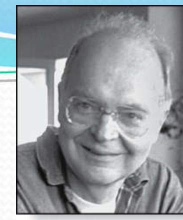
The Growth of Functions

Section 3.2

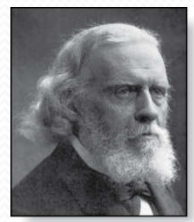
Section Summary



Donald E. Knuth
(Born 1938)

- Big-O Notation
- Big-O Estimates for Important Functions
- Big-Omega and Big-Theta Notation

Edmund Landau
(1877-1938)



Paul Gustav Heinrich Bachmann
(1837-1920)

The Growth of Functions

- In both computer science and in mathematics, there are many times when we care about how fast a function grows.
- In computer science, we want to understand how quickly an algorithm can solve a problem as the size of the input grows.
  - We can compare the efficiency of two different algorithms for solving the same problem.
  - We can also determine whether it is practical to use a particular algorithm as the input grows.
  - We'll study these questions in Section 3.3.
- Two of the areas of mathematics where questions about the growth of functions are studied are:
  - number theory (covered in Chapter 4)
  - combinatorics (covered in Chapters 6 and 8)

Big-*O* Notation

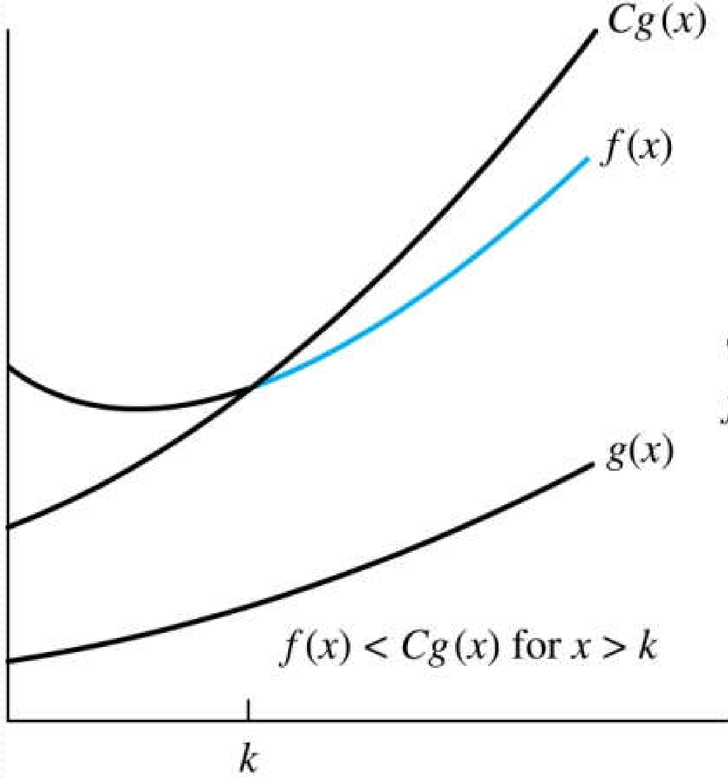
Definition: Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that *f*(*x*) is *O*(*g*(*x*)) if there are constants *C* and *k* such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. (illustration on next slide)

- This is read as "*f*(*x*) is big-*O* of *g*(*x*)" or "*g* asymptotically dominates *f*."
- The constants C and k are called *witnesses* to the relationship *f*(*x*) is *O*(*g*(*x*)). Only one pair of witnesses is needed.

Illustration of Big-*O* Notation



$f(x)$ is $O(g(x))$

The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

Some Important Points about Big-*O* Notation

- If one pair of witnesses is found, then there are infinitely many pairs. We can always make the k or the C larger and still maintain the inequality $|f(x)| \leq C|g(x)|$.
  - Any pair C' and k' where $C < C'$ and $k < k'$ is also a pair of witnesses since $|f(x)| \leq C|g(x) \leq C'|g(x)|$ whenever $x > k' > k$.

You may see "$f(x) = O(g(x))$" instead of "$f(x)$ is $O(g(x))$."

  - But this is an abuse of the equals sign since the meaning is that there is an inequality relating the values of f and g, for sufficiently large values of x.
  - It is ok to write $f(x) \in O(g(x))$, because $O(g(x))$ represents the set of functions that are $O(g(x))$.
- Usually, we will drop the absolute value sign since we will always deal with functions that take on positive values.

Using the Definition of Big-*O* Notation

Example: Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

Solution: Since when $x > 1$, $x < x^2$ and $1 < x^2$

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

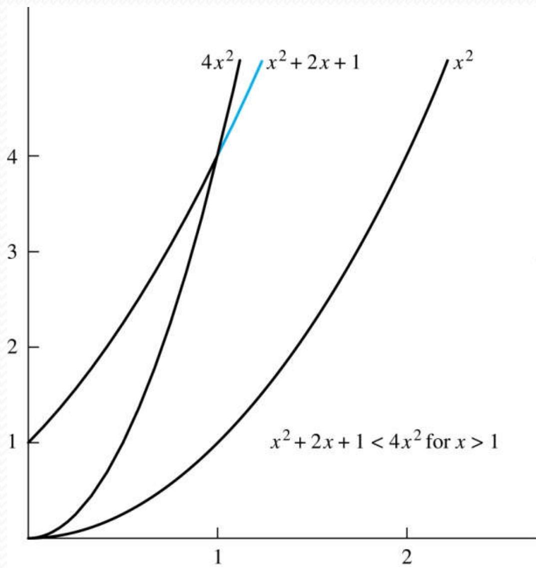
- Can take $C = 4$ and $k = 1$ as witnesses to show that

  $f(x)$ is $O(x^2)$ (see graph on next slide)

- Alternatively, when $x > 2$, we have $2x \leq x^2$ and $1 < x^2$. Hence, $0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2$ when $x > 2$.
  - Can take $C = 3$ and $k = 2$ as witnesses instead.

Illustration of Big-*O* Notation

$f(x) = x^2 + 2x + 1$ is $O(x^2)$



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in blue.

Big-*O* Notation

- Both $f(x) = x^2 + 2x + 1$ and $g(x) = x^2$ are such that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$. We say that the two functions are of the *same order*. (More on this later)

- If $f(x)$ is $O(g(x))$ and *h(x)* is larger than *g(x)* for all positive real numbers, then $f(x)$ is $O(h(x))$.

- Note that if $|f(x)| \leq C|g(x)|$ for *x* > *k* and if $|h(x)| > |g(x)|$ for all *x*, then $|f(x)| \leq C|h(x)|$ if *x* > *k*. Hence, $f(x)$ is $O(h(x))$.

- For many applications, the goal is to select the function *g*(*x*) in *O(g(x))* as small as possible (up to multiplication by a constant, of course).

Using the Definition of Big-*O* Notation

Example: Show that $7x^2$ is $O(x^3)$.

Solution: When $x > 7$, $7x^2 < x^3$. Take $C = 1$ and $k = 7$ as witnesses to establish that $7x^2$ is $O(x^3)$.

(Would $C = 7$ and $k = 1$ work?)

Example: Show that n^2 is not $O(n)$.

Solution: Suppose there are constants C and k for which $n^2 \leq Cn$, whenever $n > k$. Then (by dividing both sides of $n^2 \leq Cn$) by n, then $n \leq C$ must hold for all $n > k$. A contradiction!

Big-*O* Estimates for Polynomials

Example: Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_o$ where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$.

Then $f(x)$ is $O(x^n)$.

Uses triangle inequality, an exercise in Section 1.8.

Proof: $|f(\text{x})| = |a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0|$

Assuming $x > 1$

$$\leq |a_n| x^n + |a_{n-1}| x^{n-1} + \cdots + |a_1| x^1 + |a_0|$$

$$= x^n (|a_n| + |a_{n-1}| / x + \cdots + |a_1| / x^{n-1} + |a_0| / x^n)$$

$$\leq x^n (|a_n| + |a_{n-1}| + \cdots + |a_1| + |a_0|)$$

- Take $C = |a_n| + |a_{n-1}| + \cdots + |a_0|$ and $k = 1$. Then $f(x)$ is $O(x^n)$.
- The leading term $a_n x^n$ of a polynomial dominates its growth.

Big-*O* Estimates for some Important Functions

Example: Use big-*O* notation to estimate the sum of the first *n* positive integers.

Solution: $1 + 2 + \cdots + n \ \leq \ n + n + \cdots n \ = \ n^2$

$$1 + 2 + \ldots + n \text{ is } O(n^2) \text{ taking } C = 1 \text{ and } k = 1.$$

Example: Use big-*O* notation to estimate the factorial function $f(n) = n! = 1 \times 2 \times \cdots \times n$.

Solution:

$$n! = 1 \times 2 \times \cdots \times n \ \leq \ n \times n \times \cdots \times n \ = \ n^n$$

$$n! \text{ is } O(n^n) \text{ taking } C = 1 \text{ and } k = 1.$$

Continued →

Big-O Estimates for some Important Functions

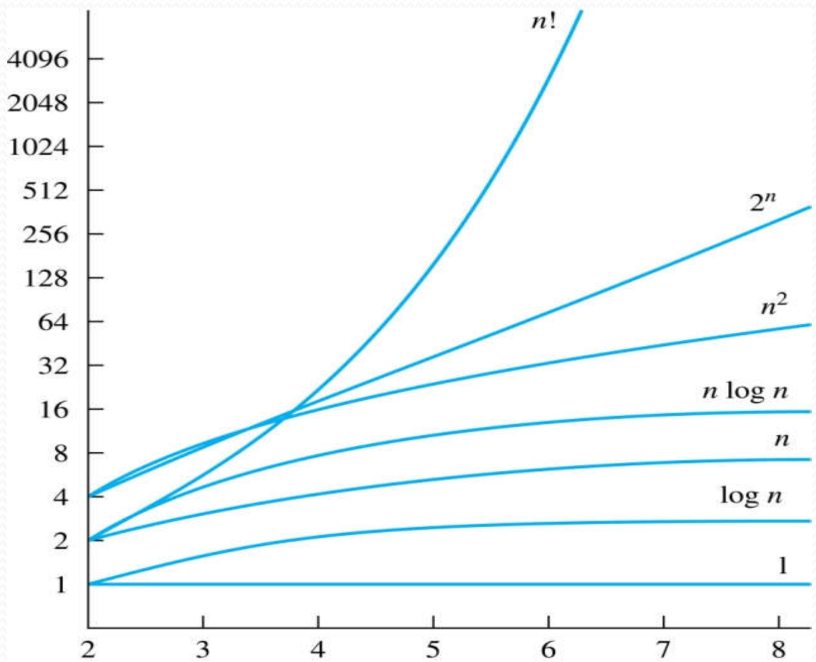
Example: Use big-O notation to estimate log n!

Solution: Given that $n! \leq n^n$ (previous slide)

then $\log(n!) \leq n \cdot \log(n)$.

Hence, log(n!) is $O(n \cdot \log(n))$ taking $C = 1$ and $k = 1$.

Display of Growth of Functions



Note the difference in behavior of functions as *n* gets larger

Useful Big-O Estimates Involving Logarithms, Powers, and Exponents

- If $d > c > 1$, then

  n^c is $O(n^d)$, but n^d is not $O(n^c)$.

- If $b > 1$ and c and d are positive, then

  $(\log_b n)^c$ is $O(n^d)$, but n^d is not $O((\log_b n)^c)$.

- If $b > 1$ and d is positive, then

  n^d is $O(b^n)$, but b^n is not $O(n^d)$.

- If $c > b > 1$, then

  b^n is $O(c^n)$, but c^n is not $O(b^n)$.

Combinations of Functions

- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then $(f_1 + f_2)(x)$ is $O(\max(|g_1(x)|,|g_2(x)|))$.
  - See next slide for proof
- If $f_1(x)$ and $f_2(x)$ are both $O(g(x))$ then $(f_1 + f_2)(x)$ is $O(g(x))$.
  - See text for argument
- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then $(f_1 f_2)(x)$ is $O(g_1(x)g_2(x))$.
  - See text for argument

Combinations of Functions

- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then
  $(f_1 + f_2)(x)$ is $O(\max(|g_1(x)|,|g_2(x)|))$.

  - By the definition of big-O notation, there are constants C_1, C_2, k_1, k_2 such that $|f_1(x) \leq C_1|g_1(x)|$ when $x > k_1$ and $f_2(x) \leq C_2|g_2(x)|$ when $x > k_2$.
  - $|(f_1 + f_2)(x)| = |f_1(x) + f_2(x)|$
    $\leq |f_1(x)| + |f_2(x)|$ by the triangle inequality $|a + b| \leq |a| + |b|$
  - $|f_1(x)| + |f_2(x)| \leq C_1|g_1(x)| + C_2|g_2(x)|$
    $\leq C_1|g(x)| + C_2|g(x)|$ where $g(x) = \max(|g_1(x)|,|g_2(x)|)$
    $= (C_1 + C_2)\,|g(x)|$
    $= C|g(x)|$ where $C = C_1 + C_2$
  - Therefore $|(f_1 + f_2)(x)| \leq C|g(x)|$ whenever $x > k$, where $k = \max(k_1,k_2)$.

Ordering Functions by Order of Growth

- Put the functions below in order so that each function is big-O of the next function on the list.
- $f_1(n) = (1.5)^n$
- $f_2(n) = 8n^3+17n^2 +111$
- $f_3(n) = (\log n)^2$
- $f_4(n) = 2^n$
- $f_5(n) = \log(\log n)$
- $f_6(n) = n^2 (\log n)^3$
- $f_7(n) = 2^n (n^2 +1)$
- $f_8(n) = n^3+ n(\log n)^2$
- $f_9(n) = 10000$
- $f_{10}(n) = n!$

We solve this exercise by successively finding the function that grows slowest among all those left on the list.

- $f_9(n) = 10000$ (constant, does not increase with n)
- $f_5(n) = \log(\log n)$ (grows slowest of all the others)
- $f_3(n) = (\log n)^2$ (grows next slowest)
- $f_6(n) = n^2 (\log n)^3$ (next largest, $(\log n)^3$ factor smaller than any power of n)
- $f_2(n) = 8n^3+17n^2 +111$ (tied with the one below)
- $f_8(n) = n^3+ n(\log n)^2$ (tied with the one above)
- $f_1(n) = (1.5)^n$ (next largest, an exponential function)
- $f_4(n) = 2^n$ (grows faster than one above since $2 > 1.5$)
- $f_7(n) = 2^n (n^2 +1)$ (grows faster than above because of the $n^2 +1$ factor)
- $f_{10}(n) = n!$ ($n!$ grows faster than c^n for every c)

Big-Omega Notation

Definition: Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are constants C and k such that

$$|f(x)| \geq C|g(x)| \quad \text{when } x > k.$$

Ω is the upper case version of the lower case Greek letter ω.

- We say that "$f(x)$ is big-Omega of $g(x)$."
- Big-O gives an upper bound on the growth of a function, while Big-Omega gives a lower bound. Big-Omega tells us that a function grows at least as fast as another.
- $f(x)$ is $\Omega(g(x))$ if and only if $g(x)$ is $O(f(x))$. This follows from the definitions. See the text for details.

Big-Omega Notation

Example: Show that $f(x) = 8x^3 + 5x^2 + 7$ is $\Omega(g(x))$ where $g(x) = x^3$.

Solution: $f(x) = 8x^3 + 5x^2 + 7 \geq 8x^3$ for all positive real numbers *x*.

- Is it also the case that $g(x) = x^3$ is $O(8x^3 + 5x^2 + 7)$?

Big-Theta Notation

Θ is the upper case version of the lower case Greek letter θ.

- **Definition**: Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. The function $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$.

- We say that "f is big-Theta of $g(x)$" and also that "$f(x)$ is of *order* $g(x)$" and also that "$f(x)$ and $g(x)$ are of the *same order*."
- $f(x)$ is $\Theta(g(x))$ if and only if there exists constants C_1, C_2 and k such that $C_1 g(x) < f(x) < C_2 g(x)$ if $x > k$. This follows from the definitions of big-O and big-Omega.

Big Theta Notation

Example: Show that the sum of the first n positive integers is $\Theta(n^2)$.

Solution: Let $f(n) = 1 + 2 + \cdots + n$.

- We have already shown that $f(n)$ is $O(n^2)$.
- To show that $f(n)$ is $\Omega(n^2)$, we need a positive constant C such that $f(n) > Cn^2$ for sufficiently large n. Summing only the terms greater than $n/2$ we obtain the inequality

$$\begin{aligned} 1 + 2 + \cdots + n &\geq \lceil n/2 \rceil + (\lceil n/2 \rceil + 1) + \cdots + n \\ &\geq \lceil n/2 \rceil + \lceil n/2 \rceil + \cdots + \lceil n/2 \rceil \\ &= (\mathrm{n} - \lceil n/2 \rceil + 1)\lceil n/2 \rceil \\ &\geq (\mathrm{n}/2)(\mathrm{n}/2) = n^2/4 \end{aligned}$$

- Taking $C = 1/4$, $f(n) > Cn^2$ for all positive integers n. Hence, $f(n)$ is $\Omega(n^2)$, and we can conclude that $f(n)$ is $\Theta(n^2)$.

Big-Theta Notation

Example: Show that $f(x) = 3x^2 + 8x \log x$ is $\Theta(x^2)$.

Solution:

- $3x^2 + 8x \log x \leq 11x^2$ for $x > 1$,
  since $0 \leq 8x \log x \leq 8x^2$.
  - Hence, $3x^2 + 8x \log x$ is $O(x^2)$.
- x^2 is clearly $O(3x^2 + 8x \log x)$
- Hence, $3x^2 + 8x \log x$ is $\Theta(x^2)$.

Big-Theta Notation

- When $f(x)$ is $\Theta(g(x))$ it must also be the case that $g(x)$ is $\Theta(f(x))$.
- Note that $f(x)$ is $\Theta(g(x))$ if and only if it is the case that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
- Sometimes writers are careless and write as if big-*O* notation has the same meaning as big-Theta.

Big-Theta Estimates for Polynomials

Theorem: Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_o$ where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$.

Then $f(x)$ is of order x^n (or $\Theta(x^n)$).

(The proof is an exercise.)

Example:

The polynomial $f(x) = 8x^5 + 5x^2 + 10$ is order of x^5 (or $\Theta(x^5)$).

The polynomial $f(x) = 8x^{199} + 7x^{100} + x^{99} + 5x^2 + 25$ is order of x^{199} (or $\Theta(x^{199})$).

Complexity of Algorithms

Section 3.3

Section Summary

- Time Complexity
- Worst-Case Complexity
- Algorithmic Paradigms
- Understanding the Complexity of Algorithms

The Complexity of Algorithms

- Given an algorithm, how efficient is this algorithm for solving a problem given input of a particular size? To answer this question, we ask:
  - How much time does this algorithm use to solve a problem?
  - How much computer memory does this algorithm use to solve a problem?
- When we analyze the time the algorithm uses to solve the problem given input of a particular size, we are studying the *time complexity* of the algorithm.
- When we analyze the computer memory the algorithm uses to solve the problem given input of a particular size, we are studying the *space complexity* of the algorithm.

The Complexity of Algorithms

- In this course, we focus on time complexity. The space complexity of algorithms is studied in later courses.
- We will measure time complexity in terms of the number of operations an algorithm uses and we will use big-*O* and big-Theta notation to estimate the time complexity.
- We can use this analysis to see whether it is practical to use this algorithm to solve problems with input of a particular size. We can also compare the efficiency of different algorithms for solving the same problem.
- We ignore implementation details (including the data structures used and both the hardware and software platforms) because it is extremely complicated to consider them.

Time Complexity

- To analyze the time complexity of algorithms, we determine the number of operations, such as comparisons and arithmetic operations (addition, multiplication, etc.). We can estimate the time a computer may actually use to solve a problem using the amount of time required to do basic operations.
- We ignore minor details, such as the "house keeping" aspects of the algorithm.
- We will focus on the *worst-case time* complexity of an algorithm. This provides an upper bound on the number of operations an algorithm uses to solve a problem with input of a particular size.
- It is usually much more difficult to determine the *average case time complexity* of an algorithm. This is the average number of operations an algorithm uses to solve a problem over all inputs of a particular size.

Complexity Analysis of Algorithms

Example: Describe the time complexity of the algorithm for finding the maximum element in a finite sequence.

```
procedure max(a1, a2, …., an: integers)
   max := a1
   for i := 2 to n
      if max < ai then max := ai
   return max{max is the largest element}
```

Solution: Count the number of comparisons.

- The $max < a_i$ comparison is made $n - 1$ times.
- Each time i is incremented, a test is made to see if $i \leq n$.
- One last comparison determines that $i > n$.
- Exactly $2(n - 1) + 1 = 2n - 1$ comparisons are made.

Hence, the time complexity of the algorithm is $\Theta(n)$.

Worst-Case Complexity of Linear Search

Example: Determine the time complexity of the linear search algorithm.

```
procedure linear search(x:integer,
            a_1, a_2, …,a_n: distinct integers)
i := 1
while (i ≤ n and x ≠ a_i)
    i := i + 1
if i ≤ n then location := i
else location := 0
return location{location is the subscript of the term that equals x, or is 0 if
    x is not found}
```

Solution: Count the number of comparisons.

- At each step two comparisons are made; $i \leq n$ and $x \neq a_i$.
- To end the loop, one comparison $i \leq n$ is made.
- After the loop, one more $i \leq n$ comparison is made.

If $x = a_i$, $2i + 1$ comparisons are used. If x is not on the list, $2n + 1$ comparisons are made and then an additional comparison is used to exit the loop. So, in the worst case $2n + 2$ comparisons are made. Hence, the complexity is $\Theta(n)$.

Average-Case Complexity of Linear Search

Example: Describe the average case performance of the linear search algorithm. (Although usually it is very difficult to determine average-case complexity, it is easy for linear search.)

Solution: Assume the element is in the list and that the possible positions are equally likely. By the argument on the previous slide, if $x = a_i$, the number of comparisons is $2i + 1$.

$$\frac{3+5+7+\ldots+(2n+1)}{n} = \frac{2(1+2+3+\ldots+n)+n}{n} =$$

$$\frac{2\left[\frac{n(n+1)}{2}\right]}{n} + 1 = n + 2$$

Hence, the average-case complexity of linear search is $\Theta(n)$.

Worst-Case Complexity of Binary Search

Example: Describe the time complexity of binary search in terms of the number of comparisons used.

procedure binary search(x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1$ {i is the left endpoint of interval}
$j := n$ {j is right endpoint of interval}
while $i < j$
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ then $i := m + 1$
    else $j := m$
if $x = a_i$ **then** *location* $:= i$
else *location* $:= 0$
return *location*{location is the subscript i of the term a_i equal to x, or 0 if x is not found}

Solution: Assume (for simplicity) $n = 2^k$ elements. Note that $k = \log n$.

- Two comparisons are made at each stage; $i < j$, and $x > a_m$.
- At the first iteration the size of the list is 2^k and after the first iteration it is 2^{k-1}. Then 2^{k-2} and so on until the size of the list is $2^1 = 2$.
- At the last step, a comparison tells us that the size of the list is the size is $2^0 = 1$ and the element is compared with the single remaining element.
- Hence, at most $2k + 2 = 2 \log n + 2$ comparisons are made.
- Therefore, the time complexity is Θ ($\log n$), better than linear search.

Worst-Case Complexity of Bubble Sort

Example: What is the worst-case complexity of bubble sort in terms of the number of comparisons made?

procedure *bubblesort*($a_1,\ldots,a_n$: real numbers with $n \geq 2$)
  for $i := 1$ to $n-1$
    for $j := 1$ to $n-i$
      if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}
{$a_1,\ldots, a_n$ is now in increasing order}

Solution: A sequence of $n-1$ passes is made through the list. On each pass $n-i$ comparisons are made.

$$(n-1)+(n-2)+\ldots+2+1 = \frac{n(n-1)}{2}$$

The worst-case complexity of bubble sort is $\Theta(n^2)$ since $\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$.

Worst-Case Complexity of Insertion Sort

Example: What is the worst-case complexity of insertion sort in terms of the number of comparisons made?

procedure *insertion sort*($a_1,\ldots,a_n$: real numbers with $n \geq 2$)
  for $j := 2$ to n
    $i := 1$
    while $a_j > a_i$
      $i := i + 1$
    $m := a_j$
    for $k := 0$ to $j - i - 1$
      $a_{j-k} := a_{j-k-1}$
    $a_i := m$

Solution: The total number of comparisons are:

$$2 + 3 + \cdots + n = \frac{n(n-1)}{2} - 1$$

Therefore the complexity is $\Theta(n^2)$.