G52ADS
Quiz Revision

Craig Knott
November 23, 2012

Contents

1 Algorithm Analysis

An algorithm is a step-by-step procedure to solve a problem in a finite amount of time. Most algorithms transform input data to output data, and generally as the input size increasing, so does the run time. When analysing algorithms, we run it against a varying size of inputs, and record the execution time, this then gives us an measure of how the algorithm scales. Another way in which this can be done, independent of hardware and software, is to mark the number of primitive operations that take place throughout the algorithm (for instance, assigning a variable, indexing into an array, comparing two numbers, etc). An example of this is below, see how loops are counted n times.

Algorithm arrayMax$(A,\ n)$	
$currentMax \leftarrow A[0]$	2
$for\ i \leftarrow 1\ to\ n - 1\ do$	1
$if\ A[i] > currentMax\ then$	2(n-1)
$currentMax \leftarrow A[i]$	2(n-1) (worst case)
$\{increase\ counter,\ i++\}$	2(n-1)
$\{test\ counter,\ i \leq (n-1)\}$	2(n-1)
$return\ currentMax$	1

Which would equate to;
$= 4 + 4(2(n-1))$
$= 4 + 8(n-1)$
$= 4 + 8n - 8$
$= 8n - 4$

This means that the time taken for this algorithm to execute, $T(n)$, is bounded by constants a and b such that;
$a(8n - 4) \leq T(n) \leq b(8n - 4)$
Both of which are linear functions, meaning this function runs in linear time, $O(n)$

2 The big-Oh Family

Big-Oh and its counterparts are used to show how functions scale, and what they are bounded by. You can say that, given functions $f(n)$ and $g(n)$, and positive constants c and n_0;

$$f(n)\ is\ O(g(n)) \iff \exists\ c,\ n_0 \text{ such that } f(n) \leq c * g(n) \quad \forall\ n \geq n_0$$

For example, we can say $(4n + 5)$ is $O(3n - 6)$ when using positive constants $c = 1.5$ and $n_0 = 28$, since $(4n+5) \leq 1.5 * (3n-6) \quad \forall n \geq 28$. You must remember, however, to choose the c **before** n, as it is a constant **independent** of n. The method of working out an appropriate c, is to rearrange the functions so that it is in the form $c =$, and cancel the ns.

A few things worth remembering about big-Oh;
Exponentials, 2^n, grow faster than any power, n^k
Powers, n^k, grow faster than any log, $log\ n$
Logs, $log\ n$, grow faster than linear time, n
Linear time, n, grows faster than constant time, 1.

The in-laws
Big-Oh is not the only tool used for analysing algorithms, we also have access to Big-Ω, Big-Θ, and Little-oh, they are defined as follows;

Big-Ω
Given functions $f(n)$ and $g(n)$ we say that $f(n)$ is $\Omega(g(n))$ if there are positive constants c and n_0 such that

$f(n)\ is\ \Omega(g(n)) \iff \exists\ c,\ n_0$ such that $f(n)\ \geq\ c\ *\ g(n) \quad \forall\ n\ \geq\ n_0$
The function grows at least as fast as

Big-Θ
Given functions $f(n)$ and $g(n)$ we say that $f(n)$ is $\Theta(g(n))$ if there are positive constants c, c' and n_0 such that

$f(n)\ is\ \Theta(g(n)) \iff \exists\ c, c'\ and\ n_0$ such that $f(n) \leq c * g(n)\ and\ f(n) \geq c' * g(n) \quad \forall\ n\ \geq\ n_0$
The function grows exactly as fast as

Little-Oh
Given functions $f(n)$ and $g(n)$ we say that $f(n)$ is $o(g(n))$ if there are positive constants c and n_0 such that

$f(n)\ is\ o(g(n)) \iff \forall\ c \exists\ n_0$ such that $f(n) \leq c\ *\ g(n) \quad \forall\ n\ \geq\ n_0$
The function grows less fast than (notice that n_0 is allowed to depend on c, because we must consider all positive values of c).

3 ADTs and CDTs

ADTs are *Abstract Data Types*. They specify data stored, possible operations on the data, and error conditions associated with these operations. They do **not** specify the implementation itself - hence "abstract".

CDTs are *Concrete Data Types*, and are the implementation of a given ADT. The choice of CDT to implement an ADT will affect the run time, and space usage, so it is very important to get this correct.

Often, ADTs come with efficiency requirements, expressed in big-Oh notation, "cancel(order) method must be $O(1)$". It is a vital skill to think of the most efficient way to implement a method, e.g. what is the most efficient way to reverse a list? The answer is not to reverse the list at all, simply reverse the order that the elements are accessed.

3.1 "Narrow" v.s "Wide" ADTs

Narrow means that there is only a small set of methods to implement. These are less flexible to use, however they are more flexible to implement, and could be more efficient. Wide means that there is a larger set of methods to implement, which makes using it much flexible, but increases the difficulty of implementing it efficiently.

4 Stacks and Queues

4.1 Stacks

Stacks are an *Abstract data type* that stores references to objects. Inserting and deleting from the stack follow a "Last-in, first-out ($LIFO$)" scheme. The main operations stacks are to *push* elements, which inserts a new element, and to *pop* elements, which is to return the last inserted element. Some auxiliary methods include *top*, getting the first element without removing it, *size*, getting the size of the stack, and *isEmpty* indicating whether the stack is empty or not.

Applications of stacks include; page-visited history of a web browser and an undo sequence in a text editor.

```
public interface Stack {
  public void push(Object o);
  public Object pop() throws EmptyStackException;
  public Object top() throws EmptyStackException;
  public int size();
  public boolean isEmpty();
}
```

4.2 Implementation of a Stack as an Array

A simple way to implement a stack ADT is to use an array. Elements are added from left to right, and a variable keep track of the index of the top element. Arrays have the problem that they have limited size, and a $FullStackException$ will need to be implemented for when a *push* is called without adequate space in the array.

If n is the number of elements in the stack, the space of an array stack is $O(n)$ and each operation runs in time $O(1)$.

4.3 Queues

Queues are another *Abstract data type* that store arbitrary objects. Insertions and deletions follow a "First-in, first-out ($FIFO$)" scheme. Insertions are at the tail of the queue and removals are at the head. The main operations of a queue are *enqueue*, to insert an element at the **end** of the queue, *dequeue*, which removes and returns the element at the **front** of the queue. Some auxiliary operations include $front$, to return the first element without removing it, *size*, to retrieve the size of the queue, and *isEmpty* to check whether or not the queue has elements stored in it.

Applications of queues include; waiting lists and access to shared resources

```
public interface Queue {
  public int size();
  public boolean isEmpty();
  public Object front() throws EmptyQueueException;
  public void enqueue(Object o);
  public Object dequeue() throws EmptyQueueException;
}
```

4.4 Implementation of a Queue as an Array

Use an array of size N in a circular fashion. Two variables are used to keep track of the front, f, and rear r. The variable r is kept empty. An exception will be throw if there is an attempt to *enqueue* or an array if it is full or *dequeue* and empty array.

The question is, why would we just not use an array instead of the queue ADT? The answer is that there is conceptual clarity, self documentation, safety of coding, and potential compiler optimisation that are only present for the ADT.

5 Linked Lists

5.1 Singly Linked Lists

A singly linked list is a *concrete* data structure consisting of a sequence of nodes. Each node contains an "element" and a reference to the next Node in the sequence.

```
public Node {
  private Object element;
  private Node   nextNode;
}
```

Inserting to the head of the list consists of the following steps:

1. Creating a new node
2. Inserting this node to the head of the list
3. Change the head pointer to point at this new node
4. Having the new node point to the old head

As these operations have no dependence on the length of the list, the big-Oh of this is $O(1)$

Removing the head of the list consists of the following steps:

1. Set the head pointer to point at the second element in the list
2. Remove the first old head

As these operations have no dependence on the length of the list, the big-Oh of this is $O(1)$

5.2 Implementing a stack as a linked list

As we have seen above, inserting and removing from the head of a linked list is very efficient ($O(1)$). The top element of the stack would be stored at the first node of the list. The space used would be $O(n)$ and each operation of the stack ADT takes time $O(1)$. However, with a linked list, we would have no way to implement the $size()$ method that the abstract stack requires. However, we could simply take out the $size()$ method, and use $!isEmpty()$, or add a counter in the list implementation, which would have maintenance and access of $O(1)$.

5.3 Implementing a queue as a linked list

queues need to access both ends of the list, to do this efficiently, we need to store the head and tail values of the list. Insertion at the head, removal at the head and insertion at the tail would be $O(1)$, and removal at the tail would be $O(n)$. This is only if we changed how we use the linked list. Instead of placing the new elements at the start of the list, we place them at the end. The flow of the linked list is not necessarily the same as the flow of elements.

5.4 Linked lists v.s Array based CDTs

Arrays have the advantage of being contiguous in memory (as in, all the elements of an array are next to each other in memory). Whereas linked lists can be scattered around memory, and give poor usage of the cache. They also require the storage of the "next" reference, which doubles the space usage.

Linked lists have the advantage of begin able to grow and shrink automatically, whereas arrays have a fixed size, and might need a lot of unused space.

5.5 Doubly Linked Lists

A doubly linked list is akin to a singly linked list, containing an element, and a link to the next node, however it also contains a link to the *previous* node. They can also contain a header and trailer node that are used to mark the start and end of the list, both of which have no element value.

Inserting into a doubly linked list requires the following steps $addAfter(p,\ X)$

1. Find node p
2. Set the $nextNode$ of X to be $p + 1$
3. Set the $prevNode$ of X to be p
4. Set the previous node of $p + 1$ to be X
5. Set the $nextNode$ of p to be X

6 Vectors

Vectors are ADTs; the Abstract type corresponds to generalising the notion of an array. The index of an entry in an array can be thought of as the number of elements preceding it (E.g. in $A[2]$, you have two elements prior, $A[0]$ and $A[1]$). This number is called "rank". The Vector ADT stores a sequence of arbitrary objects. An element can be accessed, inserted or removed by specifying its rank (the number of elements preceding it). An exception should be thrown if an incorrect rank is specified (for example, a negative one).

The main vector operations are as follows;
object elemAtRank(integer r) - returns the element at rank r (without removing it)
object replaceAtRank(integer r, object o) - replacing the element at r with the new object o
insertAtRank(integer r, object o) - inserts a new element o into the vector, at position r
object removeAtRank(integer r) - removes and returns the element at rank r
Additional operations of *size*() and *isEmpty*() are generally implemented as well.

Applications of vectors include sorting a collection of objects (akin to a database).

6.1 Array-based Vector

Using an array V of size N as a CDT, with a variable n keeping a track of the size of the vector. The operation *elemAtRank*(r) is implemented in $O(1)$ time, simply by returning $V[r]$. When inserting (*insertAtRank*(r, o), we need to make room for the new element by shifting forward $n - r$ elements. In the worst case (when $r = 0$) this would take $O(n)$ time. When deleting (*removeAtRank*(r)), we need to fill in the hole left by the removed element by shifting backward the $n - r - 1$ elements. In the worse case of $r = 0$, this would take $O(n)$ time.

In the array based implementation of a vector, the space used by the data structure is $O(n)$. The methods *size*, *isEmpty*, *elemAtRank*, and *replaceAtRank* run in $O(1)$ time, *insertAtRank* and *removeAtRank* run in $O(n)$ time (but can run in $O(1)$ time if we use the array in a circular fashion and call them methods of element 0).

6.2 Growable array-based vector

When trying to *insertAtRank*(r) on a full array, instead of throwing an exception, we can replace the array with a larger one, the question is - how large? We could increase the array by a constant, or double the size. Comparing these two, we are looking to find out $T(n)$, which is the time taken to perform a series of n inserts. We call the *amortized* time of an insert the average time taken by an insert over the series of operations - $T(n)$ / n.

So, using the incremental strategy, we replace the array k, $\frac{n}{c}$ times. The total time of a series of n inserts is:
$= n + c + 2c + 3c + 4c + ... + kc$
$= n + c\,(1 + 2 + 3 + ... + k)$
$= n + ck\,(k + 1)/2$
This means, as c is a constant, $T(n)$ is $O(n + k^2)$ which is simply $O(n^2)$ - thus the amortized time is $O(n^2)$ / n which is $O(n)$.

Using the doubling strategy, (assuming n is a power of 2), we replace the array k, $log_2 n$ times. This total time is proportional to;
$= n + 1 + 2 + 4 + 8 + ... + 2(k-1)$
$= n + (2k-1)$
$= 2n-1$

Which would have time $T(n) = O(n)$, which has amortized time of $O(1)$. (To show that this is indeed $log_2 n$, see that $log_2 2 = 1$, $log_2 4 = 2$, $log_2 8 = 3$ etc).

Something to remember is that, to find the general geometric sum of something S, defined as $S = 1 + b + b2 + b3 + ... + bk$ is $S = ((bk+1)-1)/(b-1)$

7 Sorting

7.1 Bubble

The bubble sort involves two loops, the outer loop, which repeatedly scans through the array, and the inner loop, which, on each outer scan, compares each element with its immediate neighbour.

```
void bubbleSort (int arr[]){
  int i, j, temp;
  for(i = arr.length - 1; i > 0; i--){
    for(j = 0; j < i; j++){
      if(arr[j] > arr[j+1]){
      temp = arr[j];
      arr[j] = arr[j+1];
      arr[j+1] = temp;
      }
    }
  }
}
```

The bubble sort is *stable*, which means that any number that are equal will remain in the same order before and after the sorting.
The complexivity of the bubble sort can be summed to $c((n-1) + (n-2) + ... + 1)$, which is $O(n^2)$.

7.2 Selection

Similar to the bubble sort, on each scan, we keep a track of the largest element, and move this to the end at the end of each loop.

```
void selectionSort(int arr[]){
  int i, j, temp, pos_greatest;
  for( i = arr.length-1; i > 0; i--){
    pos_greatest = 0;
    for(j = 0; j <= i; j++){
      if( arr[j] >= arr[pos_greatest]){
        pos_greatest = j;
      }
    }//end inner for loop
    if ( i != pos_greatest ) {
      temp = arr[i];
      arr[i] = arr[pos_greatest];
      arr[pos_greatest] = temp; }
    }//end outer for loop
}//end selection sort
```

The selection sort also has $O(n^2)$.

7.3 Insertion

The idea behind the insertion sort is to keep the front of the list sort, and as we move through the back elements, we insert them into the correct place in the front.

```
void insertionSort(int arr[]){
  for(int j=1; j < arr.length; j++){
    int temp = arr[j];
    int i = j; // range 0 to j-1 is sorted
    while(i > 0 && arr[i-1] >= temp){
      arr[i] = arr[i-1];
      i--;
    }
    arr[i] = temp;
  } // end outer for loop
}  // end insertion sort
```

Insertion sort is $O(n^2)$, insertion sorting can only happen on doubly linked lists, and not their single counterpart.

8 Correctness of algorithms

How are we to be sure that our code is correct? An algorithm is correct if for any valid input, it produce the result required by the algorithm's specification. There are a few ways to check whether an algorithm is correct or not, and they are;

Correctness by design
This is the practice of construct the algorithm correctly in the first place, so that it has the desired properties. This can be done through test driven design, and theorem proving.

Testing
Testing is the practice of testing all possible inputs, against their outputs to see if they match. This is, however, a poor method of doing this, as it is almost always impossible to check all the inputs.

Model-Checking
This is widely used technique for hardware verification, and verification of concurrent process. This is the process of following an input through the system util arriving at the output.

Assertions
Formulate a property that has to hold throughout the algorithm. You should specify a precondition, which is what is expected to hold before the method is executing, and a postcondition which should hold after the method. This can be done using the assignment axiom.

Invariants
Assertions for loop are difficult, because loops may be executed many times, with slightly different assertions holding before and after each iteration. These assertion are true on each iteration through a loop. You need to make sure that you use appropriate invariants, as there are many obvious invariants that would not supply us with any addition information.

9 Big-Oh Cheat Sheet
Listed here are some useful big-Oh values to remember

Order of size $n! > 2^n > n^{>2} > n\ log\ n > n > log\ n > 1$
Space of an n **sized stack as an array** $O(n)$
Operation time on a stack as an array $O(1)$
Inserting at the head of a linked list $O(1)$
Removing the head of a linked list $O(1)$
Space used by a stack implemented as a linked list $O(n)$
Time of operations on the stack ADT $O(1)$
Maintenance and access of a counter in a stack linked list $O(1)$
Inserting at the head of a queue linked list $O(1)$
Inserting at the tail of a queue linked list $O(1)$
Removing at the head of a queue linked list $O(1)$
Removing at the tail of a queue linked list $O(n)$
Retrieving an element from an array based vector $O(1)$
Worst case insertion of an array based vector $O(n)$
Worst case deletion of an array based vector $O(n)$
Space used in an array based vector $O(n)$
Array based vector size, isEmpty, elemAtRank and replaceAtRank $O(1)$
Amortized time of incremental resize of an array based vector $O(n^2/n = O(n)$
Amortized time of double resize of an array based vector $O(n)/n = O(1)$
Complexity of bubble sort $O(n^2)$
Complexity of selection sort $O(n^2)$
Complexity of insertion sort $O(n^2)$