1 Potential Method

1. **Does it halt?** Suppose *input* is a function which returns a user-specified positive integer. For each of the following problems, do the following tasks:

   - Determine if the following program halts for all possible sequences of (valid) inputs.
   - Either provide a proof of termination for all possible sequences of (valid) inputs or provide a sequence which causes the program to loop.
   - If the program does not halt, explain why the potential method does not apply.

(a)

```
1: x ← input()
2: y ← input()
3: while x > 0 and y > 0 do
4:     z ← input()
5:     if z is even then
6:         x ← x − 1
7:         y ← y + 1
8:     else
9:         y ← y − 1
```

Solution: We claim that the value of $2x + y$ serves as a potential for this program.

Claim: If the program doesn't exit the while loop on line 3, then $2x + y > 0$ (at that moment).

If the program doesn't exit the while loop, then we have $x > 0$ and $y > 0$. Thus $2x + y > 0$.

Claim: In each iteration of the while loop, $2x + y$ is strictly decreasing.

Suppose z is even. Then the value of the potential changes to $2(x - 1) + y + 1 = 2x + y - 1 < 2x + y$.

Suppose z is odd. Then the value of the potential changes to $2x + y - 1 < 2x + y$. In either case, the value of the potential function decreases by 1.

Now suppose the program never halt. Then by the two claims above, $2x+y$ is always greater than zero and it always decreases by 1 at each iteration, which is absurd. In other words, the program must always halt.

(b)

```
1: x ← input()
2: y ← input()
3: while x > 0 and y > 0 do
4:     z ← input()
5:     if z is even then
6:         x ← x − 1
7:         y ← y + 1
8:     else
9:         y ← y − 1
10:        x ← x + 1 // This line differs from the program in (a)
```

Solution: The sequence $x \leftarrow 2$, $y \leftarrow 2$, and $z_i \leftarrow i$ (where z_i denotes the i-th input for z) causes the program to loop.

Observe that if the potential method did apply, we could use the function specified by the potential method to show that the program halts for all possible sequences of valid inputs.

For example, if we take the potential function to be $2x+y$ as with the program from part (a), we can still see that the potential has a lower bound of 0.

However, the value of the potential function is not strictly decreasing over all sequences of inputs. Consider the sequence $x \leftarrow 2$, $y \leftarrow 2$, and $z_i \leftarrow i$ (where z_i denotes the i-th input for z). Then on the first iteration the value of the potential changes from $6 = 2(2) + 2$ to $7 = 2(3) + 1$.

In general for any potential function, the value is the same after any pair of iterations where z is even in one and odd in the other, since the values of x and y are the same before and after those two iterations. Thus, the value does not strictly decrease for any potential function.

(c)

```
1: x ← input()
2: y ← input()
3: while x > 0 or y > 0 do // This line differs from the program in (a)
4:     z ← input()
5:     if z is even then
6:         x ← x − 1
7:         y ← y + 1
8:     else
9:         y ← y − 1
```

Solution: The sequence $x \leftarrow 1$, $y \leftarrow 1$, and $z_i \leftarrow 1$ for all i causes the program to loop.

Observe that if the potential method did apply, we could use the function specified by the potential method to show that the program halts for all possible sequences of valid inputs.

For example, if we take the potential function to be $2x+y$ as with the program from part (a), by the analysis from the first part of the problem, the value of the potential function is strictly decreasing over all sequences of inputs.

However, the value of the potential function is not bounded from below. In other words, there is an assignment of x and y such that $2x + y$ is arbitrarily small, but at least one of x and y is greater than 0. Regardless of the choice of z the value of $2x+y$ decreases by 1 every iteration. However, if we choose only z even, then y will always be greater than 0 so the loop condition always holds true.

More generally, any valid potential function must decrease in each iteration. For this to hold in the input sequence above, the value of the potential function must decrease when x is constant but the value of y decreases. However, this algorithm has no lower bound on the value of y when x is held constant at a positive value, so

there is no lower bound on the value of any such potential function either. Thus, it does not meet the requirements of a valid potential function.

2 Divide and Conquer

1. **Majority elements.** Given an array A of n integers, where n is a power of 2, a *majority element* of A is an element in A that appears strictly more than $\frac{n}{2}$ times. The algorithm MajorityElement defined below finds the majority element of A if it exists. If A has a majority element, MajorityElement will return that element. Otherwise, MajorityElement will return $\emptyset$.

```
1:  function MajorityElement(A[1, 2, ..., n]))
2:      if n = 1 then return A[1]
3:      x ← MajorityElement(A[1, ..., ⌊n/2⌋])
4:      y ← MajorityElement(A[⌊n/2⌋ + 1, ..., n])
5:      if x ≠ ∅ then
6:          iterate over A, counting the number of occurrences of x
7:          if the number of occurrences of x in A is > n/2 then return x
8:      if y ≠ ∅ then
9:          iterate over A, counting the number of occurrences of y
10:         if the number of occurrences of y in A is > n/2 then return y
11:     return ∅
```

(a) Analyze the time complexity of MajorityElement and give the asymptotic time complexity as a closed-form solution.

> **Solution:** Let $T(n)$ be the number of steps the algorithm takes on an array of size n. Lines 3 and 4 each recursively apply the algorithm on subarrays of size $\frac{n}{2}$, giving us a total of $2T(\frac{n}{2})$. Lines 6 and 9 iterate over the entire array, taking $O(n)$ time. The remaining operations take constant time. Thus, we have the recurrence $T(n) = 2T(\frac{n}{2}) + O(n)$. Applying the Master Theorem with $k = 2, b = 2, d = 1$ gives us $T(n) = O(n \log n)$.

(b) Show the correctness of the MajorityElement algorithm by proving the following statement:
If z is a majority element of an array A of n integers, then z must be a majority element of at least one of the subarrays $A[1, \ldots, \frac{n}{2}]$ and $A[\frac{n}{2} + 1, \ldots, n]$.

> **Solution:** Suppose z is a majority element of A. For the purposes of contradiction, assume that z is not a majority element of $A[1, \ldots, \frac{n}{2}]$ nor of $A[\frac{n}{2} + 1, \ldots, n]$. Let a be the number of occurrences of z in $A[1, \ldots, \frac{n}{2}]$. Since z is not a majority, element, $a \leq \frac{n}{2}/2 = \frac{n}{4}$. Similarly, if b is the number of occurrences of z in $A[\frac{n}{2} + 1, \ldots, n]$, $b \leq \frac{n}{4}$. Then the total number of occurrences t of z in A is $t = a+b \leq \frac{n}{4}+\frac{n}{4} = \frac{n}{2}$. This contradicts that z is a majority element of A, since that requires that the number of occurrences be $t > \frac{n}{2}$. Thus, the assumption that z is not a majority element of $A[1, \ldots, \frac{n}{2}]$ nor of $A[\frac{n}{2} + 1, \ldots, n]$ must be false, and the statement is proved. □

2. **Array Local Minimum**. Let $A[1, \ldots, n]$ be an array of distinct integers, where $n = 2^k$ for some $k \in \mathbb{N}$. The integers in the array are not sorted in any particular order. A cell $A[i]$ is a

local minimum if $A[i] < A[i-1]$ and $A[i] < A[i+1]$. (If $i = 1$ or $i = n$, it only needs to be smaller than the adjacent cell.)

Devise a divide and conquer algorithm to find a local minimum in this array in $O(\log n)$ time.

Solution: General idea: Find the middle element in the array, and check if it is the local minimum by comparing it to its two neighbors. If it is, return it. Else, compare the two neighbors and pick the element that is smaller. Recurse on that half until the local minimum is found. For the base case, if the array only contains one element, just return that element.

Input: An array $A[1, ..., n]$ of distinct integers
Output: LOCAL-MIN(A)

```
1: function LOCAL-MIN(A[1...n])
2:     if size(A) = 1 then
3:         return A[1]
4:     if size(A) = 2 then
5:         return min(A[1], A[2])
6:     middle ← size(A)/2
7:     if A[middle] < A[middle − 1] and A[middle] < A[middle + 1] then
8:         return A[middle]
9:     if A[middle] > A[middle − 1] then
10:        return LOCAL-MIN(A[1, ..., middle − 1])
11:    else
12:        return LOCAL-MIN(A[middle + 1, ..., n])
```

Note the solution here uses 1-indexing. Our recurrence relation for the runtime of this algorithm is $T(n) = T(n/2) + O(1)$, So by the Master theorem with $k = 1, b = 2, d = 0$, we get a closed form solution of $T(n) = O(\log n)$ as desired. Intuitively, since the algorithm divides the array in half at each recursive step, the algorithm runs in $O(\log n)$ time.

Now we show that this algorithm will correctly find a local minimum. We have two base cases:

- If there is one element in the array, then that is a local minimum. The algorithm returns this element, which is the correct behavior.
- If there are two elements in the array, then one must be smaller than the other, as elements are distinct. This smaller element is a local minimum, and this is the value returned by the min function.

Now consider an array of size $n \geq 4$, and assume using strong induction that our algorithm correctly returns a local minimum for an array of size $1, 2, ...n - 1$.

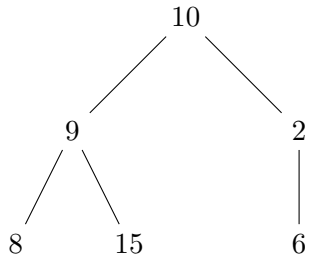
Let $m = n/2$. Note that at this size, we're guaranteed that $A[m]$ has an element on both its left and right. There are two possible cases for the middle element of the array:

1. $A[m] < A[m-1] \leq A[m+1]$: In this case, $A[m]$ is a local minimum, and is correctly returned by the function
2. $A[m] > A[m-1]$ or $A[m] > A[m+1]$: In this case, the algorithm recurses to an array of half the size, picking a side where $A[m]$'s adjacent cell is smaller. We know by our inductive hypothesis that the algorithm correctly returns on a call of size $n/2$. We also know that a solution to a subarray must be a solution to the original array because we only consider subarrays where the element after the right/left end is larger.

This concludes our proof. We've written a correct algorithm to find a local minimum in an array in $O(\log n)$ time.

3. **Binary Tree Local Max.** A complete binary tree is a binary tree in which every level, except possibly the last is completely filled and all nodes in the last level are as far left as possible.

   Consider a complete binary tree $T = (V, E, r)$ rooted at r where each vertex is labelled with a distinct integer. A vertex $v \in V$ is called a *local maximum* if the label of v is greater than the label of each of its parent and children.



In this example, the local maxima are 10, 15, and 6.

Suppose you are given such a tree where the labelling is given implicitly, i.e., the only way to determine the label of the vertex v is to visit v and query for the vertex label. Provide an algorithm that computes a local maximum of T with using $O(\log(|V|))$ vertex label queries.

Solution: We provide an algorithm that computes a local maximum of T starting from the root of T. The main idea is to start a tree traversal from the root r, check if we have already found a local maximum of T, and if not, recursively attempt the strategy on one of the subtrees rooted at the children of r.

More specifically, our algorithm is as follows,

Input: Complete, rooted, vertex labelled, binary tree $T = (V, E, r)$
Output: Local maximum v^*

```
1: function ComputeLocalMaximum(T = (V, E, r))
2:     if the label of r is greater than that of all its children's then
3:         return r
```

4: **else**
5: Let r' be a child of of r with a label greater than r
6: Let T' be the complete, rooted, vertex labelled, binary tree induced by r'
7: **return** ComputeLocalMaximum($T' = (V', E', r')$)

What remains to be shown is that the algorithm actually computes a local maximum of T and that it uses $O(\log(|V|))$ vertex label queries.

The algorithm maintains the invariant that the label of root of the (sub-)tree under consideration is greater than its parent's label (if it exists). Additionally, the root is returned if and only if the label of the root is greater than both of its children's labels. So the root is returned if and only if the label of the root is greater than all of its neighbors - in other words, if and only if the root is a local maximum.

To see that the algorithm uses at most $O(\log(|V|))$ vertex label queries, observe that per level of the tree, the algorithm uses at most 3 vertex label queries. There are $O(\log(|V|))$ levels, because T is a complete binary tree, so the number of used vertex label queries is indeed $O(\log(|V|))$.

4. **Multiplication Divide and Conquer.** Karatsuba's algorithm is a fast multiplication algorithm, which allows computing the product of two large numbers x and y using three multiplications of smaller numbers, each with half as many digits as x or y, plus some additions and digit shifts. In this question, we want to make a small change. The number of digits that each smaller number has is only one third of x or y. We will assume for this question that n is an integer power of 3.

Let x, y be two n-bit integers, and write $\begin{cases} x = a \cdot 2^{\frac{2n}{3}} + b \cdot 2^{\frac{n}{3}} + c \\ y = d \cdot 2^{\frac{2n}{3}} + e \cdot 2^{\frac{n}{3}} + f \end{cases}$. Then we have:

$$xy = ad \cdot 2^{\frac{4n}{3}} + (ae+bd)2^n + (af+be+cd)2^{\frac{2n}{3}} + (bf+ce)2^{\frac{n}{3}} + cf = A2^{\frac{4n}{3}} + B2^n + C2^{\frac{2n}{3}} + D2^{\frac{n}{3}} + E,$$

where

$$\begin{aligned} A &= ad \\ B &= ae + bd \\ C &= af + be + cd \\ D &= bf + ce \\ E &= cf \end{aligned}$$

There are two Divide-and-Conquer based algorithms described as below. In Algorithm $\mathcal{A}$, computing each value of $ad, ae, af, bd, be, bf, cd, ce, cf$ can be regarded as a sub-problem. We first solve the sub-problems and then compute xy by merging the solutions of sub-problems to get A, B, C, D, E based on the equations above. In Algorithm $\mathcal{B}$, we compute A, B, C, D, E by computing X_0, X_1, X_2, X_3, X_4 below, where the multiplication in each of them is computed recursively:

$$\begin{cases} X_0 = cf \\ X_1 = (a+b+c)(d+e+f) \\ X_2 = (4a+2b+c)(4d+2e+f) \\ X_3 = (a-b+c)(d-e+f) \\ X_4 = (4a-2b+c)(4d-2e+f) \end{cases} \Longrightarrow \begin{cases} A = \frac{1}{24}(6X_0 - 4X_1 + X_2 - 4X_3 + X_4) \\ B = \frac{1}{12}(-2X_1 + X_2 + 2X_3 - X_4) \\ C = \frac{1}{24}(-30X_0 + 16X_1 - X_2 + 16X_3 - X_4) \\ D = \frac{1}{12}(8X_1 - X_2 - 8X_3 + X_4) \\ E = X_0 \end{cases}.$$

For each algorithm $\mathcal{A}$ and $\mathcal{B}$, write down the recurrence relation $T(n)$ for the time complexity, and then compute $T(n)$ explicitly. Which of the algorithms $\mathcal{A}$ and $\mathcal{B}$ runs asymptotically faster? How do they compare (asymptotically) to Karatsuba's Algorithm?

Note: Dividing an n-bit integer by 3 can be done in $O(n)$ time, if the dividend is known to be a multiple of 3, and you can assume this for this question.

Solution:

- Algorithm $\mathcal{A}$

  We can see that there are 9 multiplications of numbers that are about $n/3$ bits long. We can do addition in linear time and multiplication by $2^{O(n)}$ can be done in linear time by bit shifting.

  $T(n) = 9T(n/3) + O(n)$

  By the Master Theorem, since $9 > 3$, $T(n) = O(n^{\log_3 9}) = O(n^2)$

- Algorithm $\mathcal{B}$

  We can see that there are 5 multiplications of numbers that are about $n/3$ bits long. We can do addition and subtraction in linear time and multiplication by constants in also linear time. Also, multiplication by $2^{O(n)}$ can be done in linear time by bit shifting.

  $T(n) = 5T(n/3) + O(n)$

  By the Master Theorem, since $5 > 3$, $T(n) = O(n^{\log_3 5}) \approx O(n^{1.46})$

The algorithm $\mathcal{B}$ runs asymptotically faster. Karatsuba's algorithm runs in $O(n^{\log_2 3}) \approx O(n^{1.58})$ so the algorithm $\mathcal{B}$ runs faster than Karatsuba's algorithm but the algorithm $\mathcal{A}$ runs slower than Karatsuba's algorithm.

3 Master Theorem

1. **Mergesort.** Recall that MergeSort has the recurrence relation

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

By using the Master Theorem, determine the runtime of MergeSort algorithm.

Solution: Since $k/b^d = 2/2^1 = 1$, we use the second case of Master Theorem, which gives $O(n \log n)$

2. **Slowsort.** Consider the sorting algorithm *slowsort*, which can be represented with the following pseudocode.

What is the most precise recurrence relation for the time complexity? What does the Master Theorem give for this relation?

```
1: function SLOWSORT(A[1, 2, ..., n]) // n is length of A
2:     SLOWSORT(A[1, ..., ⌊n/2⌋]) // sort both halves of the array recursively
3:     SLOWSORT(A[⌊n/2⌋ + 1, ..., n])
4:     if A[⌊n/2⌋] > A[n] then // largest item in first half is greater than largest in the second
5:         swap A[⌊n/2⌋] and A[n] // put largest item in the unsorted array at the end
6:     SLOWSORT(A[1, ..., n − 1]) // sort the entire array minus one element recursively
7:     return
```

- ◯ $T(n) = 2T\left(\frac{n}{2}\right) + T(n-1) + O(1)$; the Master Theorem gives $T(n) = O\left(n^2 \log n\right)$
- √ $T(n) = 2T\left(\frac{n}{2}\right) + T(n-1) + O(1)$**; the Master Theorem is unusable**
- ◯ $T(n) = 3T\left(\frac{n}{2}\right) + O(1)$; the Master Theorem is unusable
- ◯ $T(n) = 3T\left(\frac{n}{2}\right) + O(1)$; the Master Theorem gives $T(n) = O\left(n^{\log_2 3}\right)$
- ◯ This cannot be written as a relation; the Master Theorem is unusable

Solution: Lines 2 and 3 each take $T\left(\frac{n}{2}\right)$ steps. Line 6 takes $T(n-1)$ steps. The remaining lines do constant work. The resulting recurrence $T(n) = 2T\left(\frac{n}{2}\right) + T(n-1) + O(1)$ is not in the form required by the Master Theorem because of the $T(n-1)$ term.

3. **Master Theorem with Log Factors.** Refer to Master Theorem with Log Factors, provide a big-O bound for

$$T(n) = 9T(n/3) + n^2 \log^2 n$$

Solution: We have $k = 9, b = 3, d = 2, w = 2$; $\log_3 9 = 2 = d$, so we are in the second case of the Master theorem. $T(n) = O(n^2 \log^3 n)$.