Randomized Quick Sort

CS 5633 Analysis of Algorithms

Computer Science
University of Texas at San Antonio

September 18, 2024

Randomized Algorithms

Deterministic Algorithms Analysis

- Last time we considered the average case running time of a deterministic algorithm.
- That is, we assumed that the input was uniformly distributed (any permutation of the input is equally likely to occur in the case of the hiring problem), and we analyzed the running time of the algorithm of a randomly chosen input.
- Consider the average case of insertion sort when any permutation of the numbers in the array is equally likely to occur.

Deterministic Algorithms Analysis cont

- Consider the average case of insertion sort when any permutation of the numbers in the array is equally likely to occur.
  - For the *j*'th element to be inserted into the sorted portion, we assume on average, half ($\frac{j}{2}$) of the sorted portion is greater that *j*'th element.

  - That is, on average, there are $\frac{j}{2}$ comparisons to insert *j*'th item. So the total run time is $\sum_{j=2}^{n} \frac{j}{2} = \Theta(n^2)$.

The Downside of Deterministic Algorithms cont.

- There are a few drawbacks to analyzing algorithms in this way.
  - The distribution of inputs are heavily reliant on the application. Therefore the analysis may be misleading for certain applications.
  - For many applications, it is difficult to determine if the inputs come from some “well-behaved” distribution, and if we are wrong about the distribution then our analysis of the running time could be significantly off base.

From Deterministic to Random.

- One approach to deal with this would be to “shuffle” the input before running the algorithm. Essentially we can force the input to come from a known distribution. Of course the downside to this is we pay a price in the running time of the algorithm.
- A better approach to deal with this is to use a **randomized algorithm**. That is, an algorithm which uses a random number generator to determine some of the steps of the algorithm.
  - For example in insertion sort, instead of inserting the element in position j, randomly choose a one of the last $n - j$ unsorted elements to insert.

Randomized Algorithm Analysis

- The running time of a randomized algorithm is a random variable. We are interested in determining the **expected running time** of the algorithm. That is, let X be the random variable that measures the running time of a randomized algorithm. We are interested in bounding $E[X]$.
- Now we do not need to make any assumptions about the input distribution.

Randomized Algorithm Analysis cont.

- No specific input will force the algorithm's worst-case behavior (although no input will force the algorithm's best-case behavior either). The worst-case (and best-case) are determined only by the output of a random number generator.
- For these reasons, it is generally considered better to analyze the expected running time of a randomized algorithm rather than the average case running time of a deterministic algorithm.
- We will learn one example of randomized algorithm: quick sort.

Deterministic Quick Sort

The Basic Idea of Quick Sort

- For simplicity, let's start with deterministic Quick Sort.
- Quick Sort is quite similar to merge sort in that it also seeks to partition the target array into two sorted sub-arrays.
- The steps of Quick Sort:
  1. Pick the last element as the **pivot**.
  2. Partition the array into two sub-arrays: one sub-array with elements all smaller than pivot; one sub-array with elements are larger than pivot.
  3. Recursively sort the two sub-arrays with Quick Sort.
  4. Combine the sorted sub-arrays with the pivot to get the sorted array.

An Illustration of Quick Sort

1. The original unsorted array:

1	5	7	3	6	2	4

2. Pick 4 as the pivot and partition the array:

1	3	2	**4**	5	7	6

3. Recursively sort sub-arrays {1,3,2} and {5,7,6}, and join the sorted sub-arrays with the pivot.

1	2	3	**4**	5	6	7

Quick Sort Algorithm

- The Quick Sort algorithm:

Algorithm 1: Quick Sort

```
1 Function Quick_Sort(Array A[1 : n])
2     if n <= 1 then
3         return A;
4     Array B, C = Partition(A);
5     Quick_Sort(B);
6     Quick_Sort(C);
7     A = B + A[n] + C;
```

Quick Sort Algorithm cont.

- The Partition function:

Algorithm 2: Partition

```
1  Function Partition(Array A[1 : n])
2      pivot = A[n];
3      Array B = [], C = [];
4      for i ← 1 to n − 1 do
5          if A[i] < pivot then
6              Add A[i] to B;
7          else
8              Add A[i] to C;
```

Quick Sort Performance

- Let's focus on the Quick_Sort function.
- The run time of Quick_Sort is $T(n) = T(|B|) + T(|C|) + Partition$.
  - With in Partition, there are always $n - 1$ comparisons.
  - Therefore the run time of Quick_Sort is $T(n) = T(|B|) + T(|C|) + (n - 1)$.
- Clearly, the run time depends on the size of B and C.

Quick Sort Worst-case Performance

- In the worst case, A is reversely sorted.
  - In other words, there is no element smaller than $A[n]$.
  - So we always have $|B| = 0$ and $|C| = n - 1$.
- The worst-case run time of Quick_Sort is,

$$\begin{aligned} T(n) &= T(|B|) + T(|C|) + (n-1) \\ &= T(0) + T(n-1) + (n-1) \\ &= T(n-1) + (n-1) \\ &= T(0) + T(n-2) + (n-2) + (n-1) \\ &\dots \\ &= 1 + 2 + \dots + (n-2) + (n-1) = O(n^2) \end{aligned} \tag{1}$$

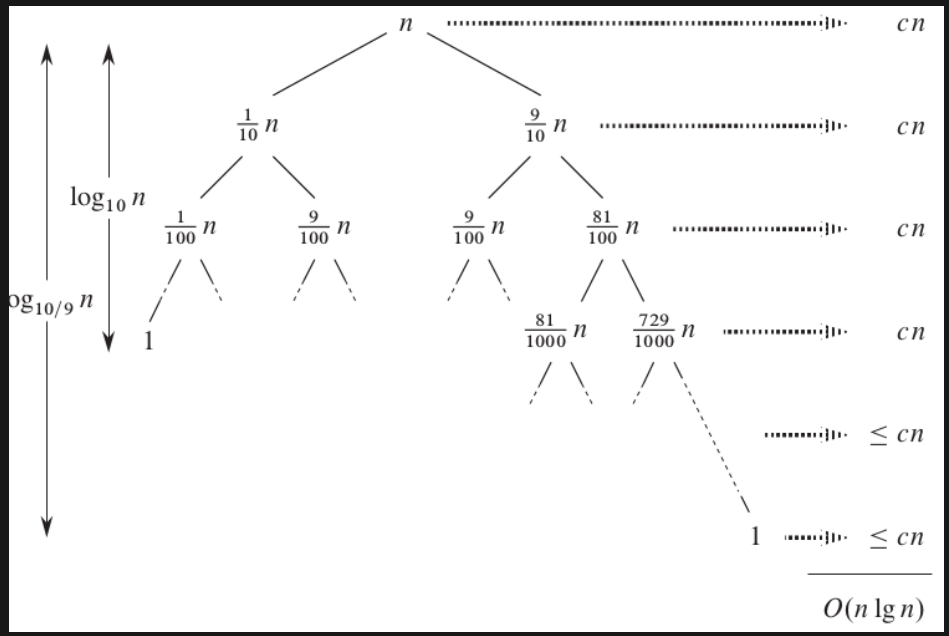
Quick Sort Best-case Performance

- In the best case, we always partition an array into two sub-arrays of the same size.
  - In other words, there are same numbers of elements smaller and larger than $A[n]$.
  - So we always have $|B| = |C| = \frac{n-1}{2}$.
- The best-case run time of Quick_Sort is,

$$\begin{aligned} T(n) &= T(|B|) + T(|C|) + (n-1) \\ &= T(\frac{n}{2}) + T(\frac{n}{2}) + (n-1) \\ &= O(n \lg n), \text{similar to merge sort} \end{aligned} \tag{2}$$

Quick Sort Common-case Performance

- Most of the time, Quick Sort has a performance close to $O(n \lg n)$.
- To see why, let's consider a case where we always has 10% elements in B, and 90% elements in C.
  - That is,
    $T(n) = T(|B|) + T(|C|) + (n-1) = T(\frac{n}{10}) + T(\frac{9n}{10}) + O(n)$.
- The recursive tree for $T(n)$ is,



Quick Sort Common-case Performance cont.

- The sorted path has $\log_{10} n$ levels.
- The sorted path has $\log_{10/9} n$ levels.
- Each level the cost is no more than $O(n)$ or $c \cdot n$, since we have at most n elements at the level.
- The execution time is then bounded by the longest path which has $\log_{10/9} n$ levels of $c \cdot n$. So the max run time is $T(n) = \log_{10/9} n \cdot c \cdot n = O(n \lg n)$ (log base is not important).
- This run time means that as long as we don't run into worse case, Quick Sort is usually very fast.
- In other words, we should try to make sure that we won't run into worse case most of the time. This is where randomized algorithm comes in handy.

Randomized Quick Sort

Randomized Quick Sort

- We only need to randomized the Partition function by adding lines 2-3:

Algorithm 3: Randomized Partition

```
1  Function R-Partition(Array A[1 : n])
2      randomly generate j, 1 ⩽ j ⩽ n;
3      pivot = A[j];
4      swap A[j] and A[n];
5      Array B = [], C = [];
6      for i ← 1 to n − 1 do
7          if A[i] < pivot then
8              Add A[i] to B;
9          else
10             Add A[i] to C;
```

Randomized Quick Sort Performance Intuition

- By randomized picking the pivot, we make it less likely that we always pick the smallest element as the pivot.
- An intuition of the run time for randomized Quick Sort:
  - The run time depends on the probability that we run into the worst case and common cases.
  - That is, the run time is, $T(n) = P(worst\_case)T(worst\_case) + P(common\_case)T(common\_case) = P(worst\_case)O(n^2) + P(common\_case)O(n \lg n)$.

Randomized Quick Sort Performance Intuition cont.

- In the worst case, we select the smallest element as the pivot. So the probability is $\frac{1}{n}$, which is roughly 0, when n is sufficiently large.
- In the common case, we do not select the smallest element. The probability of the common case is $\frac{n-1}{n}$, which is roughly 1 when n is sufficiently large.
- Put the probabilities back into the run time equation, we have $T(n) = O(n \lg n)$.

Randomized Quick Sort Performance

- A more formal analysis of randomized Quick Sort relies on determine the number of comparisons performance in R-Partition.
  - That is, the run time of randomized Quick Sort is the same as the number of executions of line 7 of R-Partition.
  - Note that, for all comparison-based sorting algorithms, we always analyze the run time based on the number of comparisons, because comparison is the most complex operation here.
- The comparison in line 7 is always about comparing an element with the pivot!

Randomized Quick Sort Performance cont.

- Let's consider the elements of A in sorted order.
  - Let Z be the list of elements in sorted order. That is $Z_i \leqslant Z_j$ if and only if $i \leqslant j$.
  - Note that we don't need to sorted the array to do the analysis. It is just more convenient when later we compute the probabilities.
- Let I_{ij} be an indicator random variable that represents whether two arbitrary elements, Z_i and Z_j, are compared in R-Partition.
  - That is,

$$I_{ij} = \begin{cases} 1, & \text{if } Z_i \text{ and } Z_j \text{ are compared in R-Partition} \\ 0, & \text{if } Z_i \text{ and } Z_j \text{ are NEVER compared in R-Partition} \end{cases}$$

Randomized Quick Sort Performance cont.

- The total number of comparisons, or the run-time of randomized Quick Sort, is then

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[I_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(Z_i \text{ is compared to } Z_j) \end{aligned} \tag{3}$$

- Since line 7 of R-Partition only compares an element to the pivot, if Z_i is compared to Z_j, then either Z_i and Z_j must be the pivot. That is,

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (P(Z_i \text{ is pivot}) + P(Z_j \text{ is pivot}))$$

Randomized Quick Sort Performance cont.

- For Z_i to be a pivot, and for it to be compared to Z_j as a pivot, it requires:
  - Z_j cannot be a pivot before Z_i is a pivot.
  - Any element $Z_l, Z_i \leqslant Z_l \leqslant Z_j$, cannot be a pivot before Z_i is pivot. Otherwise, Z_i and Z_j will be in two sub-arrays and will never be compared.
  - That is Z_i has to be the first pivot for all elements between (including) Z_i and Z_j (recall that Z is sorted).
  - Given there are $j - i + 1$ elements between Z_i and Z_j, the probability that Z_i is the first pivot is $\frac{1}{j-i+1}$. That is, $P(Z_i \text{ is pivot}) = \frac{1}{j-i+1}$.
- Similarly, for Z_j to be a pivot, and for it to be compared to Z_i as a pivot, it requires:
  - Z_j has to be the first pivot for all elements between (including) Z_i and Z_j.
  - That is, $P(Z_j \text{ is pivot}) = \frac{1}{j-i+1}$.

Randomized Quick Sort Performance cont.

- Put the probabilities back into the run time equation:

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (P(Z_i \text{ is pivot}) + P(Z_j \text{ is pivot})) \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (\frac{1}{j-i+1} + \frac{1}{j-i+1}) \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (\frac{2}{j-i+1}) \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} (\frac{2}{k+1}) \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} (\frac{2}{k+1}) \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n)
\end{aligned}
\tag{4}
$$

In-place Quick Sort Partition

- The trick of the in-place partitioning depends on two pointers *i* and *j*.
  - Pointer *j* is used to scan over *A* from beginning to the end to examine each element.
  - Pointer *i* is used to record the number of elements that are smaller than the pivot. *i* is also the index of the current array slot that can be used to store the element that is smaller than the pivot.

Quick Sort V.S. Merge Sort

- Quick Sort
  - Linear time split.
  - Constant time combine.
  - In-place sort, no extra space, except for recursive call stack (which is $O(\lg n)$.
  - Performance is un-deterministic, i.e., has a slim chance to deteriorate to $O(n^2)$.
- Merge Sort
  - Constant time split.
  - Linear time combine.
  - Very hard to be in-place. The in-place version requires significant data moves.
    - Data moves are extremely expensively today.
    - Therefore, merge sort is usually implemented to use extra $O(n)$ space.
    - Check “In-place sorting with fewer moves” by Katajainen and Pasanen, 1999, if you are interested.
  - Performance is stable.

Quick Sort V.S. Merge Sort cont.

- Real life performance depends on computer architecture features.
  - Quick Sort may be faster, because it is in-place and more cache friendly.
  - According to “The Algorithm Design Manual” , Quick Sort is faster because the in-place inner-loop is simpler (at least one less memory read, but I don’t have first hand experience).