About This Lecture

- In this lecture we will learn how to compare the relative space and time efficiencies of different software solutions.

Outline

- Implementation choice criteria
- Time and space complexity
- Asymptotic analysis
- Worst, best and average case complexity

Implementation Choices

- A programmer has many choices to make when implementing classes.
  - For example, in a banking system, should an Account object remember an Array of Transactions or a Vector of Transactions?
  - For example, should a sort method use a selection sort or a merge sort?

Choice Criteria

- What criteria should be used to make implementation choices?
  - Historically, these two criteria have been considered the most important:
    - The amount of space that an object uses and the amount of temporary storage used by its methods.
    - The amount of time that a method takes to run.

Other Choice Criteria

- However, we should also consider:
  - The simplicity of an object representation.
  - The simplicity of an algorithm that is implemented as a method.
  - These criteria are important because they directly affect:
    - The amount of time it takes to implement a class.
    - The number of defects (bugs) in classes.
    - The amount of time it takes to update a class when requirements change.
Comparing Methods

- What is better, a selection sort or a merge sort?
  - We can implement each sort as a method.
  - We can count the amount of storage space each method requires for sample data.
  - We can time the two methods on sample data and see which one is faster.
  - We could time how long it takes a programmer to write the code for each method.
  - We can count the defects in the two implementations.

Comparing Average Methods

- Unfortunately, our answers would depend on the sample data and the individual programmer chosen.
- To circumvent this problem, we could compute averages over many sample data sets and over many programmers.
- This would be a long and expensive process.
- We need a way to predict the answers by analysis rather than experimentation.

Time and Space Complexity

- The time and space requirements of a method can be determined by analysis.
- Since these values depend on the data size, we often describe them as a function of the data size (referred to as n).
- These functions are called respectively, the time complexity, time(n), and space complexity, space(n), of the algorithm.

Space Complexity

- It is easy to compute the space complexity for many algorithms:
  - A selection sort of one word elements has space complexity: space(n) = n + 1 words
    - It takes n words to store the elements.
    - It takes one extra word for swapping elements.
  - A merge sort of one word elements has space complexity: space(n) = 2n words
    - It takes n words to store the elements.
    - It takes n words for swapping elements during merges.

Time Complexity - Operations

- Actual execution time depends on
  - the number and kind of machine instructions that are generated.
  - the time it takes to run machine instructions.
- To do analysis in a machine-independent way: pick some basic operation and count how many times this operation is executed.
  - we will count access operations
Exact analysis: Example 1

The time (number of accesses) to execute this code is

\[ 1 \times \text{number of times statement (1) is executed} \]
\[ + 1 \times \text{number of times statement (2) is executed} \]
\[ = 1 + n \]

\( \text{time}(n) = n + 1 \) "linear time"

Linear time

Exact analysis: Example 2

The time (number of accesses) to execute this code is

\[ 1 \times \text{number of times statement (1) is executed} \]
\[ + 1 \times \text{number of times statement (2) is executed} \]
\[ + 1 \times \text{number of times statement (3) is executed} \]
\[ = 3 \]

\( \text{time}(n) = 5 \) "constant time"

Constant time

Exact analysis: Example 3

The time (number of accesses) to execute this code is

\[ 1 \times \text{number of times statement (1) is executed} \]
\[ + 2 \times \text{number of times statement (2) is executed} \]
\[ + 1 \times \text{number of times statement (3) is executed} \]
\[ = 2 \times n + n \]

\( \text{time}(n) = 1 + 2n^2 + n \) "quadratic time"

Example 3 (cont’d)
**Order** of Complexity

- When comparing algorithms what matters most is the “order” of complexity, not the exact details.
- “order” = quadratic, linear, constant, logarithmic, exponential,…
- For large \( n \), quadratic will always be bigger than (worse than) linear, linear bigger than logarithmic:
  \[ \text{exponential} > \text{polynomial} > \log > \text{constant} \]

**Size matters**

- If we are sorting 100 values, it probably doesn’t matter which sort we use:
  - time and space requirements are so tiny they’re irrelevant.
  - However, the simplicity may be a factor.
- If we are sorting 20,000 values, it may matter:
  - 1 sec (MS) versus 16 sec (SS) on current hardware.
  - 20,001 words (SS) versus 40,000 words (MS).
  - The simplicity difference is still the same.
- If we are sorting 1,000,000 values, it does matter.

* MS = merge sort
  SS = selection sort

**Asymptotic Analysis**

- The aim of asymptotic analysis is to determine the complexity order of an algorithm.
- This can be done in two ways:
  - Do an exact analysis and then throw away the constants and the low order terms
    - e.g. \( \text{time}(n) = 1 + 2^n + n = \text{order } 2^n \)
  - Directly analyze the order, never bother to work out the constants and low order terms in detail

**Big-O definition**

- If an algorithm has time complexity order \( n^2 \) we write \( \text{time}(n) = O(n^2) \)
- The technical definition of big-O is:
  \[ \text{time}(n) = O(g(n)) \text{ if and only if } \lim_{n \to \infty} \frac{\text{time}(n)}{g(n)} = c, \text{ where } c \text{ is a constant} \]
- In English, this means time(n) grows no faster than the function \( g(n) \)
- Two important rules:
  - make \( g(n) \) as small as possible
  - \( g(n) \) never contains unnecessary constants or terms

**Direct Asymptotic Analysis**

- Individual statements (other than method calls) can only do a fixed number of memory accesses (at most the number of variables in the statement).
- Identify which statements do one or more accesses, don’t need to know exactly how many accesses are done.
- Identify which of these statements is executed the greatest number of times (“order”, not exact number). Do this by looking at the loops which contain the statements.
Asymptotic analysis: Example 1

5 is the statement that
- makes 1 or more accesses
- will be executed more times than any other statement that makes 1 or more accesses

\[ n = \text{array.length}; \]
\[ \text{for} (i = 5; i < n/3; i++) \{
  c = 0;
  \text{for} (j = (n - 11); j >= 7; j--)
    \text{if} (\text{array}[i] < \text{array}[j]) c++;
  \text{count}[i] = c;
\}\]

Example 1 (cont’d)

is inside the j-loop, which is inside the i-loop. The j-loop executes O(n) times for each iteration of the i-loop. The i-loop executes O(n) times.

is therefore executed \(O(n) \times O(n) = O(n^2)\) times.

Asymptotic analysis: Example 2

how many accesses?

5 is the statement that
- makes 1 or more accesses
- will be executed more times than any other statement that makes 1 or more accesses
- The j-loop depends on the i-loop, so they must be analyzed together

\[ n = \text{array.length}; \]
\[ \text{for} (i = 0; i < n; i++) \{
  \text{count}[i] = 0;
  \text{for} (j = 0; j < i; j++)
    \text{if} (\text{array}[i] < \text{array}[j]) \text{count}[i]++;
\}\]

Example 2 (cont’d)

\( i=0, \) j-loop iterates 0 times
\( i=1, \) j-loop iterates 1 time
\( i=2, \) j-loop iterates 2 times
... \( i=(n-1), \) j-loop iterates \((n-1)\) times

Total number of times \#5 is executed

\[ = 0 + 1 + 2 + \ldots + (n-1) \]
\[ = (n-1)n/2 \]
\[ = O(n^2) \]

Log Time - \(\text{O}(\log(n))\) Algorithms - 1

The time for a binary search on an array of size \(n\) that does not find the key.

\[ \text{found} = \text{false}; \]
\[ \text{low} = 0; \]
\[ \text{high} = n - 1; \]
\[ \text{while} (((! \text{found}) \&\& (\text{low} <= \text{high})) \{
  \text{guess} = (\text{high} + \text{low}) / 2;
  \text{if} (\text{key} = = \text{data}[\text{guess}]) \text{found} = \text{true};
  \text{else if} (\text{key} < \text{data}[\text{guess}]) \text{high} = \text{guess} - 1;
  \text{else} \text{low} = \text{guess} + 1;
\}\]

Log Time - \(\text{O}(\log(n))\) Algorithms - 2

Each time through the loop, the size of the range [low, high] is divided in half.

For example, if \(n=16\) and key is smaller than the smallest value in data, low will remain 0 throughout and high will take on these values:

\[ 15, 6, 2, 0 \]

If the range starts at size \(n\), it takes \(\log(n)\) times through the loop to reduce the range size to 1.

The total number of times the loop is executed is therefore \(\text{O}(\log(n))\).
Common complexity orders

<table>
<thead>
<tr>
<th>NAME</th>
<th>Big-O</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>linear</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>cubic</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^n)$</td>
</tr>
</tbody>
</table>

Where does $O(n^{\log n})$ belong in this table?

Variable Time Algorithms

The time complexity of many algorithms depends not only on the size of the data (n), but also on the specific data values:
- The time to find a key value in an indexed container depends on the location of the key in the container.
- The time to sort a container depends on the initial order of the elements.

Worst, Best and Average Complexity

- We define three common cases.
- The worst case complexity is the largest value for any problem of size n.
- The best case complexity is the smallest value for any problem of size n.
- The average case complexity is the weighted average of all problems of size n, where the weight of a problem is the probability that the problem occurs.

Worst Case Complexity Example

- Assume we are doing a sequential search on a container of size n.
- The worst case complexity is when the element is not in the container.
- The worst case complexity is $O(n)$.

Average Case Complexity Example

- To compute the average case complexity, we must make assumptions about the probability distribution of the data sets.
- Assume a sequential search on a container of size n.
- Assume a 50% chance that the key is in the container.
- Assume equal probabilities that the key is at any position.
- The probability that the key is at any particular location in the container is $1\cdot\frac{1}{n} = \frac{1}{n}$.
- The average case complexity is:
  \[ \sum_{i=0}^{n} i \cdot \frac{1}{n} = \frac{\sum_{i=0}^{n} i}{n} = \frac{n(n+1)/2}{n} = \frac{n+1}{2} = \frac{3n}{2} = O(n) \]
Trading Time and Space

- It is often possible to construct two different algorithms, where one is faster but takes more space.
- For example:
  - A selection sort has worst case time complexity $O(n^2)$ while using $n+1$ words of space.
  - A merge sort has worst case time complexity $O(n \log(n))$ while using $2^n$ words of space.
- Analysis lets us choose the best algorithm based on our time and space requirements.