• Merge procedure runs in $\Theta(n)$ time, where 
  $n = r-p+1$
  – each of lines 1–3 and 8–11 takes constant time
  – the for loops of lines 4–7 take $\Theta(n_1+n_2) = \Theta(n)$ time
  – there are $n$ iterations of the for loop of lines 12–17, each of which takes constant time

Merge sort

• The procedure $\text{MERGE-SORT}(A, p, r)$ sorts the elements in $A[p..r]$
• If $p \geq r$, the subarray has at most one element and is therefore already sorted
• Otherwise, the divide step computes an index $q$ that partitions $A[p..r]$ into two subarrays:
  – $A[p..q]$, containing $\lfloor n/2 \rfloor$ elements
  – $A[q+1..r]$, containing $\lceil n/2 \rceil$ elements
**Merge-Sort(A, p, r)**

1. if \( p < r \)
2. \( q \leftarrow \left\lfloor \frac{(p+r)}{2} \right\rfloor \)
3. **MERGE-SORT**(A, p, q)
4. **MERGE-SORT**(A, q+1, r)
5. **MERGE**(A, p, q, r)

**Analysis of merge sort**

- Our analysis assumes that the original problem size is a power of 2
- Each divide step then yields two subsequences of size exactly \( n/2 \)
- We set up the recurrence for \( T(n) \), the worst-case running time of merge sort on \( n \) numbers
- Merge sort on just one element takes constant time
• When we have \( n > 1 \) elements, we break down the running time as follows:
  
  - **Divide**: The step just computes the middle of the subarray, which takes constant time:
    \[ D(n) = \Theta(1) \]
  
  - **Conquer**: We recursively solve two subproblems, each of size \( n/2 \), which contributes \( 2T(n/2) \) to the running time
  
  - **Combine**: the MERGE procedure on an \( n \)-element array takes time \( \Theta(n) \), and so
    \[ C(n) = \Theta(n) \]

  When we add the \( D(n) \) and \( C(n) \), we are adding functions that are \( \Theta(n) \) and \( \Theta(1) \)

  This sum is a linear function of \( n \)

  Adding it to the \( 2T(n/2) \) term from the “conquer” step gives the recurrence for \( T(n) \):

  \[
  T(n) = \begin{cases} 
  \Theta(1) & \text{if } n = 1 \\
  2T(n/2) + \Theta(n) & \text{if } n > 1 
  \end{cases}
  \]
To intuitively see that the solution to the recurrence is $T(n) = \Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$, let us rewrite it as

$$T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  2T(n/2) + cn & \text{if } n > 1 
\end{cases}$$

where constant $c$ represents time required to solve problems of size 1 and that per array element of the divide and combine steps.
• Inductive argument shows that total number of levels of the recursion tree is $\lg n + 1$
  – base case $n = 1$: tree has only one level; $\lg 1 = 0 \Rightarrow \lg n + 1$ is the correct number of levels
  – Inductive hypothesis: number of levels of a tree with $2^i$ leaves is $\lg 2^i + 1 = i + 1$
  – We assume that the input size is a power of 2, the next input size to consider is $2^{i+1}$
  – A tree with $n = 2^{i+1}$ leaves has one more level than a tree with $2^i$ leaves, and so the total number of levels is $(i+1)+1 = \lg 2^{i+1} + 1$

• To compute the total cost represented by the recurrence, we simply add up the costs of all the levels:
  – The recursion tree has $\lg n + 1$ levels, each costing $cn$, for a total cost of
    $$cn(\lg n + 1) = cn \lg n + cn$$
  – Ignoring the low-order term and the constant $c$ gives the desired result of
    $$\Theta(n \lg n)$$
3 Growth of Functions

• Order of growth of the running time
  – gives a simple characterization of the algorithm’s efficiency
  – allows us to compare the relative performance of alternative algorithms
• Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs

3.1 Asymptotic notation

• For a function $g(n)$ we denote by $\Theta(g(n))$ the set of functions

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}$$
A function $f(n)$ belongs to $\Theta(g(n))$ if there exist $c_1$ and $c_2$ such that it can be “sandwiched” between $c_1g(n)$ and $c_2g(n)$ for sufficiently large $n$.

Because $\Theta(g(n))$ is a set, we could write “$f(n) \in \Theta(g(n))$”.

We usually write “$f(n) = \Theta(g(n))$” instead.

When $f(n) = \Theta(g(n))$ we say that $g(n)$ is an **asymptotically tight bound** for $f(n)$.

The definition requires that every member $f(n) \in \Theta(g(n))$ be asymptotically nonnegative:
- $f(n)$ be nonnegative whenever $n$ is sufficiently large.

The function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty.
Let us use the formal definition to show that 
\[ \frac{1}{2}n^2 - 3n = \Theta(n^2) \]
- We must determine positive \( c_1, c_2, \) and \( n_0 \) such that \( c_1n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2n^2 \) for all \( n \geq n_0 \)
- Dividing by \( n^2 \) yields \( c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 \)
- We can make the right-hand inequality hold for any value of \( n \geq 1 \) by choosing \( c_2 \geq \frac{1}{2} \)
- Likewise, we can make the left-hand inequality hold for any value of \( n \geq 7 \) by choosing \( c_1 \leq 1/14 \)
- Thus, by choosing \( c_1 \leq 1/14, c_2 \geq \frac{1}{2}, \) and \( n \geq 7 \) we can verify that \( \frac{1}{2}n^2 - 3n = \Theta(n^2) \)

We can also use the formal definition to verify that \( 6n^3 \neq \Theta(n^2) \)
- Suppose for the purpose of contradiction that \( c_2 \) and \( n_0 \) exist such that \( 6n^3 \leq c_2n^2 \) for all \( n \geq n_0 \)
- But then dividing by \( n^2 \) yields \( n \leq c_2/6 \), which cannot possibly hold for arbitrarily large \( n \), since \( c_2 \) is constant
**O-notation**

- \( \Theta \)-notation asymptotically bounds a function from above and below.
- When we have only an asymptotic upper bound, we use \( O \)-notation.
- For a function \( g(n) \) we denote by \( O(g(n)) \) the set of functions

\[
O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}
\]

Note that \( f(n) = \Theta(g(n)) \) implies \( f(n) = O(g(n)) \), since \( \Theta \)-notation is a stronger notion than \( O \)-notation.

- When we use \( O \)-notation to bound the worst-case running time, we have a bound on the running time of the algorithm on every input.
\section*{Ω-notation}

- \( \Omega \) – notation provides an asymptotic lower bound on a function

- For a function \( g(n) \) we denote by \( \Omega(g(n)) \) the set of functions
  \[\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}\]

\section*{Theorem 3.1}
For any two functions \( f(n) \) and \( g(n) \), we have
\[f(n) = \Theta(g(n))\]
if and only if \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \).

\textbf{Proof} Exercises. \( \blacksquare \)
• The running time of insertion sort is $\Omega(n)$
• The running time of insertion sort therefore belongs to both $\Omega(n)$ and $O(n^2)$
  • E.g., the running time of insertion sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time
• It is not contradictory, however, to say that the worst-case running time of insertion sort is $\Omega(n^2)$, since there exists an input that causes the algorithm to take $\Omega(n^2)$ time

**$o$-notation**

• The asymptotic bound provided by $O$-notation may or may not be asymptotically tight:
  – The bound $2n^2 = O(n^2)$ is asymptotically tight
  – The bound $2n = O(n^2)$ is not
• Formally define $o(g(n))$ as the set
  $$o(g(n)) = \{ f(n) : \text{for any positive constant } c > 0 \text{ there exists } n_0 > 0 \text{ such that } 0 < f(n) < cg(n) \text{ for all } n \geq n_0\}$$
For example, \( 2n = \Theta(n^2) \), but \( 2n^2 \neq \Theta(n^2) \).

The main difference between \( \Theta \)-notation and \( o \)-notation is that in

- \( f(n) = \Theta(g(n)) \), the bound \( 0 \leq f(n) \leq cg(n) \) holds for some constant \( c > 0 \),
- \( f(n) = o(g(n)) \), the bound \( 0 \leq f(n) < cg(n) \) holds for all constants \( c > 0 \).

Intuitively, in \( o \)-notation, the function \( f(n) \) becomes insignificant relative to \( g(n) \) as \( n \) approaches infinity; i.e.,

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0
\]

\( \omega \)-notation

We use \( \omega \)-notation to denote a lower bound that is not asymptotically tight.

One way to define it is by \( f(n) \in \omega(g(n)) \) if and only if \( g(n) \in o(f(n)) \).

Formally we, however, we \( \omega(g(n)) \) as the set

\[
\omega(g(n)) = \{ \ f(n) : \text{for any positive constant} \ c > 0 \ \text{there exists} \ n_0 > 0 \ \text{such that} \\
0 \leq cg(n) < f(n) \ \text{for all} \ n \geq n_0 \}
\]
Comparing functions

Transitivity:
- \( f(n) = \Theta(g(n)), g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n)) \)
- \( f(n) = O(g(n)), g(n) = O(h(n)) \Rightarrow f(n) = O(h(n)) \)
- \( f(n) = \Omega(g(n)), g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n)) \)
- \( f(n) = o(g(n)), g(n) = o(h(n)) \Rightarrow f(n) = o(h(n)) \)
- \( f(n) = \omega(g(n)), g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n)) \)

Reflexivity:
- \( f(n) = \Theta(f(n)) \)
- \( f(n) = O(f(n)) \)
- \( f(n) = \Omega(f(n)) \)

Symmetry:
- \( f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n)) \)

Transpose symmetry:
- \( f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \)
- \( f(n) = o(g(n)) \iff g(n) = \omega(f(n)) \)
We can draw an analogy between the asymptotic comparison of functions and the comparison of real numbers

- \( f(n) = \mathcal{O}(g(n)) \) is like \( a \leq b \)
- \( f(n) = \Omega(g(n)) \) is like \( a \geq b \)
- \( f(n) = \Theta(g(n)) \) is like \( a = b \)
- \( f(n) = o(g(n)) \) is like \( a < b \)
- \( f(n) = \omega(g(n)) \) is like \( a > b \)

### 3.2 Standard notations and common functions

- For all real constants \( a \) and \( b \) such that \( a > 1 \),
  \[
  \lim_{n \to \infty} \frac{n^b}{a^n} = 0
  \]
  
  from which we can conclude that \( n^b = o(a^n) \)
- Thus, any exponential function with a base strictly greater than 1 grows faster than any polynomial function
Exponentials

- Let $e$ to denote $2.71828...$, the base of the natural logarithm, we have for all real $x,$
  \[ e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} \]
- For all real $x$, we have the inequality $e^x \geq 1 + x,$ where equality holds only when $x = 0$
- When $|x| \leq 1$, we have the approximation $1 + x \leq e^x \leq 1 + x + x^2$

- When $x \to 0$, the approximation of $e^x$ by $1 + x$ is quite good:
  \[ e^x = 1 + x + O(x^2) \]
- In this equation, the asymptotic notation is used to describe the limiting behavior as $x \to 0$ rather than as $x \to \infty$
- We have for all $x$
  \[ \lim_{n \to \infty} \left( 1 + \frac{x}{n} \right)^n = e^x \]
Factorials

- Notation \( n! \) is defined for integers \( n \geq 0 \) as
  \[
  n! = \begin{cases} 
  1 & \text{if } n = 0 \\
  n(n-1)! & \text{if } n > 0 
  \end{cases}
  \]
- Thus, \( n! = 1 \cdot 2 \cdot 3 \ldots n \)
- A weak upper bound is \( n! \leq n^n \), since each of the terms in the factorial product is at most \( n \)
- *Stirling’s approximation* gives a tighter upper bound, and a lower bound as well,
  \[
  n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)
  \]

Fibonacci numbers

- We define the Fibonacci numbers by the following recurrence:
  - \( F_0 = 0 \)
  - \( F_1 = 1 \)
  - \( F_i = F_{i-1} + F_{i-2} \) for \( i \geq 2 \)
- Thus, each Fibonacci number is the sum of the two previous ones, yielding the sequence
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,...
Fibonacci numbers are related to the golden ratio $\phi$ and to its conjugate $\hat{\phi}$, which are the two roots of the equation $x^2 = x + 1$

$$\phi = \frac{1 + \sqrt{5}}{2} = 1.61803...$$

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} = -0.61803...$$

By induction we have

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$$

Since $|\hat{\phi}| < 1$, we have

$$\frac{|\hat{\phi}^i|}{\sqrt{5}} < \frac{1}{\sqrt{5}} < \frac{1}{2}$$

which implies

$$F_i = \left| \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right|$$

Hence, the $i$th Fibonacci number $F_i$ is equal to $\phi^i / \sqrt{5}$ rounded to the nearest integer.

Thus, Fibonacci numbers grow exponentially.
4 Divide-and-Conquer

• We examine three methods for obtaining asymptotic $\Theta$ or $O$ bounds on the solution:
  – In substitution method, we guess a bound and then use induction to prove it correct
  – recursion-tree method converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.

• The master method provides bounds for recurrences of the form
  \[ T(n) = aT(n/b) + f(n), \]
  where $a \geq 1$, $b > 1$, and $f(n)$ is a given function

• This characterizes an algorithm that creates $a$ subproblems, each of which is $1/b$ the size of the original problem, and in which the divide and combine steps together take $f(n)$ time
4.2 Strassen’s algorithm for matrix multiplication

Let $A = (a_{ij})$ and $B = (b_{ij})$ be square $n \times n$ matrices; in the product $C = A \cdot B$, we define the entry $c_{ij}$, for $i, j = 1, 2, \ldots, n$, by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

We must compute $n^2$ matrix entries, and each is the sum of $n$ values.

---

Square-Matrix-Multiply(A, B)

1. $n = A.\text{rows}$
2. let $C$ be a new $n \times n$ matrix
3. for $i = 1$ to $n$
4. for $j = 1$ to $n$
5. $c_{ij} = 0$
6. for $k = 1$ to $n$
7. $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
8. return $C$
Each of the nested for loops runs exactly $n$ iterations, execution of line 7 takes constant time, the procedure takes $\Theta(n^3)$ time.

- We have a way to multiply matrices in $o(n^3)$ time
- Strassen’s remarkable recursive algorithm for multiplying $n \times n$ matrices runs in $\Theta(n^{\lg 7})$ time
- $\lg 7$ lies between 2.80 and 2.81, Strassen’s algorithm runs in $O(n^{2.81})$

---

A simple divide-and-conquer algorithm

- We assume that $n$ is an exact power of 2 in each of the $n \times n$ matrices
- Suppose that we partition each of $A$, $B$, and $C$ into four $n/2 \times n/2$ matrices

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}
\]

so that we rewrite $C = A \cdot B$ as

\[
\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}
\]
This corresponds to the four equations
\[ C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \]
\[ C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \]
\[ C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \]
\[ C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \]

Each of these four equations specifies two multiplications of \( n/2 \times n/2 \) matrices and the addition of their \( n/2 \times n/2 \) products

Based on them we can create a straightforward, recursive, divide-and-conquer algorithm

---

**SQUARE-MATRIX-MULTIPLY-RECURSIVE** \((A, B)\)

1. \( n = A.\text{rows} \)
2. let \( C \) be a new \( n \times n \) matrix
3. if \( n == 1 \)
4. \( c_{11} = a_{11} \cdot b_{11} \)
5. else partition \( A, B, \) and \( C \)
6. \( C_{11} = \text{RECURSE}(A_{11}, B_{11}) + \text{RECURSE}(A_{12}, B_{21}) \)
7. \( C_{12} = \text{RECURSE}(A_{11}, B_{12}) + \text{RECURSE}(A_{12}, B_{22}) \)
8. \( C_{21} = \text{RECURSE}(A_{21}, B_{11}) + \text{RECURSE}(A_{22}, B_{21}) \)
9. \( C_{22} = \text{RECURSE}(A_{21}, B_{12}) + \text{RECURSE}(A_{22}, B_{22}) \)
10. return \( C \)
When $n = 1$, we perform just the one constant-time scalar multiplication in line 4.

When $n > 1$, partitioning the matrices in line 5 takes $\Theta(1)$ time, using index calculations.

Lines 6–9 recurse eight times, the time taken by all recursive calls is $8T(n/2)$.

We also must account for the four matrix additions in lines 6–9.

Each matrix contains $n^2/4$ entries, and so each of the four additions takes $\Theta(n^2)$ time.

The number of matrix additions is a constant, the total time spent adding matrices in lines 6–9 is $\Theta(n^2)$.

Total time for the recursive case is the sum of

– the partitioning time,
– the time for all the recursive calls, and
– the time to add the matrices resulting from the recursive calls:

$$T(n) = \Theta(1) + 8T(n/2) + \Theta(n^2) = 8T(n/2) + \Theta(n^2)$$
The recurrence for the running time of SQUARE-MATRIX-MULTIPLY-RECURSIVE is

\[ T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) & \text{if } n > 1 \end{cases} \]

We shall see that this recurrence has the solution \( T(n) = \Theta(n^3) \).

The approach is no faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure.

In accounting for the eight recursive calls we cannot just subsume the constant factor of 8.

We must say that together they take \( 8T(n/2) \) time, rather than just \( T(n/2) \).

If we were to ignore the factor of 8, the recursion tree would just be linear, not “bushy,” and each level would contribute only one term to the sum.

Although asymptotic notation subsumes constant multiplicative factors, recursive notation such as \( T(n/2) \) does not...
Strassen’s method

- Makes the recursion tree slightly less bushy
  - Instead of performing eight recursive multiplications, it performs only seven
  - The cost is several new additions of $n/2 \times n/2$ matrices, but only a constant number of them
- Constant number of additions is subsumed by $\Theta$-notation when we set up the recurrence to characterize the running time

1. Divide matrices $A$, $B$, and $C$ into $n/2 \times n/2$ submatrices in $\Theta(1)$ time by index calculation
2. Create matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time
3. Using the matrices created, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix $P_i$ is $n/2 \times n/2$
4. Compute the submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ by adding and subtracting various combinations of the $P_i$ matrices. We can compute all four submatrices in $\Theta(n^2)$ time
We have enough information to set up a recurrence for the running time of the method.

Let’s assume that once the matrix size $n$ gets down to 1, we perform a simple multiplication.

When $n > 1$, steps 1, 2, and 4 take a total of $\Theta(n^2)$ time, and step 3 requires us to perform seven multiplications of $n/2 \times n/2$ matrices.

Hence, we obtain the following recurrence:

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
7T(n/2) & \text{if } n > 1
\end{cases}$$

By the *master method* this recurrence has the solution $T(n) = \Theta(n^{\log_7 7})$.

---

**Matrices of Step 2**

- $S_1 = B_{12} - B_{22}$
- $S_2 = A_{11} + A_{12}$
- $S_3 = A_{21} + A_{22}$
- $S_4 = B_{21} - B_{11}$
- $S_5 = A_{11} + A_{22}$
- $S_6 = B_{11} + B_{22}$
- $S_7 = A_{12} - A_{22}$
- $S_8 = B_{21} + B_{22}$
- $S_9 = A_{11} - A_{21}$
- $S_{10} = B_{11} + B_{12}$

we must add or subtract $n/2 \times n/2$ matrices 10 times, so this step does take $\Theta(n^2)$ time.
In **step 3**, we recursively multiply $n/2 \times n/2$ matrices seven times to compute the following $n/2 \times n/2$ matrices:

- $P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22}$
- $P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22}$
- $P_3 = S_3 \cdot B_{11}$
- $P_4 = A_{22} \cdot S_4$
- $P_5 = S_5 \cdot S_6$
- $P_6 = S_7 \cdot S_8$
- $P_7 = S_9 \cdot S_{10}$

**Step 4** adds and subtracts the $P_i$ matrices created in step 3 to construct the four $n/2 \times n/2$ submatrices of the product $C$

- $C_{11} = P_5 + P_4 - P_2 + P_6$
- $C_{12} = P_1 + P_2$
- $C_{21} = P_3 + P_4$
- $C_{22} = P_5 + P_1 - P_3 - P_7$

E.g.,

\[
\begin{align*}
A_{11} \cdot B_{12} - A_{11} \cdot B_{22} &= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
A_{11} \cdot B_{12} + A_{12} \cdot B_{22} &= C_{12}
\end{align*}
\]