# Final Exam: Reference Solution

**The Art of Computer Programming**

2 February 2026

## Part A: Warm-up (2 points)

**Q1.** After the three insertions, the priority queue contains

$$(a, 4),\ (b, 7),\ (c, 5).$$

After increase_key(a,8), it contains

$$(a, 8),\ (b, 7),\ (c, 5).$$

Therefore, the three calls to extract_max return, in order,

$$(a, 8),\ (b, 7),\ (c, 5).$$

Hence the objects returned are $a, b, c$, in that order.

**Q2.** One can implement increase_key(object, priority) using only extract_max and insert as follows:

a) Repeatedly call extract_max, storing the extracted pairs aside, until the desired object is found.

b) Replace its priority by the new one.

c) Reinsert all extracted elements.

If insert has complexity $\Theta(f(n))$ and extract_max has complexity $\Theta(g(n))$, then in the worst case we extract $n$ elements and reinsert $n$ elements. Thus the complexity is

$$\Theta(n\, g(n) + n\, f(n)) = \Theta(n(f(n) + g(n))).$$

## Part B: Unsorted Array in Python (5 points)

**Q3.** A simple implementation is:

```
class PriorityQueue:
    def __init__(self):
        self._queue = []

    def insert(self, obj, prio):
        self._queue.append((obj, prio))
```

```
    def extract_max(self):
        max_index = 0
        for i in range(1, len(self._queue)):
            if self._queue[i][1] > self._queue[max_index][1]:
                max_index = i
        return self._queue.pop(max_index)
```

**Q4.** One possible implementation of `increase_key` is:

```
def increase_key(self, obj, prio):
    for i, (o, p) in enumerate(self._queue):
        if o == obj:
            self._queue[i] = (o, prio)
            return
```

**Q5.** Worst-case complexities:

- `insert`: $O(1)$ amortized, because appending to a Python list is amortized constant time.
- `extract_max`: $O(n)$, because we must scan the whole list to find the maximum, and removing an element from the middle of a list may also shift elements.
- `increase_key`: $O(n)$, because we may need to scan the whole list to find the object.

# Part C: Binary Heap in C++ (7 points)

**Q6.** Reading the tree level by level from left to right, the corresponding array is

$$A = \big[(a,14),(g,11),(h,10),(d,9),(b,7),(c,5),(j,2),(f,3),(e,1),(i,4)\big].$$

**Q7.**
```
function make_heap(A, i):
    largest = i
    left = 2*i + 1
    right = 2*i + 2

    if left < len(A) and A[left].priority > A[largest].priority:
        largest = left

    if right < len(A) and A[right].priority > A[largest].priority:
        largest = right

    if largest != i:
        swap A[i] and A[largest]
        make_heap(A, largest)
```

**Q8.**
```cpp
#include <iostream>
#include <string>

BinaryHeap<std::string> heap;
heap.insert("a", 4);
heap.insert("b", 7);
heap.insert("c", 5);
heap.increase_key("a", 8);

auto x1 = heap.extract_max();
std::cout << x1.object << " " << x1.priority << std::endl;

auto x2 = heap.extract_max();
std::cout << x2.object << " " << x2.priority << std::endl;

auto x3 = heap.extract_max();
std::cout << x3.object << " " << x3.priority << std::endl;
```

**Q9.**
```cpp
template<class T>
void BinaryHeap<T>::insert(T o, int p) {
    array.push_back({o, p});
    unsigned i = array.size() - 1;

    while (i > 0) {
        unsigned parent = (i - 1) / 2;
        if (array[parent].priority >= array[i].priority)
            break;
        std::swap(array[parent], array[i]);
        i = parent;
    }
}

template<class T>
typename BinaryHeap<T>::ObjectPriorityPair BinaryHeap<T>::extract_max() {
    ObjectPriorityPair result = array[0];

    if (array.size() == 1) {
        array.pop_back();
        return result;
    }

    array[0] = array.back();
    array.pop_back();
    make_heap(0);
    return result;
}
```

The idea is:

- for `insert`, place the new element at the end of the array and repeatedly swap it with its parent while the heap property is violated;
- for `extract_max`, save the root, replace it by the last element, remove the last array cell, then restore the heap property by calling `make_heap(0)`.

**Q10.** Worst-case complexities:

- `insert`: $O(\log n)$, because the inserted element may move up along a root-to-leaf path.
- `extract_max`: $O(\log n)$, because `make_heap` may move the new root down along a root-to-leaf path.

# Part D: Fibonacci Heap (6 points)

**Q11.** An empty Fibonacci heap can be initialized as follows:

- the circular doubly linked list of roots is empty;
- the pointer *max* is set to `null`;
- the number of trees is $t = 0$;
- the number of marked nodes is $m = 0$.

**Q12.** To implement insert(object, priority):

- create a new node containing the object and its priority;
- make it a one-node tree;
- insert this node into the circular doubly linked list of roots;
- update the pointer *max* if the new node has larger priority.

This takes actual cost $O(1)$, since it only performs a constant number of pointer manipulations and comparisons.

**Q13.** a) For an empty heap, $t = 0$ and $m = 0$, so

$$\phi(H) = \gamma(t + 2m) = \gamma(0 + 0) = 0.$$

b) By definition,

$$\hat{c}_{x_i}(H_{i-1}) = c_{x_i}(H_{i-1}) + \phi(H_i) - \phi(H_{i-1}).$$

Summing for $i = 1, \ldots, k$, we obtain

$$\sum_{i=1}^{k} \hat{c}_{x_i}(H_{i-1}) = \sum_{i=1}^{k} c_{x_i}(H_{i-1}) + \sum_{i=1}^{k} \big(\phi(H_i) - \phi(H_{i-1})\big).$$

The second sum telescopes:

$$\sum_{i=1}^{k} \big(\phi(H_i) - \phi(H_{i-1})\big) = \phi(H_k) - \phi(H_0).$$

Hence

$$\sum_{i=1}^{k} \hat{c}_{x_i}(H_{i-1}) = \sum_{i=1}^{k} c_{x_i}(H_{i-1}) + \phi(H_k) - \phi(H_0).$$

Since $\phi(H_0) = 0$ and $\phi(H_k) \geqslant 0$, we get

$$\sum_{i=1}^{k} c_{x_i}(H_{i-1}) \leqslant \sum_{i=1}^{k} \hat{c}_{x_i}(H_{i-1}).$$

Dividing by $k$ yields

$$\frac{1}{k}\sum_{i=1}^{k} c_{x_i}(H_{i-1}) \leqslant \frac{1}{k}\sum_{i=1}^{k} \hat{c}_{x_i}(H_{i-1}).$$

**Q14.** Let $r$ be the root removed by extract_max, and let

$$D \coloneqq \max\left( \max_{u \text{ node of } H} \delta(u), \max_{u \text{ node of extract\_max}(H)} \delta(u) \right).$$

We analyze the actual cost and the variation of the potential.

*Step 1: remove the maximum root and promote its children.* If $r$ has degree $\delta(r)$, removing $r$ and making its children roots costs $O(\delta(r))$, hence $O(D)$.

Before the operation, suppose there are $t$ trees. After removing $r$ and promoting its $\delta(r)$ children, the number of trees becomes

$$t' = t - 1 + \delta(r).$$

*Step 2: consolidate roots of equal degree.* Whenever two roots have the same degree, we link them: one becomes the child of the other. Each such link has actual cost $O(1)$, and decreases the number of trees by 1.

After consolidation, all roots have distinct degrees. Since every root has degree at most $D$, there can be at most $D+1$ roots left (one per possible degree $0, 1, \ldots, D$). Therefore the number of links performed is at least

$$t' - (D+1),$$

and at most $t'$. In particular, all the work of repeated linking is proportional to the number of links.

Now observe that each link decreases the number of trees $t$ by 1. Moreover, when a node becomes a child during a link, its mark is reset to `false`. Since roots are unmarked before linking, this operation cannot increase the number of marked nodes $m$, and it may decrease it if a previously marked node becomes a child.

Therefore the potential

$$\phi(H) = \gamma(t + 2m)$$

decreases by at least $\gamma$ for each link (and possibly more if some marks are reset). Choosing $\gamma$ large enough ensures that this decrease in potential pays for the constant actual cost of each linking operation.

Hence, amortized, the consolidation phase costs only $O(D)$: after the links have been paid for by the decrease in potential, only the work proportional to the possible degrees remains.

Therefore the amortized complexity of extract_max is

$$O(D) = O\left(\max\left(\max_{u \text{ node of } H} \delta(u), \max_{u \text{ node of extract\_max}(H)} \delta(u)\right)\right).$$

**Q15.** Since the maximum degree of any node is $O(\log n)$, we conclude that extract_max has amortized complexity $O(\log n)$.

Thus Fibonacci heaps have the following amortized complexities:

- insert: $O(1)$,
- increase_key: $O(1)$,
- extract_max: $O(\log n)$.

Comparison with the previous implementations:

- compared with an unsorted array, Fibonacci heaps improve extract_max from $O(n)$ to $O(\log n)$, and also provide $O(1)$ amortized increase_key;
- compared with a binary heap, Fibonacci heaps have the same asymptotic complexity for extract_max, but better amortized complexity for insert and increase_key.

Their drawback is that they are much more complicated to implement, and their bounds are amortized rather than worst-case.