CSE 321 - Homework 4

Due date: 25/12/2022, 23:59

1. In this question, we are recursively looking for the best possible score. The recursive algorithm computes 2 different scores at each step: the score if we continue by stepping right and the score if we continue by stepping down. Then we choose the maximum value.
   To go from the top left to the bottom right, we should take $n + m - 1$ steps, if we include the first and the last step. For each step, we either accept it or not (according to the score that path generates). Therefore the time complexity is $O(2^{n+m})$.

2. Finding the median is easily done by sorting the array. But we just want to find the element in the middle, not the sorted version of the whole array. So, we can find the elements that should be on the left (or right) side of the median, and then the next minimum element should be our median. The algorithm in the .py file works as follows:

   - Save the frequency of each item, done in $O(n)$.
   - Initialize a variable as 0 (to keep the number of elements on the left side of the median). Let's call it sum.
   - Find the minimum element (takes linear time) in the array and reassign sum as $sum + frequency(min\_element)$.
   - Until $sum >= \frac{n}{2}$ (where n is the length of the array) repeat the previous step.
   - When $sum >= \frac{n}{2}$, return the most recent minimum element that you have found (if n is not even, calculate the average of two elements in the middle).

   Simply, none of the operations takes longer than linear time. But we repeat a step that takes linear time again and again until a condition ($sum >= \frac{n}{2}$) holds. In the worst case, we have to check all $\frac{n}{2}$ elements on the left side of the median. In this case, the time complexity becomes $O(n * \frac{n}{2}) = O(n^2)$, while the best-case time complexity is $O(n)$. On average, this algorithm would work faster than sorting the array and then finding the median, if there are repetitive elements.

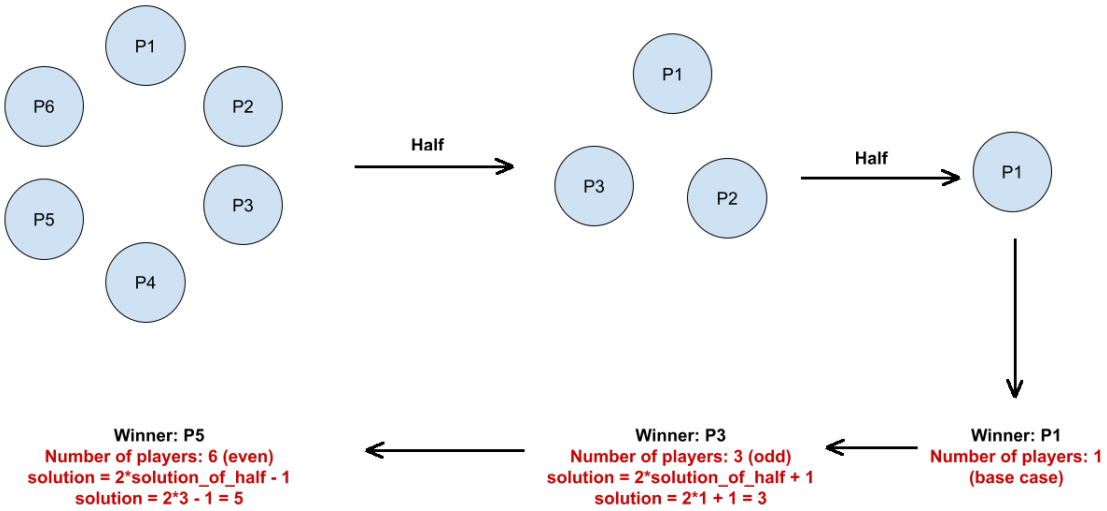
3. (a) In a circular linked list we can use a loop and eliminate a player at each step. In this way, we can achieve the solution in $n - 1$ steps (because we have to eliminate $n - 1$ players). Simply, we start with the first player and eliminate the next one. Then the third player makes their move so on and so forth. Since this operation takes $n - 1$ steps, the worst-case time complexity is $O(n)$.

   (b) We can solve this problem by a decrease-by-half algorithm. Simply, solving half of the problem is enough. We can then calculate the solution for the main problem by using the solution of half problem (this calculation takes constant time). But this calculation might be tricky.

   - If n is an even number, then the solution is going to be $2 * solution\_of(first\_half) - 1$. Because the winner of the first half will eliminate the first element of the second half (In the example below, P3 will eliminate P4). Therefore the solution will be shifted by 1.

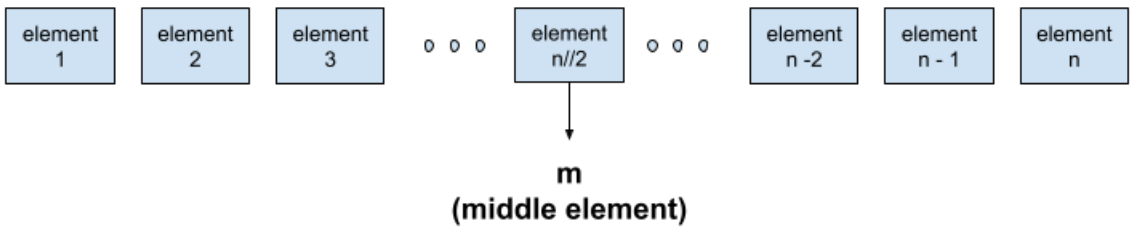
- If n is an odd number, then the solution is going to be $2 * solution\_of(first\_half) + 1$. Because this time, the winner of first half will eliminate the winner of the second half, then the last player will eliminate the winner of the first half. (In the example below, for P1, P2, P3: P1 is first half and P2 is second half. First half eliminates the second half and then the last element, P3, eliminates the first half).

Example:



Since the problem solves only half of the input size at each step, the worst-case time complexity is $O(logn)$.

4. • **Binary search:**



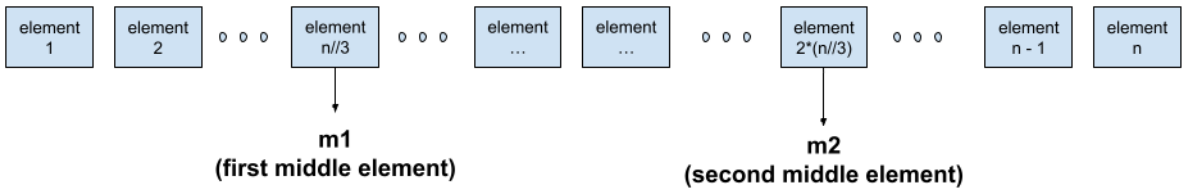
To find a $value$ we make at most 2 calculations at each step:

(a) $m == value$?

(b) If not, $m > value$? (or $m < value$?)

So the recurrence equation looks like this:
$T(n) = T(\frac{n}{2}) + 2c$
Thus, the time complexity is $O(2log_2 n)$

- **Ternary search:**



To find a $value$ we make at most 4 calculations at each step:

(a) $m_1 == value$?

(b) If not, $m_2 == value$?

(c) If not, $m_1 > value$?

(d) If not, $m_2 < value$?

So the recurrence equation looks like this:
$T(n) = T(\frac{n}{3}) + 4c$
Thus, the time complexity is $O(4log_3 n)$

- **Comparison:** Ternary search looks faster than binary search since $log2_n < log_3 n$ but at each step, ternary search makes 2 times more comparisons than binary search. If we compare their running times, we obtain $\frac{2log_2 n}{4log_3 n} = \frac{log_2 n}{2log_3 n}$. Since $2log_3 n > log_2 n$, we find $2log_2 n < 4log_3 n$ which means the time complexity of binary search is less than the time complexity of ternary search. When we divide the array into more pieces, it looks like we are going to work on a smaller part of the array and therefore the time complexity will be better. But, to decide on which part of the array we should continue, we make more comparisons. Therefore, time complexity increases. If we try to divide the array into n elements, it means that we will have $n - 1$ middle elements. And we will compare each of them with the value we are looking for. The complexity becomes $2 * (n - 1) * log_n n$. Even though $log_n n = 1$, the time complexity ends up linear. This is the reason for not dividing the array into smaller parts (dividing it into n pieces simply generates a linear search algorithm).

5. (a) The best-case happens when our approximation is the value we are looking for. But this happens only if the array is uniformly distributed. The time complexity is constant, $O(1)$.

   (b) Binary search simply divides the problem into 2, at each step. To decide its next step, it goes to the middle of the array first. Interpolation search, on the other hand, goes to any location according to the value we are looking for. Basically, binary search tries to find the part of the array that the value we are looking for presents. But ternary search tries to find it at once, according to the distribution of the elements in the array. Their best-case time complexities are the same, but in the worst-case the interpolation search takes linear time.