# DS & A Questions, supv 1

Matthew Johnson
mjj29@srcf.ucam.org

November 11, 2002

## 1 $O$ & $\Theta$ Notation

1. $\Theta(n^3)$

2. This is a badly phrased question. Exactly what do you mean?

3. $100n^2 \geq 2^n$ solved numerically:

| $n$ | $100n^2$ | $2^n$ |
|---|---|---|
| 1 | 100 | 2 |
| 2 | 400 | 4 |
| 3 | 900 | 8 |
| 4 | 1600 | 16 |
| 5 | 2500 | 32 |
| 6 | 3600 | 64 |
| 7 | 4900 | 128 |
| 8 | 6400 | 256 |
| 9 | 8100 | 512 |
| 10 | 10000 | 1024 |
| 11 | 12100 | 2048 |
| 12 | 14400 | 4096 |
| 13 | 16900 | 8192 |
| 14 | 19600 | 16384 |
| 15 | 22500 | 32768 |
| 16 | 25600 | 65536 |
| 17 | 28900 | 131072 |
| 18 | 32400 | 262144 |
| 19 | 36100 | 524288 |

The smallest value of $n$ at which $100n^2$ is faster than $2^n$ is 15

4. The Phrase "The running time of algorithm $A$ is at least $O(n^2)$" is content-free because it ignores constants, and hence the unit time which it is relative to is ignored. This is because we are only considering asymptotic time relative to other algorithms operating on the same data, and with the same unit time operations

5. True or False:

| | |
|---|---|
| $2^{n+1} = O(2^n)$ | *true* $(2^{n+1} = 2 * 2^n)$ |
| $2^{2n} = O(2^n)$ | *false* $(2^{2n} = 4^n)$ |
| $n^5 = O(\log n^{\log n})$ | *true*, for very large values of $N$, since $\log n$ will eventually be $> 5$, and exponentials grow faster than polynomial terms |
| $\log n^{\log n} = O(n^5)$ | *false*. See above |

# 2 Data Structures

## 2.1 Stacks & Queues

1. **2 Stacks in an array**
You can implement 2 stacks using an array, by having one grow down from the top of the array, and the other up from the bottom. You then store 2 pointers, to the active end of each stack. When these 2 meet the stacks are both full. Push & Pop operations are merely indexing the array by the pointer you hold for that stack, modifying one data item, and incrementing or decrementing the pointer as appropriate. These are both $O(1)$ operations.

```
var
   stack_array : array[1..MAXSTACKS] of StackType;
   stack1 : Integer = 1;
   stack2 : Integer = MAXSTACKS;

procedure push1(foo: StackType);
begin
   if stack1 = stack2 then
      Writeln("Out of stack space!");
   else begin
      stack_array[stack1] := foo;
      inc(stack1);
   end;
end;
```

```
function pop1: StackType;
begin
   if stack1 = 1 then
      Writeln("Empty Stack!");
   else begin
      dec(stack1);
      pop1 := stack_array[stack1];
   end;
end;

procedure push2(foo: StackType);
begin
   if stack1 = stack2 then
      Writeln("Out of stack space!");
   else begin
      stack_array[stack2] := foo;
      dec(stack2);
   end;
end;

function pop2: StackType;
begin
   if stack2 = MAXSTACKS then
      Writeln("Empty Stack!");
   else begin
      inc(stack2);
      pop1 := stack_array[stack2];
   end;
end;
```

2. A queue can be implemented using 2 stacks. Items are pushed onto one stack, and pop-ed from the second. When the second stack becomes empty, the first is reversed and copied into the second.

```
var
   pInS : ^Stack = New(Stack);
   pOutS : ^Stack = New(Stack);

procedure push(foo : StackType);
begin
   pInS^.push(foo);
end;
```

```
function pop : StackType;
begin
   if pOutS^.empty then
      if pInS^.empty then
         Writeln("Queue Empty!");
         Exit;
      else begin
         Dispose(pOutS);
         pOutS := pInS^.reverse;
         New(pInS);
      end;
   end;
   pop := pOutS^.pop;
end;

begin
   New(pInS);
   New(pOutS);
end;
```

The running time of this will be constant $O(1)$ for push & pop, except in the case that the stack has to be reversed. This is an $O(n)$ operation, but depending on the relative frequencies of push & pop operations, will be amortized over many operations.

3. I can't easily see how you could implement a stack using 2 queues, without using a really inefficient system, such as reversing queues on each pop operation - which is an $O(n)$ operation, and very inefficient.

4. Stack with push, pop & min operations.
   This can be implemented simply by keeping a separate variable that contains the smallest value. Push & pop operations compare their value with the minimum value, and update it if necessary. The min operation can the just return the contents of this variable.

```
var
   stack_array : array[1..MAXSTACK] of StackType;
   stackp : Integer = 1;
   minval : Integer;
```

```
procedure push(foo: StackType);
begin
   if stackp = MAXSTACK then
      Writeln("Out of stack space!");
   else begin
      stack_array[stackp] := foo;
      inc(stackp);
      if foo < minval then
         minval := foo;
   end;
end;

function min : StackType;
begin
   min := minval;
end;

function pop: StackType;
begin
   if stackp = 1 then
      Writeln("Empty Stack!");
   else begin
      dec(stackp);
      pop := stack_array[stackp];
   end;
end;
```

This code doesn't check if you remove the minimum value from the stack. When you do this you either have to search the stack for what is now the minimum value, which is an $O(n)$ operation, or you have to store a second stack that contains all the previous minimum values. When you pop the current minimum from the main stack, you also pop it from the stack of previous minimums. The value then at the top of the stack is the minimum remaining value.

## 2.2 Binary Search Trees

1. When traversing a binary tree, all nodes to the left of a given node should be less than it. In list (c), we get to node 911. This is greater than the 363 we are looking for. All nodes we look at after that should be $\leq 911$. Later on in the list there is the node 912, which is $> 911$, and hence the condition doesn't hold.

2. Sorting $n$ elements by building a binary search tree. This has 2 operations, building the tree, and walking over it. Walking the tree with be an $O(n)$ operation. In the best case, building the tree will require $\log n$ operations per insertion, and hence will be $O(n \log n)$, and in the worst case it will be $n$ operations per insert, and hence $O(n^2)$. Both these are larger than $O(n)$ for walking the tree, and hence that can be discarded in the asymptotic case. The time complexity is $O(n \log n)$ or $O(n^2)$ in the best & worse cases respectively

# 3 Sorting & Searching

1. Given an arbitrary sequence of numbers, determine if there are any repeats. This operation can definitely be done by sorting the numbers, and merely checking adjacent ones for duplicates. Since a sort can be performed in $O(n \log n)$ time (see above), then it is definitely possible in that time, although faster algorithms might exist, since the sort isn't necessary.

2. assuming the above were integers $x_i$ s.t. $1 \leq x_i \leq i$, then you know there is a lexicographical order over the keys. Hence a radix sort can be used to sort the numbers (which is $O(n)$), and as above this is sufficient to discover duplicate keys. Again, a faster algorithm may be possible

3. in this case take $x_i$ to be integers s.t. $1 \leq x_i \leq n$. The radix sort method still holds, and hence duplicates can be found in $O(n)$ time.

4. find $s_1, s_2 \in S$ s.t. $s_1 + s_2 = x$ in $O(n \log n)$
   I can do this in $O(n^2)$ trivially, don't know about $O(n \log n)$.

5. Again, it depends on the array. If you can search along the top row, and decide which column to search in, then you can do a binary search in both direction, which is $O(\log n)$. However, the condition given on the array isn't tight enough, so I'm not sure how you can do it in $< O(n^2)$.

6. *Find k in a strictly sorted vector.* Assuming that accesses into the vector are constant time operations, then you employ a binary search method, looking at the middle element, and then discard half the vector, depending on whether the middle element is larger or smaller than the number we're looking for. Since this is a constant time operation, repeated on half the array, we get $C(n) = k + C(\frac{n}{2})$. This is $O(\log n)$.