weights be 1 and running FLOYD-WARSHALL. If $d_{ij} < n$ then there is an edge, and if $d_{ij} = \infty$, then there isn't. But there's also another way using less space.

To do this, we can do the bitwise equivalent of our previous algorithms...

$$+ \rightarrow \min \rightarrow OR \qquad \cdot \rightarrow + \rightarrow AND \qquad 0 \rightarrow \infty \rightarrow FALSE \qquad 1 \rightarrow 0 \rightarrow TRUE$$

Here's our algorithm for transitive closure...

```
TRANSITIVE-CLOSURE($G$)
 1  $n = |G.V|$
 2  let $T^{(0)} = \left(t_{ij}^{(0)}\right)$ be a new $n \times n$ matrix
 3  for $i = 1$ to $n$
 4      for $j = 1$ to $n$
 5          if $i == j$ or $(i, j) \in G.E$
 6              $t_{ij}^{(0)} = 1$
 7          else $t_{ij}^{(0)} = 0$
 8  for $k = 1$ to $n$
 9      let $T^{(k)} = \left(t_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
10      for $i = 1$ to $n$
11          for $j = 1$ to $n$
12              $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$
13  return $T^{(n)}$
```