VIET NAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COMPUTER ARCHITECTURE

MAR MIPS Assignment

Sorting and Searching on Array

Instructor: Phạm Quốc Cường
Student: Trần Hoàng Long - 1852545

Ho Chi Minh City, Dec 2019

Mục lục

1 Introduction

This paper summarizes my research, notes and implementation on the Computer Architecture MIPS Assignment. There are two exercises:

- Implement QuickSort in MIPS, sort an array decendingly and report it's instruction count, execution time (assuming $1ns$ each instruction)
- Implement BinarySearch in MISP. First, sort an array assendingly and then use QuickSort implemented to find *all indexes* of an element.

For exercise 1, i will show my algorithms and ideas. To calculate the execution, i would analyze my algorithm and test my program in different cases: Worst, Best and Average.
For exercises 2, i will show the original BinarySearch, then explan my modifications to get BinarySearch working with duplicated elements and then show my implementation with a test example.
There's another section in the end where i show some notes and ideas i came accross while working with MIPS for the project, as well as reviewing my implementation for the exercises from the "MIPS" perspective.

2 Exercise 1

2.1 Quick Sort Algorithm

Quick sort algorithm is highly efficient sorting algorithm. It's based on:

1. Piking an item as a pivot (many different ways pivot can be chosen) but in my implementation, i always pick pivot as array's last element
2. Partitioning the array into two smaller sub-arrays around pivot, one half is larger and the other is smaller than pivot. Thus putting pivot at the correct position.
3. Then, recursively call the Quick sort function again to sort the sub-arrays until all are sorted.

Algorithm 1: QuickSort

Input: *head* and *tail* address of array
Output: Void
Result: Array from *head* to *tail* sorted

```
1 begin
2     if (head < tail) then
3         pivot = Partition(head, tail)
4         QuickSort(head, pivot − 1)
5         QuickSort(pivot + 1, tail)
6     end
7 end
```

The function $Partition(head, tail)$ takes two arguments: the head address and tail address of the array, it spit the array into two halves one on the left is larger and the right is smaller than pivot *(since we are sorting in decending order)*. After partitioning, it reposition the pivot in the

middle of the two sub-sets and return the new address of pivot.

Algorithm 2: Partition

Input: *head* and *tail* address of array
Output: New correct address of *pivot*
Result: Array partition into two halves with pivot in the middle

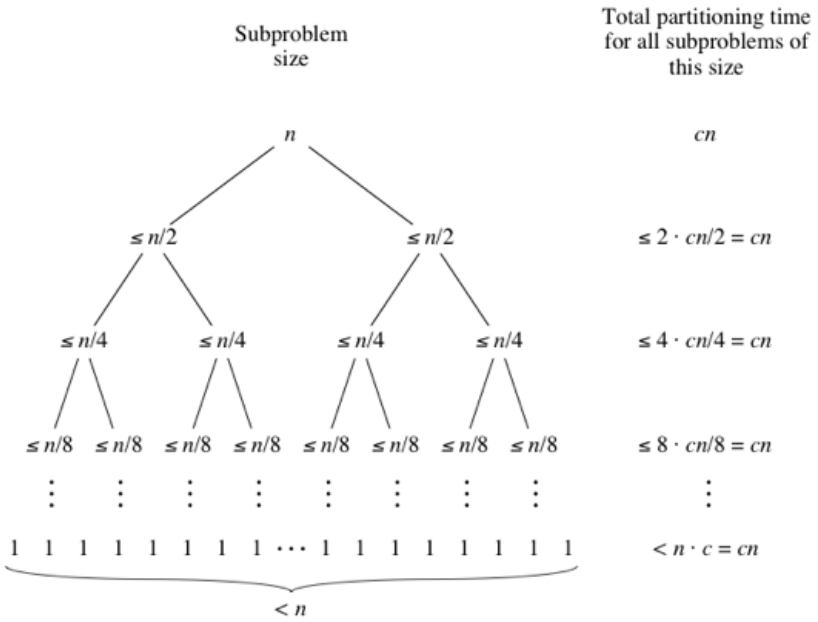
```
1  begin
2      pivot = tail
3      while (1) do
4          while (head.Value > pivot.Value) do
5              head++
6          end
7          while (tail.Value ⩽ pivot.Value)&&(tail > head) do
8              tail−−
9          end
10         if (head < tail) then
11             Swap(head.Value, tail.Value)
12         else
13             break
14         end
15     end
16     Swap(pivot.Value, head.Value)
17     return head
18 end
```

In the partition algorithm, we constantly check the value to the left and the right of the array and move inward to continue checking, if we find a pair of value in the incorrect position, we swap them. After this, we swap the pivot into the middle of two partitions and return it's new address.

2.2 Complexity

2.2.1 Best-case analysis

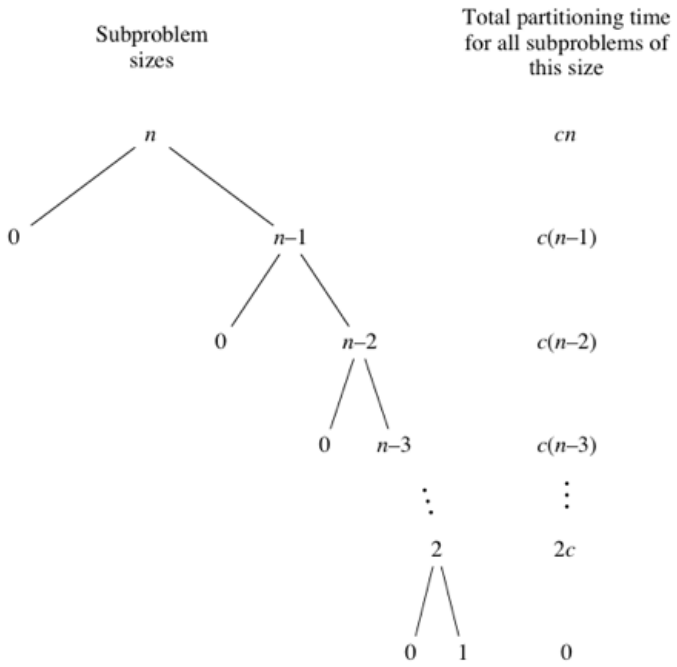
In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Each of the sub-set can only be equal or within one of each other in this case.
In this case, complexity is same as Merge Sort, $O(n \log_2 n)$



Hình 1: Tree of best case sub-problem sizes

2.2.2 Worst-case analysis

The most unbalanced case occurs when the recursive call on the i^{th} element takes $i - 1$ time. ($i \in \{n, n-1, ..., 0\}$)

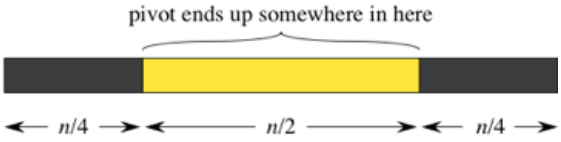


Hình 2: Tree of worst case sub-problem sizes

If this happens repeatedly in every partition, then each recursive call processes a list of size one less than the previous list. And in that case, Quicksort takes $O(n^2)$ time.

2.2.3 Average-case analysis

In the average case, the complexity is $O(n \log_2 n)$.



Hình 3: Tree of best case sub-problem sizes

2.3 Execution time example

Knowing the diffrent cases of complexity of QuickSort, i've tested my implementation to see the instruction count and execution time. *Assuming that all the instructions executes equally, requiring **1 ns** of processing.* I tested by running the program on some random sorted array of different sizes (dupplication allowed). (Generated through https://www.calculatorsoup.com and https://www.random.org)
Instructions are counted using the Instruction Counter intergrated in MARS 4.5.

2.3.1 Worst Case Performance

In my implementation, the pivot is chosen as the last element, so it's easy to see that the worst case will be when our array is already sorted. If this happens, the partition function will only process one element each time and the worst case senario happens.
Test:

No.	Array	Size	Instruction			
			I-Type	R-Type	J-Type	Total
1	$X \in [1, 100]$	10	542	267	68	877
2	$X \in [1, 1000]$	100	21332	14037	5198	40567
3	$X \in [1, 10000]$	1000	1788482	1265487	501998	3555967
4	$X \in [1, 100000]$	10000	175384982	125154987	50019998	350559967

Bảng 1: Instruction count for sorted random generated arrays

Given, theoretically, one instruction takes **1 ns**, the tested cases above would take: $877ns$, $40.567\mu s$, $3.556ms$ and $0.35s$ respectively
But, in realtime, when simulating the worst case for the array of size 10000 elements, my code took real long too finish (a few hours), so i would take this as the limit for the worst case implementation.
The arrays used here are located in this DataFile

2.3.2 Best and Average Case Performance

Sometimes, the Partition function will split the array into two perfect half, and when this happens for all recursive calls of QuickSort, we get the best case scenario, if not, we would usually get the average case for QuickSort.
Test: I will test my implementation on random, unordered arrays (dupplication allowed).

No.	Array	Size	Instruction			
			I-Type	R-Type	J-Type	Total
1	$X \in [1, 100]$	10	434	217	51	702
2	$X \in [1, 1000]$	100	5657	3266	868	9791
3	$X \in [1, 10000]$	1000	82358	48021	15723	146102
4	$X \in [1, 10000]$	10000	988861	637136	188500	1814497
5	$X \in [1, 100000]$	100000	12582457	8214842	2581215	23378514

Bảng 2: Instruction count for random generated arrays

Given, theoretically, one instruction takes **1 ns**, the tested cases above would take: $702ns$, $9.791\mu s$, $0.1461ms$, $1.815ms$ and $0.0234s$ respectively
Diffrent from the worst case, my implementation can get to much better array size in random situation, up to 100000 element and might be more. The arrays used here are located in this DataFile

3 Exercise 2

3.1 Binary Search

Binary Search is searching a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Algorithm 3: Binary Search

Input: *head* and *tail* address of array, search_value to find in array
Output: Address of searched value, $NULL(0x00000000)$ if not found

```
1  begin
2      mid = head + (tail − head)/2
3      if (mid.Value == search_value) then
4          return mid
5      else if search_value < mid.Value then
6          BinarySearch(head, mid − 1)
7      else
8          BinarySearch(mid + 1, tail)
9      end
10 end
```

That, was the key concept of binary search, but, in order to search and display all the occurrences of an element in the array, i made some adjustments. Here, i would search for the first occurence and if that exists in the array, continue to search for last occurence and print all indexes between, else, print -1.

Algorithm 4: Binary Search Print (all occurences)

Input: *head* and *tail* address of array, *search_value* to find in array
Result: All indexes of *search_value* printed, separated by "space" character, print -1 if none is found

```
1  begin
2      first_occur = BinarySearchFirst(head, tail, search_value)
3      if firsr_occur ≠ NULL then
4          last_occur = BinarySearchLast(head, tail, search_value)
5          while first_occur ⩽ last_occur do
6              print(first_occur − head)
7              first_occur + +
8          end
9      else
10         print(−1)
11     end
12 end
```

For the BinarySearchFirst and BinarySearchLast, they're the same as normal binary search, but, instead of returning the address of value when found, they store the found address and keep searching to the left (Search First case) or to the right (Search Last case) and return the address later at the end.

Algorithm 5: Binary Search First OR Last

Input: *head* and *tail* address of array, *search_value* to find in array
Output: Address of first OR last occurence of *search_value*. *NULL* if not found

```
1  begin
2      first OR last = NULL // initial found address is NULL
3      while head ⩽ tail do
4          middle = head + (tail − head)/2
5          if middle.Value == search_value then
6              first OR last = middle  //if found, save address
7               //if looking for first, keep search left; if last, search right.
8              tail = middle − 1 OR head = middle + 1
9          else if search_value < middle.Value then
10             tail = middle − 1
11         else
12             head = middle + 1
13         end
14     end
15     return first OR last
16 end
```

3.2 Example

Assume an array as follows:

$$10, 5, 1, 3, 10, 7, 1, 3, 1, 2, 3, 7, 7, 1, 8, 10, 3, 2, 1, 10$$

My implementation's procedure is as follows:

1. Print initial array
2. Sort ascendingly using QuickSort
3. Ask for user input for element to search for binarily
4. Display all found indexes

```
10 5 1 3 10 7 1 3 1 2 3 7 7 1 8 10 3 2 1 10
1 1 1 1 1 2 2 3 3 3 3 5 7 7 7 8 10 10 10 10
Input integer to search for: 1
0 1 2 3 4
-- program is finished running --
```

Hình 4: MIPS output for searching element 1

Note that the BinarySearch implemented by me does not actually store the found indexes in any way, but instead, find the first, last address and print all indexes.

4 MIPS Implementation Notes and Ideas

4.1 Calling Convention

As known, CPU only have a handful of register for us to work with on all of our functions, and it's possible for different function to use same registers, leading to overwriting and corruption during callling process. This leads us to the use of stack to save/restore register value to help preserve our registers. The conventional use of this is described briefly below:

1. Argument registers $\$a$ are usually copied into other registers for usage. If function want to moddfy directly, must be pushed to stack before use and restored later.
2. Temporary $\$s$ registers are expected to be unchanged in a function call, so the callee must save/restore them if need for usage.
3. Temporaty $\$t$ registers are seen as "could be changed", so the caller need to save them before making a function call (restore after call) and the callee does not care when using them.
4. The return address $\$ra$ register is always saved and restore if the function is not a leaf (means it calls others at some point)

Note: Careful when designing a function that's a caller and also a callee.
When coding the recursive function QuickSort in the assignment, it's important to notice that it is a caller as well as a callee, so all register modified must be saved, including the $\$ra$ register.

4.2 Implement MIPS QuickSort

My implementation of QuickSort in MIPS is rather simple:

1. Print the array one before sorting
2. Call the QuickSort (decending) function:
   - Argument is $\$a1$ and $\$a2$ as head and tail address of array
   - Pivot is always the element at the tail
   - Call the Partition function (also take argument $\$a1$ and $\$a2$ as head and tail to of array) to sort the array into to halves and put the pivot in the middle, returning new address of pivot.
   - Continue to sort on the two sides of the new location of pivot until all sorted.
3. Print the array after sorting

4.3 Implement MIPS BinarySearch

For exercise 2, it's required to sort and binarily search for all indexes of a value.
First, i sort the given array using QuickSort that i've already implemented (but asscending this time).
And then initialize BinarySearch (dupplicated element allowed):

1. BinarySearch calls for BinarySeachFirst and BinarySearchLast (all three functions take arguments: $\$a1 - head$,$\$a2 - tail$ and $\$a3 - search\_value$)

2. BinarySearchFirst/Last would find the **address** of first/last occurence of element using BinarySeachAlgorithm (modified) and return it in $\$v1$
3. After getting the first and last address, BinarySearch will translate the address into the element's relative index to the head of array and print all index between.
4. Case element not found, the return address from BinarySearchFirst is $NULL$, and BinarySeach would print -1 instead.
5. Note that i don't store the found index in anyway but instead only get the first,last address, then translate them and print all index got.