Data Structures

Sorting

Hikmat Farhat

June 20, 2018

Introduction

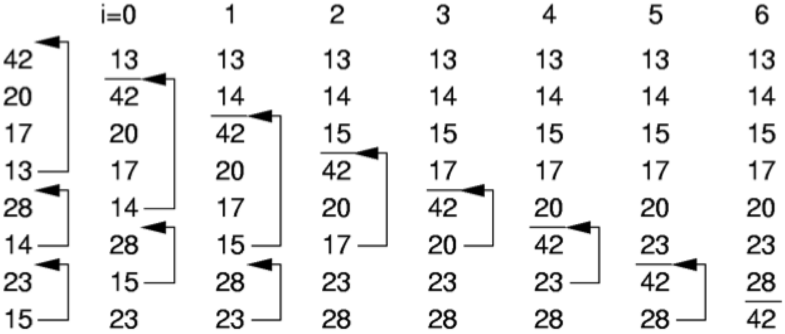
- Many applications need to sort an array of n elements.
- We will cover the following algorithms

1. Bubble Sort
2. Insertion Sort
3. QuickSort
4. MergeSort
5. HeapSort

Bubble sort

- The input is an array of n elements $a[0] \dots a[n-1]$.
- The idea of bubble sort is to keep swapping an element with its right neighbor as long as it is larger than the neighbor.
- This is done $n-1$ times because
  - the first time the largest element is moved all the way to the end of the array.
  - the second pass will put the second largest element in the slot before the last ... etc
  - The $n-1^{th}$ pass will put the $n-1^{th}$ element in its proper place.
  - The smallest element is already in its proper place so there is no need for the n^{th} pass.

Example


i=0 1 2 3 4 5 6
42 13 13 13 13 13 13 13
20 42 14 14 14 14 14 14
17 20 42 15 15 15 15 15
13 17 20 42 17 17 17 17
28 14 17 20 42 20 20 20
14 28 15 17 20 42 23 23
23 15 28 23 23 23 42 28
15 23 23 28 28 28 28 42

Bubble sort code

```
BUBBLE-SORT(a,n)
for i = 1 to n − 1 do
    for k = 0 to n − 2 do
        if a[k] > a[k + 1] then
            tmp ← a[k + 1]
            a[k + 1] ← a[k]
            a[k] ← tmp
        end
    end
end
```

- Inner loop operations do not depend on i so the algorithm will perform $\Theta(n^2)$ **iterations** no matter what the input is.
- That is why the number of **comparisons** is $\Theta(n^2)$ on all input.
- The number of **swaps** depends on the number of times the **if** statement evaluates to true. We'll get back to this later.

Insertion Sort

- A similar but better algorithm is insertion sort.
- Insertion sort saves on unnecessary comparisons.
- The basic idea is that after pass $k-1$ the portion of the array: $a[0] \dots a[k-1]$ is sorted.
- Pass k depends on that property as follows:
- Repeatedly compare $a[k]$ with $a[i]$, $i = k-1, k-2, \dots$
- If at any point $a[k] > a[i]$ stop and the subarray $a[0], \dots, a[k]$ is sorted.

Example insertion sort

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Code for insertion sort

```
INSERTION-SORT(a,n)
```

for $i = 1$ *to* $n - 1$ **do**

  $tmp \leftarrow a[i]$

  $k \leftarrow i$

  while $k > 0$ *and* $tmp < a[k-1]$ **do**

    $a[k] = a[k-1]$

    $k \leftarrow k - 1$

  end

  $a[k] \leftarrow tmp$

end

- Notice that the algorithm exits the inner loop whenever $tmp \geq a[k-1]$.

Comparison

- First we compare the two algorithm, by counting the number of comparisons and the number of swaps on the input array
- 17,1,2,8,3,9,15,16
- Bubble sort we do $7 \times 7 = 49$ comparisons.
- first pass we do 2 swaps and subsequently 1 swap each pass for a total of 8 swaps.
- insertion sort gives 7 comparisons and 8 swaps.

Complexity

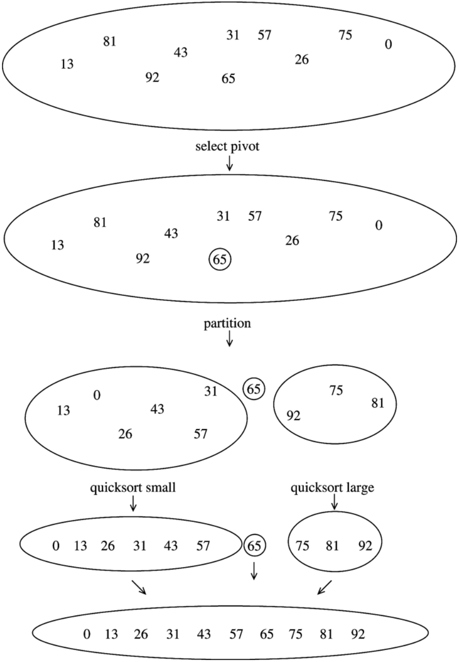
- In the case of bubble sort worst case, average case and best case are all $O(n^2)$ operations (total=comparison+swaps).
- In insertion sort the best case is $O(n)$ if the array is already sorted and $O(n^2)$ if the array is reversely sorted.
- The average case is $O(n^2)$ for both.
- If we consider swaps only both have the same number of operations which is $0, O(n^2), O(n^2)$ for best,worst and average case respectively.
- Therefore insertion sort saves on comparisons only.

General result

- Both bubble and insertion sort exchange adjacent elements.
- Given an array $a[0] \ldots a[n]$ if $i < j$ and $a[i] > a[j]$ then (i, j) is called an inversion.
- The average the number of inversions in an array with n distinct elements is $n(n-1)/4$.
- This is because the number of pairs is $\sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 = n(n-1)/2$. On average, for a random input, half of them are inverted for average number of inversions of $n(n-1)/4$.
- When we swap two adjacent elements only one inversion is removed
- Therefore, on average, we need $\Omega(n^2)$ number of swaps to sort an array.
- **Any** algorithm that works by swapping adjacent element will be $\Omega(n^2)$ on average.

Quicksort

- quicksort is a divide and conquer algorithm.
- Given an array a it works as follows
  1. If the number of elements is 0 or 1 then nothing is done so return.
  2. pick an element v from the array called the **pivot**.
  3. Partition $a - v$ into two groups: $a_1 = \{x \in a - v \mid x \leq v\}$ all elements that are smaller than v, $a_2 = \{x \in a - v \mid x \geq v\}$ all elements greater than v are in the second group.
  4. the results is quicksort(a_1) followed by v followed by quicksort(a_2).


81
31 57
75
0
43
13
26
92
65
select pivot
81
31 57
75
0
43
13
26
92
65
partition
0
31
65
75
13
43
81
92
26
57
quicksort small
quicksort large
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

Partitioning Algorithm

- We put aside for now the question of choosing the pivot and assume it is selected in some manner.
- The idea is to group the elements of the array into a group that is smaller than the pivot and another group that is larger than the pivot.
- Given a subarray with index p to r
  1. First select the pivot, $a[v]$, $p \leq v \leq r$.
  2. Swap $a[v]$ with the last element $a[r]$.
  3. run a partitioning algorithm that keeps two indices i and j
  4. At every iteration $a[k] \leq a[r]$ for $k < i$ and $a[k] \geq a[r]$ for $k > j$.

Quicksorting

- Once we have a partitioning algorithm quicksort is performed as follows

```
QUICKSORT(A,p,r)
q ← PARTITION(A,p,r)
QUICKSORT(A,p,q-1)
QUICKSORT(A,q+1,r)
```

Partitioning Algorithm

```
PARTITION(a,p,r)
i ← p − 1
pivot ← a[r]                                    // pivot assumed in place

for j ← p to r − 1 do
    if a[j] ≤ pivot then
        i ← i + 1
        swap(a[i], a[j])
    end
end
swap(a[i + 1], a[r])
return i+1
```

- We claim that the above algorithm maintains the following loop invariant
  1. If $p \le k \le i$ then $A[k] \le pivot$
  2. If $i + 1 \le k \le j - 1$ then $A[k] > pivot$
  3. If $k = r$ then $A[k] = pivot$
  4. If $j \le k \le r - 1$ under consideration.

Loop Invariant

- **Initialization**: initially $i = p - 1$, $j = p$ and since there are no values between p and i and $i + 1$ and $j - 1$ thus conditions 1 and 2 are satisfied trivially. Also condition 3 is satisfied by the assignment $pivot \leftarrow A[r]$.
- **Maintenance**:There are two possible outcomes when the loop is executed
  1. If $A[j] \leq x$ then $A[i + 1]$ and $A[j]$ are swapped and i and j are incremented. The result satisfies conditions 1 & 2.
  2. If $A[j] > x$ then j is incremented and this satisfies condition 2.
- **Termination**: The algorithm terminates when $j = r$.
- Try the algorithm on the sequence 3,5,4,1,9,5,7,8,5. Assuming the pivot is in place (last one).

Choosing the pivot

- As we will see the performance of quicksort depends on how balanced the partitioning is, on average.
- A good strategy is to select the pivot in a uniformly random fashion.
- Sometimes it is useful to choose a pivot in a deterministic fashion.
- a good deterministic choice is the median of three method:
  1. Given an array A to be partitioned between the indices p and r.
  2. Select the three elements $A[p], A[\lfloor(r-p)/2\rfloor], A[r]$ and sort them.
  3. The middle one is chosen as the pivot.
  4. Note that in this case the middle is less than the right so the swapping is done with element **before** the last.

Median of Three

```
/**
 * Return median of left, center, and right.
 * Order these and hide the pivot.
 */
template <typename Comparable>
const Comparable & median3( vector<Comparable> & a, int left, int right )
{
    int center = ( left + right ) / 2;
    if( a[ center ] < a[ left ] )
        swap( a[ left ], a[ center ] );
    if( a[ right ] < a[ left ] )
        swap( a[ left ], a[ right ] );
    if( a[ right ] < a[ center ] )
        swap( a[ center ], a[ right ] );

        // Place pivot at position right - 1
    swap( a[ center ], a[ right - 1 ] );
    return a[ right - 1 ];
}
```

Example

- As an example, run the algorithm on the sequence 3,5,4,1,5,5,7,8,9

Complexity

- We analyze the best and worst case complexity of quicksort.
- In general the cost of quicksorting an array of size n is equal to the sum of partitioning the array plus quicksorting the two smaller subarrays:
- It takes $\Theta(n)$ to partition the array into two subarrays of size i and $n-i-1$, thus:

$$T(n) = T(i) + T(n-i-1) + \Theta(n)$$

- The best case is when $i = n/2$ and the worst case is when $i = 0$.

Worst case complexity of quicksort

- The worst case occurs when one subarray is 0 and the other is $n-1$ thus the recurrence becomes

$$T(n) = T(n-1) + cn$$

- We will show that $T(n) = \Theta(n^2)$ by iterating the recurrence relation.

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + cn + c(n-1) \\ &= \ldots \\ &= T(i) + c\sum_{k=i+1}^{n} k \\ &= \ldots \\ &= T(1) + c\sum_{k=2}^{n} k = \Theta(n^2) \end{aligned}$$

Best case complexity of quicksort

- The best case is when the problem is divided into two equal subarrays then

$$T(n) = 2T(n/2) + cn$$

- By using the Master theorem we get $(a = 2, b = 2, d = 1)$

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

Average case complexity

- To compute the average case complexity we assume that the pivot is selected uniformly randomly between 0 and $n-1$.
- Recall that if the selected pivot has index $0 \leq i \leq n-1$ then the recurrence relation of the complexity of quicksort is

$$T(n) = T(i) + T(n-i-1) + c \cdot n$$

- Using different values of i we get

$$\begin{aligned}
T(n) &= T(0) + T(n-1) + c \cdot n \\
T(n) &= T(1) + T(n-2) + c \cdot n \\
T(n) &= T(2) + T(n-3) + c \cdot n \\
\ldots\ldots \\
T(n) &= T(n-2) + T(1) + c \cdot n \\
T(n) &= T(n-1) + T(0) + c \cdot n
\end{aligned}$$

- Adding the above and dividing by n we get the recurrence of the average complexity

$$T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + cn$$

- Multiplying both sides by n we get

$$nT(n) = 2\sum_{k=0}^{n-1} T(k) + cn^2$$

- Replacing n by $n-1$ we get

$$(n-1)T(n-1) = 2\sum_{k=0}^{n-2} T(k) + c(n-1)^2$$

- Subtracting the above two equations we get

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$$

- Rearranging terms and dropping the c term

$$nT(n) = (n+1)T(n-1) + 2cn$$

- Dividing both sides by $n(n+1)$ we get

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

- We iterate the above equation for different values

$$\begin{aligned}
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2c}{n+1} \\
\frac{T(n-1)}{n} &= \frac{T(n-2)}{n-1} + \frac{2c}{n} \\
\frac{T(n-2)}{n-1} &= \frac{T(n-3)}{n-2} + \frac{2c}{n-1} \\
\ldots &= \ldots \\
\frac{T(2)}{3} &= \frac{T(1)}{2} + \frac{2c}{3}
\end{aligned}$$

- By adding,term by term, the above equations we get

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + c\sum_{k=3}^{n+1}\frac{1}{k}$$

Harmonic sum

- the sum $\sum_{k=1}^{n} \frac{1}{k}$ is called the harmonic sum. We obtain an upper bound as follows

$$\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n+1} \leq \int_{u=2}^{n+1} \frac{du}{u-1}$$
$$\leq \int_{x=1}^{n} \frac{dx}{x} = \ln n$$

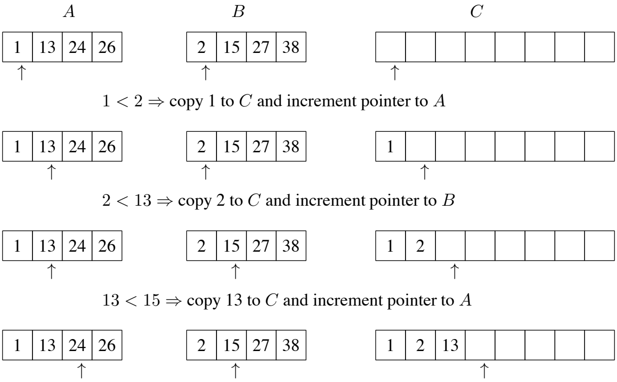
- Therefore

$$T(n) \leq (n+1)\left(\frac{T(1)}{2} + \ln n\right) = \Theta(n \log n)$$

Merge sort

- Merge sort is another divide and conquer algorithm.
- The basic idea is based on the **merging** of **two sorted lists**
- An input array *a* is divided into two parts, left and right
- A recursive call is made to sort left and right independently.
- The merge routine will merge the sorted lists together.
- As an example suppose the input is 1,26,13,24,15,27,2,38
- 1,26,13,24 is sorted to get 1,13,24,26
- 15,27,2,38 is sorted to get 2,15,27,38
- the two halves are **merged** to get 1,2,13,15,24,26,27,38
- Next we describe the merging procedure.

Example merge sort



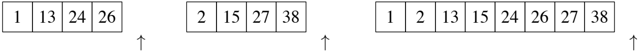
Example merge sort

$15 < 24 \Rightarrow$ copy 15 to C and increment pointer to B

$24 < 27 \Rightarrow$ copy 24 to C and increment pointer to A

$26 < 27 \Rightarrow$ copy 26 to C and increment pointer to A

the remainder of B is copied to C



Complexity of Merge sort

- Let $T(n)$ be the cost of mergesort for an array of size n. This is equal to twice the cost of mergesort for $n/2$ plus the additional cost of merging which is $O(n)$.
- Thus $T(n)$ satisfies the recurrence

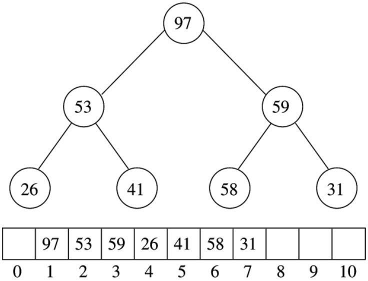
$$T(n) = 2T\left(n/2\right) + dn$$

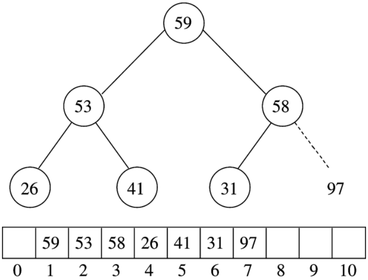
- this is the same recurrence for the best case (and average case) of quicksort with the solution $O(n \log n)$.

Heap Sort

- Heap sort is based on the property of max heap.
- Given an array of n elements $a[0] \dots a[n-1]$, we build a max heap in $O(n)$ operations as we have seen before.
- each deleteMax operation takes $O(\log n)$.
- Thus the complexity of sorting using a max heap is $O(n \log n)$.

Example Heap Sort

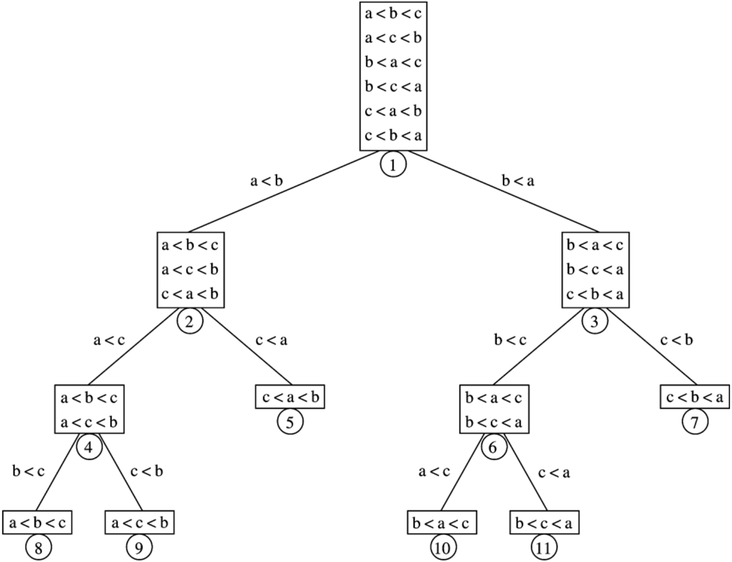




A lower bound for comparison sorting

- All algorithms we have considered so far are based on comparing numbers.
- One can show that any such algorithm is $\Omega(n \log n)$.
- Our proof depends on what is called a **decision tree**.
- Each node of the tree represents a set of orderings **consistent** with all the decisions made so far.
- After each **decision** the number of possibilities is reduced.

Decision Trees



- It is clear from the previous example that in the worst case, the number of comparisons is equal to the **depth** of the tree.
- We will show that the number of comparisons for n elements is $\Omega(n \log n)$ in the worst case.
- to do so we need
- **Lemma**: The number of leaves of a tree of depth d is at most 2^d.
- This is shown by induction on d. The base case is clearly true since the root is the only leaf for $d = 0$.
- Suppose it is true for depth d.
- Any tree of depth $d + 1$ contains the root and two subtrees of depth of at most d.
- By the hypothesis each subtree can have at most 2^d leaves for a total of $2^d + 2^d = 2^{d+1}$ leaves.

- As a corollary to the previous results we have:
- **Lemma** : The depth of a tree of L leaves is at least $\lceil \log L \rceil$, $d \geq \lceil \log L \rceil$.
- In any comparison of n elements there are $n!$ permutations and thus $n!$ leaves for the decision trees which means the decision tree has depth of at least $\log n!$.

$$\begin{aligned}
\log n! &= \log n \cdot (n-1) \ldots 1 \\
&= \log n + \log(n-1) + \ldots + \log 1 \\
&\geq \log n + \log(n-1) + \ldots + \log n/2 \\
&\geq (n/2) \log(n/2) \\
&= \Omega(n \log n)
\end{aligned}$$

Counting Sort

- The previous lower bound does not mean that sorting is $\Omega(n \log n)$.
- It means that **comparison** sorting is $\Omega(n \log n)$.
- Some sorting algorithm do not do any comparison.
- As an example we look at **counting sort**.
- If we know that the numbers we need to sort are all less than certain number k then we can use counting sort

- Given an array A with n elements all less or equal to some value k.
- Maintain an array C such that for each $C[i] = j$, j is the number of values in A that are equal to i

  for $i = 0$ **to** $n - 1$ **do**
  $\quad C[A[i]] \leftarrow C[A[i]] + 1;$
  end

- Once we are done each value is in its "relative position". We scan C again and transfer the values back to A.

$offset \leftarrow 0;$
for $i = 0$ **to** k **do**
$\quad$ **for** $j = 0$ **to** $C[i]$ **do**
$\quad\quad A[offset] \leftarrow i;$
$\quad\quad offset \leftarrow offset + 1;$
$\quad$ **end**
end

Example Counting Sort

- As an example consider the array shown in the figure below
- After scanning it and putting each element in the proper place in C we find that there are two 0's,no 1's,two 2's,three 3's, no 4's and one 5.
- Next we scan C left to right and write the appropriate value in A.

	0	1	2	3	4	5	6	7
A	2	5	3	0	2	3	0	3

C	2	0	2	3	0	1

Different implementation

- Counting sort can be implemented in a more convenient manner
- This is done by doing an extra pass on the array C where we add the value of a given bucket with the previous value.
- from the previous example the array C becomes

	0	1	2	3	4	5	6	7
A	2	5	3	0	2	3	0	3

C	2	2	4	7	7	8

- Now the code is simplified

for $i = n - 1$ **to** 0 **do**

$val \leftarrow A[i]$;

$C[val] \leftarrow C[val] - 1$;

$index \leftarrow C[val]$;

$B[index] \leftarrow val$;

end

Radix sort

- What if the maximal value is large? can we still use counting sort?
- It turns out that we can use **multiple passes** of counting sort in such a situation.
- The basic idea is to do counting sort on each digit separately starting with the least significant digit.