Q1

Let $n' = n/2 = 2^{k-1}$, and $h_{ij} =$ ith row $\wedge$ jth column entry of H_{k-1}. Let the first n' entry of v as v_1 and the later n' entries as v_2. Then,

$$H_k v = \begin{bmatrix} \sum_{i=1}^{n'} h_{1i}v_i + \sum_{i=1}^{n'} h_{1i}v_{i+n'} \\ \sum_{i=1}^{n'} h_{2i}v_i + \sum_{i=1}^{n'} h_{2i}v_{i+n'} \\ \vdots \\ \sum_{i=1}^{n'} h_{n'i}v_i + \sum_{i=1}^{n'} h_{n'i}v_{i+n'} \\ \sum_{i=1}^{n'} h_{1i}v_i - \sum_{i=1}^{n'} h_{1i}v_{i+n'} \\ \sum_{i=1}^{n'} h_{2i}v_i - \sum_{i=1}^{n'} h_{2i}v_{i+n'} \\ \vdots \\ \sum_{i=1}^{n'} h_{n'i}v_i - \sum_{i=1}^{n'} h_{n'i}v_{i+n'} \end{bmatrix} = \begin{bmatrix} H_{k-1}v_1 \\ H_{k-1}v_1 \end{bmatrix} + \begin{bmatrix} H_{k-1}v_2 \\ -H_{k-1}v_2 \end{bmatrix} \tag{1}$$

We claim that the following algorithm, which uses the above equation, calculates the desired matrix-vector productruns in $\theta(n \log n)$ operations.

We first recursively compute $H_{k-1}v_1$ and $H_{k-1}v_2$ (which are of size $n' = n/2$).

Extending $H_{k-1}v_1$ by adding itself to create $\begin{bmatrix} H_{k-1}v_1 \\ H_{k-1}v_1 \end{bmatrix}$ takes constant operation.

Extending $H_{k-1}v_2$ by adding itself multiplied by -1 to create $\begin{bmatrix} H_{k-1}v_2 \\ -H_{k-1}v_2 \end{bmatrix}$ takes $\theta(n/2) \in \theta(n)$ operation. Thus, the total time complexity of this algoruthm is given by recurrence relation $T(n) = 2T(n/2) + \theta(n)$, which solves to $\theta(n \log n)$ by master theorem.

Q2

In this question, we assume that:

- Computing $n/3$ takes $O(1)$ time for all $n \in \mathbb{N}$.
- Shifting bits (multiplication by a power of 2) takes $O(n)$ time.
- Addition and subtraction takes $O(n)$ time.

(a)

Let $m = n/3$ (we ignore ceiling), $x = a_2 2^{2m} + a_1 2^m + a_0$ then we compute the following.

Define

- $p(B) = a_2 B^2 + a_1 B + a_0$
- $P(B) = (p(B))^2 = c_4 B^4 + c_3 B^3 + c_2 B^2 + c_1 B + c_0$ for some $c_4, c_3, c_2, c_1, c_0 \in \mathbb{R}$
  Note that $c_4 = a_2^2$ and $c_0 = a_0^2$

and let

- $r_0 = a_0^2$
- $r_1 = (a_2 + a_1 + a_0)^2 = (p(1)^2) = P(1)$
- $r_2 = (a_2 - a_1 + a_0)^2 = (p(-1)^2) = P(-1)$
- $r_3 = (4a_2 + 2a_1 + a_0)^2 = (p(2)^2) = P(2)$
- $r_4 = a_2^2$

Thus,

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix}$$

Let H be the matrix above, then

$$H^{-1} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{6} & \frac{1}{6} & -2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ -\frac{1}{2} & 1 & -\frac{1}{3} & -\frac{1}{6} & 2 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix} = \begin{bmatrix} c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} r_4 \\ \frac{1}{2}r_0 - \frac{1}{2}r_1 - \frac{1}{6}r_2 + \frac{1}{6}r_3 - 2r_4 \\ -r_0 + \frac{1}{2}r_1 + \frac{1}{2}r_2 - r_4 \\ -\frac{1}{2}r_0 + r_1 - \frac{1}{3}r_2 - \frac{1}{6}r_3 + 2r_4 \\ r_0 \end{bmatrix}$$

Thus, all the coefficient can be computed using dividing by 3, bit shifting, and addition and subtraction, which in combination takes $O(n)$ time. Since

$$x^2 = (a_2 2^{2m} + a_1 2^m + a_0)^2 = c_4 2^{4m} + c_3 2^{3m} + c_2 2^{2m} + c_1 2^{1m} + c_0 = P(2^m)$$

with additional shifting and addition operation, we can compute x^2 that takes $5T(n/3) + O(n) \in O(n^{\log_3 5})$

(b)

Let $m = n/3$ (we ignore ceiling), $A = a_2 2^{2m} + a_1 2^m + a_0$, $B = b_2 2^{2m} + b_1 2^m + b_0$ then we compute the following.

Define

- $p(X) = a_2 X^2 + a_1 X + a_0$
- $q(X) = b_2 X^2 + b_1 X + b_0$
- $P(X) = p(X)q(X) = c_4 X^4 + c_3 X^3 + c_2 X^2 + c_1 X + c_0$ for some $c_4, c_3, c_2, c_1, c_0 \in \mathbb{R}$
  Note that $c_4 = a_2 b_2$ and $c_0 = a_0 b_0$

and let

- $r_0 = a_0 b_0 = c_0$
- $r_1 = (a_2 + a_1 + a_0)(b_2 + b_1 + b_0) = p(1)q(1) = P(1)$
- $r_2 = (a_2 - a_1 + a_0)(b_2 - b_1 + b_0) = p(-1)q(-1) = P(-1)$
- $r_3 = (4a_2 + 2a_1 + a_0)(4b_2 + 2b_1 + b_0) = p(2)q(2) = P(2)$
- $r_4 = a_2 b_2 = c_4$

Using the same matrix operation, we can derive coefficients, Since

$$AB = (a_2 2^{2m} + a_1 2^m + a_0)(b_2 2^{2m} + b_1 2^m + b_0) = P(2^m) = c_4 2^{4m} + c_3 2^{3m} + c_2 2^{2m} + c_1 2^{1m} + c_0$$

we can compute the multiplication in $O(n^{\log_3 5})$ due to the same reasoning as part(a).

Q3

Algorithm 1 3 Split Huffman Encoding

function HUFFMAN(S) ▷ F is the prioroty queue of (frequency, letter) where frequency is key

  if size of $S \leq 3$ ▷ Base case returns correct encoding when size ≤ 3
    return a single node tree if $|S| = 1$.
    return a tree with two leaves if $|S| = 2$.
    return a tree with three leaves if $|S| = 3$.
  end if

  If the size of S is even pad S with a dummy character with frequency $= 0$.

  $x, y, z \leftarrow$ three smallest frequency elements in S.
  Let $w :=' xyz'$
  Let $f_w := f_x + f_y + f_z$
  Push (f_w, w) to S ▷ Create input with smaller size for recursive call
  $H \leftarrow$ **Huffman**(S).

  Find the node w^* in H that corresponds to w.
  Add branches as children of w^* . ▷ This assigns encoding to branches.
  return H

end function

Time Complexity

The base case takes $O(1)$ time. For the recursive case, popping the three or two smallest elements takes $O(\log n)$ time. All operations before the recursive call take $O(1)$ time, and all operations after the recursive call take $O(1)$ time (e.g., the tree is implemented using direct access table). Since branching by factor of 3 will reduce the time complexity more than by branching factor of 2, The time complexity of this algorithm is given by the recurrence relation $T(n) \leq T(n-1) + O(\log n) =$ recurrence relation of original Huffman algorithm. From lecture, original Huffman algorithm takes $O(n \log n)$, thus, $T(n) \in O(n \log n)$.

Correctness

(Lemma 1) In an optimal ternary tree that is not a single node tree, every node other than the root must have a sibling, because if otherwise, replacing the parent of a single child with its child gives tree with lower total length, contradicting that the original tree was optimal.

(Lemma 2) The optimal ternary tree T is a full ternary tree if it has odd number of leaves ≥ 3. Suppose for contradiction that it is not. Then there is at least one node that has binary branching, let v be the deepest such node. If v is not a parent of two leaves, by moving a non-child leaf of the subtree rooted at v as the child of v we can find a tree with lower loss than T. If v is a parent of two leaves, there are two cases. If there is another binary branching at node u, moving a child of v as a child of u and replacing v with its remaining child makes a lower cost tree. If there no other binary branching, then the tree cannot have odd number of leaves, since for each node from v to root, it give rise to $3^k_1 + 3^k_2$ leaves for some k_1, k_2, and this number must be even.

(Lemma 3) In optimal ternary tree, the three lowest freqeuncy nodes x, y, z (in this order) must be the three deepest node (in that order). Suppose for contradiction that it is not. Let $p \in x, y, z$ then there exists at least one q such that q is more frequent than p but located deeper than p. swapping p and q leads to a tree with lower total length, which is a contradiction.

Proof.

As the base case, when the size of $|S| \leq 3$ the algorithm outputs optimal tree.

In any other cases the algorithm ensure that $|S|$ is odd at line 7.

Assume as IH that algotithm outputs optimal tree when size of input is less than $k \in \mathbb{N}^{\geq 3}$.

Let S be an input of size k. Let H be the output of the algorithm given S. Let S' be the S after operations at line 8 to 11. Then the recursive call at line 12 returns the optimal tree, H' over S' by IH.

Let f_w be the sum of frequencies of x, y, z. We know that total loss of S, $loss(H) = f_w + loss(H')$ (using the same reasoning as KT p.174 but using three frequencies instead of two).

Suppose for contradiction H is not optimal. Then there exists an optimal tree T over S, and thus $loss(H) > loss(T)$. Because T is optimal, the three lowest frequency nodes appear as siblings at the deepest level of T. (by lemma 2 and 3). Let T' be a tree after removing the three lowest frequency nodes from T. Similarly as above, T' is a optimal tree over S' and we can show that $loss(T) = f_w + loss(T')$ by using the same reasoning as above.

By IH, $loss(H') \leq loss(T')$. However, then $loss(H) = f_w + loss(H') \leq f_w + loss(T') = loss(T)$, which is a contradiction. $\square$

Q4

Algorithm 2 Maximize Profit

function MAXIMIZEPROFIT(E) ▷ E is the prioroty queue of (g_i, t_i) where t_i is the key
    $schedule \leftarrow []$; $time \leftarrow 0$
    while E is not empty **do**
        $Q \leftarrow$ Max-Heap of (g_i, t_i) where the key is g_i
        $(g_i, t_i) \leftarrow$ **Dequeue**(E)
        while E is not empty and $time \leq t_i < time + 1$ **do**
            Enqueue (g_i, t_i) to Q and update (g_i, t_i) with the output of **Dequeue**(E)
        end while
        If t_i of the last tuple is within $[time, time + 1)$ **Enqueue** (g_i, t_i) to Q
        Else Enqueue (g_i, t_i) to E ▷ enque the tuple back to E if it is out of the range
        If Q is not empty **Add Max**(Q) to *schedule*
        $time \leftarrow time + 1$
    end while
    return *schedule*
end function

Time Complexity

If we had to build priority queue, this takes $O(n)$ time. Every operation other than enqueing or dequeueing takes constant time.

Let j_i be the number of iterations of inner while loop at ith iteration of the outer while loop. At least j_i number of tuples are removed from E at ith iteration. Suppose that the outer loop iterates k times in total. Then the outer while loop stops when $\sum_{i=1}^{k} j_i = n$. The total number of enqueing and dequeing at ith iteration is $2 + 2j_i$. Thus, total number of enqueing and dequeing of the algorithm is $\sum_{i=1}^{k} 2 + 2j_i = 2k + 2n$.

k can exceed n iff inner loop does not iterate for many iteration of the outer loop. However, because the time gets incremented at each iteration of the outer loop, $2k + 2n \in O(max(max(t_i)_{i=1}^{n}, n))$. Let $m = max(max(t_i)_{i=1}^{n}, n)$, then the algorithm takes $O(m \log n)$ time.

Correctness

We define optimality as maximizing the sum of profit.

For each $t \in \mathbb{N}$, define $P(t)$ as: at the end of tth iteration of the puter loop, the algorithm have the optimal solution (*schedule*) among all the events i where $t_i \in [0, t)$

For the base case $t = 0$, the empty array *schedule* is the optimal solution in $[0, 0)$.

Let $t \in \mathbb{N}^+$ and assume $P(t-1)$ holds. By IH, *schedule* is the optimal solution in the time interval $[0, t-1)$.

Since *time* increases by 1 at the end of every iteration of the outer loop, $time = t-1$ during the tth iteration of the outer loop. If E is empty at the begining of tth iteration of the outer loop, then the function returns, and all the events have critical time within the range $[0, t-1)$. Thus the algorithm outputs the optimal solution within $[0, t)$.

Suppose that E is not empty and the outer loop is executed

(Case 1) E becomes empty before the inner loop gets executed.

If $t_i \in [t-1, t)$, then the last item in E is added to schedule, and result in schedule with maximum benefit among all the events with critical time within $[0, t)$.

If $t_i \notin [t-1, t)$ then it is trivial to show $P(t)$.

(Case 2) E is non-empty right before the inner loop, but loop does not iterate.

Then the only event (g_i, t_i) dequeued at line 5 is not added to schedule since $t_i \notin [0, t)$ and iteration ends. Note that t_i cannot be less than $t-1$ because in that case it would have dequeued in the previous iteration. All the events following it will have critical value of at least t_i, and hence will not be in $[0, t)$. Thus all the events in $[0, t)$ is equivalent to all the events in $[0, t-1)$, and schedule has optimal solution among all the tasks whose critical time is within $[0, t)$.

(Case 2) If E is non empty and the inner loop iterates at least once.

Then the inner loop adds all the events to Q if their critical time is within $[0, t)$. Because every event takes one unit of time and we can do only one task in Q, choosing the event with maximum profit gives the optimal solution among all the tasks in the interval $[0, t)$.

Thus in any case, at the end of tth iteration, the algorithm have the optimal solution (*schedule*) among all the events i where $t_i \in [0, t)$

Let $t_{max} = \lceil max_{i=1}^{n} t_i \rceil + 1$ then by $P(t_{max})$, the algorithm outputs the optimal solution among all the events in E.