Computer Science II

Professor Tom Ellman
Lecture 18
My **Gnarf** program is faster than yours!

No, my **Gnarf** program is faster than yours!
Efficiency of Algorithms

• How can we characterize the efficiency of an algorithm?
  – Sit in front of the computer with a stopwatch?
  – Use an operating system facility for recording elapsed CPU time?

• Unfortunately:
  – Comparisons of algorithms may depend on the type of computer.
  – Comparisons of algorithms may depend on the problem instances chosen for testing.
Asymptotic Complexity Analysis

• An approach to characterizing algorithm efficiency that is independent of any particular machine.
• Describes how the time and space used by an algorithm depends on the size of the problem.
• Makes distinctions among algorithms that hold for sufficiently large problems.
Problem Size Parameters

• The size of a problem can typically be described by an integer, or a small number of integers.
  – Sorting: The length of the array of items to be sorted.
  – Phone Number Lookup: The number of (name, number) pairs in the telephone book.
  – Spell Checking: The number of words (W) in the document and (D) in the dictionary and the maximum length (L) of a word.

• Asymptotic analysis investigates the manner in which the time \( T(n) \) or space \( S(s) \) needed to solve the problem depends on the problem size parameter (n) or parameters.
Big-O Notation

\[ T(n) = O(f(n)) \]

If there are constants \( c_1 \) and \( n_1 \) such that:

\[ T(n) \leq c_1 \cdot f(n) \quad (\text{For all } n \geq n_1) \]

We say that \( f(n) \) is an “asymptotic upper bound” on \( T(n) \).
$T(n) = O(f(n))$
Big-Ω Notation

\[ T(n) = \Omega(f(n)) \]

If there are constants \( c_0 \), and \( n_0 \) such that:

\[ c_0 \cdot f(n) \leq T(n) \quad \text{(For all } n \geq n_0) \]

We say that \( f(n) \) is an “asymptotic lower bound” on \( T(n) \).
$T(n) = \Omega(f(n))$

- $T(n)$
- $c_0 \cdot f(n)$
- $n_0$
- $n$
**Big-Θ Notation**

\[ T(n) = \Theta(f(n)) \]

if

\[ T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)) \]

We say that \( f(n) \) is an “asymptotic tight bound” on \( T(n) \).
\[ T(n) = \Theta(f(n)) \]
Constant Growth Functions

• Consider two algorithms A₁ and A₂, whose running times are T₁(n) and T₂(n).
• Let f(n) = 1. Try to find c₀, n₀, c₁, and n₁.
• Suppose: T₁(n) = 10. (Independent of n)
  1 \cdot f(n) \leq T(n) \leq 20 \cdot f(n), \text{ for all } n \geq 0.
  T₁(n) = \Theta(1).
• Suppose: T₂(n) = 100. (Independent of n)
  1 \cdot f(n) \leq T(n) \leq 200 \cdot f(n), \text{ for all } n \geq 0.
  T₂(n) = \Theta(1).
• Algorithms A₁ and A₂ are asymptotically equivalent.
• They both run in constant time.
Program to Compute Matthew Vassar’s Age

2. Let birthYear = 1792.
4. Print(age).

\[ T(n) = \Theta(1) \]
Linear Growth Functions

- Consider two algorithms $A_1$ and $A_2$, whose running times are $T_1(n)$ and $T_2(n)$.
- Let $f(n) = n$. Try to find $c_0$, $n_0$, $c_1$, and $n_1$.
- Suppose: $T_1(n) = 5n + 10$.
  \[ f(n) \leq T(n) \leq 6f(n), \text{ for all } n \geq 10. \]
  \[ T_1(n) = \Theta(n). \]
- Suppose: $T_2(n) = 20n + 100$.
  \[ f(n) \leq T(n) \leq 60f(n), \text{ for all } n \geq 3. \]
  \[ T_2(n) = \Theta(n). \]
- Algorithms $A_1$ and $A_2$ are asymptotically equivalent.
- Both algorithms run in *linear* time.
Word Count Program

\[ T(n) = \Theta(n) \]

Where \( n \) is the number of words in the file.

1. Open a file.
2. Let \( \text{count} = 0 \).
3. While (Not at End of file)
   a. Read another word.
   b. Let \( \text{count} = \text{count} + 1 \).
Examples of Growth Functions

- Consider two algorithms $A_1$ and $A_2$, whose running times are $T_1(n)$ and $T_2(n)$.
- Let $f(n) = n^2$. Try to find $c_0$, $c_1$ and $n_0$.
- Suppose: $T_1(n) = 2n^2 + 5n + 10$.
  $1 \cdot f(n) \leq T(n) \leq 3 \cdot f(n)$, for all $n \geq 7$.
  $T_1(n) = \Theta(n^2)$.
- Suppose: $T_2(n) = 20n^2 + 50n + 100$.
  $1 \cdot f(n) \leq T(n) \leq 21 \cdot f(n)$, for all $n \geq 52$.
  $T_2(n) = \Theta(n^2)$.
- Algorithms $A_1$ and $A_2$ are asymptotically equivalent.
- Both algorithms run in \textit{quadratic} time.
Program to Draw a Square Checkerboard

\[ T(n) = \Theta(n^2) \]

Where \( n \) is number of squares on a side.

1. Set up a GraphicsProgram.

2. For \( (r=0; \ r<=n; \ r++) \)
   
   For \( (c=0; \ c<=n; \ c++) \)
   
   if \( ((r+c)\%2==0) \) makeTile\((r,c,RED)\)
   
   else makeTile\((r,c,BLACK)\).
Program with Nested Loops

\[ T(n) = a_d n^d + \ldots + a_2 n^2 + a_1 n^1 + a_0 n^0 = \Theta(n^d) \]

1. Initialize level 0.

2. For \((i_0 = 1 \ldots n)\) do the following:

   a. Initialize level 1.

      b. For \((i_1 = 1 \ldots n)\) do the following:

         i. Initialize level 2.

         ii. For \((i_2 = 1 \ldots n)\) do the following:

            \[
            \ldots \text{Etc} \ldots 
            \]

            a. Initialize level d.

            b. For \((i_{d-1} = 1 \ldots n)\) Do a computation Step.
Spell Checking Algorithm

1. Read the dictionary from a file.

2. While more words remain in the document, do the following:
   a. Let word be the next word in the document.
   b. Let found be false.
   c. For (n = 0 … Length of Dictionary – 1) do the following:
      If word equals the nth dictionary entry, let found be true and break the loop.
   d. If found is false, print out word.
Approaching the Spell Checking Algorithm

• What are the appropriate problem size parameters?
  – The number of words in the document? (W)
  – The number of words in the dictionary? (D)
  – The average (maximum? minimum?) word length? (L)

• What operation(s) shall we count?
  – Number of string equality tests? (E)
  – Number of character comparisons? (C)
  – Number of misspelled words printed out? (P)

• What problem problem instances shall we consider? Best case? Worst case? Average case?
Analysis of the Spell Checking Algorithm

• Investigate dependence of $C$ on $W$, $D$ and $L$.
• Consider three cases:
  – Best case: Every word is spelled properly, and is found right at the beginning of the dictionary.
  – Worst case: Every word is misspelled, causing a search of the entire dictionary.
  – Average case: All words are spelled correctly and are distributed randomly over the dictionary.
Best Case Analysis

• Number of iterations of outer (while) loop is $W$.
• Number of iterations of inner (for) loop is 1.
• Number of string equality tests is: $W \cdot 1 = W$.
• Number of character comparisons for each equality test is $L$.

$$C(W, D, L) = \Theta(W \cdot L)$$
Worst Case Analysis

- Number of iterations of outer (while) loop is $W$.
- Number of iterations if inner (for) loop is $D$.
- Number of string equality tests is: $W \cdot D$.
- Number of character comparisons for each equality test is $L$.

$$C(W,D,L) = \Theta(W \cdot D \cdot L)$$
Average Case Analysis

- Number of iterations of outer (while) loop is $W$.
- Number of iterations if inner (for) loop is $D/2$.
- Number of string equality tests is: $W \cdot D / 2$.
- Number of character comparisons for each equality test is $L$.
- Total number of character comparisons is: $(W \cdot D \cdot L)/2$.

$$C(W,D,L) = \Theta(W \cdot D \cdot L)$$
Binary Search

• Algorithm that searches for an element in an array.
• Assumes that are elements are stored in order, e.g., according to the `compareTo` relation.
• Takes advantage of the fact that array elements can be accessed in constant (i.e., \( \Theta(1) \) ) time.
Binary Search Algorithm

begin + 0
begin + 1
mid
mid − 1
mid
mid + 1
mid + 1
end
end

Range: begin … mid
(Less than item.)

Range: mid + 1 … end
(Greater than item.)
Recursive Binary Search Algorithm

boolean search(array, begin, end, item)

// Look for item in array from begin up to end-1.
// Return true if found, otherwise return false.

If (begin=end) return false, otherwise, do the following:
1. Let mid = (begin + end)/2.
2. If item equals array[mid] return true.
3. If (item < array[mid])
   then return search(array, begin, mid, item),
   else return search(array, mid+1, end, item).
Iterative Binary Search Algorithm

boolean search(array, begin, end, item)

// Look for item in array from begin up to end-1.

// Return true if found, otherwise return false.

1. Let found = false.

2. While (found is false and begin < end) do the following:
   a. Let mid = (begin + end)/2.
   b. If item equals array[mid] let found = true, otherwise if (item < array[mid]) let end = mid, otherwise let begin = mid+1.

3. Return found.
Worst Case Analysis of Binary Search Algorithm

• Let D be the length of the array.
• Let N be the integer such that: $2^{N-1} < D \leq 2^N$.
  \[ N-1 < \log_2 D \leq N. \]
  \[ N = \lceil \log_2 D \rceil. \]
  Ceiling(r) = \lceil r \rceil is smallest integer as large as r.
• Assume search item is not found.
• After k comparisons, end-begin $\leq 2^{N-k}$.
• After N comparisons, end-begin $\leq 2^{N-N} = 1$.
• Algorithm terminates one comparison later.
• Worst case requires $N+1 = \lceil \log_2 D \rceil + 1$ comparisons.
• Worst case complexity: $C(D) = \Theta(\log D)$. 
Spell Checker with Binary Search

• Best Case: \( C(W, D, L) = \Theta(W \cdot L) \)

• Worst Case: \( C(W, D, L) = \Theta(W \cdot \log(D) \cdot L) \)

• Average Case: \( \Theta(W \cdot \log(D) \cdot L) \)
Fast Algorithms v. Fast Machines

• Consider using the following two machines:
  – $M_s$ takes 1 second to compare two characters.
  – $M_f$ takes $10^{-9}$ seconds to compare two characters.
• Run spell checker with binary search on $M_s$.
• Run spell checker with linear search on $M_f$.
• Assume everything is misspelled. (Worst case.)
• If the dictionary is large enough, the slow machine $M_s$ will beat the fast machine $M_f$. 