Lecture 12

Complexity of Algorithms 3.2-3.3
Recall Lec 10 on algorithms
How long does each algorithm take to run?

• How do the number of operations scale with input length $n$?

• **Worst-case complexity**
  • e.g. Search list of length $n$ and the item to find is in position $a_n$

• **Best-case complexity**
  • e.g. Search list of length $n$ and the item to find is in position $a_1$

• **Average-case complexity / Typical-case complexity**
Average case complexity

- e.g., for an unordered list of length $n$

- With probability $1/n$ the element is in position $a_j$ for $j$ ranging from 1 to $n$.

- Let $M(j)$ denote the number of operations needed if the desired item is in position $a_j$

- Average case:

$$\frac{1}{n} \sum_{j=1}^{n} M(j)$$
Specifying an algorithm: Our Pseudocode Language:

- **procedure**
  
  `name(argument: type)`

- **variable** := expression

- **begin statements end**

- `{comment}`

- **if condition then statement**
  [else statement]

- **for variable := initial value to final value statement**

- **while condition statement**

- **return expression**
procedure \textit{procname}(\textit{arg}: \textit{type})

• Often just say “\textit{procname}(\textit{arg})” without the type.

• Declares that the following text defines a procedure named \textit{procname} that takes inputs (\textit{arguments}) named \textit{arg} which are data objects of the type \textit{type}.
  
• Example:
  
  \begin{verbatim}
  procedure \textit{maximum}(L: list of integers)
  [statements defining \textit{maximum}...]
  \end{verbatim}
variable := expression

• An assignment statement evaluates the expression expression, then reassigns the variable variable to the value that results.
  • Example assignment statement:
    \( v := 3x + 7 \) (If \( x \) is 2, changes \( v \) to 13.)

• Also use
  • \( v <- 3x + 7 \)
Informal statement

- Sometimes we may write a statement as an informal English imperative, if the meaning is still clear and precise: *e.g.*, “swap x and y”
- Keep in mind that real programming languages never allow this.
- When we ask for an algorithm to do so-and-so, writing “Do so-and-so” isn’t enough!
  - Break down algorithm into detailed steps.
Swap two values

- V1 := 10
- V2 := 20

badswap (V1, V2)
  V1 = V2
  V2 = V1  (What ends up being the value of V2?)

swap (V1, V2)
  temp := V1
  V1 = V2
  V2 = temp  (This one works)
• Groups a sequence of statements together:
\begin{verbatim}
begin
  statement 1
  statement 2
...
  statement n
end
\end{verbatim}

• Allows the sequence to be used just like a single statement.

• Might be used:
  • After a \texttt{procedure} declaration.
  • In an \texttt{if} statement after \texttt{then} or \texttt{else}.
  • In the body of a \texttt{for} or \texttt{while} loop.

Curly braces \{\} are used instead in many languages.
{comment}

- Not executed (does nothing).
- Natural-language text explaining some aspect of the procedure to human readers.
- Also called a *remark* in some real programming languages, *e.g.* BASIC.
- Example, might appear in a *max* program:
  - {Note that \( v \) is the largest integer seen so far.}
**if** condition **then** statement

- Evaluate the propositional expression *condition*.
  - If the resulting truth value is *True*, then execute the statement *statement*;
  - otherwise, just skip on ahead to the next statement after the *if* statement.

- Variant:  **if** *cond* **then** *stmt1* **else** *stmt2*
  - Like before, but iff truth value is *False*, executes *stmt2*. 
while \textit{condition} statement

• \underline{Evaluate} the propositional (Boolean) expression \textit{condition}.

• If the resulting value is \textbf{True}, then execute \textit{statement}.

• Continue repeating the above two actions over and over until finally the \textit{condition} evaluates to \textbf{False}; then proceed to the next statement.
while \textit{condition statement}

- Also equivalent to infinite nested \textbf{ifs}, like so:

```plaintext
if \textit{condition}
begin
  \textit{statement}
  if \textit{condition}
  begin
    \textit{statement}
    ...(continue infinite nested if’s)
  end
end
```
for \( var := initial \) to \( final \) \( stmt \)

- \textit{Initial} is an integer expression.
- \textit{Final} is another integer expression.

- **Semantics:** Repeatedly execute \( stmt \), first with variable \( var := initial \), then with \( var := initial+1 \), then with \( var := initial+2 \), etc., then finally with \( var := final \).

- **Question:** What happens if \( stmt \) changes the value of \( var \), or the value that \( initial \) or \( final \) evaluates to?
**for** \texttt{var} := \texttt{initial} to \texttt{final} **stmt**

- **For** can be exactly defined in terms of **while**, like so:

```
begin
  \texttt{var} := \texttt{initial}
  \texttt{while} \texttt{var} \leq \texttt{final}
  begin
    \texttt{stmt}
    \texttt{var} := \texttt{var} + 1
  end
end
```
Now writing Alg 1: find max

**Algorithm 1** An algorithm for finding the maximum element in a finite sequence \(a = \{a_1, \ldots, a_n\}\) of integers.

**FIND-MAX (a)**

\[
\begin{align*}
  &\text{max} \leftarrow a_1 \\
  &\text{for } i \leftarrow 2, 3, \ldots, n \text{ do} \\
  &\quad \text{if } max < a_i \text{ then} \\
  &\quad\quad max \leftarrow a_i \\
  &\quad \text{end if} \\
  &\text{end for} \\
  &\text{return } max \{max \text{ is the largest element.}\}
\end{align*}
\]
Alg. #3: Binary Search

- Basic idea: On each step, look at the *middle* element of the remaining list to eliminate half of it, and quickly zero in on the desired element.
Alg. #3: Binary Search

- **procedure** `binary_search`  
  `(x:integer, a_1, a_2, ..., a_n: distinct integers)`
  
  `i := 1 {left endpoint of search interval}`  
  `j := n {right endpoint of search interval}`  
  
  **while** `i<j` **begin**  
  `{while interval has >1 item}`  
  `m := ⌊(i+j)/2⌋ {midpoint}`  
  
  **if** `x>a_m` **then** `i := m+1` **else** `j := m`  
  
  **end**  
  
  **if** `x = a_i` **then** `location := i` **else** `location := 0`  
  
  **return** `location`
Sorting Algorithms

- Sorting is a common operation in many applications.
  - *E.g.* spreadsheets and databases
- It is also widely used as a subroutine in other data-processing algorithms.
- Two sorting algorithms shown in textbook:
  - Bubble sort
  - Insertion sort

However, these are *not* very efficient, and you should not use them on large data sets!
Bubble Sort (Alg #4)

- Smallest elements “float” up to the top of the list, like bubbles in a container of liquid.
Review §3.1: Algorithms

• Characteristics of algorithms.
• Pseudocode.
• Examples: Max algorithm, linear search & binary search algorithms, sorting.
• Intuitively we see that binary search is much faster than linear search, but how do we analyze the efficiency of algorithms formally?
• Use methods of *algorithmic complexity*, which utilize the order-of-growth concepts from §1.8.
See handout 7

- **Linear search** (unsorted list)
- **Binary search** (needs sorted list)
- **Bubble sort**
- **Insertion sort**
  - (note “for k in 0 to -1” does not execute)
- **Greedy algorithms**
  - Make the best choice given what you know.
  - Internet routing algorithms
  - Making change with minimal number of coins
How long does each algorithm take to run?

- How do the number of operations scale with input length \( n \)?

- **Worst-case complexity**
  - e.g. Search list of length \( n \) and the item to find is in position \( a_n \)

- **Best-case complexity**
  - e.g. Search list of length \( n \) and the item to find is in position \( a_1 \)

- **Average-case complexity / Typical-case complexity**
How to we generalize this?

• In general we do not care about the exact number of operations required. We want to estimate or bound this by some function of $n$. (Where $n$ is the length of the input)
Growth of functions (§3.2)

• For functions over numbers, we often need to know a rough measure of *how fast a function grows*.

• If \( f(x) \) is *faster growing* than \( g(x) \), then \( f(x) \) always eventually becomes larger than \( g(x) \) *in the limit* (for *large enough* values of \( x \)).

• Useful in engineering for showing that one design *scales* better or worse than another.
Visualizing Orders of Growth

- On a graph, as you go to the right, the faster-growing function always eventually becomes the larger one...

\[ f_B(n) = n^2 + 1 \]
\[ f_A(n) = 30n + 8 \]

\[ f_B(n) > f_A(n), \text{ for } n > k \]
Another example, graphically

- Note $30n+8$ isn’t less than $n$ anywhere ($n>0$).
- It isn’t even less than $31n$ everywhere.
- But it is less than $31n$ everywhere to the right of $n=8$. 
Common reference functions
Handout 8
Relations Between the Relations

- Subset relations between order-of-growth sets.
# Complexity of Algorithms

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(1)$</td>
<td>Constant complexity</td>
</tr>
<tr>
<td>$\Theta(\log n)$</td>
<td>Logarithmic complexity</td>
</tr>
<tr>
<td>$\Theta(n)$</td>
<td>Linear complexity</td>
</tr>
<tr>
<td>$\Theta(n \log n)$</td>
<td>Lineararithmic complexity</td>
</tr>
<tr>
<td>$\Theta(n^b)$</td>
<td>Polynomial complexity</td>
</tr>
<tr>
<td>$\Theta(b^n)$, where $b &gt; 1$</td>
<td>Exponential complexity</td>
</tr>
<tr>
<td>$\Theta(n!)$</td>
<td>Factorial complexity</td>
</tr>
</tbody>
</table>
Computer time needed

**TABLE 2** The Computer Time Used by Algorithms.

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>( n )</th>
<th>( \log n )</th>
<th>( n )</th>
<th>( n \log n )</th>
<th>( n^2 )</th>
<th>( 2^n )</th>
<th>( n! )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>( 3 \times 10^{-11} ) s</td>
<td>( 10^{-10} ) s</td>
<td>( 3 \times 10^{-10} ) s</td>
<td>( 10^{-9} ) s</td>
<td>( 10^{-8} ) s</td>
<td>( 3 \times 10^{-7} ) s</td>
</tr>
<tr>
<td>( 10^2 )</td>
<td></td>
<td>( 7 \times 10^{-11} ) s</td>
<td>( 10^{-9} ) s</td>
<td>( 7 \times 10^{-9} ) s</td>
<td>( 10^{-7} ) s</td>
<td>( 4 \times 10^{11} ) yr</td>
<td>*</td>
</tr>
<tr>
<td>( 10^3 )</td>
<td></td>
<td>( 1.0 \times 10^{-10} ) s</td>
<td>( 10^{-8} ) s</td>
<td>( 1 \times 10^{-7} ) s</td>
<td>( 10^{-5} ) s</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>( 10^4 )</td>
<td></td>
<td>( 1.3 \times 10^{-10} ) s</td>
<td>( 10^{-7} ) s</td>
<td>( 1 \times 10^{-6} ) s</td>
<td>( 10^{-3} ) s</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>( 10^5 )</td>
<td></td>
<td>( 1.7 \times 10^{-10} ) s</td>
<td>( 10^{-6} ) s</td>
<td>( 2 \times 10^{-5} ) s</td>
<td>( 0.1 ) s</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>( 10^6 )</td>
<td></td>
<td>( 2 \times 10^{-10} ) s</td>
<td>( 10^{-5} ) s</td>
<td>( 2 \times 10^{-4} ) s</td>
<td>( 0.17 ) min</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
Computational complexity classes

- **Tractable** problems are polynomial complexity or less. Said to belong to *computational complexity class P*, for “polynomial”

- Exponential and factorial complexity are said to be “**intractable**”.

- **Special class NP** (non-deterministic polynomial)
NP problems

• No polynomial time algorithm for solving the problem
• But can check/validate a potential solution in polynomial time.
• E.g. factoring large integers
• Central belief: $P \neq NP$
  • Proving this is one of the Millennium Prize problems (win $1$ million!!)