Mid-Term Exam – Solution

The Art of Computer Programming

6 November 2026

General rule: syntax errors that do not change the meaning of a program are not penalized. A wrong result carried over from an earlier question is not penalized a second time. Justifications are expected to be short: one or two sentences with the right idea get full credit.

Exercise A: A Recursive Function (3 points)

Q1. The calls are `f(13)`, `f(6)`, `f(3)`, `f(1)`, `f(0)`. The returned value is $1+(0+(1+(1+0))) = 3$.

Grading: 0.5 for the calls, 0.5 for the value.

Q2. $13 = \underline{1101}_2$. The function computes the number of bits equal to 1 in the binary representation of `n`: `n % 2` is the last bit of `n`, and `n // 2` is `n` without its last bit.

Grading: 0.25 for the binary representation, 0.5 for the answer, 0.25 for the justification.

Q3. $\Theta(\log n)$: each call does a constant amount of work and n is halved at each call, so there are $\lfloor \log_2 n \rfloor + 2$ calls for $n \geqslant 1$, as many as the number of bits of n, plus one.

Grading: 0.5 for the answer, 0.5 for the justification; an answer with $O(\log n)$ gets full credit.

Exercise B: Memory in C (3.5 points)

Q4. The final values are `a = {1, 20, 30}`. In `g`, `x` is a copy of `a[0]`, which does not change, whereas `y` points to `a[1]`, which becomes 20. Then `p` points to `a[2]`, which becomes 30.

Grading: 0.5 for `a[0]`, 0.75 each for `a[1]` and `a[2]`.

Q5.

- In `triples`: the loop condition `i <= n` writes `r[n]`, one element past the end of the allocated block (a buffer overflow: undefined behavior, possibly a crash or the corruption of other data). Correction: `i < n`.
- In `main`: the array returned by `triples` is never freed, so that each iteration loses a block (a memory leak). Correction: `free(s);` after the `printf`, in the loop.

Grading: 0.75 per error, for at most 1.5 in total: 0.25 for where it is, 0.25 for what goes wrong, 0.25 for the correction. The missing check on the result of `malloc` also counts as a correct error: if the allocation fails, `r` is a null pointer, and `r[i]` dereferences it; the correction is to test `r` and stop the program with an error message. So any two of these three errors get full credit.

Exercise C: Complexity (3.5 points)

Q6. `h1` is in $\Theta(n^2)$: the instruction `c += 1` is executed $n + (n-1) + \cdots + 1 = n(n+1)/2$ times.

`h2` is in $\Theta(n)$, and not $\Theta(n \log n)$: the inner loop runs i times for $i = 1, 2, 4, \ldots, 2^p$, where $2^p \leqslant n < 2^{p+1}$, in total $2^{p+1} - 1$ times, and $n \leqslant 2^{p+1} - 1 \leqslant 2n$.

Grading: `h1`: 0.25 for the answer, 0.25 for the justification. `h2`: 0.5 for the answer, 0.5 for the justification. For `h2`, the bound $O(n \log n)$ is correct but is not a Θ: it gets 0.5 in total if justified by the number of iterations of both loops.

Q7. The worst case is $\Theta(n^2)$, reached when all elements are distinct: the test `x not in seen` scans the list `seen`, whose size grows up to n, for a total of $0+1+\cdots+(n-1)$ comparisons.

We additionally store the elements seen so far in a `set`, and test membership in it, which takes $\Theta(1)$ on average; the list is still needed to keep the order (a set alone does not keep it; a `dict`, which keeps the insertion order in Python, would also work):

```
def without_duplicates(t):
    seen = []
    seen_set = set()
    for x in t:
        if x not in seen_set:
            seen_set.add(x)
            seen.append(x)
    return seen
```

Grading: 0.25 for $\Theta(n^2)$ (or $O(n^2)$), 0.25 for the input, 0.5 for the modification (0.25 if the set replaces the list, which loses the order).

Q8. The copies take place when inserting the 2nd, 3rd, 5th, 9th, and 17th elements, and copy 1, 2, 4, 8, and 16 elements: 31 copies in total.

Grading: 0.5 for identifying when the reallocations take place, 0.5 for the total.

Exercise D: Algorithm Design (4 points)

Q9.
```
def longest_run(t):
    best = 0
    current = 0
    for i in range(len(t)):
        if i > 0 and t[i] == t[i - 1]:
            current += 1
        else:
            current = 1
        if current > best:
            best = current
    return best
```

Grading: 1 for a loop over the elements comparing adjacent elements or keeping the previous one; 0.75 for a correct count of the current run, reset to 1 (not 0) on a change;

0.75 for keeping the maximum; 0.25 for the empty sequence; 0.25 for a run at the very end of the sequence, a common error when `best` is only updated when the value changes.

A correct function with two nested loops gets at most 2 points if it is quadratic; a function restarting the inner loop after the end of each run is linear and gets full credit.

Q10. Time $\Theta(n)$ (a single pass, with a constant amount of work per element, on every input, so this is also the worst case); additional space $\Theta(1)$ (two integers and an index).

Grading: 0.5 for time, 0.5 for space, graded according to the function that was written.

Exercise E: A Linked List in C (3 points)

Q11. The list is $13 \to 8 \to 5 \to$ `nullptr`, since each insertion takes place at the front, and `first->next->value` is 8.

Grading: 0.5 for the list, in the right order (drawing the final `nullptr` is not required); 0.5 for the value, graded according to the list that was drawn.

Q12.
```
int list_count(const Node *first, int x) {
  int count = 0;
  for (const Node *n = first; n != nullptr; n = n->next)
    if (n->value == x)
      count++;
  return count;
}
```

Grading: 0.5 for following the `next` pointers until `nullptr`, 0.25 for the test, 0.25 for the counter and its return. A `while` loop, a recursive function, `NULL` instead of `nullptr`, or moving the parameter `first` itself along the list are all correct.

Q13. Elements should be added at the last node and removed at the first node. Adding after the last node takes $\Theta(1)$ with the pointer to it, and so does removing the first node. The other way round, removing the last node would require updating the `next` pointer of the node before it, which can only be reached by following the list from its first node: $\Theta(n)$.

Grading: 0.25 for each end, 0.5 for the reason.

Exercise F: A Class for Browsing History (3 points)

Q14. `current` is `"a"`, `past` is `{"home"}`, and `future` is `{"c"}`. In particular, `"b"` has disappeared: the call `h.visit("c")` emptied `future`.

Grading: 0.25 each for `current` and `past`, 0.5 for `future`, which must not contain `"b"`.

Q15.
```
void forward() {
  if (!future.empty()) {
    past.push_back(current);
    current = future.back();
    future.pop_back();
  }
}
```

Grading: 0.25 for the test, 0.25 for each of the three steps; they must come in an order that does not lose a page.

Q16. $\Theta(k)$ in the worst case: removing the first element of a vector shifts all the others, that is, $k - 1$ elements. (Otherwise `visit` is in amortized $\Theta(1)$, and clearing `future` is proportional to the number of removed elements, each of which was inserted once.)

With a `std::deque`, a double-ended dynamic array, or a `std::list`, a doubly linked list, insertion and removal at both ends are in $\Theta(1)$ (amortized for `std::deque`), and this is all that `past` needs: `push_back`, `back`, `pop_back`, and `pop_front`.

Grading: 0.25 for the answer $\Theta(k)$, 0.25 for why, 0.25 for the structure (a deque, a doubly linked list, or a circular buffer, named as such or by its C++ class), 0.25 for why. The remark in parentheses on clearing `future` is not expected. A singly linked list that adds at the front gets only 0.25 of the last 0.5: it cannot remove the oldest in $\Theta(1)$; with a pointer to the last node and additions at the back, it cannot pop from the back in $\Theta(1)$.