

In-Class Activity #1

Vancouver Summer Program

Algorithms

1. **Searching in a convex polygon.** A convex polygon has all its internal angles less than 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 $A[1], \dots, A[n]$ be arranged counter-clockwise. For simplicity, assume that the x coordinates and the y coordinates are all distinct.

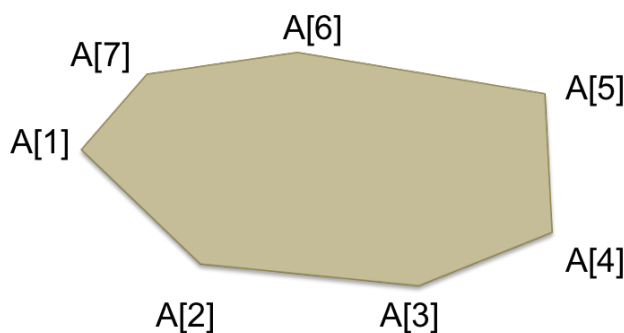


Figure 1: Example of a convex polygon with $n = 7$.

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 $O(\lg n)$.

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”s and “bbb”s. 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

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

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 $T(n) \in \Theta(f(n))$
- If $af(n/b) = Kf(n)$ for some constant $K > 1$, then $T(n) \in \Theta(n^{\log_b a})$
- If $af(n/b) = f(n)$, then $T(n) = \Theta(f(n) \log_b n)$
- 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( $A[1 \dots n]$ )
if  $n > 1$ 
     $m_1 \leftarrow \lfloor n/3 \rfloor$ 
     $m_2 \leftarrow \lfloor 2n/3 \rfloor$ 
    MERGESORT3( $A[1 \dots m_1]$ )
    MERGESORT3( $A[m_1 + 1 \dots m_2]$ )
    MERGESORT3( $A[m_2 + 1 \dots n]$ )
    MERGE3( $A[1 \dots m_1], A[m_1 + 1 \dots m_2], A[m_2 + 1 \dots n]$ )

```

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

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?)