Discrete Mathematics

Algorithms

Myles Moylan

3.1 Algorithms

1. Devise and algorithm that finds the sum of all the integers in a list (assuming the list is not empty).

procedure *summation*$(a_1, a_2, \ldots, a_n$: integers)

$sum := a_1$

for $i := 2$ **to** n

    $sum := sum + a_i$

return sum {sum is the sum of all elements in the list}

2. Describe an algorithm that takes as input a list of n integers and produces as output the largest difference obtained by subtracting an integer in the list from the one following it.

procedure *largest difference*$(a_1, a_2, \ldots, a_n$: integers)

$d := 0$

for $i := 1$ **to** $n - 1$

    $t := |a_{i+1} - a_i|$

    if $d < t$ **then**

        $d := t$

return d {d is the largest different in the list}

3. Describe an algorithm that takes as input a list of n integers in nondecreasing order and produces the list of all values that occur more than once. (Recall that a list of integers is **nondecreasing** if each integer in the list is at least as large as the previous integer in the list.)

procedure *duplicates*$(a_1, a_2, \ldots, a_n$: integers in nondecreasing order)

$k := 0$ {this counts the duplicates}

$j := 2$

while $j \leq n$

    if $a_j = a_{j-1}$ **then**

        $k := k + 1$

        $c_k := a_j$

        while $j \leq n$ and $a_j = c_k$

            $j := j + 1$

    $j := j + 1$

{$c_1, c_2, \ldots, c_k$ is the desired list}

4. Describe an algorithm that takes as input a list of n integers and finds the number of negative integers in the list.

procedure *count negatives*$(a_1, a_2, \ldots, a_n$: integers)

$count := 0$

for $i := 1$ **to** n

  if $a_i < 0$ **then**

    $count := count + 1$

return *count* {*count* is the number of negative integers in the list}

5. Describe an algorithm that takes as input a list of n integers and finds the location of the last even integer in the list or returns 0 if there are no even integers in the list.

procedure *last even location*$(a_1, a_2, \ldots, a_n$: integers)

$loc := 0$

for $i := 1$ **to** n

  if a_i is even **then**

    $loc := i$

return *loc* {*loc* is either the last even integer in the list or 0}

6. Describe an algorithm that takes as input a list of n distinct integers and finds the location of the largest even integer in the list or returns 0 if there are no even integers in the list.

procedure *largest even integer*$(a_1, a_2, \ldots, a_n$: integers)

$k := -\infty$

for $i := 1$ **to** n

  if a_i is even and $a_i > k$ **then**

    $k := i$

if $k = -\infty$ **then**

  $k := 0$

return k {k is either the largest even integer of the list or 0}

7. A **palindrome** is a string that reads the same forward and backward. Describe an algorithm for determining whether a string of n characters is a palindrome.

procedure *palindrome check*$(a_1, a_2, \ldots, a_n$: string)

answer := **true**

for $i := 1$ **to** $\lfloor n/2 \rfloor$

  if $a_i \neq a_{n+1-i}$ **then**

    answer := **false**

return *answer* {*answer* is true if and only if string is a palindrome}

8. Describe an algorithm that interchanges the values of the variables x and y, using only assignments. What is the minimum number of assignment statements needed to do this?

Three assignment statements are necessary, and sufficient, to accomplish this task:

$temp := x$
$x := y$
$y := temp$

9. Devise an algorithm to compute x^n, where x is a real number and n is an integer. [*Hint:* First give a procedure for computing x^n when n is nonnegative by successive multiplication by x, starting with 1. Then extend this procedure, and use the fact that $x^{-n} = 1/x^n$ to compute x^n when n is negative.]

procedure *compute exponent*(x : real number, n : integer)

$result := 1$

for $i := 1$ **to** $|n|$

    $result := result \cdot x$

if $n < 0$ **then**

    $result := 1/result$

return $result$ {$result$ is the exponent computed}

10. Describe an algorithm that uses only assignment statements that replaces the triple (x, y, z) with (y, z, x). what is the minimum number of assignment statements needed?

Five assignment statements are necessary, and sufficient, to accomplish this task:

$xtemp := x$
$ztemp := z$
$x := y$
$z := xtemp$
$y := ztemp$

11. Describe an algorithm that inserts an integer x in the appropriate position into the list $a_1, a_2, \ldots, a_n$ of integers that are in increasing order.

procedure *insert*($x, a_1, a_2, \ldots, a_n$: integers)

$a_{n+1} := x + 1$ {tack $x + 1$ onto the end of the list so the **while** loop will always terminate}

$i := 1$

while $x > a_i$

    $i := i + 1$ {the loop ends when i is the index for x}

for $j := 0$ **to** $n - i$ {shove the rest of the list to the right}

    $a_{n-j+1} := a_{n-j}$

$a_i := x$

{x has been inserted into the correct spot in the list, now of length $n + 1$}

12. Describe an algorithm for finding the smallest integer in a finite sequence of natural numbers.

procedure *smallest integer*($a_1, a_2, \ldots, a_n$: natural numbers)

$k := \infty$

for $i := 1$ **to** n

    if $a_i < k$ **then**

        $k := a_i$

return k {k is the smallest integer in the list}

13. Describe an algorithm that locates the first occurrence of the largest element in a finite list of integers, where the integers in the list are not necessarily distinct.

procedure *first largest*($a_1, a_2, \ldots, a_n$: integers)

$max := a_1$

$location := 1$

for $i := 2$ **to** n

  if $max < a_i$ **then**

    $max := a_i$

    $location := i$

return $location$ {$location$ is the location of the first occurrence of the largest element in the list}

14. Describe an algorithm that locates the last occurrence of the smallest element in a finite list of integers, where the integers in the list are not necessarily distinct.

procedure *last smallest*$(a_1, a_2, \ldots, a_n$: integers)

$min := a_1$

$location := 1$

for $i := 2$ **to** n

  if $min \geq a_i$ **then**

    $min := a_i$

    $location := i$

return $location$ {$location$ is the location of the last occurrence of the smallest element in then list}

15. Describe an algorithm for finding both the largest and the smallest integers in a finite sequence of integers.

procedure *minmax*$(a_1, a_2, \ldots, a_n$: integers)

$min := a_1$

$max := a_1$

for $i := 2$ **to** n

  if $max < a_i$ **then** $max := a_i$

  if $min > a_i$ **then** $min := a_i$

{min is the smallest integer in the list and max is the largest integer in the list}

16. Describe an algorithm that puts the first three terms of a sequence of integers of arbitrary length in increasing order.

procedure *first three*$(a_1, a_2, \ldots, a_n$: integers)

if $a_1 > a_2$ **then** interchange a_1 and a_2

if $a_2 > a_3$ **then** interchange a_2 and a_3

if $a_1 > a_2$ **then** interchange a_1 and a_2

{the first three elements are now in nondecreasing order}

17. Describe an algorithm that determines whether a function from a finite set of integers to another finite set of integers is onto.

procedure *onto*(f : function, $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m$: integers)

for $i := 1$ **to** m

  $hit(b_i) := 0$ {no one has been hit yet}

$count := 0$ {there have been no hits yet}

for $j := 1$ **to** n

  if $hit(f(a_j)) = 0$ **then** {a new hit}

    $hit(f(a_j)) := 1$

    $count := count + 1$

if $count = m$ **then return true else return false**

{f is onto if and only if there have been m hits}

3.2 The Growth of Functions

1. Determine whether each of these functions is $O(x)$.

a) $f(x) = 10$
Yes, since $|10| \leq |x|$ for all $x > 10$. The witnesses are $C = 1$ and $k = 10$.

b) $f(x) = 3x + 7$
Yes, since $|3x + 7| \leq |4x| = 4|x|$ for all $x > 7$. The witnesses are $C = 4$ and $k = 7$.

c) $f(x) = x^2 + x + 1$
No. There is not *constant* C such that $|x^2 + x + 1| \leq C|x|$ for all sufficiently large x. To see this, suppose this inequality held for all sufficiently large positive values of x. Then we would have $x^2 \leq Cx$, which would imply that $x \leq C$ for *all* sufficiently large x, an obvious impossibility.

d) $f(x) = 5 \log x$
Yes. This follows from the fact that $\log x < x$ for all $x > 1$ (which in turn follows from the fact that $x < 2^x$). Therefore $|5 \log x| \leq 5|x|$ for all $x > 1$. The witnesses are $C = 5$ and $k = 1$.

e) $f(x) = \lfloor x \rfloor$
Yes. This follows from the fact that $\lfloor x \rfloor \leq x$. Thus $|\lfloor x \rfloor| \leq |x|$ for all $x > 0$. The witnesses are $C = 1$ and $k = 0$.

f) $f(x) = \lceil x/2 \rceil$
Yes. This follows from the fact that $\lceil x/2 \rceil \leq (x/2) + 1$. Thus $|\lceil x/2 \rceil| \leq |(x/2) + 1| \leq |x|$ for all $x > 2$. The witnesses are $C = 1$ and $k = 2$.

2. Use the definition of "$f(x)$ is $O(g(x))$" to show that $x^4 + 9x^3 + 4x + 7$ is $O(x^4)$.

We need to put some bound on the lower order terms. If $x > 9$ then we have $x^4 + 9x^3 + 4x + 7 \leq x^4 + x^4 + x^4 + x^4 = 4x^4$. Therefore $x^4 + 9x^3 + 4x + 7$ is $O(x^4)$, taking witnesses $C = 4$ and $k = 9$.

3. Find the least integer n such that $f(x)$ is $O(x^n)$ for each of these functions.

a) $f(x) = 2x^3 + x^2 \log x$
Since $\log x$ grows more slowly than x, $x^2 \log x$ grows more slowly than x^3, so the first term dominates. Therefore this function is $O(x^3)$ but not $O(x^n)$ for any $n < 3$. More precisely, $2x^3 + x^2 \log x \leq 2x^3 + x^3 = 3x^3$ for all x, so we have witnesses $C = 3$ and $k = 0$.

b) $f(x) = 3x^3 + (\log x)^4$
We know that $\log x$ grows so much more slowly than x that *every* power of $\log x$ grows more slowly than x. Thus the first term dominates, and the best estimate is $O(x^3)$. More precisely, $(\log x)^4 < x^3$ for all $x > 1$, so $3x^3 + (\log x)^4 \leq 3x^3 + x^3 = 4x^3$ for all x, so we have witnesses $C = 4$ and $k = 1$.

c) $f(x) = (x^4 + x^2 + 1)/(x^3 + 1)$
By long division, we see that $f(x) = x +$ lower order terms. Therefore this function is $O(x)$, so $n = 1$. In fact, $f(x) = x + \frac{1}{x+1} \leq 2x$ for all $x > 1$, so the witnesses can be taken to be $C = 2$ and $k = 1$.

d) $f(x) = (x^4 + 5 \log x)/(x^4 + 1)$
By long division, this quotient has the form $f(x) = 1 +$ lower order terms. Therefore this function is $O(1)$ with $n = 0$. Since $5 \log x < x^4$ for $x > 1$, we have $f(x) \leq 2x^4/x^4 = 2$, so we can take witnesses as $C = 2$ and $k = 1$.

4. Show that $x^2 + 4x + 17$ is $O(x^3)$ but that x^3 is not $O(x^2 + 4x + 17)$.

On the one hand we have $x^2 + 4x + 17 \leq x^2 + x^2 + x^2 = 3x^2 \leq 3x^3$ for all $a > 17$, so $x^2 + 4x + 17$ is $O(x^3)$, with witnesses $C = 3$ and $k = 17$. On the other hand, if x^3 were $O(x^2 + 4x + 17)$, then we would have $x^3 \leq C(x^2 + 4x + 17) \leq 3Cx^2$ for all sufficiently large x. But this says that $x < 3C$, which is impossible for the constant C to satisfy for all large x. Therefore x^3 is not $O(x^2 + 4x + 17)$.

5. Explain what it means for a function to be $O(1)$.

A function f is $O(1)$ if $|f(x)| \leq C$ for all sufficiently large x. In other words, f is $O(1)$ if its absolute value is **bounded** for all $x > k$ (where k is some constant).

6. Determine whether each of the functions 2^{n+1} and 2^{2n} is $O(2^n)$.

Because $2^{n+1} = 2 \cdot 2^n$, clearly it is $O(2^n)$ (take $C = 2$). To see that 2^{2n} is not $O(2^n)$, look at the ratio: $2^{2n}/2^n = 2^n$. Because this is unbounded, there is no constant C such that $2^{2n} \leq C \cdot 2^n$ for all sufficiently large n.

7. Suppose that you have two different algorithms for solving a problem. To solve a problem of size n, the first algorithm uses exactly $n(\log n)$ operations and the second algorithm uses exactly $n^{3/2}$ operations. As n grows, which algorithm uses fewer operations?

Because $n \log n$ is $O(n^{3/2})$ but $n^{3/2}$ is not $O(n \log n)$, the first algorithm uses fewer operations for large n. In fact, if we solve $n \log n < n^{3/2}$, we get $n > 4$. So the first algorithm uses fewer operations for all $n > 4$.

8. Give as good a big-O estimate as possible fore each of these functions.

a) $(n^2 + 8)(n + 1)$
Here we see n^2 being multiplied by n; thus this function is $O(n^3)$.

b) $(n \log n + n^2)(n^3 + 2)$
Here, after factoring, we see that the entire function is $O(n^5)$.

c) $(n! + 2^n)(n^3 + \log(n^2 + 1))$
Fir the first factor we note that $2^n < n!$ for $n \geq 4$, so the significant term is $n!$. For the second factor, the significant term is n^3. Therefore this function is $O(n^3 n!)$.

9. Give a big-O estimate for each of these functions. For the function g is your estimate that $f(x)$ is $O(g(x))$, use a simple function g of the smallest order.

a) $n \log(n^2 + 1) + n^2 \log n$
First we note that $\log(n^2 + 1)$ and $\log n$ are in the same big-O class, since $\log n^2 = 2 \log n$. Therefore the second term here dominates the first, and the simplest good answer would be $O(n^2 \log n)$.

b) $(n \log n + 1)^2 + (\log n + 1)(n^2 + 1)$
The first term is in the same big-O class as $O(n^2(\log n)^2)$, while the second is in a slightly smaller class, $O(n^2 \log n)$. Therefore the answer is $O(n^2(\log n)^2)$.

c) $n^{2^n} + n^{n^2}$
The only issue here is whether 2^n or n^2 is the faster-growing, and clearly it is the former. Therefore the best big-O estimate we can give is $O(n^{2^n})$.

10. Show that $f(x)$ is $\Theta(g(x))$ if and only if $f(x)$ is $O(g(s))$ and $g(x)$ is $O(f(x))$.

If $f(x)$ is $\Theta(g(x))$, then $|f(x)| \leq C_2|g(x)|$ and $|g(x)| \leq C_1^{-1}|f(x)|$ for all $x > k$. Thus $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$. Conversely, suppose that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$. Then (with appropriate choice of variable names) we may assume that $|f(x)| \leq C_2|g(x)|$ and $|g(x)| \leq C|f(x)|$ for all $x > k$. (The k here will be the larger of the two k's involved in the hypotheses.) If $C > 0$ then we can take $C_1 = C^{-1}$ to obtain the desired inequalities in "$f(x)$ is $\Theta(g(x))$." If $C \leq 0$, then $g(x) = 0$ for all $x > k$, and hence by the first inequality $f(x) = 0$ for all $x > k$; thus we have $f(x) = g(x)$ for all $x > k$, and we can take $C_1 = C_2 = 1$.

11. Explain what it means for a function to be $\Theta(1)$.

Looking at the definition tells us that if $f(x)$ is $\Theta(1)$ then $|f(x)|$ has to be bounded between two positive constants. In other words, $f(x)$ can't get too large (either positive or negative), and it can't get too close to 0.

3.3 Complexity of Algorithms

1. Give a big-O estimate for the number of operations used in this segment of an algorithm.

$t := 0$

for $i := 1$ **to** 3

  for $j := 1$ **to** 4

    $t := t + ij$

Since the statement $t := t + ij$ is only executed 12 times, the number of operations is $O(1)$.

2. Give a big-O estimate for the number of additions used in this segment of an algorithm.

$t := 0$

for $i := 1$ **to** n

  for $j := 1$ **to** n

    $t := t + i + j$

The statement $t := t + i + j$ is executed here n^2 times, so the number of operations is $O(n^2)$.

3. Give a big-O estimate for the number of operations, where an operation is a comparison or a multiplication, used in this segment of an algorithm (ignoring comparisons uses to test the conditions in the **for** loops, where $a_1, a_2, \ldots, a_n$ are positive real numbers).

$m := 0$

for $i := 1$ **to** n

  for $j := i + 1$ **to** n

    $m := \max(a_1 a_j, m)$

The nested loop indicates that the number of operations is $O(n^2)$.

4. Give a big-O estimate for the number of operations, where an operation is an addition or a multiplication, used in this segment of an algorithm (ignoring comparisons used ot test the conditions in the **while** loop).

$i := 1$

$t := 0$

while $i \leq n$

  $t := t + i$

  $i := 2i$

With i doubled at every iteration starting at 1, and n used as an upper bound for i, this **while** loop looks to be working on the order of $O(\log_2 n)$ executions.

5. Suppose that an element is known to be among the first four elements in a list of 32 elements. Would a linear search or a binary search locate this element more rapidly?

The linear search would find this element after at most 9 comparisons (4 to determine that we have not yet finished with the **while** loop, 4 more to determine if we have located the desired element yet, and 1 to set the value of *location*). Binary search will take $2 \log 32 + 2 - 2 \cdot 5 + 2 = 12$ comparisons. Thus, the linear search will be faster, in terms of comparisons.

6. Give a big-O estimate for the number of comparisons used by the algorithm that determines the number of 1s in a bit string by examining each bit of the string to determine whether it is a 1 bit.

Since the algorithm simply scans one bit at a time it is clearly $O(n)$ with an n-length string of bits.

7. **a)** Suppose we have n subsets $S_1, S_2, \ldots, S_n$ of the set $\{1, 2, \ldots, n\}$. Express a brute-force algorithm that determines whether there is a disjoint pair of these subsets. [*Hint:* The algorithm should loop through the subsets; for each subset S_i, it should then loop through all other subsets; and for each of these other subsets S_j, it should loop through all elements k in S_i to determine whether k also belongs to S_j.]

procedure *disjoint pair*($S_1, S_2, \ldots, S_n$: subsets of $\{1, 2, \ldots, n\}$)
answer := **false**
for $i := 1$ **to** n
  for $j := i + 1$ **to** n
    disjoint := **true**
    for $k := 1$ **to** n
      if $k \in S_i$ and $k \in S_j$ **then** *disjoint* := **false**
    if *disjoint* **then** *answer* := **true**
return *answer*

b) Give a big-O estimate for the number of times the algorithm needs to determine whether an integer is in one of the subsets.

The three nested loops imply that the elementhood test needs to be applied $O(n^3)$ times.

8. The conventional algorithm for evaluating a polynomial $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ at $x = c$ can be expressed in pseudocode by

procedure *polynomial*($c, a_0, a_1, \ldots, a_n$: real numbers)
power := 1
$y := a_0$
for $i := 1$ **to** n
  power := *power* $*$ c
  $y := y + a_i * power$
return y $\{y = a_n c^n + a_{n-1}c^{n-1} + \cdots + a_1 c + a_0\}$

where the final value of y is the value of the polynomial at $x = c$.

a) Evaluate $3x^2 + x + 1$ at $x = 2$ by working through each step of the algorithm showing the values assigned at each assignment step.

Here we have $n = 2$, $a_0 = 1$, $a_1 = 1$, $a_2 = 3$, and $c = 2$. Initially, we set *power* equal to 1 and y equal to 1. The first time through the **for** loop (with $i = 1$), *power* becomes 2 and so y becomes $1 + 1 \cdot 2 = 3$. The second and final time through the loop, *power* becomes $2 \cdot 2 = 4$ and y becomes $3 + 3 \cdot 4 = 15$. Thus the value of the polynomial at $x = 2$ is 15.

b) Exactly how many multiplications and additions are used to evaluate a polynomial of degree n at $x = c$? (Do not count additions used to increment the loop variable.)

Each pass through the loop requires two multiplications and one addition. Therefore there are a total of $2n$ multiplications and n additions in all.

9. What is the largest n for which one can solve within one second a problem using an algorithm that requires $f(n)$ bit operations, where each bit operation is carried out in 10^{-9} seconds, with these functions $f(n)$?

Since each bit operation requires 10^{-9} seconds, we want to know for what value of n there will be at most 10^9 bit operations required. Thus we need to set the expression equal to 10^9, solve for n, and round down if necessary.

a) $\log n$

Solving $\log_2 n = 10^9$, we get $n = 2^{10^9}$. By taking $\log_{10}$ of both sides, we find that this number is approximately equal to $10^{300,000,000}$.

b) n

Clearly, $n = 10^9$.

c) $n \log n$

Solving $n \log n = 10^9$ is not trivial. There is no good formula for solving such **transcendental** equations. An algorithm that works well with a calculator is to rewrite the equation as $n = 10^9 / \log n$, enter a random starting value, say $n = 2$, and repeatedly calculate a new value of n. After a number of iterations in this way, the numbers stabilize at approximately $39,620,077$, so that is the answer.

d) n^2

Solving $n^2 = 10^9$ gives $n = 10^{4.5}$, which is $31,622$ when rounding down.

e) 2^n

Solving $2^n = 10^9$ gives $n = \log(10^9) \approx 29.9$. Rounding down gives the answer, 29.

f) $n!$

The quickest way to find the largest value of n such that $n! \leq 10^9$ is simply to try a few values of n. We find that $12! \approx 4.8 \times 10^8$ while $13! \approx 6.2 \times 10^9$, so the answer is 12.

10. How much time does an algorithm using 2^{50} operations need if each operation takes these amounts of time?

a) 10^{-6} s

$2^{50} \times 10^{-6} = 1,125,899,907$ seconds ≈ 36 years

b) 10^{-9} s

$2^{50} \times 10^{-9} = 1,125,899.907$ seconds ≈ 13 days

c) 10^{-12} s

$2^{50} \times 10^{-12} = 1,125.899907$ seconds ≈ 19 minutes