Algorithm Design and Analysis
Assignment 5

1. Consider the following 3-PARTITION problem. Given integers $a_1, ..., a_n$, we want to determine whether it is possible to partition of $\{1, ..., n\}$ into three disjoint subsets I, J, K such that
$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{1}{3} \sum_{i=1}^{n} a_i.$$
For example, for input $(1, 2, 3, 4, 4, 5, 8)$ the answer is yes, because there is the partition $(1, 8)$, $(4, 5)$, $(2, 3, 4)$. On the other hand, for input $(2, 2, 3, 5)$ the answer is no. Devise and analyze a dynamic programming algorithm for 3-PARTITION that runs in time polynomial in n and in $\sum_i a_i$.

   Solution. (Remark: all integers are positive.) Let $f(j, x, y)$ be a boolean value to show whether we can only use the first j integers to make two disjoint subsets with sum x and y. We define the transition of f to be
$$f(j, x, y) = f(j-1, x-a_j, y) \vee f(j-1, x, y-a_j) \vee f(j-1, x, y).$$
It is easy to show if any one of $f(j-1, x-a_j, y)$, $f(j-1, x, y-a_j)$ and $f(j-1, x, y)$ is true, then we can directly put the j-th integer into the corresponding subset, which means $f(j, x, y)$ is also true. On the other hand if $f(j, x, y)$ is true, remove a_j from the corresponding subset shows one of the three must be true. With the transition function, we give the DP algorithm as follows. $\square$

Algorithm 1: Determine whether we can make a feasible 3-PARTITION

```
f[0,0,0] ← 1. ;
S = ∑_{i=1}^{n} a_i/3 ;
for j = 1 to n do
    for x = 1 to S do
        for y = 1 to S do
            f[j,x,y] ← f[j−1,x,y] ;
            if x ≥ a[j] and f[j−1,x−a[j],y] = 1 then
                f[j,x,y] ← 1;
            end
            if y ≥ a[j] and f[j−1,x,y−a[j]] = 1 then
                f[j,x,y] ← 1;
            end
        end
    end
end
Output f[n, ∑_{i=1}^{n} a_i/3, ∑_{i=1}^{n} a_i/3] ;
```

2. Let $X[1..n]$ be a reference DNA sequence. Let S be a set of m exon candidates of X, where each exon candidate is represented by a triple (i, j, w), which means that the strength or probability for fragment $X[i..j]$ being an exon is w. Notice that many triples in S are false exons, and true exons do not overlap. Show how to use dynamic programming to find a maximum-weight subset of S in which all exon candidates are non-overlapping. The time complexity should be linear in terms of n and m.

   Solution. For any $s = (i, j, w) \in S$, we define $i(s) = i$, $j(s) = j$, and $w(s) = w$. Let $f(k)$ be the maximum weight construct by a feasible subset of exons with $j \leq k$. Let S_k be the subset of exons with $j = k$, we define the transition of f as follows:

   $$f(k) = \max\{f(k-1), \max_{s \in S_k}\{f(i(s)-1) + w(s)\}\}.$$

   We can prove it by induction and $f(0) = 0$ trivially holds. Assume $f(k')$ holds for any $k' < k$, we can at most choose one exon in S_k. That means we can either choose nothing in S_k and get $f(k-1)$ or choose one $s \in S_k$ so that we can only use other exons with $j < i(s)$ and get $f(i(s)-1) + w(s)$. Hence, we can conclude the transition, and we give the DP algorithm below. □

Algorithm 2: Calculate the maximum weight subset

```
f[0] ← 0. ;
for k = 1 to n do
    f[k] ← f[k − 1] ;
    for each s ∈ S_k do
        if f[i(s) − 1] + w(s) < f[k] then
            f[k] ← f[i(s) − 1] + w(s) ;
        end
    end
end
Output f[n] ;
```

3. Consider the following game. A "dealer" produces a sequence $s_1, \ldots, s_n$ of "cards" facing up, where each card s_i has a value v_i. Then two players take turns picking a card from the sequence, but can only pick the first or the last card of the (remaining) sequence. The goal is to collect cards of largest total value. Assume n is even.

   (a) Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. That is, the natural *greedy* stratgy is suboptimal.

   (b) Give an $O(n^2)$ algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute in $O(n^2)$ time some information, and then the first player should be able to make each move optimally in $O(1)$ time by looking up the precomputed information.

Solution.

(a) Consider the sequence $2, 100, 1, 1$, if the first player chooses 2, then the second player can choose 100, which means the greedy strategy is not optimal.

(b) Define $f[i,j]$ $(j - i \geq 1)$ be the optimal value of the player when he is choosing cards from the sequence $s_i, ..., s_j$. (Notice that we allow there is an odd number of cards). We define the transition as follows:

If $j - i = 1$, $f[i,j] = \max\{a_i, a_j\}$.

If $j - i > 1$,

$$f[i,j] = \max\{a_i + \sum_{i+1 \leq k \leq j} a_k - f[i+1,j], a_j + \sum_{i \leq k \leq j-1} a_k - f[i,j-1]\}.$$

When $j-i=1$, it's easy to see the best strategy is to choose the larger card. If $j-i>1$, there are two choices for the player, choosing a_i or a_j. Assume we get the correct value of $f[i',j']$ for all $j'-i'<j-i$, when the player (player A) chooses a_i, the other player will get $f[i+1,j]$, and player A will get $a_i+\sum_{i+1\le k\le j} a_k - f[i+1,j]$. With the same reason, the player will get $a_j+\sum_{i\le k\le j-1} a_k - f[i,j-1]$ if he chooses a_j. We give the algorithm below and we record the optimal strategy by the array g. That means, when the first player is choosing card from the subsequence from i to j, he should choose i or j following $g[i,j]$.

Algorithm 3: Calculate the optimal strategy.

$\forall 1\le i\le n, f[i] \leftarrow i.$;
for $k=1$ *to* $n-1$ **do**
    for $i=1$ *to* $n-k$ **do**
        $j\leftarrow i+k$;
        if $a_i+\sum_{i+1\le k\le j} a_k - f[i+1,j] > a_j+\sum_{i\le k\le j-1} a_k - f[i,j-1]$ **then**
            $f[i,j]\leftarrow a_i+\sum_{i+1\le k\le j} a_k - f[i+1,j]$;
            $g[i,j]\leftarrow i$;
        end
        if $a_i+\sum_{i+1\le k\le j} a_k - f[i+1,j] \le a_j+\sum_{i\le k\le j-1} a_k - f[i,j-1]$ **then**
            $f[i,j]\leftarrow a_j+\sum_{i\le k\le j-1} a_k - f[i,j-1]$;
            $g[i,j]\leftarrow j$;
        end
    end
end
Output g ;

$\square$

4. Assume points $v_1, v_2, \ldots, v_n$ form a convex polygon in $\mathbb{R}^2$. Let $d(i,j)$ be the Euclidean distance between v_i and v_j if $i \leq j$ and $d(i,j) = -\infty$ if $i > j$. For every $r \geq 0$, we use $d^{(r)}(i,j)$ to denote the length of the the *longest paths* from v_i to v_j using *at most* r edges. Therefore, $d(i,j) = d^{(1)}(i,j)$.

   (a) Let $s, t \geq 0$ be any two any integers satisfying $r = s + t$. For every $i \leq j$, prove that $d^{(r)}(i,j) = \max_{i \leq k \leq j} \left\{ d^{(s)}(i,k) + d^{(t)}(k,j) \right\}$.

   (b) Prove that the distance $d(\cdot,\cdot)$ satisfies the *inverse Quadrangle Inequality* (iQI):

$$\forall i \leq i' \leq j \leq j' : d(i,j) + d(i',j') \geq d(i',j) + d(i,j').$$

   (c) Prove that for any integer $r \geq 0$, $d^{(r)}(\cdot,\cdot)$ satisfies iQI as well.

   (d) If we let $K^{(r)}(i,j)$ denote $\max \left\{ k \mid i \leq k \leq j \text{ and } d^{(r)}(i,j) = d^{(s)}(i,k) + d^{(t)}(k,j) \right\}$, prove that

$$K^{(r)}(i,j) \leq K^{(r)}(i,j+1) \leq K^{(r)}(i+1,j+1), \quad \text{for} \quad i \leq j.$$

   (e) Give an algorithm to compute $d^{(r)}(i,j)$ for all $1 \leq i < j \leq n$ in $O(\log r \cdot n^2)$ time [1].

Solution.

(a) Assume $d^{(r)}(i,j)$ using $r' \leq r$ edges $(i_1 = i, i_2), (i_2, i_3) ...., (i_{r'}, j)$. $\forall 1 \leq k \leq r'$, $i_k < j$ because otherwise there is $-\infty$ distance. For any s and t, we can partition the r' edges into two subsets by the pivot k, with s edges and $r' - s$ edges where $r' - s \leq t$. Thus, it is one of the choice of $\max_{i \leq k \leq j} \left\{ d^{(s)}(i,k) + d^{(t)}(k,j) \right\}$.

(b) The iQL trivially holds when $i = i'$ or $j = j'$, the remaining case is 1) $i < i' = j < j'$ and 2) $i < i' < j < j'$. In case 1), the iQL becomes the triangle inequality $d(i,j) + d(j,j') \geq d(j,j')$ and it holds for the Euclidean distance. In case 2), (v_i, v_i', v_j, v_j) is a convex quadrilateral so the diagonals (v_i, v_j) and $(v_{i'}, v_{j'})$ are inside the quadrilateral and their intersection point o are also inside. Refer to Figure 1, we have $io + oj' \geq ij'$ and $i'o + oj \geq i'j$, so $d(i,j) + d(i',j') \geq d(i',j) + d(i,j')$.

[1] That is, your algorithm needs to compute all the $\binom{n}{2}$ values within $O(\log r \cdot n^2)$ time.

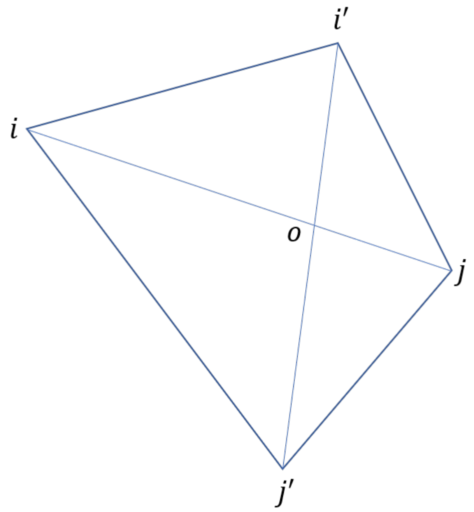

Figure 1: example

(c) We prove it by induction on r, and the base case is proved above. To prove iQI holds inductively on r, we assume it holds for any $r' < r$. Fix $s = r - 1$ and $t = 1$, we have that $d^{(r)}(i,j) = \max_{i \le k \le j} d^{(r-1)}(i,k) + d(k,j)$. Consider the two terms at the RHS of the iQL, assume $d^{(r)}(i',j)$ is maximized at $k = x$ and $d^{(r)}(i,j')$ is maximized at $k = y$. That means

$$d^{(r)}(i',j) = d^{(r-1)}(i',x) + d(x,j),\ d^{(r)}(i,j') = d^{(r-1)}(i,y) + d(y,j') \tag{1}$$

If $y \ge x$, because $i \le i' \le x \le j$ and $i \le x \le y \le j'$, we have

$$d^{(r)}(i,j) \ge d^{(r-1)}(i,x) + d(x,j),\ d^{(r)}(i',j') \ge d^{(r-1)}(i',y) + d(y,j').$$

Then, because $i \le i' \le x \le y$, by the induction hypothesis, we have

$$d^{(r-1)}(i,x) + d^{(r-1)}(i',y) \ge d^{(r-1)}(i',x) + d^{(r-1)}(i,y). \tag{2}$$

Hence,

$$\begin{aligned} d^{(r)}(i,j) + d^{(r)}(i',j') &\ge d^{(r-1)}(i,x) + d(x,j) + d^{(r-1)}(i',y) + d(y,j') \\ &\ge d^{(r-1)}(i',x) + d^{(r-1)}(i,y) + d(x,j) + d(y,j') \quad \text{By Eqn (2)} \\ &= d^{(r)}(i',j) + d^{(r)}(i,j') \quad \text{By Eqn (1)} \end{aligned}$$

If $y \le x$, symmetrically, we have

$$d^{(r)}(i,j) \ge d^{(r-1)}(i,y) + d(y,j),\ d^{(r)}(i',j') \ge d^{(r-1)}(i',x) + d(x,j').$$

Because $y \leq x \leq j \leq j'$, by the induction hypothesis, we have

$$d(y,j) + (x,j') \geq d(x,j) + d(y,j'). \tag{3}$$

Hence,

$$\begin{aligned} d^{(r)}(i,j) + d^{(r)}(i',j') &\geq d^{(r-1)}(i,y) + d(y,j) + d^{(r-1)}(i',x) + d(x,j') \\ &\geq d^{(r-1)}(i,y) + d^{(r-1)}(i',x) + d(x,j) + d(y,j') \quad \text{By Eqn (3)} \\ &= d^{(r)}(i',j) + d^{(r)}(i,j') \quad \text{By Eqn (1)} \end{aligned}$$

(d) To prove the first inequality, we plan to show that

$$\begin{aligned} \forall i \leq k < j, \quad & d^{(s)}(i,k+1) + d^{(t)}(k+1,j) - d^{(s)}(i,k) - d^{(t)}(k,j) \\ & \leq d^{(s)}(i,k+1) + d^{(t)}(k+1,j+1) - d^{(s)}(i,k) - d^{(t)}(k,j+1). \end{aligned} \tag{4}$$

If it holds, then move k from i to $K^{(r)}(i,j)$, $d^{(s)}(i,k) + d^{(t)}(k,j+1)$ must increase at least as much as $d^{(s)}(i,k) + d^{(t)}(k,j)$. Thus, $K^{(r)}(i,j+1) \geq K^{(r)}(i,j)$. To prove it, we should have

$$d^{(t)}(k+1,j) - d^{(t)}(k,j) \leq d^{(t)}(k+1,j+1) - d^{(t)}(k,j+1).$$

Notice that it is implied by the iQL of $d^{(t)}(\cdot,\cdot)$ for $k \leq k+1 \leq j \leq j+1$.

Similarly, to prove the second one, we plan to show that

$$\begin{aligned} \forall i < k < j, \quad & d^{(s)}(i,k+1) + d^{(t)}(k+1,j+1) - d^{(s)}(i,k) - d^{(t)}(k,j+1) \\ & \leq d^{(s)}(i+1,k+1) + d^{(t)}(k+1,j+1) - d^{(s)}(i+1,k) - d^{(t)}(k,j+1). \end{aligned} \tag{5}$$

That means we need to prove

$$d^{(s)}(i,k+1) - d^{(s)}(i,k) \leq d^{(s)}(i+1,k+1) - d^{(s)}(i+1,k).$$

It holds by the iQL of $d^{(s)}(\cdot,\cdot)$ for $i \leq i+1 \leq k \leq k+1$. □

(e) It is similar to the exponentiation by squaring method to calculate $d^{(r)}(\cdot,\cdot)$. Initially, we set $x = 1$ and calculate $d^{(x)} = d(\cdot,\cdot)$ in $O(n^2)$ time. Then, we write r in binary, and scan it from left to right from the second position.

- If the binary code we scan is 1, let $x' = 2x+1$, we calculate $d^{(2x)}$ from $d^{(x)}$ and then calculate $d^{(x')}$ from $d^{(2x)}$ and $d^{(1)}$, and then update $x = x'$.
- If the binary code we scan is 0, let $x' = 2x$, we calculate $d^{(2x)}$ by $d^{(x)}$, and then update $x = x'$.

Finally, we get $x = r$ and we get $d^{(r)}$ in $\log r$ rounds. Next, we give the DP algorithm to calculate $d^{(s+t)}$ by $d^{(s)}$ and $d^{(t)}$ in $O(n^2)$ time and so that our algorithm totally runs in $O(\log r \cdot n^2)$.

Algorithm 4: calculate $d^{(x)}$ from $d^{(s)}$ and $d^{(t)}$

```
∀1 ≤ i ≤ n, d[x, i, i] ← 0 ;
∀1 ≤ i ≤ n, K[i, i] ← i ;
for l = 2 to n − 1 do
    for i = 1 to n − l + 1 do
        j ← i + l − 1.;
        d[x, i, j] ← −∞;
        for k′ = K[i, j − 1] to K[i + 1, j] do
            if d[s, i, k′] + d[t, k′, j] > d[x, i, j] then
                K[i, j] ← k′;
                d[x, i, j] ← d[s, i, k′] + d[t, k′, j]
            end
        end
    end
end
```

Remark that we can conclude the algorithm above runs in $O(n^2)$ time because for each l, the "If" subroutine runs

$K[2][l] - K[1][l-1] + K[3][l+1] - K[2][l] ... + K[n-l+2][n] - K[n-l+1][n+1]$ times.

It equals to $K[n-l+2][n] - K[1][l-1] \leq n$, so the algorithm runs in $O(n^2)$.