In-Class Activity #1

Vancouver Summer Program
Algorithms

---

1. **Searching in a convex polygon.** A convex polygon has all its internal angles less then 180 degrees and no two edges cross each other. Such a polygon can be stored using an array A of (x, y) coordinates; each element of the array contains one (x, y) coordinate corresponding to one corner of the polygon. Let $A[1]$ correspond to the vertex with the smallest x coordinate, and let $A[1], \ldots, A[n]$ be arranged counter-clockwise. For simplicity, assume that the x coordinates and the y coordinates are all distinct.

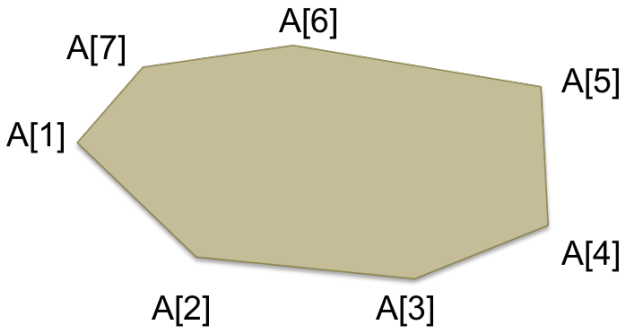

Figure 1: Example of a convex polygon with $n = 7$.

Develop an algorithm that finds the vertex with the largest x coordinate in $O(\lg n)$ time. You must prove that your algorithm has a runtime complexity of $O(\lg n)$.

Solution. Notice that with the counter-clockwise ordering of the corners of the polygon, convexity implies that the x coordinates first increase up to a maximum x, then they start to decrease back to the starting point. That is, array A is sorted in such a way that for some $i \in [n]$, $A[1].x < A[2].x \cdots < A[i-1].x > A[i].x > A[i+1].x > \cdots > A[n].x$, and our task is to find the this i. The x coordinates in their ordering in A, taken in isolation, can be viewed as a triangle, and our task is to find the location of its peak. Both subarrays around the peak x value are sorted, so we may a procedure very similar to binary search (where we compare $A[m]$ and $A[m-1]$ instead of needle and list$[m]$ in updating the L and R indexes).

2. (The aabbb Language) In language aabbb, all words consist of "aa"s or "bbb"s or some combination of these two patterns. Devise an algorithm that takes as input an integer $n \geq 0$ and computes the number of words of length n in this language.

Solution. By default, there are no words of 0-length and this language has no words of length 1. Let f_n denote the number of strings of length n that are combinations of aa and bbb. Then $f_0 = f_1 = 0$, $f_2 = 1$ (aa), and $f_3 = 1$ (bbb). Experimenting with values of f_n for sufficiently large n, you will see that

$$f_n = f_{n-2} + f_{n-3}, \qquad n \geq 4.$$

To see this, note that every string that is a combination of "aa"s and "bbb"s must end in either aa or bbb. Thus, the number of strings of length n that are combinations of "aa"s and "bbb"s is the number of strings of length $n-3$ that are combinations of "aa"s and "bbb"s with bbb appended to the end (here one cannot append aa because the resulting strings will have length $n-1$), plus the number of strings of length $n-2$ that are combinations of "aa"s and "bbb"s with aa appended to the end. Note that appending bbb to the end of the f_{n-3} distinct strings of length $n-3$ that satisfy our property produces f_{n-3} distinct strings of length n. The same holds for f_{n-2} when appending aa. Also, note that the strings of length n produced by both append operations are distinct. This gives the recurrence relation.

3. Consider a variant of MERGESORT wherein the input array is divided into 3 almost equally sized thirds. Assume for simplicity that $n \geq 3$. Assume that you have a procedure MERGE that takes 3 different sorted arrays and correctly combines them into one sorted array, and runs in time cn (such a procedure indeed exists.) Write the pseudo-code for this variant and analyze its running time. To solve the running-time recurrence, you may utilize the following result.

   The Master Theorem. The recurrence $T(n) = aT(n/b) + f(n)$ can be solved as follows:

   - If $af(n/b) = \kappa f(n)$ for some constant $\kappa < 1$, then $T(n) \in \Theta\big(f(n)\big)$
   - If $af(n/b) = Kf(n)$ for some constant $K > 1$, then $T(n) \in \Theta\big(n^{\log_b a}\big)$
   - If $af(n/b) = f(n)$, then $T(n) = \Theta\big(f(n) \log_b n\big)$
   - If none of these three cases apply, you're on your own.

   Are there any efficiency gains (asymptotically) in having finer equally-sized (but constant, i.e., not depending on n) splittings of the input array? What if the number of divisions is a function of n?

   Solution. The algorithm is MERGESORT3 (below).

```
MERGESORT3(A[1...n])
if n > 1
    m1 ← ⌊n/3⌋
    m2 ← ⌊2n/3⌋
MERGESORT3(A[1...m1])
MERGESORT3(A[m1 + 1...m2])
MERGESORT3(A[m2 + 1...n])
MERGE3(A[1...m1], A[m1 + 1...m2], A[m2 + 1...n])
```

   Here one or both of MERGE3's input arrays might be empty: If $n = 2$, then $m_1 = 0$ and $m_2 = 1$, so $A[1 \ldots m_1] = A[1 \ldots 0] = \emptyset$, $A[m_1 + 1 \ldots m_2] = A[1 \ldots 1] = A[1]$, and $A[m_2 + 1 \ldots n] = A[2 \ldots 2] = A[2]$.

   Running-time: We have $T(n) = 3T(n/3) + cn$. With $f(n) = cn$ and $a = b = 3$ in the Master Theorem, we conclude that $T(n) \in \Theta(n \log n)$. This is the same asymptotic complexity as the 2-way splitting MERGESORT, so there is no performance gain in splitting the array into finer (equally-sized) subarrays.

4. (Regrouping.)

   BC's Ministry of Advanced Education has built up a list of students enrolled at UBC, SFU and UVic. The list is essentially an array of objects, each object representing a student. The list is currently sorted by the GPA of the students, across the three institutions. The ministry would now like to rearrange the students such that the array contains all UBC students first, then all SFU students, and finally all UVic students. (The students no longer need to be sorted by GPAs.) The ministry's computing systems are out-dated and do not have additional memory. Develop an *efficient* algorithm to regroup students as specified without using an additional array. (The problem can easily be solved in $\Theta(n \lg n)$ time in the worst case. Can you do better?)

   Solution. This problem can be solved in $\Theta(n)$ time as follows.

   We will define two simple constant-time operations to simplify the explanation of the algorithm. Examine(i) returns the institution of the student at position i in the list. Swap(i, j) swaps the entries at locations i and j of the list.

We will use a to represent the position of the last UBC student in the list, and b to represent the position of the last SFU student in the list, and c to represent the position of the last UVic student in the list. (Eventually, we will want all UBC students to have indices $1 \ldots a$, all SFU students to have indices $a+1 \ldots b$ and all UVic students to have indices $b+1 \ldots c$. We can initially assume $a = b = c = 0$.

We will use index i to represent our current position in the list, which we can initialize to 1 (the start of the list).

Our algorithm then proceeds as follows as long as $i \leq n$:

- If Examine(i) returns UVic: increment c and i.
- If Examine(i) returns SFU: Swap$(i, b+1)$ and increment b, c and i.
- If Examine(i) returns UBC: Swap$(i, b+1)$ then Swap$(b+1, a+1)$ and increment a, b, c and i.

Running time: This algorithm runs in $\Theta(n)$ time because each of the n entries in the array is examined once and after each examination a constant number of constant-time operations are performed.

Proof of correctness: The algorithm iterates through the list. At the start of each loop iteration, we have the following invariant that is preserved: Elements $1 \ldots a$ are UBC students, elements $a+1 \ldots b$ are SFU students and elements $b+1 \ldots c$ are UVic students and $i = c + 1$ represents the index of the next student to process. If we show that this invariant is true after each iteration then we have the desired list when $i = n + 1$.

To show that the invariant holds after each iteration, let us consider the three cases that may occur within each iteration.

Case 1. If the student record at index i represents a UVic student then we increment c and i. The invariant holds after this operation.

Case 2. If the student record at index i represents an SFU student then we swap the entries at locations $b+1$ and i. Now entries $a+1 \ldots b+1$ are SFU students and entries $b+2 \ldots c+1$ are UVic students. (The UBC students are untouched.) After incrementing b, c and i the invariant holds.

Case 3. If the student record at index i represents a UBC student then we swap the entries at locations $b+1$ and i and then we swap the entries at locations $a+1$ and $b+1$. Now entries $1 \ldots a+1$ are UBC students, entries $a+2 \ldots b+1$ are SFU students and entries $b+2 \ldots c+1$ are UVic students. (The UBC students are untouched.) After incrementing a, b, c and i the invariant holds.

The invariant holds after each loop iteration and that is sufficient to conclude that the algorithm produces the desired sorted list. ■