CMPSCI 630 Systems **Fall 2019**

Lecture 5

Lecturer: Emery Berger *Scribe: Mayank Jha, Vincent Pun*

We started with a quick review of the first-fit and best-fit memory allocation schemes. The first-fit scheme is potentially linear and can waste space. The best-fit scheme is costly because it iterates through the whole list to find the smallest piece that fits. This led us to the topic of Pythons own memory allocator.

5.1 Python's Memory Allocator

Python uses a special custom memory allocator. Python maintains an array of free list pointers, pointing to free lists, each managing memory of sizes in multiples of 8. Objects are rounded up to the next multiple of 8 till 256. Requests of 1 byte would go into the 8-byte free list, and requests of 252 bytes would go into the 256-byte free list, for example.

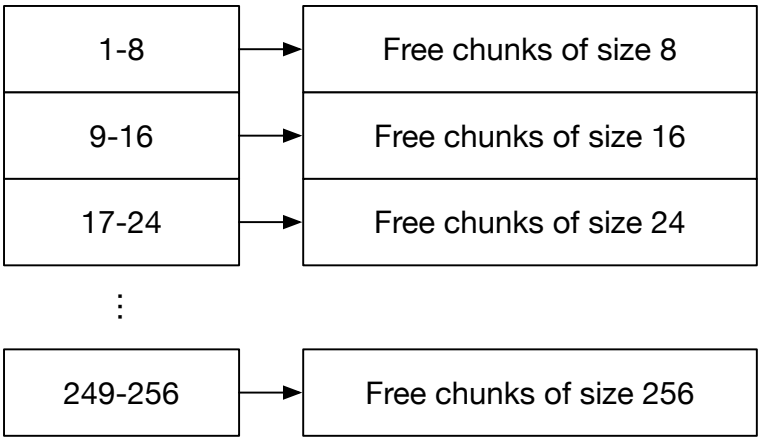

Figure 5.1: A visualization of the free lists in Python.

Internally, Python does this:

```
if (size <= 256) {
  // use small objs
} else {
  // use malloc
}
```

Python's data types are very big and space-inefficient. For example, an integer is 24 bytes. A one-character string is 50 bytes, in stark contrast to C which is 2 bytes. Empty dictionaries in Python take up 240 bytes.

5.1.1 Mechanics

The mechanism is a linked list implementation of stacks. Using a request for a 48-byte object as an example:

1. The allocator goes to the 6th free list (48 bytes).
2. If the free list is empty, it gets a chunk of memory from malloc and carves it into 48-byte chunks.
3. An object is popped from the list.
4. When free is called, that object is pushed back to the free list.

Size classes are segregated in Python. If a 48-byte object is freed and is put back on the free list, it cannot be reused for requests for other sized-objects.

A problem can occur in the following situation:

1. 1 MB of 8-byte objects are allocated.
2. All of them are freed.
3. They all go back to the 8-byte free list. 1 MB of free memory is around.
4. 1 MB of 16-byte objects are allocated.
5. All of them are freed.
6. The same goes on until there have been, say, 32 of such requests.

In this case, actual memory usage will keep increasing but no memory is in use. Although the peak memory usage is 1 MB, the actual memory consumption is actually 32 MB—a 32x increase in memory!

5.1.2 Notes on Python Lists

Lists are implemented internally as vectors. Lists grow by doubling the size if the list is full and no longer fits new items. This is an instance of the *ski rental problem*.

5.1.2.1 Ski Rental Problem

The ski rental problem is a problem where a decision to buy or a decision to rent is to be made. Using an example where renting a ski costs $50, and buying a ski costs $500. Assuming the skier rents in the beginning. At which point should the skier buy instead of renting?

Figure 5.2 shows the cumulative cost of renting ski and buying a ski over the number of times the skier skis. If the skier buys a ski at the point where the cost of renting skis and buying a ski meets (i.e. renting 10 times), the skier will never have to lose money as compared to continuing to rent after the point. The skier will not have to pay more than twice the amount of buying a ski.

This strategy amortizes the cost of doubling, and is the optimal strategy in an environment with no information.

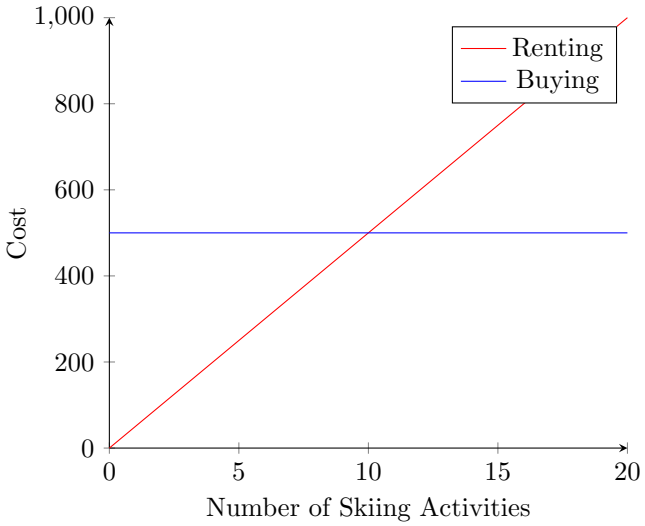

Figure 5.2: The cost of renting skiis versus buying a ski over the number of skiing activities.

5.2 Garbage Collection

The most common garbage collection algorithm is mark-and-sweep. Reference counting is a common technique where the reference to a memory location is marked, and it cannot be removed until the count is zero. However this technique is incomplete, as the cases where we have self-referential pointers. So, there might still be certain sections of the memory which remain unclaimed after the garbage collection.

- Python, LISP, Java, PHP, JavaScript are some of the modern day computer languages which are garbage collected.
- Pythons GC is a reference counting GC, however a real better GC can be imported into it, using the `gc` module.

5.2.1 Mark-and-Sweep Algorithm

Initially, all objects have a color bit set to white. All objects found by the collector are marked black. The remaining white objects are then considered unreachable and thus suitable for collection.

Essentially, this builds a transitive closure of the graph from the roots.

5.2.2 Generational Garbage Collection

This is a technique introduced by David Ungar, where objects are split into sections/classes/generations. It is mainly based on the concept that "most objects die young," so it makes sense to keep the immortal or objects which are not likely to die in a separate section, and not iterate through them while doing garbage collection, thus saving time.

- David Ungar and his team are mostly credited for making the Smalltalk language which was a super-dynamic language where everything is an object. It was a bad idea as it was super expensive. A JIT

compilation was introduced as a workaround fix.

- His students associated with the group are the same team who later went on to build the V8 engine for the Google chrome browser.
- Generational GC takes pressure of the full heap GC. It can work quite well if can weak if lifetime is not much. Python uses approximate generational GC.
- In a generational GC, young objects or objects which are likely to be destroyed soon, are placed in a separate zone, and we apply our GC only on this set. This process is repeated over time.