Example: Bubble Sort


Idea: Repeatedly "percolate" the next largest element up to its correct position.

Bubble sort algorithm:

```
for $i := n–1$ downto 1 do
   for $j := 1$ to $i$ do
```

Note: We will see faster ways to sort in later chapters.
Complexity of Bubble Sort:

For all inputs, the if statement is executed once for every iteration of the inner loop and consumes $O(1)$ time (it always does the work of testing if $A[j]>A[j+1]$ even if an exchange is not necessary).

The total time to perform the if statement along with any time expended for the exchange, along with the overhead of the control for the for loops, etc. is $O(1)$.

Thus, the time for bubble sort is proportional to the number of times the if statement is executed.

That is, the big O complexity of bubble sort can be determined by counting how many times the if statement is executed.
(complexity of bubble sort continued)

Since the inner loop goes from 1 to \(i\), and \(i\) is going from \(n-1\) down to 1 with each iteration of the outer loop, the number of times the \(if\) statement is executed is:

\[
= (n-1) + (n-2) + \cdots + 1
= 1 + 2 + 3 + \cdots + (n-1)
= \left(\text{number of terms}\right) \left(\text{average value of a term}\right)
= \left(\text{number of terms}\right) \left(\frac{\text{smallest term} + \text{largest term}}{2}\right)
= \frac{1}{2} n^2 - \frac{1}{2} n
\]
(complexity of bubble sort continued)

Given that the number of times the if statement is executed is

\[ \frac{1}{2} n^2 - \frac{1}{2} n \]

we can now conclude:

**Bubble sort is \( O(n^2) \).**

Let \( a = 1 \) and \( b \geq 1/2 \) in the big O definition.

**Bubble sort is \( \Omega(n^2) \).**

\[
\frac{1}{2} n^2 - \frac{1}{2} n = \frac{1}{4} n^2 + \left( \frac{1}{4} n^2 - \frac{1}{2} n \right)
\]

So we can choose \( c = 1/4 \) in the \( \Omega \) definition (the expression in parentheses is greater than 0 for all \( n \geq 2 \)).

**Bubble sort is \( \Theta(n^2) \).**

Since it is both \( O(n^2) \) and \( \Omega(n^2) \).
Space Used By Bubble Sort

$O(1)$ space

(in addition to the $O(n)$ space used by $A$)
Example: Run-Length Codes

**Idea:** Given a string that has many *runs* (blocks of repeated characters), it may take less space to represent it by the lengths of each run. We limit our attention to binary strings. For example, the binary string \(011110011100001010111\) could be encoded as \(1423511112\).

**Implementation issues:**

- Assume that the string starts with 0 (if not, begin with zero 0's)
- Assume the encoder and decoder have agreed in advance on the number of bits \(k\) to be used to represent each integer in the encoding.
- When \(2^k - 1\) is received by the decoder, it expects that the next run will be more of the same bit.
- When the count is a multiple of \(2^k - 1\), it is followed by a zero count for the same bit (unless it is the last count).
Binary run-length encoding algorithm:

\[ \text{MaxCount} := 2^k - 1 \]
\[ \text{count} := 0 \]
\[ \text{PrevBit} := "0" \]

\textbf{while} input remains \textbf{do begin}
\n\hspace{1em} \text{CurBit} := the next input bit
\hspace{1em} \textbf{if} \text{PrevBit}=\text{CurBit} \text{ and } \text{count}<\text{MaxCount} \text{ then } \text{count} := \text{count}+1
\hspace{1em} \textbf{else begin}
\hspace{2em} \text{Output } \text{count} \text{ using } k \text{ bits.}
\hspace{2em} \textbf{if} \text{count}=\text{MaxCount} \text{ and } \text{PrevBit} \neq \text{CurBit} \text{ then } \text{Output 0 using } k \text{ bits.}
\hspace{2em} \text{count} := 1
\hspace{2em} \textbf{end}
\hspace{1em} \text{PrevBit} := \text{CurBit}
\hspace{1em} \textbf{end}
\nOutput \text{count} \text{ using } k \text{ bits.}
Binary run-length decoding algorithm:

\[ \text{MaxCount} := 2^k - 1 \]
\[ \text{parity} := 0 \]
while input remains do begin
  Read \( k \) bits from the input stream to get the integer \( x \).
  if \( \text{parity} = 0 \) then output \( x \) "0" bits else output \( x \) "1" bits
  if \( x < \text{MaxCount} \) then \( \text{parity} := 1 - \text{parity} \)
end

**Complexity:** Space is \( O(1) \) for both encoding and decoding. Assuming that input or output of \( k \) bits is \( O(k) \) time, for a string of \( n \) bits, encoding and decoding is \( O(kn) \) in the worst case (e.g., an alternating sequence of 0's and 1's), which is \( O(n) \) assuming that \( k \) is constant with respect to \( n \).
Example: Horner’s Method for Polynomial Evaluation

Problem: Given the $n \geq 1$ coefficients $A[0] \ldots A[n–1]$, evaluate:

Straightforward algorithm:
\begin{verbatim}
input x
value := 0
for i := (n–1) downto 0 do begin
    temp := 1
    for j := 1 to i do temp := temp*x
    value := value + A[i] * temp
end
output value
\end{verbatim}

Time: $\Theta(n^2)$.
Space: $O(1)$ space in addition to the $O(n)$ space used by $A$. 
(Horner's method continued)

**Idea:**

compute $A[n-1]$
and so on ...

**Horner's algorithm:**

```plaintext
input x
value := 0
for i := (n-1) downto 0 do value := value*x + A[i]
output value
```

**Time:** $O(n)$.

**Space:** $O(1)$ space in addition to the $O(n)$ space used by $A$. 
Example: Matrix Multiplication

**Definition:** The product of two "square" \( n \) by \( n \) matrices is:

\[
\begin{pmatrix}
A[1,1] & \cdots & A[1,n] \\
\vdots & \ddots & \vdots \\
A[n,1] & \cdots & A[n,n]
\end{pmatrix}
\begin{pmatrix}
B[1,1] & \cdots & B[1,n] \\
\vdots & \ddots & \vdots \\
B[n,1] & \cdots & B[n,n]
\end{pmatrix} =
\begin{pmatrix}
C[1,1] & \cdots & C[1,n] \\
\vdots & \ddots & \vdots \\
C[n,1] & \cdots & C[n,n]
\end{pmatrix}
\]

\( C[i,j] = \) (row \( i \) of \( A \))(column \( j \) of \( B \))
\[
\]

**Example:**

\[
\begin{pmatrix}
1 & 2 \\
3 & 4
\end{pmatrix}
\begin{pmatrix}
5 & 6 \\
7 & 8
\end{pmatrix} =
\begin{pmatrix}
1*5+2*7 & 1*6+2*8 \\
3*5+4*7 & 3*6+4*8
\end{pmatrix} =
\begin{pmatrix}
19 & 22 \\
43 & 50
\end{pmatrix}
\]
(matrix multiplication continued)

For non-square matrices, we multiply a $m$ by $w$ matrix $A$ with a $w$ by $n$ matrix $B$ to form a $n$ by $m$ matrix $C$.

That is, computing an entry of $C$ by multiplying a row of $A$ by a column of $B$ is well defined as long as the length of a row in $A$ is the same as the height of a column in $B$.

The multiplication of a row by a column is called a *dot product* or an *inner product*; in general, for $1 \leq i \leq n$ and $1 \leq j \leq m$:


For example:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \times 1 + 2 \times 2 \\ 3 \times 1 + 4 \times 2 \\ 5 \times 1 + 6 \times 2 \\ 7 \times 1 + 8 \times 2 \end{pmatrix} = \begin{pmatrix} 5 \\ 11 \\ 17 \\ 21 \end{pmatrix}$$
Standard matrix multiplication algorithm:

\[
\begin{align*}
\text{for } 1 \leq i \leq m, 1 \leq j \leq n \text{ do begin} \\
& \quad C[i,j] := 0 \\
& \quad \text{for } k=1 \text{ to } w \text{ do } C[i,j] = C[i,j] + A[i,k]B[k,j] \\
\text{end}
\end{align*}
\]

**Time:** The inner \textit{for} loop is \( O(w) \), and it is executed by the outer \textit{for} loop for all \( mn \) possible values of \( i,j \), for a total of \( O(mwn) \) time. For the case of square matrices where \( m=w=n \), the time is \( O(n^3) \).

**Space:** \( O(1) \) in addition to the space used by \( A, B, \) and \( C \).

*** There are asymptotically faster methods.