CSL 3010: Design and Analysis of Algorithms July-November 2022

Lecture 5: Quicksort and Recursion

Lecturer: Dr. Saket Saurabh *Scribe: Mukul, Sawan, Jaimin, Tanay*

1 Quicksort

Input: Given an array A consisting of numbers in unsorted manner
Output : Return the sorted array A.

Example case 1: A = [8, 6, 7, 5, 3, 10, 9]
O/P: A = [3, 5, 6, 7, 8, 9, 10]

Example case 2: X = [11, 6, 5, 1, 7, 13, 12]
O/P: A = [1, 5, 6, 7, 11, 12, 13]

Approach : We split the array into smaller subarrays before recursion, so that merging the sorted subarrays is trivial.

- 1.) Choose a pivot element from the array
- 2.) Partition the array into three subarrays containing the elements smaller than the pivot, the pivot element itself, and the elements larger than the pivot.
- 3.) Recursively quicksort the first and last subarrays.

Pseudo code:

```
def Partition(A[1 ... n], p):
    swap A[p] with A[n]
    l = 0
    for i = 1 to n-1:
        if A[i] < A[n]:
            l = l + 1
            swap A[l] with A[i]
    swap A[n] with A[l+1]
    return l+1
```

Listing 1: Pseudo code for partitioning

```
def Quicksort(A[1 ... n]):
    if n > 1:
        choose a pivot element A[p]
        x = Partition(A,p)
        Quicksort(A[1 ... x-1])
        Quicksort(A[x+1 ... n])
```

Listing 2: Pseudo code for Quicksort

Proof of correctness :-
We need to first prove that the Partition works correctly in partitioning the array and then assuming it dos we'll prove quicksort is correct.
We'll prove that partition is correct by using loop invariant

- **initialization :** Before the loop, all the conditions of the loop invariant are satisfied, because x is the pivot and the subarrays A[1 ... x-1] and A[x+1 ... n] are empty.

- **maintainance :** After each iteration in the loop, if A[i] is less than equal to A[n], then l is incremented and the values at l'th index and i'th index are swapped
- **termination :** When the loop terminates, i = n and we make a final swap of A[n] with A[l+1], so all elements in A are partitioned into one of the three cases:
  - A[1 ... p] is <= pivot
  - A[p+1 ... r] is > pivot
  - A[p] = pivot

  Now let's prove that **quicksort is correct by method of Induction**

  P(n) = Quicksort is always correct for arrays of length n

  - **For n = 1**
    P(1) is single element which is always sorted, so correct for n = 1
    Let's assume P(k) is correct for k < n.
  - So, P(n) holds because, say the length of array on left of pivot is x and on right of pivot is y, then since x < n and y < n, so by our hypothesis left and right array are correctly sorted
  - The partition loop guarantees that the algorithm is correct and the pivot we've selected is correct so that is also in right position and hence the complete array is sorted.

1.1 Analysis

Partition function clearly runs in O(n) time, because it involves a simple for .loop For QuickSort, we get a recurrence that depends on r, the rank of the chosen pivot element :

$$T(n) = T(r\text{-}1) + T(n\text{-}r) + O(1)$$

The **worst case** of the algorithm is when the **array is already sorted**, in this case the recursive calls split the array into 1 and n-1 elements at each recursive call. In this case, we have:

$$\begin{gathered} T(n) = cn + T(1) + T(n - 1) \\ = cn + T(1) + c(n - 1) + T(1) + T(n - 2) \\ = c(n + (n - 1)) + 2T(1) + T(n - 2) \\ = c(n + (n - 1) + (n - 2) + ...) \\ \text{Which will eventually evaluate to} \\ \mathbf{T(n) = O(n^2)} \end{gathered}$$

If we could somehow always magically choose (**Lucky Hit**) the pivot to be the median element of the array A (or an element whose rank is in the range of [n/4, 3n/4]), we would have r = ceil(n/2), the two subproblems would be as close to the same size as possible, the recurrence would become

$$T(n) = T((n/2)\text{-}1) + T((n/2)) + O(n) \leq 2T((n/2)) + O(n)$$

and we'd have $\mathbf{T(n) = O(nlogn)}$ (solved various times)

2 Recursion Trees

- Tree Method
  - elementary methods, geometric series etc

- integration

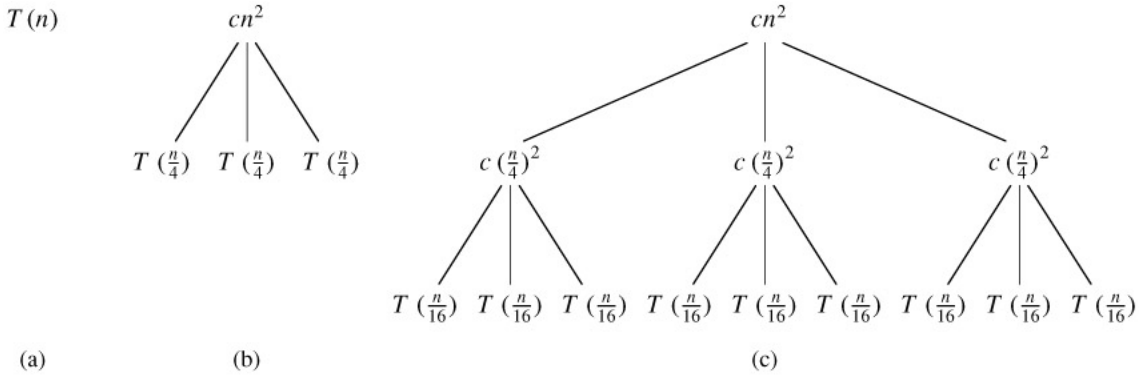
- Divide and Conquer
  - Guessing involves intuition, trial and error
  - verification is via induction

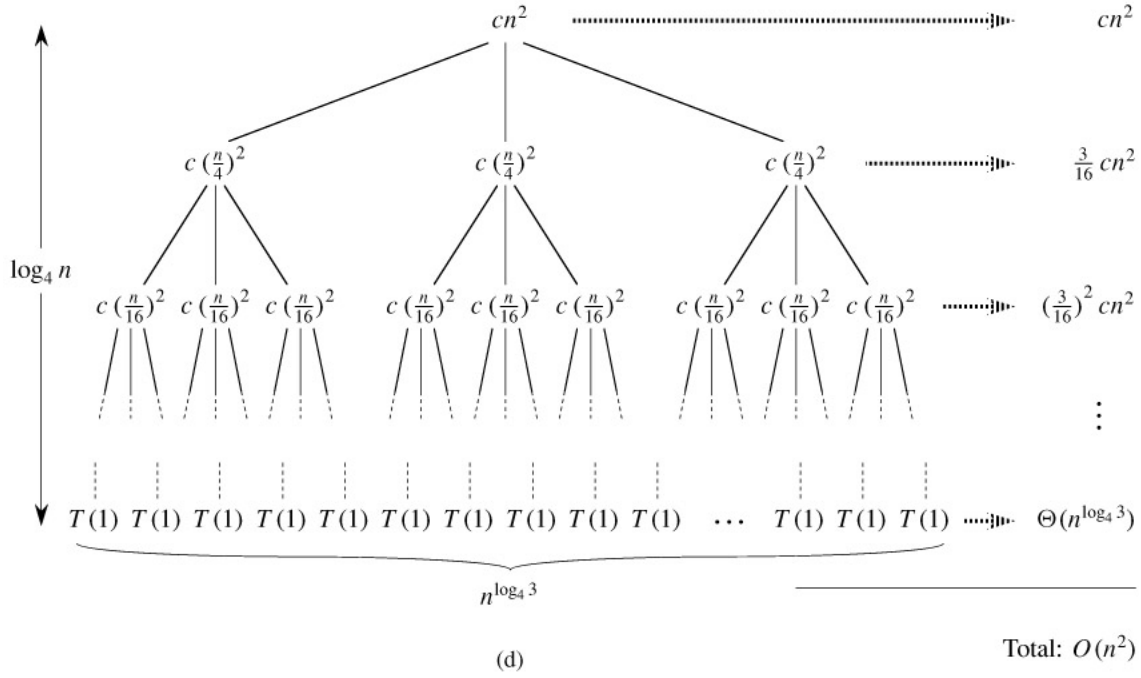
so, closer look on Divide and Conquer Algorithm

- spends O(f(n)) time on non recursive tasks
- makes r recursive calls on each problem of size n/c, so recurrence relation is

$$T(n) \leq rT(n/c) + O(f(n))$$

Some examples of what a recursion tree looks like are given below :





Some important points to consider :-

- The base cases of the recurrence are represented by the leaves of recurrence tree.
- T(n) is the sum of all values in the recursion tree, we can evaluate this sum by considering the tree in level-wise manner. For each integer i, the i'th level of the tree has exactly r^i nodes, each with value $f(n/c^i)$.*Thus*,

$$T(n) = \sum_{i=0}^{L} r^i.f(n/c^i)$$

where L is the depth of the tree. base case n = 1 implies L = $\log_c n$, since number of leaver in recursion tree is

$$r^L = r^{log_c n} = n^{log_c r}$$

Let's now see 3 cases where this level-wise consideration is easy to conclude :-

- **Decreasing:** If the series decays exponentially, sum is dominated by the value at the root of the recursion tree and every term is a constant factor smaller than the previous term, then **T(n) = O(f(n)).**
- **Equal:** If all terms in the series are equal, we immediately have **T(n)= O(f(n).L) = O(f(n)log n)**
- **Increasing:** If the series grows exponentially, sum is dominated by the number of leaves in the recursion tree and every term is a constant factor larger than the previous term then **T(n) = O($n^{log_c r}$)**.

2.1 Solving Recurrence Relations

2.1.1 Example 1

Input: T (n) = T($\sqrt{n}$) + 1 for $n > 2$ T(2)=1

T (n) = T($\sqrt{n}$) + 1 for $n > 2$ T(2)=1
From this we will have
$\sqrt{n}$, $\sqrt{\sqrt{n}}$,$\sqrt{\sqrt{\sqrt{n}}}$, ... O(1)
$n^{\frac{1}{2}}$, $n^{\frac{1}{2^2}}$, $n^{\frac{1}{2^3}}$, ...
so for L steps
$n^{\frac{1}{2^L}} = 2$
log $n^{\frac{1}{2^L}} = log_2 2$
L=$log_2(log_2 n)$
T(n)=O($log_2(log_2 n)$)

2.1.2 Example 2

Input: T(n) = $\sqrt{n}.T(\sqrt{n})$+ n, for n > 2, T(2) = 1

T(n) = $\sqrt{n}.T(\sqrt{n})$+ n
T($\sqrt{n}$) = $n^{1/4}.T(n^{1/4}) + \sqrt{n}$
T($n^{1/4}$) = $n^{1/8}.T(n^{1/8}) + n^{1/4}$
so if we again consider T(n)
T(n) = $\sqrt{n}.T(\sqrt{n})$+ n
= $n^{3/4}.T(n^{1/4}) + n + n$
= $n^{7/8}.T(n^{1/8}) + n + 2n$
after k steps we'll get

.

.

$= \mathrm{n}^{1-1/2^k}.T(n^{1/2^k}) + kn$

let $\mathrm{n}^{1/2^k} = 2$

$=> \log_2 n = 2^k$

$=> \log_2 log_2 n = k$

$=> \mathrm{T(n)} = (\mathrm{n}/2).1 + \mathrm{n}\log_2 log_2 n$

$=> \mathbf{T(n) = O(n\log_2 log_2 n)}$
