COMP10002 Foundations of Algorithms

Semester Two, 2017

Arrays and Algorithms

Overview

Chapter 7 (Part I)

Chapter 12 (Part I)

Assertions and correctness

Measuring efficiency

Binary search

Quicksort

Program examples

Exercises

Chapter 7 – Concepts (Part I)

In an array, every element is of the same type, and the `i`'th value can be accessed independently of other elements via `A[i-1]`. An array is an address mapping directly on to the computer's memory structure.

In K&R C, arrays must be sized at compilation time. In 1990 standard C, that restriction is lifted in functions – the array size may be an `int` function parameter.

There is no execution-time array bounds checking, and responsibility rests with the programmer.

Chapter 7 – Arrays and functions

If used without a subscript, the array name itself is interpreted as being a pointer constant. When used in a function call (or anywhere else), the address of the array gets passed. The size of each element of the array must be declared, but not the number of elements.

In the function, the array argument is received as a pointer variable, and can be used to alter array elements. The notation `A[i]` is merely shorthand for `*(A+i)`.

Functions cannot return arrays, but (Chapter 10) are able to allocate new memory space and return a pointer to it.

Chapter 7 – Program examples (Part I)

- `array1.c`
- `insertionsort.c`
- `matrixadd.c`
- `twodarray.c`
- `pointer4.c`

Chapter 7 – Exercises

Exercise 1

Write a function that takes two arguments, an array `A` of type `int`, and an integer `n` indicating how many elements there are in `A`; and returns `IS_SORTED` if `A` is in (ascending) order, and `NOT_SORTED` if there are ordering violations.

Exercise 2

Ditto arguments, but returning the number of distinct values in the array `A`. You may not alter the array.

Exercise 3

Ditto arguments, but identifying the start point of the longest run of ascending values. You may not alter the array.

Chapter 7 – Summary (Part I)

Key messages:

- Arrays are accessed by pointers; subscript indexing is just another form of pointer dereference
- Because of this relationship, there is no compile-time or run-time bounds checking
- Functions receive arrays as a pointer to an element of the array base type
- Two-dimensional (and higher) arrays need to the thought of as being arrays of arrays, including when being passed into functions.

Chapter 12 – Concepts (Part I)

Algorithms are at the heart of computing.

The most important attribute of any algorithm is correctness – it must solve the problem it claims to.

After that, we are interested in resources – the memory space required to execute it, and its execution time.

Both of these resource requirements are likely to grow as some function of n, where n is the "size" of the problem instance.

Array search

Given: an array `A` of some type, and an integer `n` indicating how many elements there are in `A` that may be inspected.

Question: does the element `x` (of the same type as the elements of `A`) appear in `A`? If so, which location?

Linear search

$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq x$
  $i \leftarrow i + 1$
if $i \geq n$
  return *not_found*
else
  return i

Can we demonstrate that this algorithm is correct?

Can we predict its behavior in terms of execution time?

Linear search – Correctness

Consider this augmented version:

define $P \equiv (0 \leq i \leq n)$ **and** $(x \notin A[0 \ldots i-1])$

$i \leftarrow 0$
assert: P
while $i < n$ **and** $A[i] \neq x$
    assert: P **and** $(i < n$ **and** $A[i] \neq x)$
    $i \leftarrow i + 1$
    assert: P

assert: P **and** $(i \geq n$ **or** $A[i] = x)$
if $i \geq n$
    assert: P **and** $(\ldots)$ **and** $(i \geq n) \implies x \notin A[0 \ldots n-1]$
    return *not_found*
else
    assert: P **and** $(\ldots)$ **and** $(i \ngeq n) \implies A[i] = x$
    return i

Linear search – Correctness

Assertions are argued statements about what must be true as a program executes.

With the use of logic, and precise rules associated with the semantics of executable statements, invariants can be used to provide formal proofs of program correctness.

Thinking in terms of invariants – especially loop invariants – will help you develop safe programs.

The C pseudo-function `assert` is defined in `assert.h` and can be inserted into your programs. If the argument expression is violated, program execution will be halted and the invalid assertion identified.

Linear search – Efficiency

The number of loop iterations varies between zero and n.

In the best case, the item x is found in the first location. But in the worst case, n iterations are required.

While it might be tempting to hope for the best, it is more professional (and much safer) to instead plan for the worst.

In particular, an adversary is able to rearrange the array values to force the bad behavior to occur.

Defining efficiency

If the program requires n steps on an input of size n, it is linear.

If it takes one second when $n = 1{,}000$, it will take around two seconds if n is increased to 2,000.

It is the rate of growth that is important, not the exact running time on any input.

A different algorithm that requires $2n$ steps is still linear; as is a third method that requires $15n + 27\sqrt{n} - e \log n$ steps.

Defining efficiency

To capture this idea, we define sets of functions that are all asymptotically equivalent in terms of their eventual long term growth rate:

$$f(n) \in O(g(n))$$

if and only if

$$\exists n_0, c > 0 : \forall n > n_0, f(n) \leq c \cdot g(n) .$$

This is a complex definition!

Defining efficiency

In words:

> *$f(n)$ is a function that is order $g(n)$ when a positive threshold n_0 and a positive constant c can be identified such that, for every n larger than n_0, $f(n)$ is bounded above by c times $g(n)$.*

Example 1: Take $f_1(n) = 2n$. Then $f_1(n) \in O(n)$.
Demonstration: take $n_0 = 1$ and $c = 2$.

Example 2: Take $f_2(n) = 15n + 27\sqrt{n} - e \log n$.
Then $f_2(n) \in O(n)$.
Demonstration: take $n_0 = 1$ and $c = 42$.

Defining efficiency

Example 3: Take $f_3(n) = f_2(n) \times f_1(n)$.
Then $f_3(n) \in O(n^2)$.
(And also $\in O(n^2 \log n), O(n^3)$, and so on).
Demonstration: take $n_0 = 1$ and $c = 84$.

Example 4: Take $f_4(n) = f_2(n)/f_1(n)$.
Then $f_4(n) \in O(1)$.
(And also $\in O(\log n), O(\sqrt{n}), O(n)$, and so on).

Example 5: Take $f_5(n) = f_2(n) - f_1(n)$.
Then $f_5(n) \in O(n)$.
(And also $\in \ldots$).

Defining efficiency

Note that $O(g(n))$ is a set of functions, including all functions that have the same or smaller growth rate.

Given a function $f(n)$ to be categorized, it is usual to make use of the *simplest* such $g(n)$ function, by dropping constants and secondary terms; and also to choose $g(n)$ so that the *smallest* set $O(g(n))$ is generated.

So, while it is correct that $2n \log_2 n \in O(5n^2 - 3)$, it is usual to keep it simple, and say that $2n \log_2 n \in O(n \log n)$.

Defining efficiency

Why???

The "order" notation provides a tremendously useful abstraction that lets algorithms be compared.

If $f(n) \in O(g(n))$, and $g(n) \notin O(f(n))$, then an algorithm that requires $f(n)$ steps will, for large enough inputs, be faster than an algorithm that takes $g(n)$ steps.

Searching – Sorted array

If array is *sorted*, can exit loop early:

$i \leftarrow 0$
while $i < n$ **and** $A[i] < x$
    $i \leftarrow i + 1$
if $i \geq n$ **or** $A[i] > x$
    return *not_found*
else
    return i

But worst-case execution time is still linear. And average case is only half that, so no better.

(Can you alter the previous predicate P and show that this method is also correct?)

Binary search

If array is sorted, search by repeated range halving is better:

define $P \equiv (0 \le lo \le hi \le n)$ **and**
$(x \notin A[0 \ldots lo - 1])$ **and** $(x \notin A[hi \ldots n - 1])$

Initialization:
$lo, hi \leftarrow 0, n$
assert: P

Loop:
while $lo < hi$
assert: P **and** $lo < hi$

Binary search

$m \leftarrow (lo + hi)/2$
assert: P **and** $lo \leq m < hi$
if $x < A[m]$
  assert: P **and** $x < A[m \ldots n-1]$
  $hi \leftarrow m$
  assert: P
else if $x > A[m]$
  assert: P **and** $x > A[0 \ldots m]$
  $lo \leftarrow m + 1$
  assert: P
else
  assert: P **and** $x \not< A[m]$ **and** $x \not> A[m] \implies x = A[m]$
  return m

Binary search

Upon loop exit:
assert: P **and** $lo \geq hi \implies x \notin A[0 \ldots n-1]$
return *not_found*

Binary search

Without the assertions it is *shorter*, but the intellectual complexity is unchanged:

$lo, hi \leftarrow 0, n$
while $lo < hi$
    $m \leftarrow (lo + hi)/2$
    if $x < A[m]$
        $hi \leftarrow m$
    else if $x > A[m]$
        $lo \leftarrow m + 1$
    else
        return m
return *not_found*

Binary search – Termination

It is also important to demonstrate that loops terminate. All the assertions in the world won't help if the loop doesn't actually exit to yield the required post-condition.

For a loop to terminate:

1. Define a function $t()$ over suitable variables
2. Demonstrate that the loop pre-condition results in $t()$ having a finite initial value t_0
3. Demonstrate that every iteration reduces $t()$ by at least k, for some fixed $k > 0$
4. Demonstrate that the loop ends when $t()$ reaches some lower bound t_e.

Binary search – Termination

In the case of binary search:

1. Define $t() = hi - lo$
2. Loop precondition requires that $t_0 = n$
3. Two alternatives exist for loop iteration:
   - $hi \leftarrow m$, knowing $lo \leq m < hi$
   - $lo \leftarrow m + 1$, knowing $lo \leq m < hi$

   Both alternatives reduce $t()$ by at least $k = 1$
4. Loop only continues if $lo < hi$, that is, stops once $t() \leq t_e = 0$.

Don't underestimate the difficulty of getting even simple algorithms *exactly* right!

Binary search as a recursive C function


```
int
binary_search(data_t A[], int lo, int hi,
            data_t *key, int *locn) {
    int mid, outcome;
    /* if key is in A, it is between A[lo] and A[hi-1] */
    if (lo>=hi) {
        return BS_NOT_FOUND;
    }
    mid = (lo+hi)/2;
    if ((outcome = cmp(key, A+mid)) < 0) {
        return binary_search(A, lo, mid, key, locn);
    } else if (outcome > 0) {
        return binary_search(A, mid+1, hi, key, locn);
    } else {
        *locn = mid;
        return BS_FOUND;
    }
}
```

Binary search as a recursive C function

The function `cmp` is a standard C paradigm.

It is usually passed in to a sorting or searching function as a function argument (Chapter 10).

It compares (via pointers to underlying objects) two elements, and returns:

- $-$ve, if first item should come prior to second
- 0, if two items can be considered to be equal
- $+$ve, if first item should come after second.

The string comparison functions `strcmp` and `strncmp` (Chapter 7, Part II) fit this template, as does the integer comparison function shown in `binarysearch.c`.

Binary search – Efficiency

How long does binary search require?

Based on the recursive C function, the time taken is bounded above by $T(n)$, where

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 1 + T(\lfloor n/2 \rfloor) & \text{if } n > 1 \end{cases}$$

Recurrence relations often arise in the analysis of algorithms.

The exact solution of this one is
$T(n) = 1 + \lfloor \log_2 n \rfloor \in O(\log n)$.

Binary search – Efficiency

As a sanity check, suppose linear search takes (say) $2.5n$ fundamental (whatever that means) operations.

And that binary search, which is more complex, takes (say) $10 \log_2 n$ operations.

When is $2.5n < 10 \log_2 n$?

Only when $n < 16$.

The rapid superiority of binary search is unsurprising, since $\log n$ grows much more slowly than n.

Sorting

How did the array get sorted in the first place?

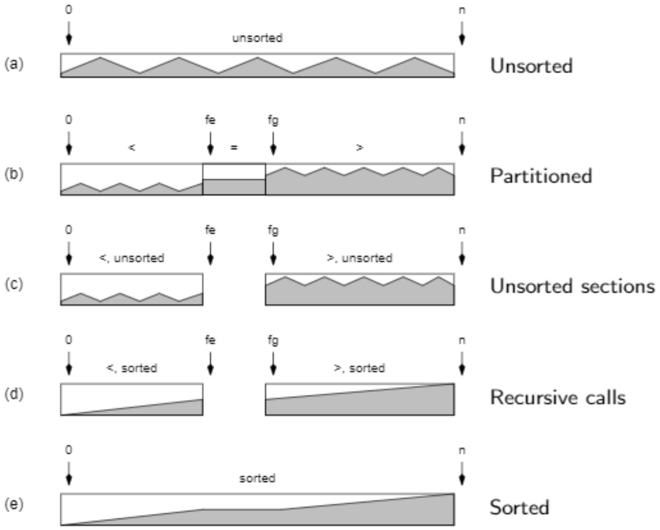
Chapter 7 describes insertionsort.

In the worst case, insertionsort requires $n(n-1)/2$ comparisons between items. It is an $O(n^2)$-time method. The time taken grows as a quadratic function of the size of the problem.

That is bad news, and insertionsort is not a good algorithm. Bubblesort, described in the yellow edition, has the same problem.

Does sorting *need* to take quadratic time?

Quicksort – The idea



Quicksort – Algorithm

define $R \equiv 0 \leq fe < fg \leq n$ **and** $A[0 \ldots fe-1] < p$ **and**
$A[fe \ldots fg-1] = p$ **and** $A[fg \ldots n-1] > p$

if $n \leq 1$
    return
else
    $p \leftarrow$ any element in $A[0 \ldots n-1]$
    assert: $n > 1$ **and** $p \in A[0 \ldots n-1]$
    $(fe, fg) \leftarrow \mathit{partition}(A, n, p)$
    assert: R
    $\mathit{quicksort}(A[0 \ldots fe-1])$
    $\mathit{quicksort}(A[fg \ldots n-1])$
    assert: $A[0 \ldots fe-1]$ is sorted **and** $A[fg \ldots n-1]$ is sorted
        and $R \implies A[0 \ldots n-1]$ is sorted

Quicksort – Algorithm

Now have specification for *partition* – it permutes A in to three sections, $<$, $=$, and $>$ relative to pivot value p known to be initially present, and returns segment boundaries:

> assert: $n > 1$ **and** $p \in A[0 \ldots n-1]$
> $(fe, fg) \leftarrow partition(A, n, p)$
> assert: R

PS. The assertions provided here don't require that the elements in A get permuted, so *partition* could in fact "cheat", and assign complying values in to A.

Partition – Algorithm

define $P \equiv 0 \le fe \le next \le fg \le n$ **and** $A[0 \ldots fe-1] < p$ **and**
$A[fe \ldots next-1] = p$ **and** $A[fg \ldots n-1] > p$ **and**
$(p \in A[next \ldots fg-1]$ **or** $fe < next)$

Initialization:
$next, fe, fg \leftarrow 0, 0, n$
assert: P

Loop:
while $next < fg$
    assert: P **and** $next < fg$

Partition – Algorithm

if $A[next] < p$
  swap $A[fe]$ and $A[next]$
  $fe, next \leftarrow fe + 1, next + 1$
  assert: P
else if $A[next] > p$
  swap $A[next]$ and $A[fg - 1]$
  $fg \leftarrow fg - 1$
  assert: P
else
  $next \leftarrow next + 1$
  assert: P **and** $fe < next$

assert: P **and** $next \geq fg$ **and** $fe < next \implies R$

Partition – Analysis

To analyze execution time (and at the same time, to show termination), define $t() = fg - next$.

By the initial assignment, $t_0 = n$.

At every iteration, either *next* goes up by one, or *fg* decreases by one. Therefore, $t()$ decreases by $k = 1$ at every iteration.

By the loop guard, the loop ends when $t() = t_e = 0$.

Hence, the number of iterations is less than $(t_0 - t_e)/k = n$, and so $partition \in O(n)$.

Quicksort – Analysis

Hope for the best?

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + 2 \times T(\lfloor n/2 \rfloor) & \text{if } n > 1 \end{cases}$$

In this case, $T(n) \in O(n \log n)$. Wonderful...

(Actually, there is an even better situation, can you see it?)

Quicksort – Analysis

Plan for the worst!

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + T(0) + T(n-1) & \text{if } n > 1 \end{cases}$$

Now $T(n) \in O(n^2)$.

Awful...

Quicksort – Analysis

Is there something in between? Suppose the chosen pivot is equally likely to end up in any element in A. Then

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + \frac{2}{n}\sum_{i=0}^{n-1} T(i) & \text{if } n > 1 \end{cases}$$

Now what?

Quicksort – Analysis

Turns out (after doing maths) that average case of Quicksort is also $O(n \log n)$, with the exact solution to the recurrence yielding

$$T(n) \approx 1.44 n \log_2 n + O(n) .$$

But, what does average case really mean?

Quicksort – Analysis

Suppose $A[0 \ldots n-1]$ is a *random permutation* of final array. Then first item is also random. If so, can take:

$$p \leftarrow A[0]$$

This approach means that for the average case analysis to be valid, randomness is required to be present in the input data.

Quicksort – Analysis

But if there is any chance at all of input being already sorted or nearly sorted or nearly reverse sorted, taking first item as pivot is a high risk strategy.

Better idea: Instead, take

$$p \leftarrow A[(n-1)/2]$$

Harder now to say what kind of sequence will force worst-case behavior. (But there are still such sequences.)

Or, could even be more defensive, and take

$$p \leftarrow median(A[0], A[(n-1)/2], A[n-1])$$

Quicksort – Analysis

Brilliant idea: Instead, take

$$p \leftarrow A[random(0, n-1)]$$

Now the randomness required for the analysis is automatically supplied by the algorithm.

The expected running time is $O(n \log n)$, *regardless* of the particular input sequence.

Analysis of algorithms

Types of analysis:

1. Best case: of academic interest only, only fools make use of it.
2. Average case:
   - 2a. Randomness required in input: high risk unless input can be guaranteed to be randomized.
   - 2b. Randomness enforced by algorithm, regardless of input: robust, but still not for true safety-critical applications.
3. Worst case: bet your (and others') life on it.

So, a question to ponder: can sorting be done in $O(n \log n)$ time in the worst case?

Tattslotto

Quicksort versus Quickpick?

	Quicksort	Quickpick $20
Best case	$O(n)$	$1,000,000
Average case	$O(n \log n)$	$10
Worst case	$O(n^2)$	$0

In Quicksort, the average is close to the best. Desirable outcome, and worth the risk.

In Quickpick, the average is close to the worst. Smarter to spend your money on something else.

Chapter 12 – Program examples

- `binarysearch.c`
- `quicksort.c`
- `sortscaffold.c`

Chapter 12 – Exercises

Exercise 1

Suppose that the sorted array may contain duplicate items. Linear search will find the *first* match.

Modify the binary search algorithm so that it also identifies the first match if there are duplicates.

Modify the demonstration of correctness so that it matches your altered algorithm.

Chapter 12 – Exercises

Exercise 2

A sorting algorithm is stable if the original array order is preserved among items with equal sort key.

2a. Show that Quicksort is not stable.

2b. How might it be modified so that it becomes stable?

Chapter 12 – Exercises

Exercise 3

As shown, Quicksort requires two recursive calls.

But the final *tail recursion* can be replaced by inserting a while loop into the Quicksort function, leaving just one recursive call.

Implement this change, and test your implementation for correctness.

Exercise 4

A slightly different approach to *partition* is described by this more relaxed specification:

assert: $n > 1$ **and** $p \in A[0 \ldots n-1]$
$f \leftarrow partition(A, n, p)$
assert: $0 < f < n-1$ **and** $A[0 \ldots f-1] \leq p$ **and**
$A[f] = p$ **and** $A[f+1 \ldots n-1] \geq p$

4a. Design a function that meets this specification.

4b. Does this approach have any disadvantages or advantages compared to the first one presented?

Exercise 5

Suppose that the kth smallest item is to be identified in an unsorted array $A[0 \ldots n-1]$.

5a. Design and analyze an algorithm for this problem that requires $O(nk)$ time in the *worst* case.

5b. Design and analyze an algorithm for this problem that requires $O(n)$ time in the *average* case.

You may not modify the array.

Exercise 6

Suppose that $A[0 \ldots n-1]$ is a sorted array, and that the position of x is to be found.

Suppose further that, although x might appear anywhere, it is often expected to be located relatively early in the array.

Design and analyze an algorithm for this problem that requires $O(\log d)$ time, where d is the index in A at which x (or the first element larger than it) appears, that is, $A[d-1] < x \leq A[d]$.

Chapter 12 (Part I) – Summary

Key messages:

- Algorithm correctness can be demonstrated via logic
- $O()$ notation allows asymptotic costs to be compared
- Best case analysis, and average case analysis where randomness is required in the input, are risky
- Worst case analysis, and average case analysis where randomness arises in the algorithm, are much better.
- Quicksort requires $O(n \log n)$ time, on average, but is $O(n^2)$ in the worst case.