

Here is a set of questions and possible answers for your reference.

1. *Why is it that we have to do: $hi - (hi-lo)/2$ instead of $(lo+hi)/2$ to compute the mid point*
 - a. **Answer:** Because $(lo+hi)$ can overflow if the sum becomes greater than the `max_int`. But subtraction can never overflow.
2. *Why is it that we double the array instead of incrementing by 1,2,3 etc as needed? Isn't that a waste of memory?*
 - a. **Answer:** We can prove mathematically, when you double the array, the cost per operation is constant time. In this case, worst case (copying the old array) happens at powers of 2, and they are sufficiently spread out that it does not affect the constant cost. Sure you are wasting a bit of memory, but it is worth for the better performance
3. *Why do we try to make lower bound higher when trying to find an optimal solution?*
 - a. **Answer:** there can be lots of lower bounds for an algorithm. One may observe the lower bound is constant for a class of input. But that does not mean that it is the best lower bound you can find. There may be a higher lower bound that can help us get to the optimal solution. Similarly, you need to lower the upper bound. Both are very difficult algorithmic challenges in general. But this is what algorithm researchers (like Bob Tarjan, a Princeton faculty who won the Turing award for Computer Science) do.
4. *When counting memory should you ignore the constant terms?*
 - a. **Answer:** yes, in the tilde notation analysis, you can ignore all constants. For example, if you come up with a sum : $32N + 48$ you can just say this is $\sim 32N$ (because those constants are negligible as N gets larger)
5. *What is the difference between a reference and a memory?*
 - a. **Answer:** A reference is a "name" that points to some memory location. All java references take 8 bytes of memory. A memory is allocated when you can `new()`. For example: `Node ptr = new Node();` statement, the `ptr` is just a reference that takes 8 bytes. But when you say `new`, compiler can allocate enough memory to hold a `Node` type (for example: 48 bytes etc). The equal sign in the statement indicates that you would like `ptr` to take the address (shallow memory) of the block of memory (deep memory) allocated for the node.
6. *Why is it that we just use linked lists instead of arrays when implementing data structures like stacks?*
 - a. **Answer:** sure you can use linked lists. But there are some disadvantages such as additional memory for links and complexity of accessing random elements in a list. On the other hand arrays are straight forward and random access is constant time. Arrays have to be resized but linked lists do not, but still it is best to use arrays when possible.
7. *Why do we need to learn resizable arrays? Java contains something called an ArrayList.*
 - a. **Answer:** Yes, but the internal implementation of `ArrayList` is really a linked list. So `ArrayList` is a disguised linked list.

8. *Why do we need iterators?*

- a. **Answer:** When you implement a generic collection like a Stack, you need to provide a way for the client to access elements in the collection. It is best, not to expose the internal data structures of your collection. With iterators, client don't need to know anything about that and that is the way clients like it. They just want the API, so they can solve their problem using your library.

9. *What is the difference between Iterator and Iterable?*

- a. **Answer:** They are both interfaces. Generic classes can implement the Iterable interface and provide a method called iterator() that returns an iterable object. The client can then write simple code using statements like : for (Item it : Collection) to iterate through the items in the collection. The iterator() method must returns an iterator object and it is implemented as an inner class.
- b. **Answer:** If the collection implements Iterator interface, then it must at least provide, next() and hasNext() to the client to iterate through the collection. A bit more complicated for the client than Iterable interface.
- c. See the provided sample files for some examples of implementation

10. *What math formulas do I need to know in this class?*

- a. **Answer:** Minimally, the arithmetic sum and geometric sum. You can write this on a cheat sheet and bring to exams.
- b. **Answer:** You also need to know that discrete sums can be approximated using continuous integrals. For example, if you need to find the sum: $\sum f(n)$ then you can find an upper bound to this using $\int f(n) dn$ which is easier to find. For example, find an upper bound for the discrete sum: $\sum n^{1/2}$ using the integral $\int n^{1/2}$

11. *How do I know if $\lg n$ or $n^{1/3}$ grows faster/slower?*

- a. **Answer:** The best way to do this is to take the limit (as n goes to infinity) of $\lg n / n^{1/3}$. Use a rule like L'hopitals rule to find this limit. See if it goes to 0 or infinity. Make your own conclusion then.

12. *Java arrays take extra 24 bytes in addition to the size of stored data. Why?*

- a. **Answer:** There can be many reasons for Java using this memory, such as garbage collection information and keeping the length of the array as an attribute. It is an internal overhead and does not matter much in the overall analysis of program performance.

13. *What are differences between, tilde, big_O, big_Omega and big_theta?*

- a. **Answer:** Tilde gives an approximation for the runtime using only the most dominant power of N and some factor. Eg: $\sim 5N$. The big_O is an upper bound. So technically $\sim 5N$ is $O(N)$, $O(N \lg N)$, $O(N^2)$ etc. The big_omega is a lower bound. That is $\sim 5N$ can be $\Omega(1)$ or $\Omega(N)$ etc. The big_theta is the tight bound. For example, we showed that finding the max in an unsorted array is $O(n)$ and $\Omega(N)$ and therefore, it is $\Theta(N)$

14. *How do I find some estimates for the complexity of the algorithm using just the runtime data?*

- a. **Answer:** This is one of the benefits of running your algorithm with many data sets. You can start with $N=10$ (say) and then double the input and keep running the algorithm to find run time $T(N)$ for input of size N . Then you assume a model: $T(N) = aN^b$ and use the logarithm to plot the line: $\lg(T(N)) = \lg a + b \lg(N)$ to find a and b . The coefficient b informs us the complexity of the algorithm ($b=0$ is constant time, $b=1$ is linear etc)
- b. **Answer:** You also need to pick samples that provide reasonably large n and times > 1 second. You can keep max time to about 5 seconds. But more you can get the better the approximation is. That is, for large N and longer runtimes, all the other overhead of running the program as discounted or negligible.