VIETNAM NATIONAL UNIVERSITY - HCM
Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

COMPUTER ARCHITECTURE (CO2008)

Topic

MIPS
Quicksort and Binary Search

Instructor: Mr. Phạm Quốc Cường
Students: Trần Quốc Anh - 1852247

Ho Chi Minh City, Dec 2019

Outline

1 Problem

.data
nums.word <*an integer number*>
elems.word <*array elements*>

- Where <*an integer number*> will store the number of elements in the array elems. elems is an integer array whose size is equal to value <*an integer number*>. You are required to choose those values when developing and testing your program.
  - 1. Write a MIPS program that sort the the array elems in descending order using the quick sort algorithm.
    2. Calculate the execution time of your program if one instruction requires 1 ns for processing.
  - 1. Sort the the array elems in ascending order at first;
    2. Allow users to input an integer number. The program reports position(s) of the number if it exists in the array. Binary search mus be used in this step.

2 Theorem

a) Quicksort:

- Definition

Quicksort (sometimes called **partition-exchange sort**) is an efficient sorting algorithm, serving as a systematic method for placing the elements of a random access file or an array in order. Developed by British computer scientist Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heap-sort.[1]

Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. Efficient implementations of Quicksort are not a stable sort, meaning that the relative order of equal sort items is not preserved. Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting. It is very similar to selection sort, except that it does not always choose worst-case partition.[1]

Mathematical analysis of quicksort shows that, on average, the algorithm takes O(n.log(n)) comparisons to sort n items. In the worst case, it makes O(n^2) comparisons.[1]

- Algorithm

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array

into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are:

1. Pick an element, called a *pivot*, from the array.
2. *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. *Recursively* apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

- Pseudo Code

Quicksort process can be divided into two part:

+ Partition:

- Step 1: Choose the highest index value has pivot.
- Step 2: Take two variables to point left and right of the list excluding pivot.
- Step 3: Left points to the low index.
- Step 4: Right points to the high.
- Step 5: While value at left is less than pivot move right.
- Step 6: While value at right is greater than pivot move left.
- Step 7: If both step 5 and step 6 does not match swap left and right.
- Step 8: If left greater or equal right, the point where they met is new pivot.

```
partition(array<int>, low<int>, high<int>)
.       pivot = array[high]
.       left = low
.       right = high - 1
.       while true do
.       .       while left <= right && arr[left] < pivot do
.       .       .       left = left + 1
.       .       end
.       .       while right >= left && arr[right] > pivot do
.       .       .       right = right - 1
.       .       end
.       .       if left >= right then
.       .       .       break
.       .       end
.       .       swap(arr[left], arr[right])
.       .       left = left + 1
.       .       right = right - 1
.       end
end partition
```

+ Quicksort:

- Step 1: Make the right-most index value pivot.
- Step 2: Partition the array using pivot value.
- Step 3: Quicksort left partition recursively.
- Step 4: Quicksort right partition recursively.

```
binarySearch(array<int>, key<int>)
.       if left < right then
.       .       pivot = partition(pivot, left, right)
.       .       quicksort(array, low, pivot - 1)
.       .       quicksort(array, pivot + 1, high)
.       end
end quicksort
```

b) Binary Search:

- Definition

In computer science, **binary search** is a type of method used for finding a certain element in a sorted list. It is an algorithm that is more efficient than linear

search, which will go through every element in the list of items to find a particular element. However, the given list of elements must be sorted first before a binary search algorithm can be used.[3]

The binary search algorithm works by repeatedly splitting the sorted list into two and working on the part of the list that may contain the element that you are looking for, until the final list contains only one element.[3]

- Algorithm

Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.[4]

- Pseudo code

+ Binary Search:

```
binarySearch(array<int>, n<int>, key<int>)
.       left = 0
.       right = n - 1
.       while left <= right do
.       .       mid = (left + right) / 2;
.       .       if key == array[mid] then
.       .       .       return mid
.       .       else if key < array[mid]
.       .       .       right = mid - 1
.       .       else left = mid + 1
.       .       end
.       end
end quicksort
```

3 Solving in MIPS

a) "First one":

1. Write a MIPS program that sort the the array elems in descending order using the quick sort algorithm.

According to the previous pseudo code, I divided this problem into two parts and here is the main code, which I implemented for easier visualization about my solution.

For example, I will generate an array: 1, 9, 3, 4, 5, 2, 6, 3 with the size of 8 elements.

1	9	3	4	5	2	6	3

For the **main** code, I load 0 to *Low*, (length - 1) to *High* and the given array to an argument, and in this example *High* will be 7. Then I move to **quicksort** function, which I used jump-and-link instruction to return the result to main function. Finally is the **display** function to display the sorted array as an result.

- Firstly, I will express about **partition** for easier to understand because it will appear later in **quicksort**:

In Quick Sort method, there are four ways to choose a pivot:

- Choose first element.
- Choose last element.
- Choose median of the recent array.
- Choose random element.

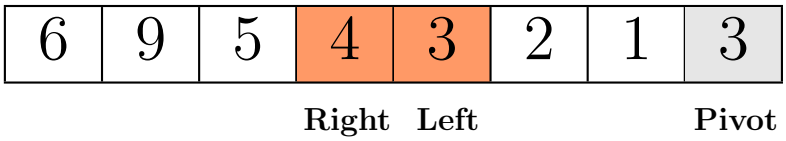
In this case, I choose the last element (array[length - 1] = 3) as a pivot. Then assign *Low* and (*High* - 1) sequentially to temporary *Left* and *Right*.

1	9	3	4	5	2	6	3
Left						**Right**	**Pivot**

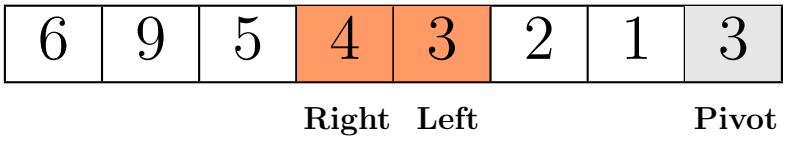
After choosing a pivot, I implemented a while loop to move all the elements that higher than pivot to the left side of the array and lower than pivot to the right side, since the requirement is to have an descending order.

While (Left smaller than or equal to Right) and (array[Left] greater than pivot), do increases Left by 1.
While (Left smaller than or equal to Right) and (array[Right] smaller than pivot), do decreases Right by 1.
If Left greater than Right, exit the **partition** and swap Pivot with array[Left]. Else swap array[Left] with array[Right], also increases Left by 1 and decreases Right by 1. Then, jump back to the first while loop in **partition**.

After doing those steps, the array will be sorted like this:



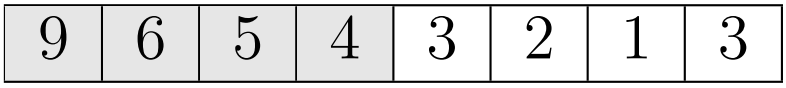
At final step, swap pivot with array[Right] and return Left as a partition result.



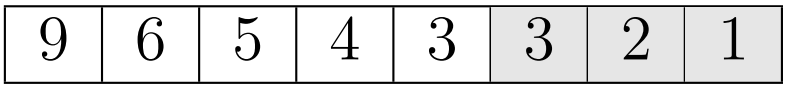
- Secondly, I will express about **quicksort**:

In this function, *if* Low greater than or equal to High, stop it. *Else* set an integer midpoint equal to the returned result of **partition** function, then recursively call **quicksort** with passing the following values to the parameters:

- (Low, midpoint - 1) to sort the left side of the midpoint.



- (midpoint + 1, High) to sort the right side of the midpoint.



2. Calculate the execution time of your program if one instruction requires 1 ns for processing.

4 Conclusion

In conclusion, with applications of these operating systems, the technology is developing day by day, more and more commonly. However, besides these benefits that also have many issues, so, inventors should find out solutions to deal with these problems. Hopefully, there are more and more useful and effective applications for human beings in the near future.

References

[1] *Quicksort*, WIKIPEDIA, ,
https://en.wikipedia.org/wiki/Quicksort

[2] Data Structure and Algorithms - Quick Sort, tutorialpoint,
https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm

[3] Binary search, Simple English WIKIPEDIA,
https://simple.wikipedia.org/wiki/Binary$_search$

[4] Binary search algorithm, WIKIPEDIA,
https://en.wikipedia.org/wiki/Binary$_search_algorithm$

[5] Utility Software - Quertime - Page 2, 6 Dec. 2018,
www.quertime.com/article/what-happened-to-the-mobile-os-market/