COMP26120 Lab 5

Ziad Salem

December 10, 2019

1 Complexity Analysis

1.1 Iteration Sort

Insertion sort is a sorting algorithm that builds a final sorted array (sometimes called a list) one element at a time.

Statement	Effort

`void insertion_sort(struct darray* array)`
`{`
`  for (int i = 1; i < array->size; i++)`
`  {`
`  int z = i;`
`  while (z > 0 && compare(array->cells[z-1] , array->cells[z]) > 0)`
`  {`
`  swap(&array->cells[z - 1], &array->cells[z]);`
`  z--;`
`  }`
`  }`
`}`
	$c1n$
	$c2(n-1)$
	$c3T$
	$c4(T-(n-1))$
	$c5(T-(n-1))$

T = t2 + t3 + . . . + tn where ti is number of while expressions valuations for the (ith) for loop iteration
n is array.size
T(n) = c1n + c2(n-1) + c3(T) + c4(T - (n-1)) + c5(T - (n-1)

1.1.1 Best Case

The best case is when the inner while loop body never executed. So, when checking the running time it will be T(n) = an - b. The big-O notation is O(n).

1.1.2 Worst Case

The worst case will be when inner loop body executed for all previous elements. This happens when the array is not sorted at all and it needs to sort every element. The running time will be

$$T(n) = an^2 + bn - c \tag{1}$$

The big-O notation will be

$$O(n^2) \tag{2}$$

1.1.3 Average Case

The big-O notation gonna be the same as the worst case which is $O(n^2)$. This is because the average case happens when the array is somehow sorted(partially) so the running time will be half the worst case but we don't care about constants.

1.2 Quick Sort

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are 3 main steps in quick sort:

1. Choose the last element as the pivot
2. Create 2 new arrays to the left and right of the pivot.
3. Sort and then keep calling the method until everything is sorted.

Statement	Effort
`void quick_sort(struct darray* arr)`	$T(n)$
`{`	
`  if (arr->size > 1)`	$\Theta(1)$
`  {`	
`    struct darray* greater_than = initialize_set(1);`	
`    struct darray* less_than = initialize_set(1);`	
`    int pivot = arr->size - 1;`	
`    for (int i = 0; i < arr->size - 1; i++)`	$\Theta(n)$
`    {`	
`      if (compare(arr->cells[i], arr->cells[pivot]) >= 0)`	$\Theta(1)$
`      {`	
`        greater_than = insert(arr->cells[i], greater_than);`	
`      }`	
`      else{`	
`        less_than = insert(arr->cells[i], less_than);`	
`      }`	
`    }`	
`    quick_sort(less_than);`	$T(n1)$
`    quick_sort(greater_than);`	$T(n2)$
`    for(int j = 0; j < less_than->size; j++)`	$\Theta(n1)$
`    {`	
`      swap(&arr->cells[j], &less_than->cells[j]);`	$\Theta(1)$
`    }`	
`    swap(&arr->cells[less_than->size], &arr->cells[pivot]);`	
`    int initial = less_than->size + 1;`	
`    for(int z = 0; z < greater_than->size; z++)`	$\Theta(n2)$
`    {`	
`      swap(&arr->cells[z + initial], &greater_than->cells[z]);`	$\Theta(1)$
`    }`	
`    tidy(greater_than);`	
`    tidy(less_than);`	
`  }`	0
`}`	

n is array.size
$\Theta(n)$ is for partitioning
T(n1) and T(n2) are for recursive calls where n1 and n2 are the partitions depending which one is greatest.
T(n) = O(n) + T(n1) + T(n2)

n = n1 + n2

1.2.1 Best Case

$$T(n) = \begin{cases} C & \text{if} n = 1 \\ \Theta(n) + T(\frac{n}{2}) + T(\frac{n}{2}) & \text{if} otherwise \end{cases}$$

The best case is reached when the pivot is chosen at the middle so the two arrays created are of the same size and balanced. Using the master method we get:

$T(n) = \Theta(n) + 2T(\frac{n}{2})$

If we set a and be to 2 we have:

$f(n) = \Theta(n^{\log_b a})$
$f(n) = \Theta(n^{\log_2 2}) = \Theta(n)$
$T(n) = \Theta(n^{\log_b a} logn)$
$= \Theta(nlogn)$

1.2.2 Worst Case

The worst case is reached when you set the pivot either at the start or at the end of array.

$$T(n) = \begin{cases} C & \text{if} n = 1 \\ \Theta(n) + T(n-1) + T(1) & \text{if} otherwise \end{cases}$$

$T(n) = \Theta(n) + T(n-1) + T(1)$
$T(n) = \Theta(n) + T(n-1)$
$T(n) = \Theta(n) + (\Theta(n-1) + T(n-2))$
...
$T(n) = \Theta(n) + (\Theta(n-1) + (\Theta(n-2) + ... + (T(1)))$
$T(n) = \Theta(n) + (\Theta(n-1) + (\Theta(n-2) + ... + (\Theta(1)))$
$T(n) = \sum_{i=0}^{n} i$
$T(n) = \Theta(n^2)$

1.2.3 Average Case

Complexity is : $\Theta(nlogn)$

We get this complexity when the partition we do is partially balanced so the complexity will be almost the best case.

2 Experimental Analysis

In this section we consider the question

> [Under what conditions is it better to perform linear search rather than binary search?]

2.1 Experimental Design

I did 8 experiments to know in which condition does linear do better and I found it performs better when it is unsorted, small query and large dictionary. To know that, I varied the conditions each time between Sorted and Unsorted array, Different size of queries and Different sizes of dictionaries. Linear is better in this case because binary sorts the array first before searching so it takes more time than linear search.

2.2 Experimental Results

Sorted, Small Query and Small Dictionary	
Linear Search	**Binary Search**
0.008s	0.008s
0.005s	0.009s
0.006s	0.009s
0.007s	0.010s
0.004s	0.010s

Sorted, Large Query and Small Dictionary	
Linear Search	**Binary Search**
1.545s	0.576s
1.605s	0.869s
1.435s	0.661s
1.372s	0.890s
1.459s	0.743s

Sorted, Small Query and Large Dictionary	
Linear Search	**Binary Search**
0.347s	1.149s
0.271s	1.163s
0.490s	1.138s
0.576s	1.147s
0.249s	1.552s

Sorted, Large Query and Large Dictionary	
Linear Search	**Binary Search**
1m 08.25s	1.451s
1m 29.05s	1.362s
1m 11.86s	1.498s
1m 35.16s	1.261s
1m 20.62s	1.537s

UnSorted, Small Query and Small Dictionary	
Linear Search	**Binary Search**
0.003s	0.004s
0.005s	0.006s
0.004s	0.003s
0.004s	0.004s
0.004s	0.006s

UnSorted, Large Query and Small Dictionary	
Linear Search	**Binary Search**
0.741s	1.105s
0.903s	1.082s
1.244s	1.056s
0.689s	1.073s
0.859s	1.085s

UnSorted, Small Query and Large Dictionary	
Linear Search	**Binary Search**
0.176s	0.600s
0.184s	0.600s
1.209s	0.739s
0.192s	0.578s
0.201s	0.602s

UnSorted, Large Query and Large Dictionary	
Linear Search	**Binary Search**
0.176s	0.738s
0.184s	0.800s
1.209s	0.803s
0.192s	0.955s
0.201s	0.815s

3 Extending Experiment to Data Structures

We now extend our previously analysis to consider the question

> Under what conditions are different implementations of the dictionary data structure preferable?

There is always a best implementation but it depends on certain conditions. So, when the array is sorted binary search is the fastest. If it is not sorted, tree is the best especially when we want to insert. But considering all conditions hash set is the best implementation. Its complexity is lower than the other implementations. It is O(1) for most methods used in Hash set.

4 Conclusions

For insertion sort, the best case is O(n) while the worst case(and the average case) is O(n to the power of 2). For quick sort the best case(average as well) is O(nlogn) while the worst case is O(n to the power of 2). So, quick sort is much faster than insertion sort. Also, based on the experiments, I conclude that binary search is better in the overall performance but linear search is better when it is unsorted, small query and large dictionary.