Complexities

The time and space complexities of each of the sorting algorithms included within the project.

Done By: Arooba Moin (20K-0213)
Sabah Mawani (20K-0393)
Submitted to: Dr. Atif Tahir
Section: BCS-5D

Insertion Sort

Insertion Sort

Line#	Cost	time	Insertion-Sort(A)
1	c_1	n	for $j = 2$ to A.length
2	c_2	$n-1$	key = A[j]
3	0	$n-1$	//insert A[j] into the sorted sequence A[1...j-1]
4	c_4	$n-1$	i = j-1
5	c_5	$\sum_{j=2}^{n} t_j$	while i>0 and A[i]>key
6	c_6	$\sum_{j=2}^{n} (t_j-1)$	A[i+1] = A[i]
7	c_7	$\sum_{j=2}^{n} (t_j-1)$	i = i-1
8	c_8	$n-1$	A[i+1] = key

$t_j = j$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} j + c_6\sum_{j=2}^{n}(j-1) + c_7\sum_{j=2}^{n}(j-1) + c_8(n-1)$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$2T(n) = n^2(c_5 + c_6 + c_7) + n(2c_1 + 2c_2 + 2c_4 + c_5 - c_6 - c_7) - (2c_2 + 2c_4 - 2c_5 + c_6 + c_7 + 2c_8)$$

$$2T(n) = An^2 + Bn + C$$

$$T(n) = O(n^2)$$

Space Complexity:

Since we use only a constant amount of additional memory apart from the input array, the space complexity is $O(1)$.

Bubble Sort

Bubble sort

```
def bubble_sort(array):
    for i in range(len(array),1,-1):
        for j in range(1,i):
            if array[j-1] > array[j]:
                array[j-1], array[j] =
                array[j], array[j-1].
```

Time complexity: outer for loop → n times

~~(n-1)~~ Inner for loop → n-1, n-2

$$(n-1)+(n-2)+(n-3)+\ldots 3+2+1 = \frac{N(N-1)}{2}$$

$$= O(N^2)$$

Space complexity measures the amount of extra space that is needed for sorting the list. Bubble sort only requires one (1) extra space for the temporal variable used for swapping values. Therefore, it has a space complexity of O(1).

Merge Sort

Merge-Sort

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4T\left(\frac{n}{4}\right) + 2n$$

$$= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n = 8T\left(\frac{n}{8}\right) + 3n$$

$$\vdots$$

$$= kn + 2T\left(\frac{n}{2^k}\right)$$

$$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log n$$

$$T(n) = n\log n = O(n\log n)$$

Space Complexity: Auxiliary Space: $O(n)$

Heap Sort

Date:

Heap Sort

maxheap (A)
A.heap.size = A.length
for i = ⌊A.length/2⌋ down to 1
heapify (A,i) → $O(n)$

Heapify (A,i)
l = left (i)
r = right (i)
if l ≤ A.heap.size and A[l] > A[i]
largest = l
else largest = i
if r ≤ A.heap.size and A[r] > A[largest]
largest = r.
if largest ≠ i
exchange A[i] with A[largest]
heapify (A, largest) → $O(\log n)$

$T(n) = T(2n/3) + O(1)$

HeapSort (A)
maxheap (A)
for i = A.length downto 2.
exchange A[1] with A[i]
A.heapsize = A.heapsize − 1
heapify (A,1)

Time Complexity : $O(n\log n)$
$: O(n) + (n-1)\,O(\log n)$
$= O(n) + O(n\log n)$
$= O(n\log n)$

Space complexity : $O(1)$.

Quick Sort

Date:

Quick Sort

Quick sort algorithm also applies the divide and conquer principle to divided the input array into lists. The first with small items, and the second w/ large items. The algorithm then sorts both lists recursively until resultant list is sorted

$$T(n) = 2T(n/2) + c \cdot n$$
$$4T(n/4) + 2cn$$
$$8T(n/8) + 3cn$$
$$2^k T\left(n/2^k\right) + kcn$$

$\frac{n}{2^k} = 1 \quad k = \log_2 n$

$$= 2^{\log_2 n} + T(1) + c \cdot n \cdot \log_2 n$$
$$= n \cdot c_1 + cn \cdot \log n$$
$$T(n) = 2T(n/2) + O(n)$$
$$k = \log_2 n$$
$$O(n \log n)$$

Space complexity : $O(\log n)$

Radix Sort

Radix Sort

Radix sort depends on counting sort, otherwise it doesn't achieve O(nk) in total.

```
// counting sort
  # counting - O(n)
    # Accumulating - O(k)
  # sorting - O(n)
```

Time complexity of counting sort → O(n+k)

```
~~def (radix s~~
def radix sort(arr, max_val):
  num = getnum(max_val)
  # O(k(n+k)
    for d in range(num):
      # count sort takes O(n+k)
    arr = count_sort(arr, max_val)
```

Time complexity of Radix:

O(k(n+k)),

Space complexity: ~~[illegible]~~ O(n+k)

Bucket Sort

Bucket Sort

Line #	Time	Space	Bucket Sort (A)
1			$n \leftarrow$ length[A]
2	$O(n)$		for $i = 1$ to n do
3	$O(n)$	$O(n+k)$	Insert $A[i]$ into list $B[nA[i]]$
4	$O(n)$		for $i = 0$ to $n-1$ do
5			Sort list B with insertion sort
6	$O(k)$		Concatenate the lists $B[0], \dots, B[n-1]$ together in order

for line #5 in the worst case it will take $O(n^2)$, while on an average it takes $O(n)$ time

Time complexity = $O(n)$ Space complexity = $O(n+k)$

Counting Sort

Counting Sort

Line #	Time	Space	Algorithm:
1		$O(u)$	Create a counter array $C[1, \dots, u]$
2		$O(n)$	Create an auxillary array $B[1, \dots, n]$
3	$O(n)$		Scan A once, record element frequency
4	$O(u)$		Calculate prefix sum in C
5	$O(n)$		Scan A in the reverse order, copy each element to B at the correct position
6	$O(n)$		Copy B to A

Time complexty:

$$T(n) = O(n) + O(u) + O(n) + O(n)$$
$$= O(n+u)$$
if ($u == n$)
$$O(n)$$

Space Complexity

$$O(n+u)$$
if ($u == n$)
$$O(n)$$

Quick Sort Adaptation

Quicksort Adaptation.

- combination of Quicksort and Insertion sort.

Insertion_Sort(A)
$T(n) = O(n^2)$.

Quick_Sort (B)
$T(n) = 2T(n/2) + O(n)$
$= O(n \log n)$

```
def hybrid_quicksort
        while low < high;            ↳ O(nlogn)
          if high-low + 1<10:        ↳ O(logn)
             insertionsort(A)        ↳ O(n²).
      else
          partition(c);              → O(n)
          if pi_low < high- pi:      ↳ O(logn)
             hybrid_quicksort
             low = pi+1.
          else
             hybrid_quicksort ()
                high = pi-1.
```

Time complexity: $O(n^2)$
Space: $O(n)$.

Counting Sort Adaptation

Counting Sort Adaptation

Line#	Time	Space	Preprocessing
1		O(u)	Create a counter array C[1,...,u]
2	O(n)		Scan A once, record element frequency
3	O(u)		Calculate prefix sum in C

$T(n) = O(n+u)$ Space $= O(u)$

Line#	Time	Space	Adaptation
1		O(1)	Take input 1
2		O(1)	Take input 2
3	O(1)	O(1)	Access C[input1]
4	O(1)	O(1)	Access C[input2]
5	O(1)	O(1)	Compute C[input2] - C[input1]

$T(n) = O(1)$ Space $= O(1)$