Maximum subsequence sum problem

Given a sequence $(A_1, A_2, \ldots, A_n)$ of (maybe negative) integers,

Goal: find max sum $\sum_{k=i}^{j} A_k$
Answer: $20 = 11 + (-4) + 13$
```java
public static int maxSubSum1(int[] a) {
    int maxSum = 0;
    for (int i = 0; i < a.length; i++) {
        for (int j = i; j < a.length; j++) {
            int thisSum = 0;
            for (int k = i; k <= j; k++) {
                thisSum += a[k];
            }
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```

**Time for loops**

\(0(1) \cdot 0(n) \cdot 0(n) = O(n^3)\)
total: $O(n^3)$

$O$ is an upper bound.

A times inner loop runs:

\[
\begin{align*}
\ell & \leq n - 1 \\
i & = 0 \\
j & = i \\
k & = i
\end{align*}
\]

\[
\frac{n^3 + 3n^2 + 2n}{6} = \Theta(n^3)
\]
There are \( n/4 = \Omega(n) \) iterations of outer loop with \( i < n/4 \).

\( \Rightarrow n/4 = \Omega(n) \) iterations of middle loop with \( j > 3n/4 \).

\( \Rightarrow \) inner loop has \( \geq n/2 = \Omega(n) \) iterations
time is

\[ \Omega(n), \Omega(n), \Omega(n) = \Omega(n^3) \]

\[ O(n^3) + \Omega(n^3) \]

\[ \Rightarrow \Theta(n^3) \]

Usually, don't worry about the \( \Omega \) bound,
public static int maxSubSum2(int[] a) {
    int maxSum = 0;

    for (int i = 0; i < a.length; i++) {
        int thisSum = 0;
        for (int j = i; j < a.length; j++) {
            thisSum += a[j];
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }

    return maxSum;
}

$O(n^2)$: doubly nested for loop
An example of dynamic programming.
Divide-and-conquer:
- divide input into smaller instances of exact same problem
- recursively solve smaller instances by calling our alg on each.

(Try not to overthink; these calls just work!)
"conquer" by combining solutions. maybe a bit more work
Max subsequence sum - divide array into left half and right half of about same size recursively find max sums in left and right halves
- If best subsequence is not entirely in one half, it contains rightmost/last member of left & leftmost first member of right.
```java
private static int maxSumRec(int[] a, int left, int right) {
    if (left == right) { // base case
        if (a[left] > 0) {
            return a[left];
        } else {
            return 0;
        }
    }

    int center = (left + right) / 2;
    int maxLeftSum = maxSumRec(a, left, center); // recurse left,
    int maxRightSum = maxSumRec(a, center + 1, right); // recurse right

    // compute max sum containing last element on left
    int maxLeftBorderSum = 0, leftBorderSum = 0;
    for (int i = center; i >= left; i--) {
        leftBorderSum += a[i];
        if (leftBorderSum > maxLeftBorderSum) {
            maxLeftBorderSum = leftBorderSum;
        }
    }

    // compute max sum containing first element on right
    int maxRightBorderSum = 0, rightBorderSum = 0;
    for (int i = center + 1; i <= right; i++) {
        rightBorderSum += a[i];
        if (rightBorderSum > maxRightBorderSum) {
            maxRightBorderSum = rightBorderSum;
        }
    }

    return max3(maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum);
}

public static int maxSubSum3(int[] a) {
    return maxSumRec(a, 0, a.length - 1);
}
```
Recurrence

A function defined in terms of itself

\[ T(n) \text{: the alg's running time on n elements} \]

\[ T(1) = O(1) \]

\[ T(n) = 2T\left(\frac{n}{2}\right) + O(n) \]
Need to solve recurrence.

\[ T(n) = \Theta(n \log n) \]

Will see how to solve later.
public static int maxSubSum4(int[] a) {
    int maxSum = 0, thisSum = 0;

    for (int j = 0; j < a.length; j++) {
        thisSum += a[j];

        if (thisSum > maxSum) {
            maxSum = thisSum;
        } else if (thisSum < 0) {
            thisSum = 0;
        }
    }

    return maxSum;
}
Given a sorted array of ints \( A[0, \ldots, n-1] \).

A search for \( x \) returns \( i \) s.t. \( A[i] = x \) or -1 if no such \( i \) exists.
Binary search:

Check middle element $A[\text{mid}]$.

If $A[\text{mid}] = x$, return $\text{mid}$.

If $A[\text{mid}] < x$,

If $i < \text{mid}$ then $A[i] \leq A[\text{mid}] < x$, so search right half.
public static <AnyType extends Comparable<? super AnyType> int binarySearch(AnyType[] a, AnyType x) {
    int low = 0, high = a.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (a[mid].compareTo(x) < 0) {
            low = mid + 1;
        } else if (a[mid].compareTo(x) > 0) {
            high = mid - 1;
        } else {
            return mid; // found
        }
    }
    return NOT_FOUND; // defined elsewhere as -1
}

average case? what is time for "average input?" practical but hard to work with
Worst-case:

for $O$ means every case just pick big, true, reasonable bounds for things easier to analyze
Can divide 2 \ln n times, so \(O(\log n)\).

// assumes m >= n
public static long gcd(long m, long n) {
    while (n != 0) {
        long rem = m % n; // remainder from m / n
        m = n;
        n = rem;
    }
    return m;
}

If \(m = 2n\), then \(n\) does not change after one iteration.
Thm: If \( m > n \), then \( n \equiv x \mod m \)
where \( x < \frac{m}{2} \).

\[ \Rightarrow \] \( n \) halves every 2 iterations

\[ \Rightarrow \] \( 2 \log n \) iterations

\[ \Rightarrow \] \( O(\log n) \) time