Garbage Collector

Generally involve 2 steps: detection and collection.
Some comon methods are:

- Reference counting: If nothings point to it, garbage! Collect as soon as it detects.
  Drawback: inefficient handling of circular references.
- Mark-sweep: If cant be reached from roots, garbage! Collect phase happens after marking phase.
  Drawback: memory leak if disruption happens.
- Generational: Most young objects become garbage quickly, divide the heap into young / old sections, young one collected more frequently

Mark-Sweep: Performance $O(n)$

The simple algorithm is

```
def mark(node)
    node.status = "reachable"
    for ref in node.references:
        mark(ref)
```

drawback: if the process gets disrupted, dead objects will be left in the heap that are not reclaimed

Solution: Tricolor marking

```
def tricolor_mark(node):
    return if node.color ≠ "white"      # prevents circular reference
    node.color = "grey"
    for ref in node.references:
        tricolor_mark(ref)
    node.color = "black"
```

Why? 3 colors represent 3 statuses:

- White: unreached (will be cleaned)
- Grey: in progress
- Black: reachable (will not be cleaned)

If disruption happens, we can restart from grey nodes.

Generational Garbage Collector: Performance O(n)

Core idea:

- Split heap into young and old sections.
- When young section is full, do a "minor collection" on just the young section.
- Any objects survive the minor collection(s), gets promoted to old section.
- When memory is low, do a "major collection" on just the old section, Mark-Sweep is usually deployed here.

Algorithm

```
def main():
    for obj in all_objs:
        allocate(obj)
    if memory_is_low():    collect_major()

def allocate(obj):
    young_generation.append(obj)
    if len(young_generation) > MAX_ALLOWED_SIZE:
        collect_minor()

def collect_minor:
    for obj in young_generation:
        if is_reachable(obj)
            old_generation.append(obj)    # simple promotion policy
    young_generation.clear()

def collect_major:
    # implement Mark-Sweep here
```

Performance Analysis: $O(m \cdot n)$

- $O(m)$: loop over young objects
- $O(n)$: start from roots traverse all the nodes

We can do better than $O(m.n)$, with tracking intergeneration references.

- Core idea:
  - Keep a remembered_set, to store the index of an old objects, if it referencing young objects.
  - Whenever there is a change in reference, update the remembered_set.
  - When running collect_minor, we check from both:
    - roots: mark all young objects reachable, skip old objects.
    - rememered_set: mark all young objects reachable from the set elements.

Track Intergenerational Reference Algorithm

```
def mark_reachable(obj):
    if obj.is_oldgen or mark_set.has(obj):
        return
    mark_set.add(obj)
    for ref in obj.references:
        mark_reachable(ref)

def find_reachable_young_objs():          --> O(n)
    mark_set = set()                          n: nodes reached
    for root in roots:                           from roots
        mark_reachable(root)

    for old_index in remembered_set:
        mark_reachable(old_gen[old_index])
    return mark_set

def collect_minor():
    reachables = find_reachable_young_objs()

    (young_gen - reachables).clear()      --> O(m)
                                              m: young gen nodes
```

Performance: $O(m + n)$

Reference Counting:

Counts the number of reference (pointers) to that object.
Anytime the count reach 0, reclaim the memory space.

Increase count:

```
// Example in C++
MyObject* obj = new MyObject(); // obj now has a reference count of 1
MyObject* anotherRef = obj;      // reference count increases to 2
```

Decrease count:

```
delete anotherRef; // reference count decreases to 1
delete obj;        // reference count decreases to 0
```

Limitation of Reference Counting

- Cannot handle circular reference i.e two objects pointing to each other.
- Performance overhead, each time we increase / decrease the reference count

↳ To mitigate some of these limitations, "weak reference" can be used.

Jargons

- Directed cycles:

  Object in heap that reference each other, create a cycle

  Example:

```
class Node()
    self.next = None

a = Node()
b = Node()
a.next = b
b.next = a
```

- Conservative / Optimistic approximation:

  Basically upper bound / lower bound of the accuracy of method used for live object detection

  - Conservative : identify more live objects than it should
  - Optimistic : identify less live objects than it should

- True liveness:

  Means an object on the heap is:

  - Reachable through reference from program roots (global var and stack).
  - Could potentially be acessed and used by the program in the future.

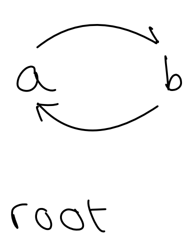
- "Approximation of true liveness"

  Fancy way of saying "how we detect live objects", or vice versa, "how we detect garbage".

- "Reference counting" is a conservative approximation

  it "cautiously" reclaim memory space, only when the reference count drop to zero

  Example:



Here we have 2 objects referencing each others, and root is not pointing to any of them. "Reference counting" will not reclaim the space here, which highlights its conservative nature.