CSCI 4470 Algorithms

Part II Sorting and Order Statistics

- 6 Heapsort
- **7 Quicksort**
- 8 Sorting in Linear Time
- 9 Medians and Order StatisticsHeapsort Algorithms Notes

Chapter 7: Quicksort

```
7 Quicksort
  7.1 Description of quicksort
  7.2 Performance of quicksort
  7.3 A randomized version of quicksort
  7.4 Analysis of quicksort
```

> Algorithms: https://algostructure.com/sorting/selectionsort.php

Algorithm Name	Best-case	Average-case	Worst-case	Memory	Stable
MergeSort	$n \log(n)$	$n \log(n)$	$n \log(n)$	worst: n	yes
HeapSort	$n \log(n)$	$n \log(n)$	$n \log(n)$	1	no
InsertionSort	n	n^2	n^2	1	yes
QuickSort	$n \log(n)$	$n \log(n)$	n^2	average: $\log(n)$ worst: n	no
Bubblesort	n	n^2	n^2	1	yes
SelectionSort	n^2	n^2	n^2	1	no

QuickSort Overview

- **Definition**, Quicksort is a divide-and-conquer algorithm utilized for sorting arrays or lists.

Algorithm Structure

```
QuickSort(A, p, r)
if p < r      // Partition the subarray around the pivot, which ends up in A[q].
    q = Partition(A, p, r)
    QuickSort(A, p, q - 1)    // recursively sort the low side
    QuickSort(A, q + 1, r)    // recursively sort the high side
```

Quicksort Algorithm Process

1. Divide

- **Partition**: The array is rearranged into two subarrays: `A[p...q-1]` and `A[q+1...r]`. Each element in `A[p...q-1]` is $\leq$ `A[q]`, and each element in `A[q+1...r]` is $>$ `A[q]`. The `q` index is computed in this step, placing `A[q]` in the correct sorted position, a pivotal step in quicksort.

2. Conquer

- **Recursive Sorting**: The two subarrays `A[p...q-1]` and `A[q+1...r]` are sorted recursively through quicksort.

3. Combine

- **No Additional Steps Required**: The array is sorted in place, eliminating the need for extra steps during the combination phase.

4. Base Case

- **Termination**: Recursion ends when a subarray has one or no elements, inherently sorted at this point.

Properties of Quicksort Algorithm

- **In-Place**: Quicksort sorts the elements directly within the dataset and does not require additional space.
- **Not Stable**: Quicksort may change the relative order of equal elements, making it unstable.

Why Not Stable?

Consider an array A[p,…,r] = 2 9 9 6 7.
During the partitioning in Quicksort:

- The element 6 may be swapped with the first 9.
- This swap reverses the order of the two 9's, demonstrating the instability of Quicksort.

Performance

- ***Best Case**: When the pivot is ideally chosen, it results in a time complexity of $\Theta(n \log n)$.*
- ***Worst Case**: In the worst scenario, the time complexity can degrade to $\Theta(n^2)$.*

- ***Average Case**: Generally, it tends to have a time complexity of $\Theta(n \log n)$.*
- **To avoid the WORST-CASE scenario:**
    - Use a good pivot strategy, such as choosing the median element as the pivot.
    - Randomly select the pivot element to ensure the algorithm has an average-case time complexity of $\Theta(n \log(n))$.

Four Regions Maintained by Partition Function

```
PARTITION(A, p, r)
x = A[r]     // the pivot
i = p - 1       //highest index into the low side
for j = p to r - 1       // process each element other than the pivot
     if A[j] ≤ x        // does this element belong on the low side?
         i = i + 1       // index of a new slot in the low side
         exchange A[i] with A[j]     // put this element there
exchange A[i + 1] with A[r]      // pivot goes just to the right of the low side
return i + 1      // new index of the pivot
```

Initialization:

```
| Unrestricted Area | Pivot (x)|
```

- **Unrestricted Area**: All elements except the pivot.
- **Pivot x**: The element to partition the array around.

During Partition:

```
| ≤ Pivot | > Pivot | Unrestricted Area | Pivot |
```

- $\leq$ **Pivot**: Elements found to be less than the pivot.
- $\geq$ **Pivot**: Elements greater than the pivot.
- **Unrestricted Area**: Unexamined elements.
- **Swapping**: Elements less than the pivot in the unrestricted area are swapped into the < Pivot region.

Final Step of Partition:

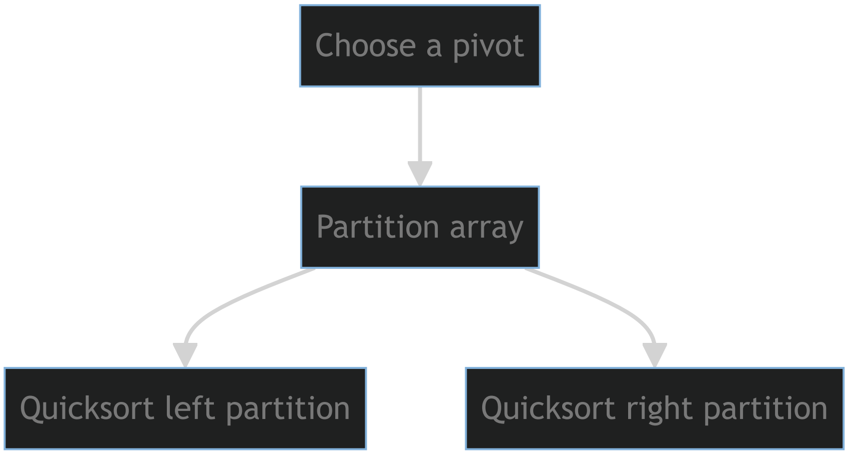
```
| ≤ Pivot | Pivot (x) | > Pivot |
```

- **Final Swap**: *The Unrestricted region is empty*. The pivot x swaps with the last element of the < Pivot region, finding its correct position.

Post Partition:

```
| ≤ Pivot | Pivot (x) | > Pivot |
```

- **Partition Result**: The array is now segmented into two halves for subsequent recursive calls, with the pivot in its correct sorted position.



The Partition Function Example from Class Note

- The partition process step by step using the array $[3, 7, 8, 2, 10, 5]$ and the partition function you provided. choose the last element as the pivot, which is 5 in this case.

Initial Array:

- **Pivot**: 5, **p**: Start of the array (index 0) and **r**: End of the array (index 5)

```
p                               r   x = 5
A[0]  A[1]   A[2]  A[3]  A[4]  A[5]
3      7      8     2    10     5
```

Step 1: Set x as the pivot element

```
x = 5   // pivot x = A[r]
```

Step 2: Set i to one less than the starting index

```
i = p - 1 = -1 // p = A[0] = 0
```

Step 3-6: Start the loop with j = p and iterate until r-1

First Iteration (j = 0) (j = p)

```
3  7  8  2  10  5
i  j            r
```

- $A[j] = 3$ which is less than $x = 5$, so we increase i by 1 and swap $A[i]$ and $A[j]$.
- i becomes 0.

After First Iteration

```
3  7  8  2  10  5
i     j         r
```

Second Iteration (j = 1)

- $A[j] = 7$ which is greater than $x = 5$, so we do nothing and move to the next iteration.

After Second Iteration

```
3  7  8  2  10  5
i        j      r
```

Third Iteration (j = 2)

- $A[j] = 8$ which is greater than $x = 5$, so we do nothing and move to the next iteration.

After Third Iteration

```
3  7  8  2  10  5
i            j  r
```

Fourth Iteration (j = 3)

- $A[j] = 2$ which is less than $x = 5$, so we increase i by 1 and swap $A[i]$ and $A[j]$.
- i becomes 1.

After Fourth Iteration

```
3  2  8  7  10  5
   i         j  r
```

Fifth Iteration (j = 4)

- $A[j] = 10$ which is greater than $x = 5$, so we do nothing and end the loop.

Step 7: Swap $A[i + 1]$ and $A[r]$

- i is currently 1, so we swap $A[i+1]$ and $A[r]$.

After Step 7

```
3  2  5  7  10  8
      i  r
```

Step 8: Return $i+1$

- The function returns $i+1$ which is 2.

After Partition Function called, the array is partitioned into two parts, elements less than 5 and elements greater than 5, with 5 in its correct position.

The next steps in quicksort would be to recursively sort the subarrays on either side of the pivot.

Example of Partition Function of Quicksort Algorithm from Class Note

Using the array: `{5,18,10,16,9,12,56,20,13}` and using the `PARTITION` function of the quicksort algorithm and pivot `13`.

1. Initial Setup

- **Array**: `{5, 18, 10, 16, 9, 12, 56, 20, 13}`
- **Pivot**: `13`, **i**: `-1`, **j**: `0`, **p**: 5, A[0], and **r**: 13 A[8]

```
i
↓
-1 [5, 18, 10, 16, 9, 12, 56, 20, 13]
    ↑
    j
```

2. First Iteration

- **i**: `-1 -> 0` (incremented as `5 < 13`)
- **j**: `1`

```
i
↓
0 [5, 18, 10, 16, 9, 12, 56, 20, 13]
   ↑
   j=1
```

3. Second Iteration

- **i**: `0` (remains same as `18 > 13`)
- **j**: `2`

```
i=0
↓
0 [5, 18, 10, 16, 9, 12, 56, 20, 13]
       ↑
       j=2
```

4. Third Iteration

- **i**: `0 -> 1` (incremented as `10 < 13`)
- **j**: `3`
- Swap `18` and `10`

```
i
↓     18⇆10
1 [5, 10, 18, 16, 9, 12, 56, 20, 13]
           ↑
           j=3
```

⇆

5. Fourth Iteration

- **i**: `1` (remains same as `18 > 13`)
- **j**: `4`

```
i
↓
1 [5, 10, 18, 16, 9, 12, 56, 20, 13]
               ↑
               j=4
```

6. Fifth Iteration

- **i**: `2` (incremented as `9 < 13`)
- **j**: `5`
- Swap `18` and `9`

```
i
↓           18⇆9
2 [5, 10, 9, 16, 18, 12, 56, 20, 13]
                  ↑
                  j=5
```

7. Sixth Iteration

- **i**: `3` (incremented as `12` < `13`)
- **j**: `6`
- Swap `16` and `12`

```
i
↓             16⇆12
3 [5, 10, 9, 12, 18, 16, 56, 20, 13]
                     ↑
                     j=6
```

8. Seventh Iteration

- **i**: `3` (remains same as `56` > `13`)
- **j**: `7`

```
i
↓
3 [5, 10, 9, 12, 18, 16, 56, 20, 13]
                         ↑
                         j=7
```

9. Eighth Iteration

- **i**: `3` (remains same as `20` > `13`)
- **j**: `8`

```
i
↓
3 [5, 10, 9, 12, 18, 16, 56, 20, 13]
                             ↑
                             j=8
```

10. Final Swap

- **i**: `4` (incremented)
- Swap `18` and `13`

```
i
↓
4 [5, 10, 9, 12, 13, 16, 56, 20, 18]
                 ↑
                 j
```

11. Conclusion

- The array is now partitioned around `13`.
- Left subarray: `{5, 10, 9, 12}`
- Right subarray: `{16, 56, 20, 18}`

This detailed step-by-step walkthrough should help visualize the `PARTITION` function's execution on the given array.

The Generic Recurrence Relations for the Quicksort Algorithm

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

- $T(n)$: Represents the time complexity to sort an array of n elements
- $T(q-1)$: Represents the time complexity to sort the left subarray, which contains $q-1$ elements
- $T(n-q)$: Represents the time complexity to sort the right subarray, which contains $n-q$ elements
- $\Theta(n)$: Represents the time complexity to partition the array, which is linear with respect to n

Notes for q : q (Pivot)

- q: Pivot's position post-partition. It varies based on pivot choice and element distribution:
  - Smallest/largest pivot: $q = 1$ or n, **worst-case**.
  - Pivot splits array nearly in half: $q \approx \frac{n}{2}$, **best-case**.
  - Otherwise, the choice of the pivot can be $1 < q < n$.

Recurrence Relation Piecewise Function:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T(q-1) + T(n-q) + \Theta(n) & \text{if } n > 1. \end{cases}$$

> This Recurrence Relation is more general and can represent various cases including the best, average, and worst cases depending on the values of q.

> ***To find out the case scenarios of the recurrence relation, which are finding the Minimum or Maximum of the function***

Step 1: Setup the Recurrence Relation, and Assumption

- Assume, $T(n) \leq cn^2$, and $1 \leq q < n$
- $T(n) \leq c(q-1)^2 + c(n-q)^2 + kn$, k is a constant

From the Expression

- $T(n) \leq c\left[(q-1)^2 + (n-q)^2\right] + kn$

Step 2: Differentiate with Respect to q

To find Minimum and Maximum of the function, differentiate each term in the expression with respect to q.

- ***Second Derivative of q is positive, it is minimum, Second Derivative of q is negative, it is maximum***
- $\frac{d}{dq}\left[c\left[(q-1)^2+(n-q)^2\right]+kn\right]=0$

Step 3: Apply the Chain Rule
Applying the chain rule to differentiate the squares:

- $\frac{d}{dq}\left[c(q-1)^2\right]+\frac{d}{dq}\left[c(n-q)^2\right]+\frac{d}{dq}[kn]=0$

Using the chain rule:

- $\frac{d}{dq}\left[c(q-1)^2\right]=2c(q-1)1$
- $\frac{d}{dq}\left[c(n-q)^2\right]=-2c(n-q)1$
- $\frac{d}{dq}[kn]=0$ Since this term does not contain q, its derivative with respect to q is zero:

Step 4: Set Up the Equation to find the critical points

- $2c(q-1)1-2c(n-q)1+0=0$

> to find the critical points where the derivative is zero, which are essential in analyzing the behavior of the function in terms of q.

Step 5: Simplify and Solve for q

- $2cq-2c-2c(n-q)=0$
- $2cq-2c-2cn+2cq=0$

Combine like terms:

- $4cq-2c-2cn=0$

Now, solve for q:

- $4cq=2c+2cn, q=\frac{2c+2cn}{4c}$
- $q=\frac{c(2+2n)}{4c}, q=\frac{2(1+n)}{4}$

Finally, we find:

- $q=\frac{n+1}{2}$

Step 6: Finding the Second Derivative

To find the **second derivative**, differentiate the first derivative with respect to q again. Differentiating the terms we obtained in step 3 gives:

- $f'(n)=2c(q-1)1-2c(n-q)1=0$

So the second derivative of the function with respect to q is:

- $\frac{d^2}{dq^2}\left[2c(q-1)-2c(n-q)\right]=4c$
  - $\frac{d^2}{dq^2}\left[2c(q-1)\right]=2c$
  - $\frac{d^2}{dq^2}\left[-2c(n-q)\right]=-2c$
  - $2c-(-2c)=4c$

Analysis:

- Since the second derivative is positive ($4c>0$), the function has a minimum at the critical point.
- In the worst case, q is either 1 or $n-1$ (unbalanced partition), leading to a higher time complexity.

Step 7: Conclusion

From the analysis, the worst-case scenario for quicksort, where the partition is always unbalanced, leads to a time complexity of $O(n^2)$ based on the given general recurrence relation and the assumption $T(n)\leq cn^2$.

This approach adheres to the requirements you listed, using the general recurrence relation, an assumption, and the first and second derivatives with respect to q to analyze the worst-case scenario for quicksort.

Practical Considerations

- **Pivot Strategy**: Implementing strategies like randomized or median-of-three pivot selection can often optimize performance
- **Small Arrays**: For small arrays, other sorting algorithms like insertion sort might be more efficient

The Recurrence Relation Analysis of Worst Case

$$T(n)=\begin{cases}\Theta(1) & \text{if } n\leq 1,\ \text{(base case)}\\ T(n-1)+\Theta(n) & \text{if } n>1,\ \text{(general case)}\end{cases}$$

- **Base Case**: When $n\leq 1$, the array has at most one element, so it is already sorted, and the time complexity is constant, denoted as $\Theta(1)$.
- **General Case**: When $n>1$, we perform a partition and then recursively sort an array of size $n-1$, incurring a time complexity of $T(n-1)$. The partitioning process itself has a time complexity of $\Theta(n)$, so we add this to the recursive term.

Step 1: Setup the Recurrence Relation
Given the recurrence relation:

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

Assume that

- $T(n) \leq cn^2$, for some constant c, and $(1 \leq q < n)$

Step 2: Substitute the Assumption into the Recurrence
Substituting our assumption into the recurrence gives:

- $T(n) \leq c(n-1)^2 + \Theta(n)$
- Expand the Squared Term $(a-b)^2 = a^2 - 2ab + b^2$
  - $(n-1)^2 = n^2 - 2(n)(1) + 1^2$
  - $(n-1)^2 = n^2 - 2n + 1$

Step 3: Expand and Simplify
Expand and simplify the expression:

- $T(n) \leq c(n^2 - 2n + 1) + \Theta(n)$, then $T(n) \leq cn^2 - 2cn + c + \Theta(n)$

Step 4: Find the Constant Term

To identify the right constant c, set the $\Theta(n)$ term to kn, where k is a constant.

- $T(n) \leq cn^2 - 2cn + c + kn$

Step 5: Derivative to Find Slope Points
To find the slope points, we take the derivative of the right-hand side with respect to n:

- Apply the power rule of differentiation individually to each term. The power rule states that the derivative of n^x with respect to n is $x \cdot n^{(x-1)}$.
  - Derivative of cn^2 with respect to n is $2cn$.
  - Derivative of $-2cn$ with respect to n is $-2c$.
  - Derivative of c with respect to n is 0 because it's a constant.
  - Derivative of kn with respect to n is k.
- $\frac{d}{dn}(cn^2 - 2cn + c + kn) = 2cn - 2c + k$

Step 6: Find Max and Min Slope Points
Setting the derivative equal to zero gives the slope points:

- $2cn - 2c + k = 0$

Solving for n gives:

- $n = \frac{2c-k}{2c}$
- Substitute the value of n, $2c\left(\frac{2c-k}{2c}\right) - 2c + k = 0$

- $\frac{2c \times 2c}{2c} - \frac{2c \times k}{2c} - 2c + k = 0$, $\frac{4c^2}{2c} - \frac{2ck}{2c} - 2c + k = 0$
- $\frac{4c^2 - 2ck - 4c^2 + 2ck}{2c} = 0, 0 = 0$

Step 7: Verify the Solution

verified that substituting the value of n back into the derivative equation results in zero, confirming it is a critical point.

Step 8: Conclusion

From the above steps, we can conclude that $T(n) = \Theta(n^2)$ under the assumption $T(n) \leq cn^2$.

Step 9: To find the suitable constant c and k, we can look at the equation derived in step 6:

- $2cn - 2c + k = 0$

Solving for c gives:

- $2c(n-1) + k = 0$

This equation gives a relationship between c, k, and n. To find the exact values of c and k that satisfy the condition $c > \frac{k}{2}$ and $k < 2c$, we would need additional information or constraints on the values of c and k.

Solution 2:

Given the recurrence relation:

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

Step 1: Setup the Recurrence Relation

- Assume $T(n) \leq cn^2$ for some constant c, and let the constant term in $\Theta(n)$ be kn, where k is a constant.

Step 2: Substitute the Assumption into the Recurrence

- $T(n) \leq c(n-1)^2 + kn$

Step 3: Expand and Simplify

- $T(n) \leq c(n^2 - 2n + 1) + kn$
- $T(n) \leq cn^2 - 2cn + c + kn$

Step 4: Rearrange the Terms

- $T(n) \leq cn^2 - n(2c - k) + c$

Step 5: Find the Condition for n

- To ensure the inequality holds for all n, we need $n(2c-k) > c$, which gives us:
  - $n > \frac{c}{2c-k}$

Step 6: Find the Conditions for c and k

- From the above inequality, we can derive the conditions for c and k:
  - $2c - k > 0$
  - Which gives us two conditions:
    - $c > \frac{k}{2}$
    - $k < 2c$

Step 7: Conclusion

- We have found a solution where $T(n) = \Theta(n^2)$ under the assumption $T(n) \leq cn^2$, and we have derived the conditions for c and k to satisfy the inequality.

Base Case:
For $n = 1$, $T(1) = \Theta(1)$. Let's assume $T(1) = a$ for some constant a.

Recurrence Relation Piecewise Function:

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

Step 1: Setup the recurrence relation, and Assumption

- Assume that $T(n) \geq c \cdot n \log(n)$, where $c > 0$, and $1 \leq q \leq n-1$
- $T(n) \geq cq \log(q) + c(n-q-1)\log(n-q-1) + kn$, k is a constant
- $T(n) = c\,(q\log(q) + (n-q-1)\log(n-q-1) + kn$

Step 2: Differentiate the terms with respect to q, the constants won't affect the result, minimize the function:

- $f(q) = q\log(q) + (n-q-1)\log(n-q-1)$

Apply the product rule of Differentiation $q\log(q)$, where $u = q$ and $v = \log(q)$
$u' = 1$, and $v' = \frac{1}{q}$

- $\frac{d}{dq}(q\log(q)) = q \cdot \frac{1}{q} + \log(q) \cdot 1 = \log(n) + 1$

Apply the product rule of Differentiation $(n-q-1)\log(n-q-1)$, where $u = n-q-1$ and $v = \log(n-q-1)$
$u' = -1$, and $v' = \frac{-1}{n-q-1}$

- $\frac{d}{dq} = ((n-q-1)\log(n-q-1)) = (n-q-1) \cdot \frac{-1}{n-q-1} + \log(n-q-1) \cdot (-1)$
- $= -1 - log(n-q-1)$

Combine two terms,

- $f'(q) = \log(q) + 1 - \log(n-q-1) - 1$
- Simplify, $f'(q) = \log(q) - \log(n-q-1)$

Step 3: Find q for which $f'(q) = \log(q) - log(n-q-1) = 0$, we get:

Apply the properties of logarithms

- $\log(n-q-1) - \log(q) = \log(\frac{q}{n-q-1}) = 0$
- $\frac{q}{n-q-1} = 0$,$q = n-q-1$, then $2q = n-1$, $q = \frac{n-1}{2}$

Step 4: Find n, using $q = \frac{n-1}{2}$, $1 \leq q \leq n-1$
$n = 1$, makes $q = 0$ which is out of the validate bound, so pick $n \geq 2$

- $f(q) = q\log(q) + (n-q-1)\log(n-q-1)$
- $\geq \frac{n-1}{2}\log(\frac{n-1}{2}) + \left(n - \frac{n-1}{2} - 1\right)\log\left(n - \frac{n-1}{2} - 1\right)$
- $= (n-1)\log(\frac{n-1}{2})$

Apply to the $T(n)$, for $n \geq 2$

- $T(n) \geq c \cdot (n-1)\log(\frac{n-1}{2}) + \Theta(n)$
- $= c \cdot (n-1)\log(n-1) - c \cdot (n-1) + \Theta(n)$
- $= cn\log(n-1) - c\log(n-1) - c(n-1) + \Theta(n)$
- $\geq cn\log(\frac{n}{2}) - c\log(n-1) - c(n-1) + \Theta(n)$, since $n \geq 2$
- $= cn\log(n) - cn - c\log(n-1) - cn + c + \Theta(n)$
- $= cn\log(n) - (2cn + c\log(n-1) - c) + \Theta(n)$
- $\geq cn\log(n)$, $T(n) \in \Omega(n\log(n))$ is true

Step 5: Find out Minimum (second derivative of q''). If $f''(q) > 0$, then $q = \frac{n-1}{2}$ is indeed a minimum.

By differentiating $f'(q) = \log(q) - log(n-q-1)$:

- $\frac{d}{dq}\log(q) = \frac{1}{q}$,

Apply the chain rule $f(g(x)) = f'(g(x)) \cdot g'(q)$ for the term $\log(n-q-1)$
where $f(x) = log(x)$ and $g(q) = n-q-1$

- $f'(x) = \frac{1}{x}$, and $g'(q) = -1$
- $\frac{d}{dq}\log(n-q-1) = \frac{-1}{n-q-1}$

Combine the terms for $f''(q)$

- $f''(q) = \frac{1}{q} + \frac{1}{n-q-1}$

For $q = \frac{n-1}{2}$, $f''(q) > 0$, confirming that it's a minimum.

Therefore, since we can pick the constant c small enough so that the $\Theta(n)$ term dominates the quantity $2cn + c\log(n-1) - c$. Thus, the best-case running time of quicksort is $\Omega(n\log(n))$.

The Recurrence Relation Analysis of Average Case

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Case 1: $n \leq 1$

- For n less or equal to 1, the time complexity is constant, denoted as $\Theta(1)$.

Case 2: $n > 1$

- For n greater than 1, the array is split into two parts to be sorted recursively, with a linear time complexity for partitioning and merging.

```
     _______Level 0: T(n) = cn_______
        /                           \
Level 1: T(n/10)          Level 1: T(9n/10)
      /        \                /           \
Level 2: T(n/100)  Level 2: T(9n/100)  Level 2: T(n/10)  Level 2: T(81n/100)
    /    \         /    \        /    \        /    \        /    \
   ...   ...     ...   ...     ...   ...     ...   ...     ...   ...

Level n: T(1) ... T(1) ...    T(1) ...      T(1)

all terms eventually become T(1), biggest size will take longest time
```

Given that: $T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + \Theta(n)$

- prove that $T(n) \in \Theta(n\log(n))$
- $T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + cn$

Level 0: The initial problem size is n.

- **L0**: $T(n) = cn$

Level 1: First Level of Recursion

- $T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right)$
- **L1:** The problem is divided into two ***subproblems: one of size $\frac{n}{10}$ and the other of size $\frac{9n}{10}$.***

Level 2: Second Level of Recursion

Apply the recurrence relation to each term (2 Terms from Level 1):

- ***the fraction rule:*** $\frac{\frac{a}{b}}{c} = \frac{a}{b \cdot c}$

For $T\left(\frac{n}{10}\right)$**:**

- $T\left(\frac{n}{10}\right) = T\left(\frac{\frac{n}{10}}{10}\right) + T\left(\frac{9\frac{n}{10}}{10}\right) + \left(\frac{cn}{10}\right) = T\left(\frac{n}{(10)^2}\right) + T\left(\frac{9n}{(10)^2}\right) + \left(\frac{cn}{10}\right)$

For $T\left(\frac{9n}{10}\right)$**:**

- $T\left(\frac{9n}{10}\right) = T\left(\frac{\frac{9n}{10}}{10}\right) + T\left(\frac{9\frac{9n}{10}}{10}\right) + \left(\frac{9n}{10}\right) = T\left(\frac{9n}{(10)^2}\right) + T\left(\left(\frac{9}{10}\right)^2 n\right) + \left(\frac{9cn}{10}\right)$

Combining the results gives the terms at level 2:

- $T(n) = \left[T\left(\frac{n}{10^2}\right) + T\left(\frac{9n}{10^2}\right) + \Theta\left(\frac{n}{10}\right)\right] + \left[T\left(\frac{9n}{10^2}\right) + T\left(\frac{9^2 n}{10^2}\right) + \Theta\left(\frac{9n}{10}\right)\right] + \Theta(n)$

Identifying the terms at level 2:

- $T\left(\frac{n}{10^2}\right),\ T\left(\frac{9n}{10^2}\right),\ T\left(\frac{9n}{10^2}\right),\ T\left(\frac{9^2 n}{10^2}\right)$
- $T\left(\frac{n}{10^2}\right) + T\left(\frac{9n}{10^2}\right) + T\left(\frac{9n}{10^2}\right) + T\left(\frac{9^2 n}{10^2}\right)$

Observe that: $T(\frac{n}{10^2})$ is the smallest size of problem, and $T\left(\frac{9^2 n}{10^2}\right)$ is the biggest size of the problem

- At any level, the size of problem will be $\frac{n}{10^i}$,
- The height of the left most is $\log_{10}(n)$
  - $\frac{n}{10^i} = 1, (10^i)\frac{n}{10^i} = (10^i)1, n = 10^i$
  - $\log_{10}(n) = \log_{10}(10^i), \log_{10}(n) = i$
- The size of problem of the right most tree, $\left(\frac{9n}{10}\right)^i$
  - $\left(\frac{9n}{10}\right)^i = 1, \left(\frac{9n}{10}\right)^i = 1,$
  - $\log_{\frac{10}{9}} \cdot \left(\frac{9n}{10}\right)^i = \log_{\frac{10}{9}} \cdot (n)$

$T(n) \in O(n \log(n)),$
L.M.B $T(n) \geq cn \log_{10}(n)$
R.M.B $T(n) \leq cn \log_{\frac{10}{9}} n$

Last Level of Recursion

- Apply recurrence until $n \leq 1$ (base case).
- **Last Level:** Tree expands until $n \leq 1$, reaching a constant time complexity, $\Theta(1)$.

AVERAGE CASE BEHAVIOR (平均情況行為)

Given a split a to $(1-a)$, where $0 \leq a \leq \frac{1}{2}$,

Calculation Steps:

1. Split Ratio:

- Given a 7-to-3 split: $7(left) + 3(right) = 10$ total parts.

2. Fraction:

- Larger partition is $\frac{7}{10}$ of the total.

3. Reciprocal:

- Reciprocal of the fraction: $\frac{10}{7}$.

4. Height Calculation:

- Height of tree: $\log_{\frac{10}{7}} n$, using base $\frac{10}{7}$.

This is where $\frac{10}{7}$ comes from in the height calculation.

What if the partition always produces a 7-to-3 proportional split?

What is the cost of each level?

- Cost per level = $O(n)$

What is the height?

- Height of tree = $\log_{\frac{10}{7}} n = \Theta(\log(n))$

What is $T(n)$? $T(n)$

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

Randomized Quicksort Algorithm

Overview:

- Utilizes a random number generator for behavior determination.

Advantages:

- Ensures uniform data distribution, unaffected runtime by input order.

Effect:

- Doesn't alter the worst-case runtime, enhances average case reliability.

Randomizing Significance:

- Ensures predictable average case scenarios.

Partition Process:

- Random pivot selection for improved efficiency.

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)     // Randomly select a pivot index `i` between `p` and `r`.
exchange A[r] with A[i]    // Swap the randomly selected element with the last element.
return PARTITION(A, p, r)    // `PARTITION` function to partition the array around the pivot.

RANDOMIZED-QUICKSORT(A, p, r)
if p < r       // Continue if the start index `p` is less than the end index `r`.
 q = RANDOMIZED-PARTITION(A, p, r)   // `RANDOMIZED-PARTITION` to get partition index `q`.
 RANDOMIZED-QUICKSORT(A, p, q - 1)     // Recursively sort the subarray b/4 partition index.
 RANDOMIZED-QUICKSORT(A, q + 1, r )     // Recursively sort the subarray after the partition index.
```

Purpose:

- `RANDOMIZED-PARTITION(A, p, r)` : Randomly selects and partitions around a pivot.
- `RANDOMIZED-QUICKSORT(A, p, r)` : Sorts `A` using randomized partition recursively.

Analysis of RANDOMIZED-QUICKSORT Algorithm

`Quicksort()` Function

```
QUICKSORT(A, p, r)
if p < r
   q = PARTITION(A, p. r )
   QUICKSORT(A, p, q - 1)
   QUICKSORT(A, q + 1, r)
```

How many times is Quicksort function called?

> $n-1$ times, recursively until array is divided into size 1 subarrays.

How many elements become a pivot?

> $n-1$ pivot, one pivot in each recursive call.

What makes the runtime of QuickSort differ for two inputs of size n?

> Pivot choice and initial element order. Good pivot choices lead to faster sorts.

What is the complexity of the partition function inside of the quicksort function?

> $\Theta(n)$, iterating over the entire array segment (p to r) in the worst case.

What is the complexity of two recursive calls of quicksort function inside of the quicksort?

> **Best Case:** $\Theta(\log n)$, equal array division, the depth of the recursive tree is $\log(n)$.

> **Worst Case:** $\Theta(n)$, uneven array division, one subarray is 0, another subarray is $n-1$.

```
PARTITION(A, p, r)
x = A[r]    // the pivot
i = p - 1     // highest index into the low side
for j = p to r - 1    // process each element other than the pivot
  if A[j] ≤ x     // does this element belong on the low side?
    i = i + 1     // index of a new slot in the low side
    exchange A[i] with A[j]   // put this element there
exchange A[i+1] with A[r]     // pivot goes just to the right of the low side
return i + 1    // new index of the pivot
```

`PARTITION() Function` as below:

- **p (p)**: The start index of the array segment to partition.
- **r (r)**: The end index of the array segment, where the pivot element is located.
- **i (i)**: Tracks the last index of an element ≤ pivot. Initially set to p-1.
- **j (j)**: Used to iterate over the array segment from p to r-1 to find elements ≤ pivot.
- Each variable plays a vital role in partitioning the array correctly around the pivot element.

How many times is the Partition function called?

> $n-1$ times in the worst case

> In the worst case, the partition function is called $n-1$ times, once for each element except the last one.

How much work is done in Partition outside the for loop?

> Constant work
> Outside the loop, only a few operations are performed, which take a constant amount of time.

What is doing inside of the for loop?

> **Iterating:** From p to $r-1$, checking each element against the pivot.

> **Comparing:** Each element with the pivot element $A[r]$.

> **Swapping:** If $A[j]$ ≤ $A[r]$, then i is increased by 1, and $A[i]$ is swapped with $A[j]$.

> **Partitioning:** Ensuring elements ≤ pivot are on the left, and elements > pivot are on the right.

How many times is the loop executed?

> $r - p$ times
> The loop iterates from p to $r - 1$, so it is executed $r - p$ times.

If the data is sorted, which lines execute more often overall?

> **Lines 3 to 6**
>
> In a sorted array, the loop will always find that $A[j] \leq x$ (since x is the last element), causing lines 5 and 6 to execute for each element in the array segment, leading to more swaps.

> **Line 7**
>
> Line 7 will also execute more often as it is outside the loop and will be executed each time the `PARTITION` function is called.

EXPECTED RUNNING TIME

EXPECTED RUNNING TIME, the Costs of functions `Quicksort(A,p.r)` , and `Partition(A,p,r)` , and the number of comparisons

Corollary: Expected running time of Quicksort is $n + E[X]$

Randomized Quicksort

Introduction

- **Definition:** An algorithm that uses randomness as part of its logic.
- **Benefits:**
    - Removes bias from data, making it appear uniformly distributed.
    - Average case becomes the most likely scenario.
- **Implementation:** Each partition selects the pivot randomly.

Timing Analysis

- **Best Case:**
    - Occurrence: When each pivot is the median of the segment under consideration.
    - Recurrence: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
    - Running Time: $\Theta(n \log n)$

- **Worst Case:**
  - Occurrence: Very specific data conditions.
  - Running Time: $\Theta(n^2)$
- **Average Case:**
  - Recurrence: Based on the random choice of pivots, the average depth of the recursion tree is about $2 \log n$.
  - Running Time: $\Theta(n \log n)$

Lemma

- **Statement:** If line 4 of the partition is executed X times, the running time is $O(n + X)$.
- **Proof:**
  - Outside loop work: $O(1)$
  - Loop execution: X times

Computing E[X]

Definition: $E[X]$ is just the total number of comparisons performed.

Expression:
$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{(j-i+1)}$

Let elements of A be labeled $Z_1, Z_2, ..., Z_n$, where $rank(z_i) = i$

Assumes distinct keys

Let $X_{ij} = 1$ if z_i is compared to z_j (0 otherwise)

Express X in terms of X_{ij}

$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

$E[X] = E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$

Once z_i and z_j are in different partitions, they cannot be compared

Why?

Once partitioned, do not compare

Comparisons only happen from p to r (i.e., w/in partition)

Partitions do not get merged

Let $Z_{ij} = \{Z_i, ..., Z_j\}$, When are z_i and z_j are compared ?

z_i and z_j are compared when they are in the same partition and 1 of them is the pivot

$E[X_{ij}] = Pr\{z_i$ or z_j is 1st pivot in $Z_{ij}\}$

$= Pr\{z_i$ is 1st pivot in $Z_{ij}\}$ + $Pr\{z_j$ is 1st pivot in $Z_{ij}\}$

$Pr\{z_i$ is 1st pivot in $Z_{ij}\} =$?

$Pr\{z_j$ is 1st pivot in $Z_{ij}\} =$?

Assume each element is equally likely to be a pivot

Each probability is $\frac{1}{(j-i+1)}$

So $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{(j-i+1)}$

$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{(j-i+1)}$

2. Let $k = j - i$
   $E[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{(k+1)}$
   $< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$ note < and sum bounds changed and denominator changed
   $= \sum_{i=1}^{n-1} O(\log(n))$

Thus, $E[x] = O(n\log(n))$

Result: $E[X] = O(n \log n)$

Running time of Quicksort $= n + O(n\log(n)) = O(n\log(n))$

Runtime of Quicksort is $O(n + X)$
$X =$ total number of iterations of Partition loop

Randomized-Quicksort has expected runtime of $E[n + X] = O(n + E[X])$
Counting expected number of comparisons gives $E[X] = O(n\log(n))$

Thus, Randomized-Quicksort has expected runtime of $O(n + n\log(n)) = O(n\log(n))$

SUMMARY

Quicksort is usually an efficient algorithm

- Under most cases, the runtime is $O(n\log(n))$
- Very rarely, the runtime can be $\Theta(n^2)$

Disregarding, stack space from recursive calls, the algorithm is in place

- Even taking the stack space into account, the space is usually $O(\log(n))$, which grows very slowly

This is the most popular sorting algorithm for general input values

- Sometimes with variations

We can do better if we know things about the input, though!

Conclusion

- **Runtime:** $O(n \log n)$
- **Space:** Usually $O(\log n)$, which grows very slowly.
- **Popularity:** The most popular sorting algorithm for general input values, sometimes with variations.