year11-algo

November 15, 2024

1 Algorithms and Data Structures for the IB

1.0.1 Covers topics 4.2 and 5

2 What will not be covered

- Collections (see JETS/textbook and topic 4.2.2)
- Pseudocode (see exam handout material, Python will be used instead)
- Basic programming skills (see topic 4.1)
- Java (see Option D material, implementations will be provided when relevant)

All material on the slides is explicitly assessed by the IB unless stated otherwise.

3 Directions of thought

1. Algorithm complexity analysis (4.2.8, 4.2.9)
2. Array algorithms revisited (4.2.1)
3. Advanced data structures (5.1.4 - 5.1.19)
    - 2D arrays
    - Stacks
    - Queues
    - Linked lists
    - Trees
4. Recursion and tracing (5.1.1 - 5.1.3)

The slides will be available on https://maleksware.github.io/stuff-place/y11-algo.

4 Complexity analysis

4.0.1 What the IB cares about

- Very basic knowledge of Big O
- Useful in IAs to cover Criterion C
- Identify/outline/state questions (associate complexities with algorithms)

4.0.2 Factors that affect code performance

- Skill of the programmer
- Hardware (caching/core performance/parallelism)
- Compiler efficiency

- Advanced data types (ADTs)
- *Complexity of underlying algorithm*
- *Size of input*

5 Big O notation

5.0.1 How do we measure code performance?

5.0.2 What about algorithms?

- Timing works for particular hardware/implementation, bad for algorithms
- Number of operations is better but hard to measure (and different operations take a different amount of time)

6 Scaling

Big O measures how the number of operations (and time to execute the algorithm implementation) **scales** with the size of input in the **worst case**.

n indicates a measurable size of input (length of an array that is put in the algorithm, number of vertices in the graph, etc).

The expression inside of big O indicates what the execution time is proportional to (the trend towards very big input sizes).

$O(n)$ means that the execution time grows linearly when the input size is increased. If the input size is doubled, the execution time is doubled as well (because the output time is proportional to n).

What about $O(n^2)$?

$O(n^2)$ means that if the input is doubled, the number of operations quadruples.

7 Other common complexities

- $O(1)$ - *constant time operations*. This means their execution time does not depend on the input size. Accessing an array (not LinkedList) element or performing arithmetic counts as such
- $O(n)$ - linear. Examples - removing an array element or calculating the mean of a number array.
- $O(n^2)$ - quadratic. Examples - insertion/selection sort, iterating over pairs.
- $O(\log n)$ - logarithmic (base 2). Example - binary search in a sorted array.
- $O(n \log n)$ - complexity of most Divide and Conquer algorithms (merge sort, quick sort). **Not assessed.** Not knowing what Divide and Conquer means is fine.

For all complexities, state what happens to the number of iterations (or operations) if the size of input is quadrupled.

8 Time complexity vs space complexity

All complexities can also relate to the memory blueprint (number of primitively typed entities stored as opposed to the number of simple operations). The TC and SC are not always the same: consider making n calculations in a loop, the memory used is $O(1)$ but the TC is $O(n)$.

9 Determining algorithm complexities

Get ready to propose ideas and make mistakes - some of the examples are tricky!

Identifying complexities from code is **not assessed** but extremely helpful to be able to do.

Remember the following tips:

1. Arithmetic operators, accessing array elements and printing (small) data are $O(1)$.
2. Powers of n come from nested loops with n.
3. Logarithms are unlikely to show up in the exam at all. But if they do, look for bisections and binary search.

In the following examples, consider n to be the length of the iterable (list, array or string) if it's not explicitly stated.

[ ]:
```
# Example 1

s = input()

print("Hello World!")
a = 1
b = 2
```

$O(1)$ (if you don't account for the time it takes to execute the data input, in which case it's $O(n)$ where n is the buffer size).

[1]:
```
# Example 2

s = [int(i) for i in input().split()] # reads an array of space-separated␣
 ↪integers

count = len(s) # Bonus marks if you state the complexity of this function
element_sum = 0

for i in s:
    element_sum += i

print(element_sum / count)
```

```
1 2 3
2.0
```

$O(n)$. A single loop over n elements.

[ ]:
```python
# Example 3

s = [int(i) for i in input().split()]

inversions = 0

for i in range(len(s)):
    for j in range(len(s)):
        if i < j and s[i] > s[j]:
            inversions += 1

print(inversions)
```

$O(n^2)$. Each of n iterations of the first loop results in n iterations of the second, getting us $n \cdot n = O(n^2)$.

[ ]:
```python
# Example 4

s = input() # reads a string

for i in range(42):
    for j in s:
        # iterates over all characters of s
        if j == "w":
            print("W")
```

$O(n)$. Remember that big O is a measure of *scaling*, meaning that **all constants are omitted**. We only care about the general law of scaling, not its particular aspects such as constants.

Well… kinda. The rule to ignore constants comes from the definition of big O which accounts for all constant factors.

[ ]:
```python
# Example 5

s = [int(i) for i in input().split()]

inversions = 0

for i in range(len(s) - 1):
    for j in range(i + 1, len(s)):
        if s[i] > s[j]:
            inversions += 1

print(inversions)
```

Left as an exercise for the reader

Just kidding, this one is also $O(n^2)$ - future Alex, please show this on the board

[29]:
```python
# Example 6

s = [int(i) for i in input().split()]
el_count = 0

for i in range(len(s)):
    for j in range(len(s)):
        for k in range(len(s)):
            el_count += (s[i] + s[j] + s[k])

for i in range(len(s)):
    for j in range(len(s)):
        el_count -= (s[i] + s[j])

print(el_count)
```

```
1 2 3
126
```

$O(n^3)$. Yes, it is $O(n^3 + n^2)$, but for very large n the n^2 term becomes insignificant (remember that we care about the trend towards large inputs).

However, there is nothing wrong in including all terms particularly if it's not obvious which one "wins".

NB: Look for features in the code that need to examine all elements - for example, using `sum(s)` to count the sum of elements introduces a $O(n)$ factor, even though we can't see a loop.

10 Array algorithms revisited

The IB only requires you to know 4 of them:

1. Linear search
2. Bubble sort
3. Selection sort
4. Binary search

11 Linear search

- Time complexity: $O(n)$
- Used in all arrays
- Easy to implement
- Relatively slow

[33]:
```python
s = [1, 4, 3, 2, 8, 7]

for i in range(len(s)):
    if s[i] == 3:
        print(f"Found at index {i}")
```

```
Found at index 2
```

11.0.1 Improving performance

Early exit: if the element is found, quit the loop.

NB: Do **not** use `break` in Paper 1. Use a `while` loop with a flag instead:

[36]:
```
s = [1, 4, 3, 2, 8, 3]

found = False
index = 0

while index < len(s) and not found:
    if s[index] == 3:
        print(f"Found at index {index}")
        found = True
    index += 1
```

```
Found at index 2
```

Bonus question: what is the time complexity *with early exit*?

Still $O(n)$. Early exits are a *heuristic*, meaning they improve practical performance in some cases without changing the worst case time complexity.

12 Bubble sort

- Time complexity: $O(n^2)$
- Slow but famous

n iterations, on each iteration adjacent elements are compared starting from the first one and swapped if needed.

Implementation details: if a `swap` function is provided, use it, otherwise swap with a temporary variable.

[44]:
```
s = [2, 3, 8, 1, -2, 4, -3, 7, 0, 12]
num_of_iterations = 0

for iteration in range(len(s)):
    for i in range(len(s) - 1):
        num_of_iterations += 1
        if s[i] > s[i + 1]:
            tmp = s[i]
            s[i] = s[i + 1]
            s[i + 1] = tmp

print(s)
print(num_of_iterations)
```

```
[-3, -2, 0, 1, 2, 3, 4, 7, 8, 12]
90
```

13 Improving performance

NB: remember these 2 points for the exam! You can get 4 marks with those.

1. Reducing the number of iterations

Note that the last `iteration` elements of the array are fixed. Let's not look at them at all!

Again, this does not change time complexity (remember example 5)

[42]:
```
s = [2, 3, 8, 1, -2, 4, -3, 7, 0, 12]
num_of_iterations = 0

for iteration in range(len(s)):
    for i in range(len(s) - 1 - iteration):
        num_of_iterations += 1
        if s[i] > s[i + 1]:
            tmp = s[i]
            s[i] = s[i + 1]
            s[i + 1] = tmp

print(s)
print(num_of_iterations)
```

```
[-3, -2, 0, 1, 2, 3, 4, 7, 8, 12]
45
```

2. Early exit if no swaps are made

If a list is almost sorted, this will make the algorithm terminate more quickly. Still won't help in the worst case though!

[46]:
```
s = [2, 3, 8, 1, -2, 4, -3, 7, 0, 12]
num_of_iterations = 0
swap_made = True
iteration = 0

while iteration < len(s) and swap_made:
    swap_made = False
    for i in range(len(s) - 1 - iteration):
        num_of_iterations += 1
        if s[i] > s[i + 1]:
            swap_made = True
            tmp = s[i]
            s[i] = s[i + 1]
            s[i + 1] = tmp
    iteration += 1
```

```
print(s)
print(num_of_iterations)
```

```
[-3, -2, 0, 1, 2, 3, 4, 7, 8, 12]
42
```

14 Selection sort

- Time complexity: $O(n^2)$
- Also slow and famous

A *sorted prefix* is maintained. On each stage, the smallest element in the unsorted part is found and moved to the end of the sorted part, extending it by 1.

The following implementation is the easiest to remember:

[49]:
```
s = [2, 3, 8, 1, -2, 4]

for i in range(len(s)): # the length of the sorted part
    min_value = 1_000_000 # just a big number representing infinity
    min_index = 0
    for j in range(i, len(s)):
        if s[j] < min_value:
            min_value = s[j]
            min_index = j

    tmp = s[i]
    s[i] = s[min_index]
    s[min_index] = tmp

print(s)
```

```
[-2, 1, 2, 3, 4, 8]
```

15 Bubble vs selection

Exam type of question: distinguish between the two

- Bubble can terminate early
- Bubble works better on arrays that are almost sorted
- Selection does less swaps (one per iteration)
- Both work in $O(n^2)$

16 Binary search

- Only on sorted arrays
- $O(\log n)$
- Gotta remember how it's written

- Fast

17 How it works

Easy to think about in terms of a *monotonous function* (see the board).

1. Establish a range where you expect the answer to be
2. Pick its midpoint
3. If you've overshot it, everything past it is redundant - shrink
4. Shrink the other side otherwise
5. Repeat until you hit what you need

There are 2 understandable implementations: 1. The good one with educational value 2. The IB one

And 2 common approaches: 1. Iterative 2. Recursive

Both can be on exams.

The most common thing on the exam is to use binary search to locate an element in a sorted array. Note the equivalence with the "function" way of thinking about it.

[63]:
```
# Good implementation
s = [1, 2, 5, 8, 18, 42, 228]

k = 300 # want to get the index at which this element would've been inserted␣
 ↪into s

# Invariant: s[l] < k, s[r] >= k

l = -1 # set both to out of bounds
r = len(s)

while r - l > 1: # end when they touch, marking the critical condition
    mid = (r + l) // 2
    if s[mid] < k:
        l = mid
    else:
        r = mid

print(r) # where is the answer?
```

```
7
```

[61]:
```
# IB implementation (USE THIS ONE WHEN YOU SIT THE EXAM!!!)

s = [1, 2, 5, 8, 18, 42, 228]

k = 18
```

```
l = 0 # in bounds!
r = len(s) - 1

found = False
index = -1

while l <= r and not found: # end when they overlap so we can detect if we␣
 ↪found the element
    mid = (r + l) // 2
    if s[mid] == k:
        found = True
        index = mid
    elif s[mid] < k:
        l = mid + 1
    else:
        r = mid - 1

if found:
    print(index)
else:
    print("Not found")
```

```
4
```

18 Recursive implementation

- Never seen on exams before, but was on Y12 trials
- Do the same thing, pass the boundaries as function parameters and return the result to the parent caller function
- If you don't know what this slide is about don't worry (recursion is coming later)

19 Analysing complexity

Where is the log factor coming from?

What happens if the input size is doubled?

20 Advanced data types (ADTs)

Also known as data structures (not to be confused with collections)

20.0.1 2D arrays (matrices)

Simply put, an array of arrays.

Access elements by row index, then by column index:

[6]:
```python
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
    [10, 11, 12]
]

n = 4
m = 3

for i in range(n):
    for j in range(m):
        print(matrix[i][j])
    print()
```

```
1
2
3

4
5
6

7
8
9

10
11
12
```

21 Bonus content

1. State the time complexity of each snippet
2. State which one will run faster in practice

21.0.1 Snippet 1

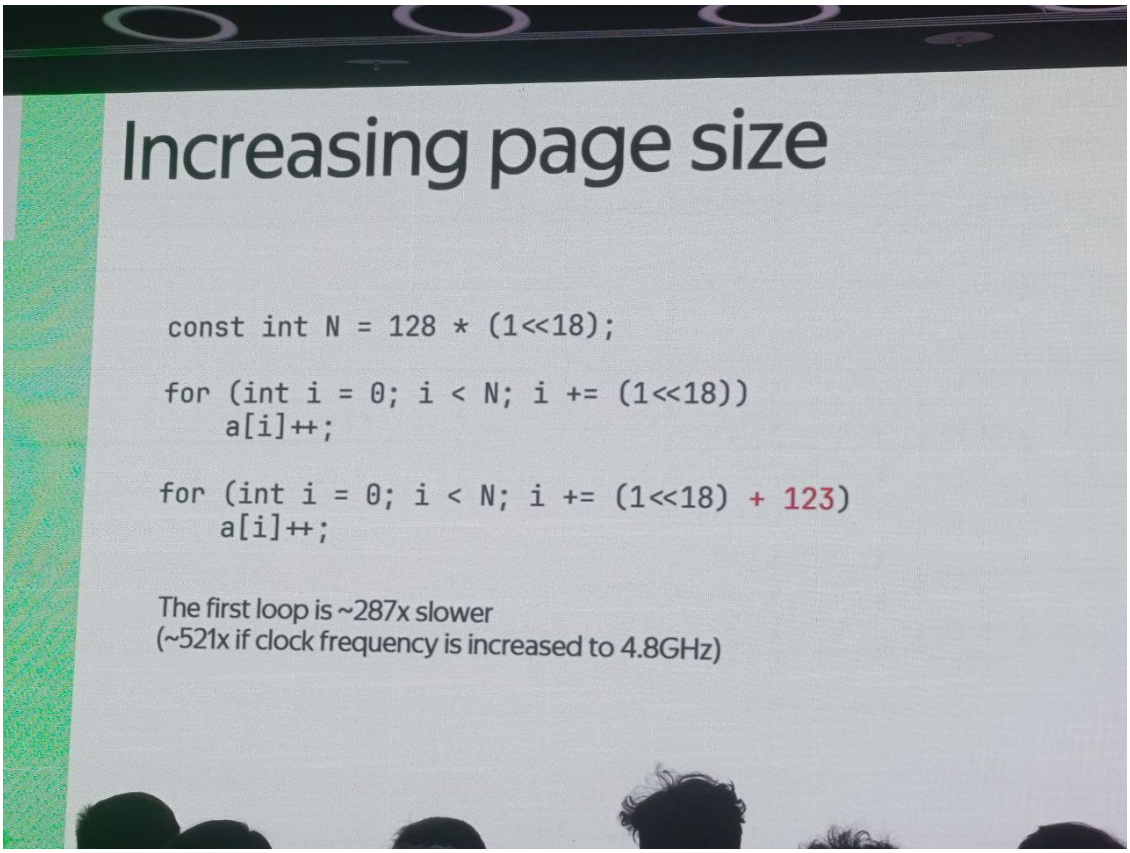
```cpp
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        element_sum += s[i][j];
    }
}
```

21.0.2 Snippet 2

```
for (int j = 0; j < m; j++) {
    for (int i = 0; i < n; i++) {
        element_sum += s[i][j];
    }
}
```

$O(n)$ for both but Snippet 2 is ~150-300x slower on big data (measured on C++) because of cache misses

22 More magic (offtop)



23 Dynamic and static data structures

23.0.1 Static

- Arrays (both linear and 2D), stacks and queues implemented on top of them
- Have a fixed amount of memory: a lot of memory has to be allocated on the start of the code even if real data is small
- Faster element access because memory locations are fixed
- Contiguous elements are faster to access (cache lines)
- More predictable to work with (can contain information such as length in the header)

- The relationship between data elements stays the same
- Simpler to implement

23.0.2 Dynamic

- Linked lists, trees
- Can expand based on runtime data
- Slower element access because memory is allocated at runtime (often don't offer access by index)
- Non-contiguous data slows access even more
- Vary in size requiring additional mechanisms to know the size etc
- The relationship between elements can change
- Harder to implement

24 Stacks

- LIFO
- Like a stack of plates (pun intended)
- Methods:
  - `push(element)`: adds an element to the stack
  - `pop()`: removes and returns an element from the top of the stack
  - `isEmpty()`: `true` if empty, `false` otherwise

24.0.1 For the exam

- Application of stacks: parsing, call stack (for functions), recursion, depth first search (DFS), undo/redo
- You will be explicitly told to use a stack in Paper 1 implementations
- Downsides of the stack: unability to read/remove from the middle, need to go over all elements and remove them to linearly search
- Use common sense to deduce advantages/disadvantages (why is a stack bad as a restaurant order processing ADT?)

25 Implementing a stack using a static array

Ideas?

[11]:
```
stack = [-1] * 100 # "static" array xD
top_ptr = -1 # topmost element index

def isEmpty():
    if top_ptr == -1:
        return True
    else:
        return False

def push(x):
    global top_ptr
```

```
    top_ptr += 1
    stack[top_ptr] = x

def pop():
    global top_ptr
    element = stack[top_ptr]
    top_ptr -= 1
    return element

push(1)
push(2)
print(pop())
print(stack)

push(42)
print(isEmpty())

pop()
pop()
print(isEmpty())
print(stack)
```

```
2
False
True
```

25.0.1 Framing this into words for the exam

1. A static array is created of sufficient size
2. A variable to keep track of the topmost element index is initialised
3. For a `push()`: the variable is incremented, then the element is assigned to the respective index
4. For a `pop()`: the element is read from the index signified by the variable, the variable is decreased and the element is returned
5. `isEmpty` compares the pointer variable to the first assigned value and returns the respective status

Use common sense as needed.

26 Queues

- FIFO
- One end in which to add and the other end from which to take
- Methods:
  - `enqueue(x)`: add the element to the **back** of the queue
  - `dequeue()`: remove and return the element from the **front** of the queue
  - `isEmpty()`
  - `peek()`: `dequeue()` without removal (why is it present here but not in the stacks?)

26.0.1 For the exam

- Application of queues: print queue, processor job queue, buffers, server request queue, breadth-first search (BFS)
- Use the methods described here (I probably lost marks on Paper 1 because I forgot their names)
- Benefits of the queue: it keeps the order, allows to process requests on first-to-come basis
- Downsides of the queue: hard linear search, can't randomly access elements, takes up more space than needed if implemented on an array

Remember that although the methods' names can be read as `nq` and `dq`, both of them have 7 letters each. Play it safe kids, you got plenty o'time.

27 Implementing a queue using a static array

Ideas?

Better ideas? If it's obvious how to make it on an array, think about how to use the least amount of memory.

[28]:
```
queue = [-1] * 10
front = -1 # points to the first element
back = 0 # points to the last element
```

[29]:
```
def enqueue(x):
    global front
    front += 1
    queue[front] = x

def dequeue():
    global back
    element = queue[back]
    back += 1
    return element

def peek():
    return queue[back]
```

[31]:
```
def isEmpty():
    if front + 1 == back:
        return True
    else:
        return False
```

[36]:
```
# reset for clarity and repeatability (you don't have to do that)
queue = [-1] * 10
front = -1
back = 0
```

```
enqueue(32)
enqueue(42)
print(front, back)
print(peek())
print(dequeue())
print(isEmpty())

enqueue(20)
dequeue()

print(dequeue())
print(isEmpty())

print(queue)
```

```
1 0
32
32
False
20
True
[32, 42, 20, -1, -1, -1, -1, -1, -1, -1]
```

28 Test your understanding

State the time complexities of all methods of 2D arrays, stacks and queues.

A thing to think about: how do we make a much better and more efficient queue in terms of memory efficiency? Constant time complexity per operation must be retained.

This is more of a maths/algorithmic thinking question, and it is by far the hardest in these slides.

29 Linked lists

- Nodes scattered around memory and pointing to each other in order
- Can be singly linked, doubly linked and circular
- Are a dynamic data structure
- Don't allow access by index (slow search)
- Easy to add, remove or insert elements ($O(1)$ per operation)
- Methods are **not explicitly assessed**, refer to JETS for Java syntax

29.0.1 For the exam

- Application of linked lists: mutable arrays, moves in the game (chess) where you don't have to jump between non-consecutive moves that often. **Should not be assessed but was asked in N24 exams**, be prepared to make stuff up.
- Understand the structure of each (see the board)
- Remember the `head` and `tail` pointers

- Check that you can describe insertion/deletion operations (see the board)

30 Trees

- Arguably the best thing IB CompSci has
- Can be seen as a linked list with branching (and without merging)
- Are a dynamic data structure
- Have no single order of processing (wait for it)
- Methods are **not assessed** but you are required to have a visual understanding of them

30.0.1 Elements of a tree

- Nodes (vertices)
- Pointers to child nodes from parent nodes
- A single root node (**IB-specific!**)
- Leaves (nodes without children)
- Subtrees

30.0.2 Binary trees

- Each node has at most 2 children (left and right, order matters)
- Are extremely useful for storing ordered data
- The left subtree has elements smaller than the parent node, the right one has the larger ones
- You traverse and pick the path you want

Exam tip: when asked to add the items to the binary tree, do it in order in which they appear, follow the smaller/greater path and just make sure that each of them ends up as a leaf on the current step.

30.0.3 Tree traversals

1. Preorder (root -> left st -> right st)
2. Inorder (left st -> root -> right st)
3. Postorder (left st -> right st -> root)

Collecting flags!

Question: which one outputs the elements of the tree in ascending order?

Question: how would you implement those traversals?

31 Recursion

Definition: situation when a function calls itself.

Useful for traversals and problem decomposition.

Adapted quote: "[recursion] is when we have a big problem that we don't know how to solve, and we break it down into smaller problems that we also don't know how to solve"

Do we believe that $7! \equiv (7 \cdot 6!)$?

[69]:
```
def factorial(x):
    if x <= 0:
        return 1
    else:
        return x * factorial(x - 1)

print(factorial(5))
```

```
120
```

31.0.1 Constructing a recursive function

1. Parameters
2. Base case (base condition) - do by hand
3. Recursive case (reduce to some simpler cases)

What about Fibonacci numbers?

[70]:
```
def fib(x): # fib(0) = 1, fib(1) = 1, fib(2) = 2
    if x < 2:
        return 1
    else:
        return fib(x - 1) + fib(x - 2)

for i in range(10):
    print(fib(i))
```

```
1
1
2
3
5
8
13
21
34
55
```

31.0.2 Downsides of recursion

- Slower (function calls take time)
- Uses more memory on the stack than iterative approaches
- Needs memoisation to work efficiently
- Hard to debug

31.0.3 Benefits of recursion

- More intuitive and abstract
- Easy to implement
- Elegant

- Uses an implicit stack (function call stack), hiding complexity from the programmer

[ ]:
```python
# State the output of this algorithm, showing all your working:

def f(x):
    if x % 3 == 0:
        return 0
    else:
        return f(x - 1) + 1

input() # ignore this line
print(f(5))
```

[ ]:
```python
# State the output of this algorithm, showing all your working:

def f(x):
    if x < 3:
        return x
    return f(x - 2) + x + f(x - 3)

input() # ignore this line
print(f(6))
```

[ ]:
```python
# State the output of this algorithm, showing all your working:
# Bonus: figure out why this is useful

def f(a, b):
    if a % b == 0:
        return b
    return f(b, a % b)

input() # ignore this line
print(f(20, 12))
```

32 Traversing trees recursively (inorder)

1. Start at the root node
2. If the node is a leaf, add it to the output and return
3. Call the function from the left subtree
4. Add the current node to the output
5. Call the function from the right subtree
6. Return