Sabancı University
Faculty of Engineering and Natural Sciences

CS301 – Algorithms

Homework 2

Due: March 12, 2024 @ 23.55
(upload to SUCourse)

PLEASE NOTE:

- Provide only the requested information and nothing more. Unreadable, unintelligible, and irrelevant answers will not be considered.
- Submit only a PDF file. (-20 pts penalty for any other format)
- Not every question of this homework will be graded. We will announce the question(s) that will be graded after the submission.
- You can collaborate with your `TA/INSTRUCTOR ONLY` and discuss the solutions of the problems. However, you have to write down the solutions on your own.
- Plagiarism will not be tolerated.

Late Submission Policy:

- Your homework grade will be decided by multiplying what you normally get from your answers by a "submission time factor (STF)".
- If you submit on time (i.e. before the deadline), your STF is 1. So, you don't lose anything.
- If you submit late, you will lose 0.01 of your STF for every 5 mins of delay.
- We will not accept any homework later than 500 mins after the deadline.
- SUCourse's timestamp will be used for STF computation.
- If you submit multiple times, the last submission time will be used.

Question 1

(a) What is the form of the input array that triggers the worst case of the insertion sort?

Answer:

Tan Ufuk Çelik
ID:28285

The worst case for the insertion sort algorithm is when the input array is reverse sorted (descending order). That is, the array is arranged, therefore that the largest element is at the beginning of the array and the smallest element is at the end of the array. In this situation, the maximum number of comparisons and substitutions will be required to place each element in its correct place.

At each step, Insertion Sort tries to insert the current element into the correct position of the sorted subarray. If the array is sorted in descending order, each element has to be placed at the very beginning of the sorted subarray. This requires all elements in the sorted subarray to be shifted one position to the right. Since this process is repeated for each element of the algorithm, the number of operations becomes $O(n^2)$, which is a function of the array size.

(b) What is the complexity of this worst–case behavior in Θ notation?

Answer:

Tan Ufuk Çelik
ID: 28285

The worst case for Insertion sort is when the input array is reverse order. In this case each element will need to be added to the beginning of the already sorted section, which will require shifting the entire sorted section one by one each time. Therefore, for each element in the array except the first element, comparison and replacement operations will need to be performed as many as the number of elements up to that element. The total number of these operations forms an arithmetic series of the form $1+2+3+\ldots+(n-1)$, and the sum of this series is calculated by the formula $n(n-1)/2$

→ Therefore, this expression transforms into a square term, where the n^2 term dominates for large values of n. So, the worst-case running time is expressed as $\Theta(n^2)$.

(c) Explain how this particular form of the array results in this complexity.

Answer:

The reason why Insertion Sort has a worst-case complexity of $O(n^2)$ is directly related to the fact that the array is sorted in descending order. This causes the algorithm to experience the worst case scenario at each step.

Tan Ufuk Çelik
ID: 28285

1. Insertion sort logic:
   Insertion sort works by creating a sorted subarray and inserts the next element at the correct location of this sorted subarray at each step. Initially, the sorted subarray consists of only the first element, and the remaining elements are added to this sorted region one by one.
2. Descending order case: If the array is sorted in descending order, each new element will be smaller than all existing elements in the sorted subarray. This means that each new element must be added to the very beginning of the sorted subarray.
3. Maximum shift operation:
   If each element is added to the beginning, all existing elements in the sorted subarray must be shifted one position to the right. this process is repeated for each new element added.
4. Square number operations:
   These shifting operations increase in proportion to the size of the array n. The first element requires no shift, the second element requires one shift, the third element requires two shift, therefore the last element requires $(n-1)$ shift. This creates an arithmetic series of the form $1+2+3+\ldots+(n-1)$ whose sum is $n(n-1)/2$. This formula turn into a square term dominated by the n^2 term for large values of n.

For example:

set = [5,4,3,2,1]

step 1 $\longrightarrow$ the first element 5 is considered in place because by itself it is sorted.

step 2 $\longrightarrow$ compare 4 and 5. 4 is smaller than 5, so put it to the front [4,5,3,2,1]

step 3 $\longrightarrow$ Firstly compare 3 and 5, then with 4. It is smaller then both, so $\longrightarrow$ [3,4,5,2,1]

step 4 $\longrightarrow$ [2,3,4,5,1]

step 5 $\longrightarrow$ [1,2,3,4,5]

Question 2

(a) What is the form of the input array that triggers the best case of the insertion sort?

Tan Ufuk Çelik
ID: 28285

The best case for the Insertion sort algorithm is when the input array is in ascending order from start to finish. In this case, since each element is already in the right place, the algorithm checks each element but does not perform any displacement. Only one comparison is made for each element and the element is already in the correct position. This ensures that the running time of the algorithm is O(n) at best, because only one comparison is made for each element, making a total of n comparisons across the array.

(b) What is the complexity of this best–case behavior in Θ notation?

Tan Ufuk Çelik
ID: 28285

The best case performance of the insertion sort algorithm is expressed as Θ(n). The notation Θ tightly defines both the lower and upper bounds of an algorithm's running time, indicating that the algorithm has at best linear time complexity. Which is, if the the input array is already sorted and only one comparison is made for each element, making n comparisons total. Therefore, the best case, the time complexity of Insertion sort is set to Θ(n).

(c) Explain how this particular form of the array results in this complexity.

Reasons:

Tan Ufuk Çelik
ID: 28285

1. Already Sorted array

If the input array is already sorted in ascending order, there is no need to change the positions of the elements as each element is in its own place. This minimizes extra operations at each step of the algorithm.

2. Single comparison Process:

While each element is added to its position in the sorted subarray, it is compared only with the previous element. Therefore algorithm immediately detects that the element is in the correct place and leaves the element in place. This means a single comparison is made for each element

3. Linear Process:

As a result, this process is repeated for all elements through the array and only one comparison is made for each. This means that the algorithm will make n comparisons in total, where n is the number of elements of the array

4. $\Theta(n)$ Complexity

Given these situation the running time of the insertion sort increases in direct proportion to the input size, and it is completed in linear time $\Theta(n)$.

For Example

array = [1, 2, 3, 4, 5]

Step 1 → the first element 1 is considered and it's place is already sorted.
step 2 → compare 2 with 1. (2 is bigger than 1. Therefore no change)
step 3 → compare 3 with 2. (no change)
step 4 → compare 4 with 3. (no change)
step 5 → compare 5 with 4. (no change)

Question 3

Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any comparison sort stable. How much additional time and space does your scheme entail?

Tan Ufuk Çelik
10:28285

A stable sorting algorithm is an algorithm that preserves the order of elements with equal key values in the input array and in the output array

Table:

	Insertion Sort	Merge Sort	Heap Sort	Quick Sort
Stable	✓	✓		
Not-Stable			✓	✓ → generally it is not stable

- Insertion sort → It is stable. Preserves the original order between elements with with equal values
- Merge sort → It is stable sorting algorithm. It preserves
- Heap sort → It is unstable sorting algorithm. It uses the stack data structure and cannot preserve the original order of elements with equal values.
- Quick Sort → It is sorting algorithm that is generally unstable. Due to pivot selection and division, the original order of elements with equal values may be dissorted.

As a simple scheme, I can say that adding an index to each element indicating its position in the original array. This method provides a stable ranking by including the original positions in the comparision criterion when elements with equal values are compared. These are steps:

1. Expand elements:
   Expand each element to include the element itself and its index in the original array
   For example, the sequence [3, 1, 4, 1] and extended sequence [(3,0), (1,1), (4,2), (1,3)]
2. Modify the comparison function:
   If the values are equal, then compare their original state
3. Sort and recycle:
   Perform sorting on extended elements. Once the sorting is complete, get back the original element values using the first element of each pair.

This scheme can make any comparative sorting algorithm stable. However it requires additional time and space to store the indexes of each element.

* Since the time taken for comparing indexes is constant (it involves a simple arithmatic operation) we say that the time complexity is $O(1)$

* Also, we use additional space to store n elements indixes, so additional $O(n)$ space is needed.

Question 4

(a) Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in .... time if the intermediate stable sorting algorithm is Counting Sort.

Answer:

Tan Ufuk Çelik
10:28285

the answer is: $O(d(n+k))$

RADIX-SORT sorts each digit of numbers separately, starting from the least significant digit to the most significant digit.

$d \rightarrow$ the number of digits of the numbers

$n \rightarrow$ the total number of digits

$k \rightarrow$ maximum possible digit value.
(the # of values that each digit can take)

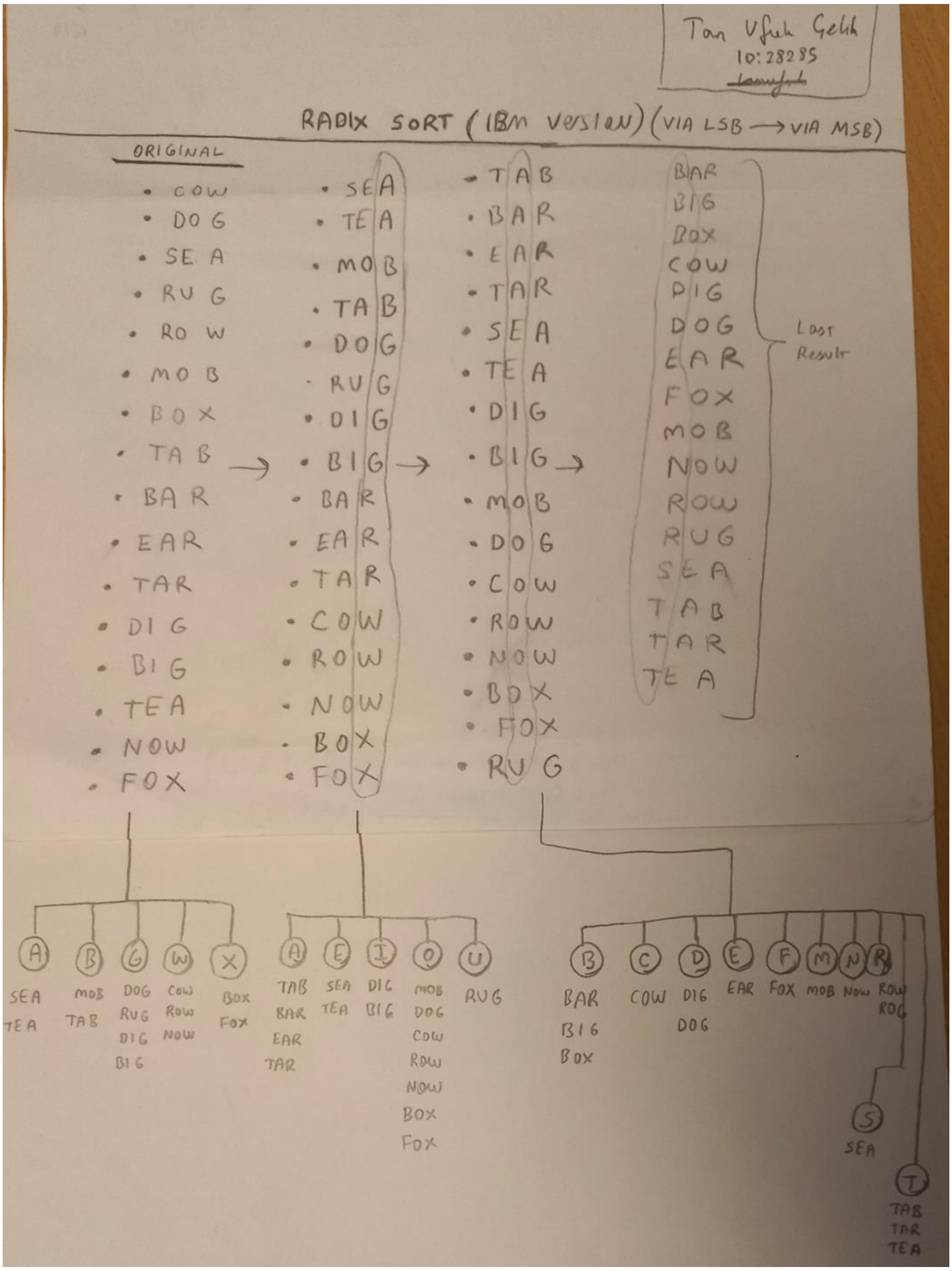
therefore, if we assume that counting sort runs in $O(n+k)$ time for each digit, the total running time of RADIX-SORT will be $O(d(n+k))$, because there are d digits in total.

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Figure 1: The operation of radix sort on seven 3-digit numbers. The leftmost column is the input. The remaining columns show the numbers after successive sorts on increasingly significant digit positions. Tan shading indicates the digit position sorted on to produce each list from the previous one.

(b) Using Figure 1 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG,SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

Answer:



Question 5

The pseudo-code for Quicksort algorithm is given below.

Algorithm 1 Quicksort algorithm

```
Function Quicksort(array A, l, r):
    if r − l + 1 ≤ 1 then
        return
    end
    p ← ChoosePivot(A, l, r)
    Partition(A, p, l, r)
    Quicksort(A, l, p − 1)
    Quicksort(A, p + 1, r)

Function Partition(A, p, l, r):
    i ← l + 1
    for j ← l + 1 to r do
        if A[j] ≤ p then
            swap A[i] with A[j]
            i ← i + 1
        end
    end
    swap A[i − 1] with p
    return i − 1
Function ChoosePivot(A, l, r):
    return A[⌊(l + r)/2⌋]
```

(a) Write down the recurrence for the running time for the case where the algorithm chooses the median as the pivot at each iteration.

Answer:

Tan Ufuk Çelik

ID:28285

$$T(n) = \underbrace{2\,T\left(\frac{n}{2}\right)}_{} + \underbrace{\Theta(n)}_{}$$

It shows that two recursive calls on each half of the array (Because the pivot is median. And also, each subarray has roughly half of the elements in the array)

For the portioning part, which is all n elements are iterated split them around the pivot.

$T(n) \rightarrow$ It is runing time for a problem of size n.

$2T(n/2) \rightarrow$ It is sum of the running time for two equal sized subproblems.

$\Theta(n) \rightarrow$ It is linear time required for the division and merging the array.

(b) Calculate a tight bound for this recurrence using the Master Theorem.

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

Tan Ufuk Çelik
ID: 28285

Formula of master theorem:

$T(n) = aT(\frac{n}{b}) + f(n)$, then if $a \geq 1, b > 1$ and $f(n)$ is asymptotically positive

You can apply one of this three case;

1. If $f(n) = O(n^{\log_b^{a-\varepsilon}})$ for $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b^a})$
2. If $f(n) = \Theta(n^{\log_b^a})$, then $T(n) = \Theta(n^{\log_b^a} \log n)$
3. If $f(n) = \Omega(n^{\log_b^{a+\varepsilon}})$ for some $\varepsilon > 0$,
   and if $af(n/b) \leq cf(n)$
   for some $c < 1$, and for large n, then $T(n) = \Theta(f(n))$

In our equation, $a = 2$, $b = 2$, $f(n) = \Theta(n)$.

Because of $f(n) = \Theta(n^{\log_2^2}) = \Theta(n)$, we can apply the second case.

* $f(n)$ rated with $n^{\log_b^a}$ (n^1)

Therefore; case 2, $T(n)$ is tight bound

$$\boxed{T(n) = \Theta(n \log n)}$$

(c) [5 points] Write down the recurrence for the running time for the case where the algorithm chooses the smallest element in the array as the pivot at each iteration.

<u>Answer:</u>

Tan Ufuk Çelik
ID: 28285

If the algorithm chooses the smallest element of the array as the pivot at each iteration, this leads to the worst-case scenerio of the quick sort algorithm. Because in this case, the entire array except one element (whis is pivot) at a time is treated as a single as a single subarray and the other subarray remains empty. this will cause unbalanced splitting of the algorithm and it is worst performance.

As a result, in this scenerio; only one element of the array at a time is "sorted" and the remaining n-1 elements will be considered from the quick sort algorithm again. therefore, the runtime iteration will be: $\boxed{T(n) = T(n-1) + \theta(n)}$

(d) [5 points] Calculate a tight bound for this recurrence using the iteration method.

Answer:

Tan Ufuk Çelik
ID: 28285

~~[illegible]~~

In the beginning, we have

$$T(n) = T(n-1) + \theta(n)$$

1. $T(n) = T(n-1) + \theta(n)$
2. $T(n-1) = T(n-2) + \theta(n-1)$
3. $T(n-2) = T(n-3) + \theta(n-2)$

$T(3) = T(2) + \theta(1)$

$T(2) = T(1) + \theta(2)$

$T(1) = T(0) + \theta(1)$

$T(0) = \theta(1) \longrightarrow$ It represents the base case (constant)

$$T(n) = \left(\theta(n) + \theta(n-1) + \theta(n-2) + \dots + \theta(2) + \theta(1)\right)$$

$$\downarrow$$

$$T(n) = \theta(1 + 2 + \dots + (n-1) + n)$$

$$T(n) = \theta\left(\frac{n.(n+1)}{2}\right)$$

$$T(n) = \theta\left(\frac{1}{2}n^2 + \frac{1}{2}n\right)$$

$$\boxed{T(n) = \theta(n^2)}$$