1. (a) Since $A[1] \le A[k]$ and $-B[1] \le -B[k]$ for any k, $A[1]-B[1]$ is the minimum. Our algorithm can always output 1 for this problem. Thus, it runs in $O(1)$ time.

   (b) Let $A[i] = 2i$ for all $i \in [1, n] \setminus \{z\}$, and let $A[z] = 2i - 1$. Let further $B[i] = 2(n - i)$ for all $i \in [1, n]$. Note that $A[z] + B[z] = 2n - 1$ is the minimum. Our algorithm needs to output z.

   z is some secret number kept in Alice's mind. For any algorithm $\mathcal{A}$, if $\mathcal{A}$ has an access to $A[i]$, then Alice says $z \ne i$. In this way, $\mathcal{A}$ cannot figure out what z is before $n - 1$ different $A[i]$'s are read. This requires $\Omega(n)$ time already.

2. We apply the substitution method to obtain upper bounds for the first two subproblems, and Master theorem for the last.

   (a) We start by guessing that $T(n) = O(S(n))$ where

   $$S(n) \le \begin{cases} S(n/3) + S(n/4) + cn & \text{if } n > 1 \\ 1 & \text{if } n \le 1 \end{cases}$$

   By the recursion-tree method,

   $$\begin{aligned} S(n) &\le S(n/3) + S(n/4) + cn \\ &\le S(n/9) + S(n/12) + S(n/12) + S(n/16) + cn + 7cn/12 \\ &\le \cdots \\ &\le cn[1 + 7/12 + (7/12)^2 + (7/12)^3 + \cdots] \\ &\le cn/(1 - 7/12) \\ &= O(n) \end{aligned}$$

   To verify whether the guess $T(n) = O(S(n)) = O(n)$ is correct, we appeal to the substitution method, as shown in Claim 1.

   Claim 1. *$T(n) \le dn$ for any $n \ge 1$ where d is some positive constant.*

   Proof. We prove this claim by induction on n. Observe that $T(1)$, $T(2)$, ..., $T(n_0)$ are all constants if n_0 is a constant. If we set $d \ge \max\{T(1), T(2), \ldots, T(n_0)\}$, the induction base holds. Assume that $T(n) \ge dn$ for any $n < k$. For $n = k > n_0$, the recurrence relation gives that $T(k) \le T(\lceil k/3 \rceil) + T(\lceil k/4 \rceil) + \alpha k \le (7d/12 + \alpha)k + 2d$ for some constant $\alpha > 0$. We need

   $$(7d/12 + \alpha)k + 2d \le kd \tag{1}$$

to complete the proof. EQ (1) holds by setting $d \geq 12\alpha$ and $k > n_0 = 6$. We are done. $\square$

(b) We guess that $T(n) = O(n)$ because the recurrence relation of this subproblem is similar to that of subproblem (a).

Claim 2. *$T(n) \leq dn$ for any $n \geq 1$ where d is some positive constant.*

Proof. We prove this claim by induction on n. If n_0 is a constant, then $T(1)$, $T(2)$, ..., $T(n_0)$ are all constants. The induction base holds by setting $d \geq \max\{T(1), T(2), \ldots, T(n_0)\}$. Assume that $T(n) \leq dn$ for any $n < k$. For $n = k > n_0$, the recurrence relation yields that $T(k) \leq T(\lceil k/3 \rceil + 5) + T(\lceil k/4 \rceil + 7) + \beta k \leq (7d/12 + \beta)k + 14d$ for some constant $\beta > 0$. We require $(7d/12 + \beta)k + 14d \leq kd$ to complete the proof. We are done by setting $d \geq 12\beta$ and $k > n_0 = 42$. $\square$

(c) Because $f(n) \leq c \log n = O(n^{(\log_2 2) - \varepsilon})$, the first case of the Master theorem applies. We have $T(n) = O(n)$.

If you have no idea why $\log n = O(n^{1-\varepsilon})$, you can prove it by induction or try to prove a stronger statement that $\log n = o(n^{1-\varepsilon})$. The latter approach requires

$$\begin{aligned}\lim_{n\to\infty} \frac{\log n}{n^{1-\varepsilon}} &= \lim_{n\to\infty} \frac{1/n}{(1-\varepsilon)n^{-\varepsilon}} \\ &= \lim_{n\to\infty} \frac{1}{(1-\varepsilon)n^{1-\varepsilon}} \\ &= 0,\end{aligned}$$

where the first equality is due to L'Hôpital's rule.

3. (a)

$$\overrightarrow{Ap} \times \overrightarrow{AB} = \begin{vmatrix} 3 & 2 \\ -1 & 1 \end{vmatrix} = 5 \geq 0.$$

$$\overrightarrow{Bp} \times \overrightarrow{BC} = \begin{vmatrix} 1 & 1 \\ -2 & -1 \end{vmatrix} = 1 \geq 0$$

$$\overrightarrow{Cp} \times \overrightarrow{CD} = \begin{vmatrix} 0 & -2 \\ -1 & -2 \end{vmatrix} = -2 < 0$$

$$\overrightarrow{Dp} \times \overrightarrow{DA} = \begin{vmatrix} 2 & -1 \\ 1 & 2 \end{vmatrix} = 5 \geq 0$$

The sign of the above cross products implies that $p \notin Q$ and the convex hull of $Q \cup \{p\}$ is $A - B - C - p - D - A$.

(b)

$$\overrightarrow{Ap} \times \overrightarrow{AB} = \begin{vmatrix} 2 & 2 \\ -1 & 1 \end{vmatrix} = 4 \geq 0.$$

$$\overrightarrow{Bp} \times \overrightarrow{BC} = \begin{vmatrix} 0 & 1 \\ -2 & -1 \end{vmatrix} = 2 \geq 0$$

$$\overrightarrow{Cp} \times \overrightarrow{CD} = \begin{vmatrix} -1 & -2 \\ -1 & -2 \end{vmatrix} = 0 \geq 0$$

$$\overrightarrow{Dp} \times \overrightarrow{DA} = \begin{vmatrix} 1 & -1 \\ 1 & 2 \end{vmatrix} = 3 \geq 0$$

The sign of the above cross products implies that $p \in Q$.

4. (a) Algorithm 1 is the pseudocode. The initial call is $opt(n, n, sol[n][n] = \{\infty\})$

```
if sol[a][b] < ∞ then
|   return sol[a][b];
end
if (a, b) equals (1, 1) then
|   return sol[a][b] = A[1][1];
end
if a equals 1 then
|   return sol[a][b] = A[a][b] + opt(a, b − 1, sol);
end
if b equals 1 then
|   return sol[a][b] = A[a][b] + opt(a − 1, b, sol);
end
return sol[a][b] = A[a][b] + min{opt(a − 1, b, sol), opt(a, b − 1, sol)};
```

Algorithm 1: $opt(a, b, sol[n][n])$

(b) Algorithm 2 is the pseudocode. The initial call is $backtrack(n, n, sol[n][n] = \{\infty\})$

```
if (a, b) ≠ (1, 1) then
|   if a equals 1 then
|   |   backtrack(a, b − 1, sol);
|   end
|   if b equals 1 then
|   |   backtrack(a − 1, b, sol);
|   end
|   if a > 1 and b > 1 then
|   |   if sol[a][b] equals sol[a − 1][b] + A[a][b] then
|   |   |   backtrack(a − 1, b, sol);
|   |   else
|   |   |   backtrack(a, b − 1, sol);
|   |   end
|   end
end
print (a, b);
return;
```

Algorithm 2: $backtrack(a, b, sol[n][n])$

(c) The degenerate cases are ignored in this discussion for simplicity. The shortest monotonic path from $(1, 1)$ to (a, b) must visit either $(a - 1, b)$ or $(a, b - 1)$ right before it reaches (a, b). Hence, $opt(a, b) = \min\{opt(a-1, b), opt(a, b-1)\} + A[a][b]$, implying the correctness of *opt* and *backtrack*. $opt(n, n)$ runs in $O(n^2)$ time because there are $O(n^2)$ different subproblems and each needs $O(1)$ time. $backtrack(n, n)$ runs in $O(n)$ time because it visits $O(n)$ subproblems and each needs $O(1)$ time.

5. (a) Let L_i be the collection of increasing subsequences of $A[1..(i-1)]$ whose last element less than $A[i]$. Let $S_A[i]$ be the longest length among all subsequences in L_i. $S_A[1..n]$ can be obtained from the computation of the LIS of $A[1..n]$. Let R be the reverse of array A, i.e. $R[i] = A[n-i]$ for each $i \in [1, n]$. Similarly, we can obtain $S_R[1..n]$ from the computation of the LIS of $R[1..n]$. The desired solution is

$$\max_{1 \leq k \leq n} S_A[k-1] + 1 + S_R[n-k],$$

which can be computed by a linear scan on arrays S_A and S_R. We define that $S_A[0] = S_R[0] = 0$.

   (b) The longest bitonic subsequence can be decomposed into three parts, L, $A[k]$, and R, where L is an IS of $A[1..k-1]$ whose last element is less than $A[k]$ and R is an DS of $A[k+1..n]$ whose first element is less than $A[k]$. By our definition, the length of L is $S_A[k-1]$ and the length of R is length $S_R[n-k]$. If we try all possible $k \in [1, n]$, the return must be the length of the longest bitonic subsequence. The running time is thus $O(n \log n) + O(n \log n) + O(n) = O(n \log n)$.

6. (a) Let $T[1..n][0..m]$ satisfy that

$$T[i][j] = \begin{cases} 1 & \text{if there is a subset of } \{1, \ldots, i\} \text{ whose weight is } j \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 3 is the pseudocode.

```
T[1..n][0..m] ← {0};
T[1][0] ← 1;
if w_1 ≤ m then
    T[1][w_1] ← 1;
end
for i ← 2 to n do
    for j ← 0 to m do
        if T[i − 1][j] equals 1 then
            T[i][j] ← 1;
        end
        if j ≥ w_i and T[i − 1][j − w_i] equals 1 then
            T[i][j] ← 1;
        end
    end
end
return (T[n][k] equals 1) ? “Yes” : “No”;
```

Algorithm 3: *subsetsum*(n, k)

(b) Let $S_{i,j}$ be the collection of all subsets $S \subseteq \{1, 2, \ldots, i\}$ that have weight $\sum_{i \in S} w_i = j$. Let $Q[1..n][0..m]$ satisfy that $Q[i][j] = q > -\infty$ if $S_{i,j} \neq \emptyset$ or otherwise $Q[i][j] = -\infty$, where q is the maximum value $\sum_{i \in S} v_i$ among all $S \in S_{i,j}$.

Algorithm 4 is the pseudocode.

```
1  Q[1..n][0..m] ← {−∞};
2  Q[1][0] ← 0;
3  if w_1 ≤ m then
4  |   Q[1][w_1] ← v_1;
5  end
6  for i ← 2 to n do
7  |   for j ← 0 to m do
8  |   |   if j ≥ w_i and Q[i − 1][j − w_i] > −∞ then
9  |   |   |   Q[i][j] ← Q[i − 1][j − w_i] + v_i;
10 |   |   end
11 |   |   if Q[i − 1][j] > −∞ and Q[i − 1][j] > Q[i][j] then
12 |   |   |   Q[i][j] ← Q[i − 1][j];
13 |   |   end
14 |   end
15 end
16 opt ← −∞;
17 for j ← 0 to m do
18 |   if Q[n][j] > opt then
19 |   |   opt ← Q[n][j];
20 |   end
21 end
22 return opt;
```

Algorithm 4: *knapsack*(n)

(c) Initially, only $T[1][0] = T[1][w_1] = 1$. $T[1][0]$ represents that $\emptyset$ has weight 0 and $T[1][w_1]$ represents that $\{1\}$ has weight w_1. For $i > 1$, we can say $T[i][j] = 1$ iff $T[i-1][j] = 1$ or $T[i-1][j-w_i] = 1$. Each of these two cases denotes whether the i-th stone is contained in the subset or not, respectively. If the i-th stone is not contained in the subset, we need a subset of $\{1, 2, \ldots, i-1\}$ of weight j. Otherwise, the i-th stone is contained in the subset, then we need a subset of $\{1, 2, \ldots, i-1\}$ of weight $j - w_i$. This relation explains why Algorithm 3 correctly solves subproblem (a). The running time of Algorithm 3 is dominated by the loops (Lines 6-15), which need $O(nm)$ time.

Similar argument can applied to explain the correctness of Algorithm 4.